11 Microprocesorul Intel 8086

11.1 Generalități

Lansat în 1978 de firma *Intel*, se prezintă sub forma unei capsule cu 40 de pini, este realizat în tehnologia HMOS şi are în structură circa 29.000 tranzistoare, pe o suprafață de siliciu de 37 mm^2. Apariția lui a fost urmată la scurt timp de o familie de componente:

Intel 8284 - generator de tact;

Intel 8288 - controler de magistrale;

Intel 8087 - coprocesor aritmetic;

Intel 8089 - coprocesor de intrare-ieşire .

În 1979 apare microprocesorul *Intel* 8088 care păstrează caracteristicile lui 8086 dar cu magistrala de date externă este de 8 biți. A cunoscut o largă utilizare prin includerea sa în numeroase produse ale firmei IBM.

Microprocesorul Intel 8086 este cel mai răspândit microprocesor pe 16 biți; registrele interne şi magistralele de date interne şi cea externă sunt de 16 biți.

Caracteristici tehnice principale :

- multiplexarea în timp a magistralelor de date, adrese şi stări, pentru păstrarea capsulei de 40 de pini;
- magistrala de adrese de 20 de biți, ceea ce permite adresarea unei memorii de capacitate maximă de 1 MB;
- o singură tensiune de alimentare : + 5 Vcc;
- frecvența semnalului de tact: 4 MHz, 5 MHz sau 8 MHz, în funcție de variantă;
- compatibilitate cu limbajul de asamblare al microprocesorului Intel-8080 şi Intel-8085.
- setul de instrucțiuni conține 94 de tipuri de instructiuni, inclusiv operații aritmetice în cod BCD şi operații de înmulțire, împărțire;
- operează cu digiți (cod BCD, 4 biți/digit sau 8 biți/digit), cu octeți (byte), cu cuvinte de 16 biți (word), cu cuvinte duble de 32 de biți (double word), şiruri de caractere de 8 biți (string) şi blocuri de date.
- acoperă o gamă largă de aplicații datorită celor două moduri de lucru: *modul minim* pentru aplicații simple, în care procesorul generează el însuşi semnalele electrice necesare transferului de date cu memoria şi porturile de intrare/ieşire şi *modul maxim*, pentru aplicații complexe, inclusiv sisteme multiprocesor, în care semnalele

de comandă pentru memorie şi porturi sunt generate de un circuit specializat, 8288 (controler de magistrale).

11.2 Structura internă

Microprocesorul Intel-8086 cuprinde două unităţi funcţionale care lucrează asincron şi independent una faţă de cealaltă:

- Unitatea de execuţie EU (*Execution Unit*), care efectuează operaţiile conţinute codificat în instrucţiuni.
- Unitatea de interfaţă cu magistralele (*Bus Interface Unit*), care are rolul de a extrage instrucţiunile din memorie şi de a transfera operanzii între unitatea de execuţie şi memorie sau porturi de intrare/ieşire.

11.2.1. Unitatea de interfaţă cu magistralele BIU (*Bus Interface Unit*)

Realizează conectarea microprocesorului cu exteriorul prin intermediul magistralelor de adrese (20 de linii) şi de date (16 linii). De asemenea, unitatea BIU generează semnalele de comandă pentru realizarea operaţiile de citire şi scriere cu memoria sau cu porturile.

Unitatea BIU realizează extragerea în avans a instrucţiunilor din memorie, pe care le stochează într-un fişier de instrucţiuni de 6 octeţi, care este de fapt o listă de tip FIFO (*First In First Out*).

Dacă în acest fişier sunt cel puţin două locaţii libere şi unitatea de execuţie nu solicită transfer de operanzi cu exteriorul, unitatea BIU va iniţia un ciclu maşină de extragere în avans a unei instrucţiuni din memorie, de la adresa următoare. Se asigură astfel un important câştig de timp, prin suprapunerea execuţiei cu extragerea instrucţiunilor, operaţie care necesită timp de acces la memorie şi timp de transfer.

Instrucţiunile de salt pot modifica succesiunea normală prin comutarea execuţiei la o adresă "de salt"; în această situaţie, unitatea BIU iniţializează fişierul prin ştergerea instrucţiunilor existente şi încărcarea lui cu o nouă secvenţă de instrucţiuni extrase începând cu adresa de salt.

Dacă în timpul execuţiei unei instrucţiuni unitatea EU solicită un operand din memorie, unitatea BIU calculează adresa fizică (20 de biţi) în funcţie de modul de adresare comandat, utilizând registrele proprii şi unitatea aritmetică proprie (UA) şi generează semnalele de comandă pentru transferul operandului către unitatea de execuţie.

11.2.2. Unitatea de execuție EU (*Execution Unit*)

Extrage succesiv instrucțiunile din fişierul de instrucțiuni, le decodifică pe baza unui microprogram rezident şi le execută prin intermediul registrelor de uz general şi unității aritmetice şi logice (UAL).

Dacă pentru execuția unei instrucțiuni este necesar accesul la memorie sau la porturi I/O, unitatea de execuție transmite către BIU o adresă de 16 biți (adresă efectivă sau *offset*) ce va fi utilizată pentru operațiile de transfer.

Unitatea de execuție dispune de un bloc de comandă care coordonează funcționarea unității. În acest bloc există o memorie ROM în care este stocat microcodul de interpretare şi execuție pentru fiecare instrucțiune.

11.2.3. Setul de registre

Registrele sunt specializate pe funcții; ele pot fi grupate în 5 categorii:

- registre de date (AX, BX, CX, DX);
- registre index, pentru accesul în interiorul unui segment (SP, BP, DI, SI);
- registre de segment (CS , DS , SS , ES) ;
- registru indicator de adresă (IP);
- registru de stare (F).

Primele două categorii alcătuiesc registrele de uz general.

Utilizatorul are acces la toate registrele, dimensiunea lor fiind de 16 biți, egală cu dimensiunea magistralei de date.

Registrele de date. Deşi sunt registre de 16 biți, pot fi utilizate ca două registre de 8 biți: registrul superior H (High) şi registrul inferior L (Low). Oricare din registrele de uz general poate fi utilizat în operații aritmetice şi logice dar au şi funcții specifice, care nu pot fi modificate de programator (tab.1).

Tabel 1. Funcțiile specifice ale registrelor de uz general

Registrul	Funcția specifică
AX	Înmulțire, împărțire, intrare / ieşire pe cuvânt
AL	Înmulțire, împărțire, intrare / ieşire pe octet
AH	Înmulțire, împărțire, pe octet
BX	Translatare (adresare indexată)
CX	Contor pentru operații cu şiruri, bucle
CL	Contor pentru deplasări, rotații
DX	Înmulțire, împărțire, intrare / ieşire indirectă

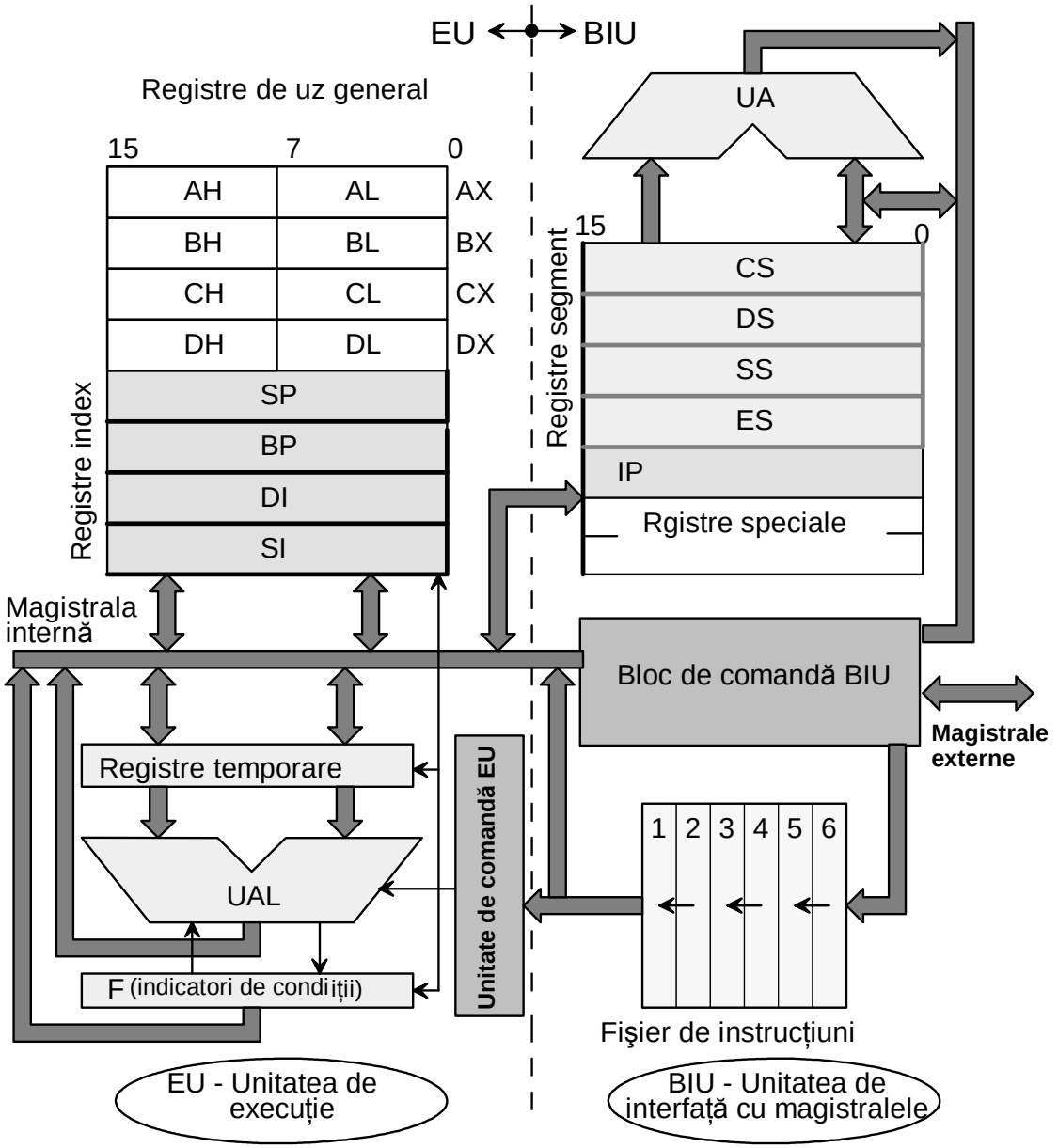


Fig.1 Arhitectura internă a microprocesorului Intel 8086

Registrele de segment

Sunt registre de 16 biți care conțin adresa de bază a unui segment de memorie. Memoria direct adresabilă de 1 MB necesită o magistrală de adrese de 20 biți.

Registrele interne ale procesorului fiind de 16 biți, memoria este divizată din punct de vedere logic în segmente de 64 kB, fiecare segment fiind astfel adresabil cu 16 biți. Procesorul poate să utilizeze simultan atâtea segmente de memorie câte registre de segment posedă.

Adresele de început (de bază) ale celor 4 segmente sunt conținute în cele 4 registre de segment (Cod, Date, Stivă şi de Date suplimentar).

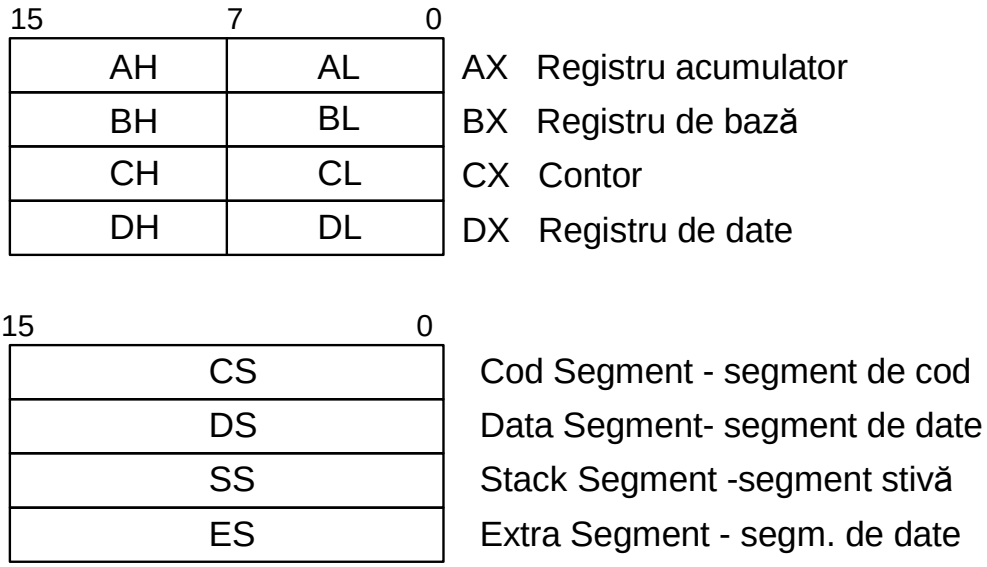


Fig. 2. Registrele de date şi de segment la Intel 8086

Registrul CS conține adresa de început a segmentului de cod unde se află codurile instrucțiunilor.

Pentru a adresa o instrucțiune, microprocesorul combină conținutul registrului segment de cod CS cu al registrului indicator de adresă, IP, obținând o adresă fizică de 20 biți .

Registrul DS conține adresa de început a segmentului de date, SS conține adresa de început a segmentului stivă iar ES conține adresa de început a unui segment de date suplimentar. Segmentul de cod conține instrucțiuni iar ultimele trei segmente de memorie sunt dedicate operanzilor (date).

Registrele pentru accesul în interiorul unui segment.

Împreună cu registrele de date, reprezintă registrele generale; SP şi BP sunt registre indicator iar SI şi DI sunt registre index .

Registrele indicator conțin adresa relativă față de baza segmentului la care se află un operand în cadrul segmentului stivă; SP indică adresa relativă a vârfului stivei în raport cu conținutul registrului SS (baza stivei), iar BP indică adresa relativă a unui operand în cadrul segmentului stivă.

Registrele index conțin adresa relativă față de baza segmentului la care se află un operand în cadrul segmentelor de date sau de date suplimentar, deci în raport DS sau ES; SI şi DI se utilizează în general în cazul operațiilor cu şiruri de date, SI indicând adresa operandului sursă iar DI adresa operandului destinație.

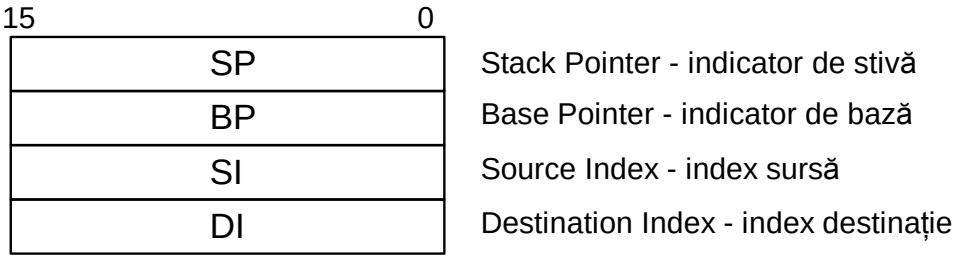


Fig. 3. Registrele indicator şi index la Intel 8086

Registrul indicator de adresă IP (*Instruction Pointer*)

Este similar registrului de adresă PC (*Program Counter*) de la Intel 8080 sau Zilog Z80, indicând adresa instrucțiunii ce se extrage din segmentul de cod. Spre deosebire de PC, IP nu conține adresa fizică a instrucțiunii ci adresa relativă față de baza segmentului de cod. Conținutul lui IP se combină cu cel al lui CS şi astfel se obține adresa fizică a instrucțiunii. După transferul fiecărui octet în fişierul de instrucțiuni, conținutul lui IP creşte cu o unitate: IP = IP + 1, fiind astfel pregătit pentru adresarea octetului următor din segmentul de cod.

Registrul de stare sau al indicatorilor de condiții F (*Flags*)

Acest registru face parte din unitatea aritmetică şi logică UAL (sau ALU - *Arithmetic and Logic Unit*). Deşi este un registru de 16 biți, doar 9 sunt semnificativi, reprezentând indicatorii de condiții ai procesorului 8086.

Indicatorii se poziționează în "0" sau "1" după efectuarea unei operații aritmetice, logice sau după o instrucțiune de control; pot fi testați prin intermediul instrucțiunilor condiționale şi se pot lua decizii în funcție de valoare lor.

CF: (*Carry Flag*) indicatorul de transport; se activează (în "1" logic) la apariția unui bit de transport (depăşirea lungimii normale a rezultatului) la adunare sau de împrumut la scădere; este asociat rezultatului unei operații aritmetice sau logice şi se comportă ca al noulea bit (b8) al acestuia.

PF: (*Parity Flag*) indicator de paritate; se activează (în "1" logic) când rezultatul unei operații aritmetice sau logice este un octet cu număr par de unități.

AF: (*Auxiliary Flag*) indicator de transport la jumătate; se activează când apare transport sau împrumut în operațiile aritmetice şi logice între biții 3 şi 4.

ZF: (*Zero Flag*) indicator de zero; se activează (în "1" logic) dacă rezultatul unei operații aritmetice sau logice este zero.

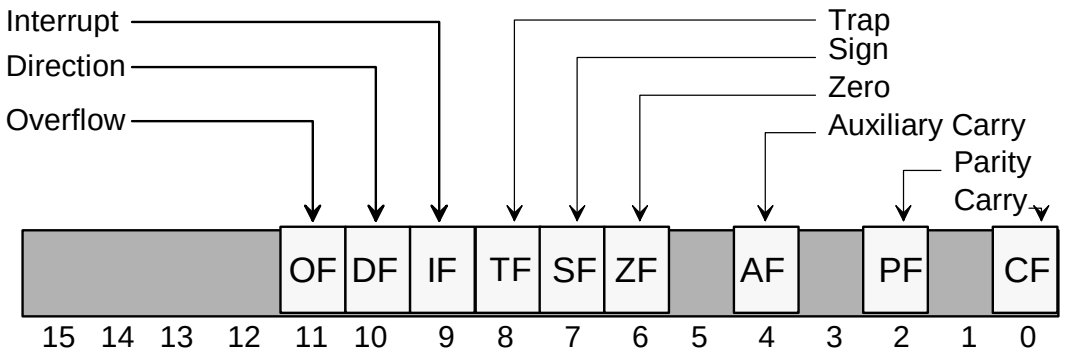


Fig.6. Registrul indicatorilor de condiții la microprocesorul Intel 8086/8088

SF: (*Sign Flag*) indicator de semn; conține bitul cel mai semnificativ al rezultatului, deci copiază bitul de semn; va fi "1" pentru rezultat negativ şi "0" pentru un rezultat pozitiv.

OF: (*Overflow Flag*) indicator de depăşire; se activează (în "1" logic) când apare depaşirea capacității registrului ce conține rezultatul, ca urmare a unei operații aritmetice cu operanzi cu semn; va fi "1" când apare transport sau împrumut în/din rangul de semn.

TF: (*Trap Flag*) indicator pentru modul de lucru "pas cu pas"; dacă este poziționat în "1" logic, microprocesorul nu mai lucrează la viteza dată de semnalul de tact ci în ritmul impus de utilizator. Acest mod de lucru este deosebit de util la depanarea programelor.

IF: (*Interrupt Flag*) indicator pentru controlul întreruperilor; dacă este poziționat în "1" logic (prin instrucțiunea corespunzătoare), sunt validate cererile de întrerupere de tip INTR (mascabile); invalidarea acestora se face prin poziționarea lui IF în "0", caz în care cererile INTR nu vor fi luate în considerație.

DF: (*Direction Flag*) indicator de "direcție", are efect în operațiile cu şiruri. Dacă este poziționat în "1", după fiecare transfer, adresa operandului din şir se incrementează (creşte cu 1) iar dacă este poziționat în "0" adresa se decrementează.

Observație: indicatorii DF, IF şi CF pot fi poziționați în "0" sau "1" prin instrucțiunile de control specifice:

CLC (*Clear Carry*): CF = 0 **STC** (*Set Carry*): CF = 1
CLD (*Clear Direction*): DF = 0 **STD** (*Set Direction*): DF = 1
CLI (*Clear Interrupt*): IF = 0 **STI** (*Set Interrupt*): IF = 1

11.2.4. Terminale şi semnificația lor

Prin multiplexarea în timp a magistralei de date cu cea de adrese la aceleaşi terminale precum şi prin dublarea rolului unor terminale ale

microprocesorului în funcţie de cele două moduri de lucru ale sale, a fost posibilă închiderea sa într-o capsulă cu 40 de terminale.

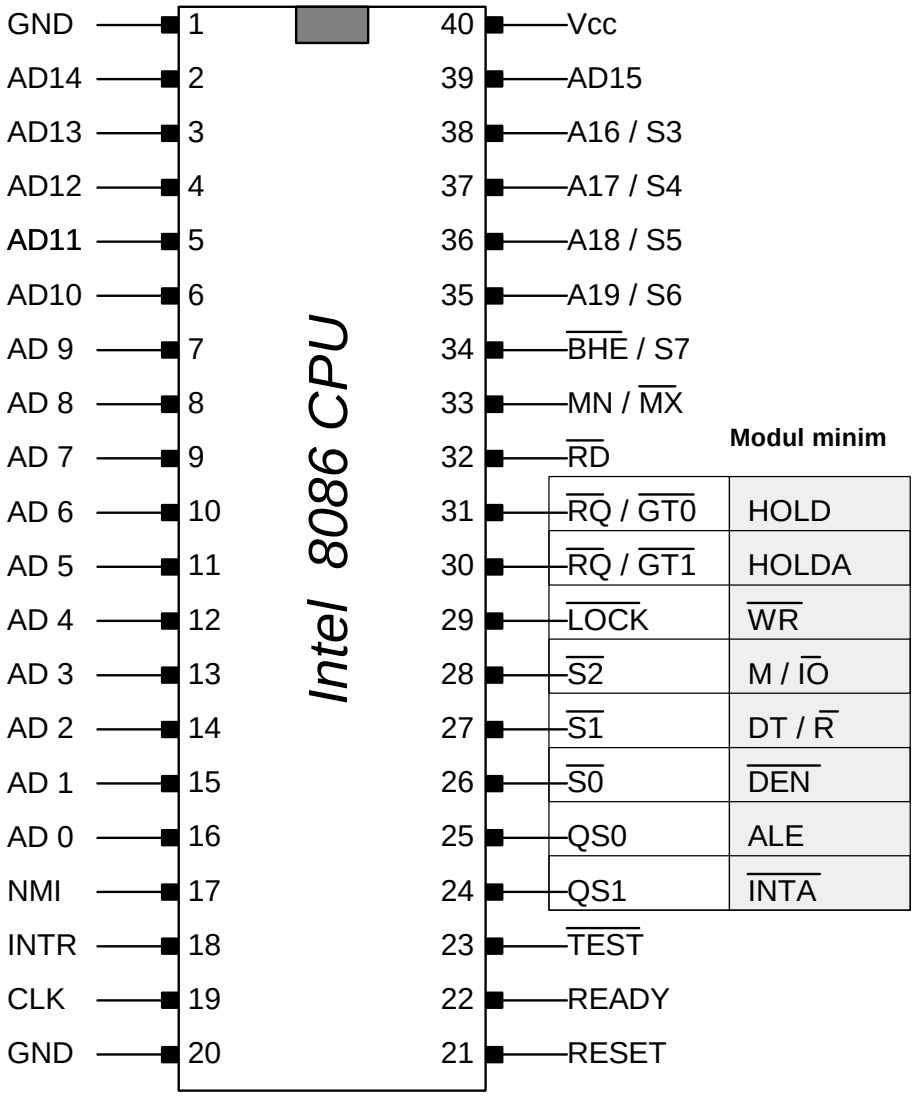


Fig.3 Intel 8086 - Configuraţia semnalelor la pini

Terminalele cu acelaşi rol în ambele moduri de lucru sunt :

AD_0 - AD_{15} : linii bidirecţionale cu trei stări (*three state*), constituie magistrala de adrese şi date multiplexate. Pe durata stării T_1 a ciclului maşină, pe cele 16 linii se încarcă o adresă de memorie sau pentru porturi I/O iar pe durata stărilor T_2, T_3, T_w şi T_4 liniile constituie magistrala de date.

A_{16} /S_3 , A_{17} /S_4 , A_{18} /S_5 , A_{19} /S_6 : ieşiri cu trei stări; în starea T_1 reprezintă cei mai semnificativi 4 biţi ai magistralei de adrese iar în stările T_2 - T_4 reprezintă informaţii de stare: S_3, S_4 indică registrul segment utilizat în calculul adresei fizice, S_5 copiază indicatorul de întreruperi (IF) iar S_6 = 0.

$\overline{\text{BHE}}$ /S_7 (*Bus High Enable*) : ieşire cu trei stări; în timpul stării T_1 se activează când are loc un transfer pe octetul superior al magistralei de date, validând acest transfer, iar în stările T_2 - T_4 este bit de stare.

$\overline{\text{RD}}$ (*Read*): ieşire cu trei stări, activă (în "0") atunci când microprocesorul execută un ciclu maşină de citire memorie sau porturi.

S4	S3	Registrul segment implicat în adresare
0	0	ES
0	1	SS
1	0	CS
1	1	DS

READY : intrare activă în "1", pentru sincronizarea procesorului cu memoria sau porturile I/O, mai lente.

INTR (*Interrupt Request*) : intrare activă în "1" reprezentând cerere de întrerupere mascabilă.

$\overline{\text{TEST}}$: intrare activă în "0", utilizată în cazul instrucțiunii ESC (Escape) care permite altui procesor să extragă instrucțiuni sau operanzi din memorie, din segmentele curente ale lui 8086/8088. Instrucțiunea ESC poate iniția o subrutină executată de un procesor concurent, ca de exemplu 8087 - coprocesor aritmetic; în timpul execuției subrutinei, procesorul de bază execută programul curent până când devine necesar rezultatul subrutinei. Dacă acesta întârzie, procesorul de bază intră în stare de aşteptare până când TEST devine inactiv.

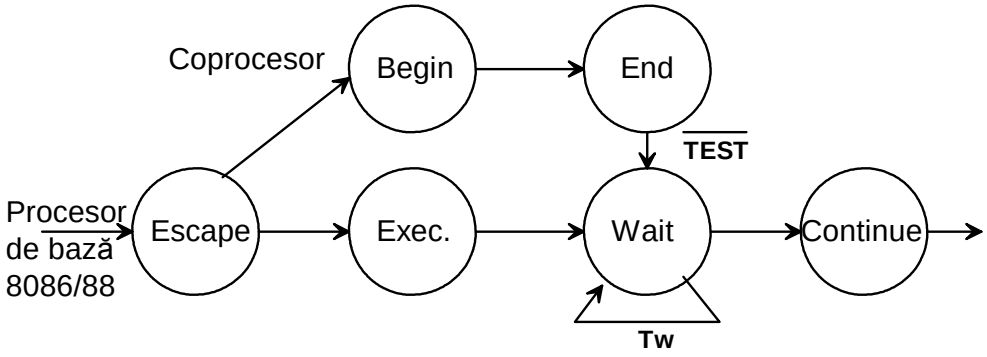


Fig. 4 Rolul semnalului TEST în sisteme multiprocesor

NMI (*Not Mascable Interrupt*) : intrare activă în "1" reprezentând cerere de întrerupere nemascabilă (care nu poate fi invalidată prin soft). Este utilizată pentru intervenția oparatorului sau pentru evenimente a căror luare în considerație nu suportă amânare (avertizare asupra unei iminente fluctuații a tensiunii de alimentare, intervenție *watch dog*).

RESET: intrare activă în "1", pentru inițializarea microprocesorului. Se aplică automat la apariția tensiunii de alimentare sau în timpul

funcționării, de către operator. Procesorul întrerupe toate activitățile până când semnalul devine inactiv. Se efectuează următoarele inițializări interne:

Indicatorii de condiții: toți în "0" IP = 0000 H
CS = FFFF H DS = 0000 H
SS = 0000 H ES = 0000 H
Fişierul de instrucțiuni: vid;
Întreruperile mascabile: invalidate.

Prima instrucțiune ce se va executa după inițializare va fi cea de la adresa FFFF0 H.

De regulă, la adresa FFFF0 se află o instrucțiune de salt către prima instrucțiune a sistemului de operare.

CLK : intrare pentru impulsurile de tact furnizate de un circuit specializat (generator de tact, tipic Intel 8284) cu frecvența uzuală de 8 MHz, nivel TTL şi factor de umplere 1/3.

MODUL MINIM

Se instalează dacă intrarea de mod este la nivel ridicat, **MN/MX**=1.

M/IO ieşire cu trei stări (*Memory / Input-Output*); determină o operație de citire sau scriere asupra memoriei sau porturilor. Dacă este "1" logic, se execută un ciclu maşină de acces la memorie iar dacă este "0" logic se execută un ciclu maşină de transfer cu porturile de intrare / ieşire.

WR: ieşire cu trei stări (*Write*) activă în "0" logic; determină o operație de scriere în memorie sau în porturi.

INTA: ieşire cu trei stări (*Interrupt Acknowledge*), activă în "1" logic, când microprocesorul execută un ciclu maşină de acceptare întrerupere.

ALE: ieşire activă în "1" (*Address Load Enable*) care se activează când pe magistrala multiplexată este încărcată o adresă.

DT/R : ieşire cu trei stări (*Data Transmission Reception*), care indică sensul transferului pe magistrala de date. Dacă este "0" logic, magistrala este orientată către microprocesor (*Reception*) iar în caz contrar, către sistem (*Transmission*).

DEN: ieşire cu trei stări (*Data Enable*) care validează transferul datelor către microprocesor.

HOLD: intrare pentru cereri de cedare magistrale. Un dispozitiv inteligent extern solicită controlul total asupra magistralelor în vederea accesului direct la memorie (tehnică *DMA - Direct Access Memory*).

HLDA: ieşire, răspuns la cererea HOLD, confirmând acceptarea acestei cereri, după trecerea magistralelor de adrese, date şi control în starea SIR (starea de înaltă impedanță sau starea "a treia").

Ultimele opt terminale prezentate au alte funcţii în modul maxim.

MODUL MAXIM

Se instalează dacă intrarea de mo este la nivel coborât, **MN/MX**=0.

RQ/GT0, RQ/GT1 (***Request/Grant Lines***) : linii bidirecţionale utilizate pentru cedarea magistralelor în urma unui dialog cerere - acceptare - renunţare.

În modul maxim, semnalele HOLD şi HOLDA evoluează în două semnale mai complexe, RQ/GT0, RQ/GT1, ce pot servi utilizării în comun a magistralelor de către 8086/8088 şi alte două procesoare.

Cererea şi cedarea magistralelor necesită trei faze distincte: cererea, alocarea, eliberarea. Un procesor cere pe o linie RQ/GT controlul magistralelor şi pe aceeaşi linie CPU 8086 confirmă trecerea lor în starea SIR, deci cedarea lor. În această situaţie, unitatea BIU este deconectată de la magistrale iar EU execută instrucţiuni din fişierul intern până la golirea acestuia sau până când o instrucţiune necesită acces la magistrale. Când procesorul extern termină operaţiile cu magistralele, transmite pe aceeaşi linie, RQ/GT = 1, cu semnificaţia că CPU 8086 poate să preia controlul asupra magistralelor.

Linia RQ/GT0 are prioritate faţă de RQ/GT1, în cazul în care apar cereri simultane pe cele două linii.

S2 , S1 , S0 : ieşiri cu trei stări care exprimă codificat tipul de ciclu maşină ce va fi va executat, conform tabelului; semnalele se aplică circuitului specializat Intel 8288 care generează semnale de comandă pentru memorie şi porturi I/O, corespunzătoare ciclului maşină curent.

S2	S1	S0	Tipul ciclului maşină
0	0	0	Acceptare întrerupere
0	0	1	Citire port I/O (intrare date)
0	1	0	Scriere port I/O (ieşire date)
0	1	1	Halt (oprire)
1	0	0	Aducere instrucţiune din memorie
1	0	1	Citire memorie
1	1	0	Scriere memorie
1	1	1	Combinaţie neutilizată

LOCK : ieşire cu trei stări, activă în "0", cu semnificaţia că magistralele nu pot fi cedate pe durata LOCK = "0", deoarece se află în execuţie o instrucţiune care nu poate fi întreruptă. Semnalul este activat de prefixul LOCK al unei instrucţiuni oarecare şi rămâne activ pe toată

durata execuţiei instrucţiunii. Prefixul constă într-un octet plasat înaintea octeţilor ce definesc instrucţiunea.

QS0 , QS1 : ieşiri (*Queue Status Lines*), care indică dispozitivelor externe tipul de informaţie preluată de unitatea de execuţie din fişierul de aşteptare în starea anterioară. Informaţia este utilă pentru un coprocesor, în cadrul instrucţiunii ESC, în vederea utilizării magistralelor la extragerea unui operand din memorie.

QS1	QS0	Semnificaţie
0	0	Fără citire din fişier
0	1	Octetul preluat a fost primul octet al instrucţiunii
1	0	Fişierul este gol
1	1	Octetul preluat nu a fost primul octet al instrucţiunii

11.3 Implementarea stivei

Stivele sunt formate în memoria RAM şi sunt adresabile cu registrul segment SS şi registrul pointer SP. Un program poate utiliza mai multe stive, fiecare cu lungimea maximă de 64 KB. Registrul SS conţine adresa de bază a stivei iar SP conţine adresa relativă (deplasamentul sau offset-ul) faţă de bază, a vârfului stivei. Stivele au locaţii de 16 biţi şi funcţionează ca liste LIFO (*Last In First Out*), adică ultimul operand introdus în stivă este primul care poate fi extras.

Un cuvânt este salvat în stivă la adresa relativă (SP-2); citirea din stivă se face prin copiere de la adresa (SP), după care se face actualizarea registrului pointer SP: (SP+2). Locaţiile din stivă sunt de 16 biţi şi ca urmare operanzii care se introduc şi se extrag în/din stivă sunt cuvinte de 16 biţi iar adresele a două locaţii consecutive diferă prin 2 (în fiecare locaţie sunt doi octeţi).

Se pot introduce în stivă operanzi de 16 biţi din registre cu excepţia lui CS şi operanzi de 16 biţi din memorie. Extragerea din stivă se face prin copiere într-un registru de 16 biţi, cu excepţia lui CS, sau în memorie.

Salvarea în stivă se face cu instrucţiunea PUSH iar extragerea din stivă cu POP. Decrementarea registrului SP la introducere şi incrementarea sa la scoatere se fac automat. Programatorul trebuie să fixeze doar baza stivei în SS şi limita superioară a stivei, în SP. Toate operaţiile cu stiva se realizează apoi foarte simplu, prin instrucţiunile PUSH şi POP. Adresele la care se fac transferurile se calculează automat şi nu prezintă nici un interes. Tot ceea ce contează este ordinea în care se

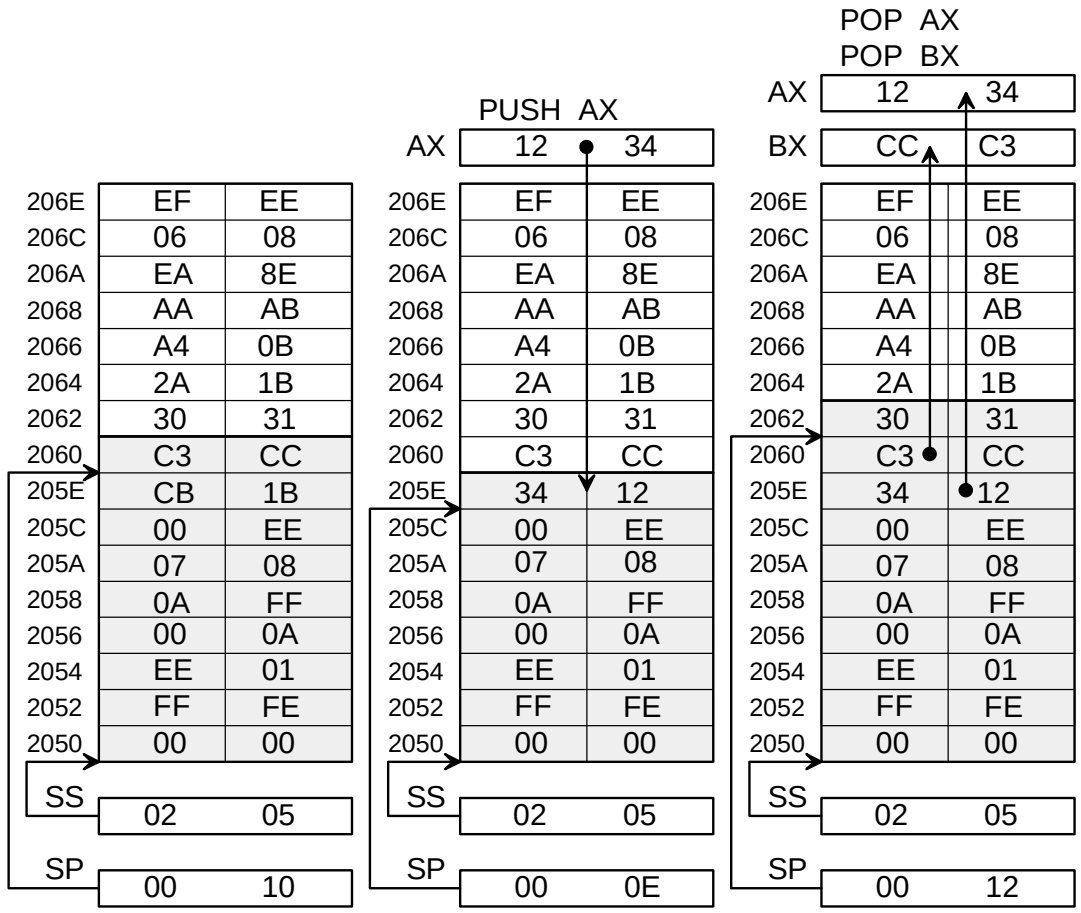


Fig.5 Funcționarea stivei: situația curentă (stânga), introducerea în stivă (mijloc) şi extragerea din stivă (dreapta).

face introducerea în stivă, deoarece la extragere aceasta se inversează: *''cel din urmă va fi cel dintâi''*. Calculul adresei se face astfel:

SS: 0 2 0 5 0 +
SP: 0 0 1 0 = 0 2 0 6 0 H = 2060 H

- *De ce sunt necesare stivele ?*

- Stivele sunt strict necesare în lucrul cu subrutine (proceduri şi funcții), când trebuie eliberate registrele interne în vederea apelării unei subrutine care va încărca registrele cu propriile sale date. Conținutul registrelor este însă necesar la revenirea din subrutină. De aceea eliberarea registrelor se face prin salvarea lor în stivă, într-o anumită ordine şi refacerea lor din stivă la revenirea din subrutină în programul apelant.

- *Nu putem folosi memoria RAM normală pentru salvarea registrelor ?*

- Ba da, dar conținutul registrelor interne este atât de important încât acesta trebuie pus la loc sigur, unde accesul să fie simplu, pentru a nu

complica inutil textul programului, dar controlat de instrucţiuni speciale, pentru evitarea erorilor de program.

11.4 Organizarea şi adresarea memoriei

Pentru microprocesorul Intel 8086, memoria este un şir de 1.048.576 octeţi, (1Megaoctet). Toate registrele sale interne de adresare fiind de 16 biţi, nu se pot transfera intern adrese mai lungi de 16 ranguri binare, adică nu se pot adresa spaţii de memorie mai mari de 64 kB. Pentru acoperirea unui spaţiu de memorie de 16 ori mai mare (64 kB x 16 = 1 MB), acesta este divizat logic (în mod virtual) în segmente de maxim 64 kB.

Microprocesorul poate lucra simultan cu 4 segmente de memorie, fiecare cu dimensiunea între 16 octeţi şi 64 kiloocteţi, adresele lor de început fiind fixate în cele 4 registre segment .

În utilizarea oricărei locaţii de memorie se folosesc două feluri de adrese: *fizică* şi *logică*. Adresa fizică este de 20 de biţi şi identifică în mod unic fiecare byte din spaţiul memoriei între 00000 H şi FFFFF H. Orice schimb de informaţie între CPU şi memorie utilizează adresa fizică, pe care o transmite prin magistrale de adrese.

O adresă logică este formată din doi parametri: *adresa de bază* şi valoarea de *offset*; pentru orice locaţie, adresa de bază indică adresa primului octet din segmentul de memorie iar valoarea de *offset* arată distanţa în octeţi de la adresa de bază la locaţia adresată. Adresa de bază şi *offset*-ul sunt de 16 biţi fiecare. Primul octet dintr-un segment de memorie are offsetul "0000".

Segmentele de memorie sunt blocuri care se pot suprapune; nu există restricţii de delimitare a segmentelor în spaţiul adreselor (fig. 6), singura restricţie fiind ca adresa de început a unui segment sa fie multiplu de 16, adică să aibă ultimii 4 biţi de valoare "0".

Pot exista segmente adiacente (A, B), segmente parţial suprapuse (B, C şi D, E) sau segmente disjuncte (C, D). Definirea unui segment logic se face prin încărcarea adresei de început într-un registru segment; nu se specifică în nici un fel cât de mare este segmentul dar nu pot fi accesaţi decât primii 64 ko, deoarece *offset* - ul este pe 16 biţi.

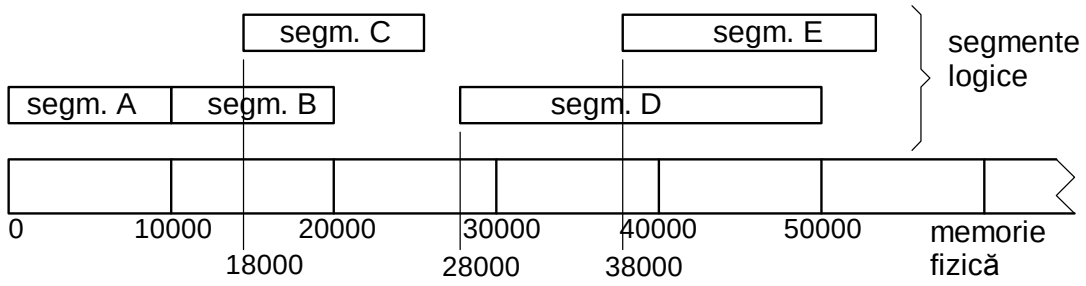


Fig.6 Relația dintre segmentele logice şi memoria fizică

Avantajele segmentarii memoriei sunt :

- Este facilitată programarea modulară; fiecare modul de program poate ocupa unul sau mai multe segmente şi poate fi dezvoltat independent față de celalalte.
- Permite scrierea programelor independent de adresele de memorie în care vor fi stocate, sau poziția lor în memorie poate fi modificată (realocarea dinamică). Pentru aceasta, este necesar ca programele să nu afecteze conținutul registrelor segment şi să nu facă referire la locații din afara segmentului curent; utilizând instrucțiuni de modificare a conținutului registrelor segment, un program poate fi mutat oriunde în memorie.
- Permite deservirea "simultană" a mai multor periferice; de exemplu, un program editor de texte care lucrează cu mai multe imprimante poate transmite câte un bloc de 64 kB fiecărei imprimante care solicită intrarea în "serviciu" prin încărcarea registrului ES cu baza segmentului de date unde se află textul corespunzător imprimantei ce solicită date.
- Se pot utiliza mai multe stive; schimbarea stivei se face prin plasarea adresei de început în registrul segment SS şi a adresei vârfului stivei (limita de sus) în registrul pointer SP.

Dezavantajele segmentării memoriei:

- Limitarea lungimii programelor la dimensiunea maximă a segmentelor.
- Adresa fizică se obține din doua adrese logice (bază şi offset), ceea ce face necesară o operație suplimentară de adunare pe 16 biți şi implicit o unitate aritmetică destinată calculului adesei.

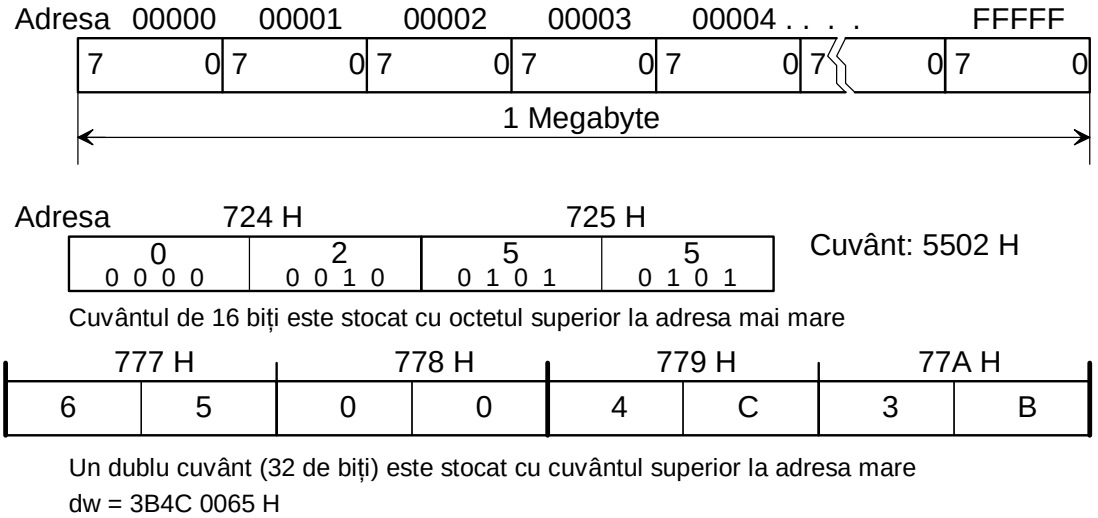


Fig.7 Organizarea informației în memorie

Conform convențiilor firmei Intel cuvântul de 16 biți este memorat în locații consecutive de 8 biți, cu octetul inferior (b_0 - b_7) la adresa de valoare mai mică (*adr*) şi cu octetul superior (b_7 - b_{15}) la *adr+1,* iar dublul cuvânt este memorat cu cuvântul inferior (b_0 - b_{15}) la *adr, adr+1* şi cu cuvântul superior (b_{16} - b_{31}) la *adr+2, adr+3* (fig.7). Adresarea se poate face la nivel de octet dar şi la nivel de cuvânt, adică doi octeți consecutivi, fără restricții cu privire la adresa de început a operanzilor cuvânt.

Microprocesorul generează cei 20 de biți ai adresei fizice prin adunarea adresei de bază cu adresa efectivă (deplasament sau offset), fig.8. Conținutul registrului segment este de 16 biți; pentru a obține adresa de bază a segmentului de memorie, de 20 de biți, se completează conținutul registrului segment cu 4 biți de valoare 0.

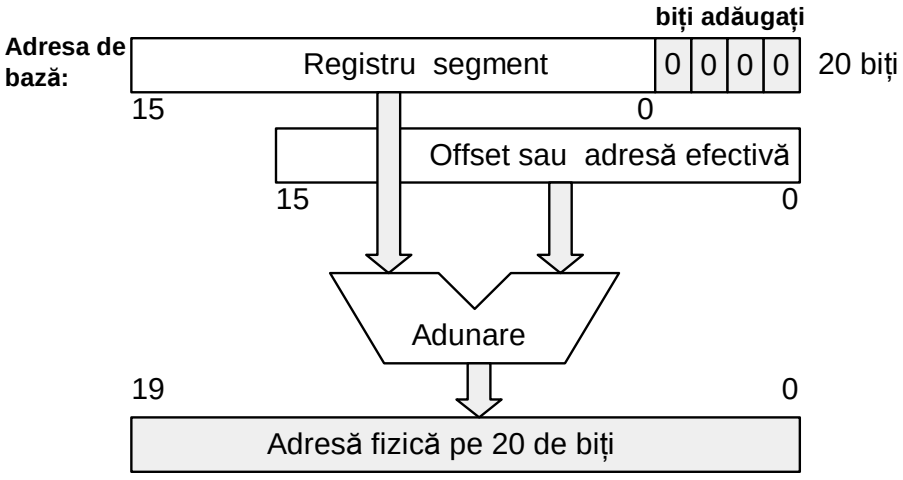


Fig.8 Calculul adresei fizice din adresa de bază şi offset

Două adrese fizice consecutive care se termină cu 0, cuprind între ele 16 locații. Dacă DS=1234 H, ES=1235 H, rezultă că segmentul de date adresat cu DS conține 16 octeți: 12340, . . . ,1234F, iar segmentul de date adresat cu ES începe la adresa de bază 12350 H. De aceea lungimea minimă a unui segment este de 16 locații.

Pentru obținerea adresei fizice se combină un registru segment cu un registru index (SI, DI), cu un registru pointer (SP, BP, IP, BX) sau cu un cuvânt de 16 biți specificat în instrucțiune (adresă efectivă, EA). Nu este posibilă orice combinație între un registru segment şi un altul care furnizează offset - ul.

Registrele segment (care conțin câte o adresă de bază fiecare) permit adresarea imediată a 4 segmente de memorie.

Programele obțin acces la un segment de cod şi 3 segmente de date din orice zonă de memorie, prin încărcarea registrelor segment cu adresele de început ale segmentulor dorite.

În tabelul de mai jos sunt date toate combinațiile posibile.

Tipul referinței la memorie	Registrul segment utilizat	Registrul segment alternativ	Sursa adresei de offset
Extragere instrucțiuni	CS	-	IP
Operații cu stiva	SS	-	SP
Şir de octeți - destinație	ES	-	DI
Şir de octeți - sursă	DS	-	SI
Variabile de program	DS	CS, ES, SS	EA
Adresare cu BP ca bază de offset	SS	CS, DS, ES	EA
Adresare cu BX ca bază de offset	DS	CS, ES, SS	EA

Orice aplicație va defini şi utiliza segmentele proprii. Registrele de adresare curentă (segment) asigură în general ca spațiu de lucru: 64 kbyte pentru coduri, 64 kbyte pentru stivă şi 128 kbytes pentru stocarea datelor. Multe aplicații pot fi srise prin simpla inițializare a registrelor segment (şi apoi le poți uita !). Structura segmentată a memoriei descurajează programele foarte lungi, monolitice.

Instrucțiunile sunt totdeauna extrase cu CS şi IP.

Operanzii din stivă sunt obținuți cu SS şi SP, care conține offset - ul față de adresa de bază din SS.

Majoritatea variabilelor (din memorie) sunt obținute cu DS - curent, dar un program poate informa BIU cu privire la segmentul în care se află

anumite variabile. Diferența față de adresa de bază este calculată de EU; calculul se bazează pe modul de adresare specificat în instrucțiune; rezultatul calculului se numeşte **adresa efectivă a operandului** (EA).

Şirurile de octeți sunt adresate diferențiat, în funcție de variabile.

Operandul sursă din şir este "citit" cu DS dar poate fi specificat şi alt registru segment. Offset-ul este furnizat de SI (registrul index sursă).

Operandul destinație din şir se obține cu ES iar offset-ul se obține din DI (registrul index destinație). Operațiile cu şiruri se execută sub controlul instrucțiunilor specifice, care modifică automt registrele SI şi DI, după cum lucrează cu octeți sau cu cuvinte de 16 biți.

Când este folosit BP ca registru pointer (sursă de offset) într-o instrucțiune, variabila este citită cu SS. Registru BP permite un mod convenabil de adresare pentru datele din stivă; BP poate fi de altfel utilizat la accesarea datelor şi cu CS, DS, ES.

În adresarea memoriei sunt facilitate modurile frecvente de adresare.

Memoria fizică de 1 MB poate fi divizată în două blocuri de câte 512 kB fiecare: blocul locațiilor cu adrese pare şi blocul locațiilor cu adrese impare.

Blocul par este conectat la liniile D0 - D7 ale magistralei de date (octet inferior) iar cel impar la liniile D8 - D15 ale magistralei (octet superior).

Liniile de adresă A1 - A19 se aplică ambelor blocuri de memorie şi ca urmare la încărcarea unei adrese se selectează simultan câte o locație din fiecare bloc.

Transferul de date între locațiile selectate şi magistrala de date se face însă cu încă două linii, A0 şi BHE (fig.9).

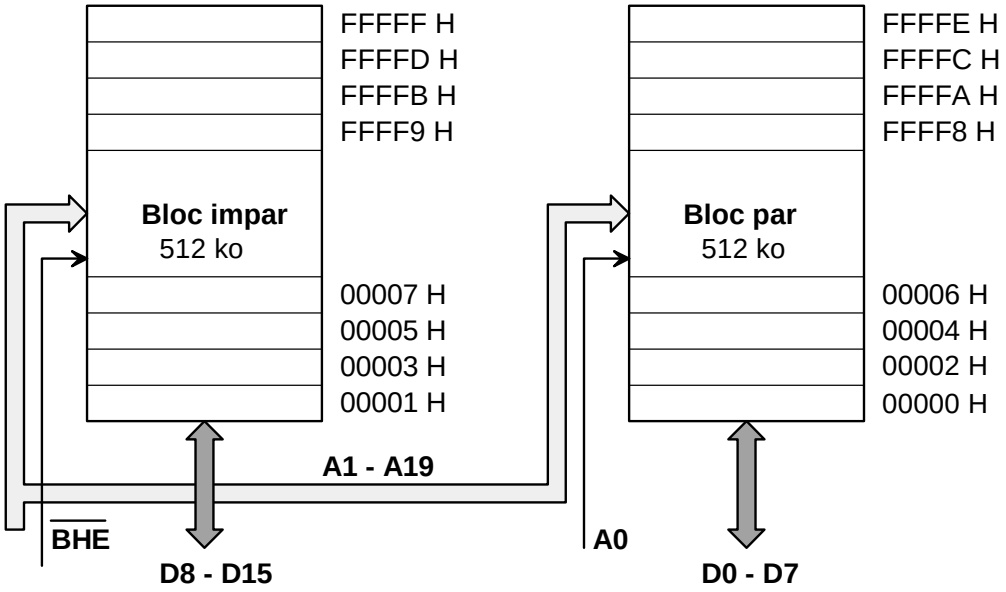


Fig.9 Conectarea blocurilor de memorie la magistrale

BHE	A0	Se transferă
0	0	Ambii octeți
0	1	Octet superior (cu adresă impară)
1	0	Octet inferior (cu adresă pară)
1	1	Fără transfer

Pentru accesul la un operand cu adresă pară, $\mathbf{A_0} = 0$ iar $\overline{\mathbf{BHE}} = 1$ (inactiv) va invalida blocul impar. Ca urmare, are loc un transfer între locația adresată şi octetul inferior al magistralei de date (fig.10).

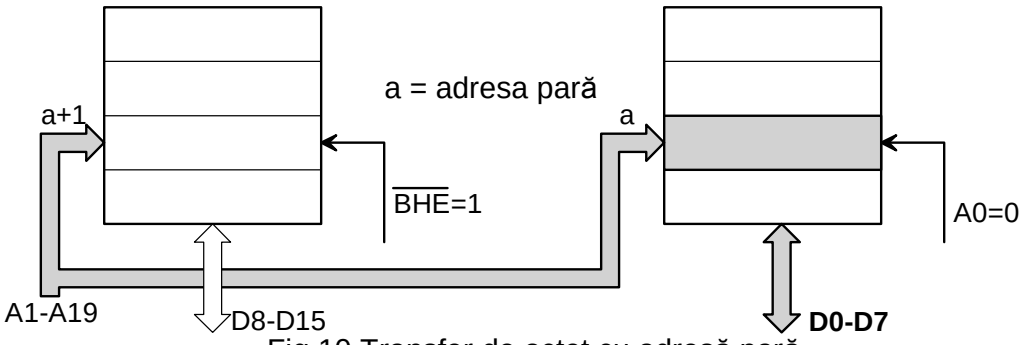


Fig.10 Transfer de octet cu adresă pară

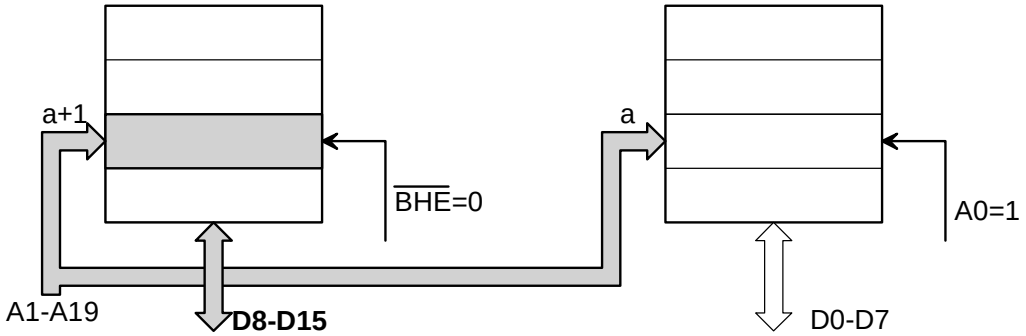


Fig.11 Transfer de octet cu adresă impară

Pentru accesul la un operand cu adresă impară, linia $\mathbf{A_0}$=1va invalida blocul par iar $\overline{\mathbf{BHE}} = 0$ va selecta blocul impar. Ca urmare, are loc un transfer între locația adresată şi octetul superior al magistralei de date (fig.11).

În cazul unui transfer pe 16 biți (cuvânt) începând de la o adresă pară, ambele blocuri vor fi selectate simultan de liniile $\mathbf{A_0}$=0 şi $\overline{\mathbf{BHE}} = 0$. Are loc transferul între cele două locații şi întreaga magistrală de date D0-D15, într-un singur ciclu maşină şi se spune că operandul este *aliniat*, adică începe de la adresă pară.

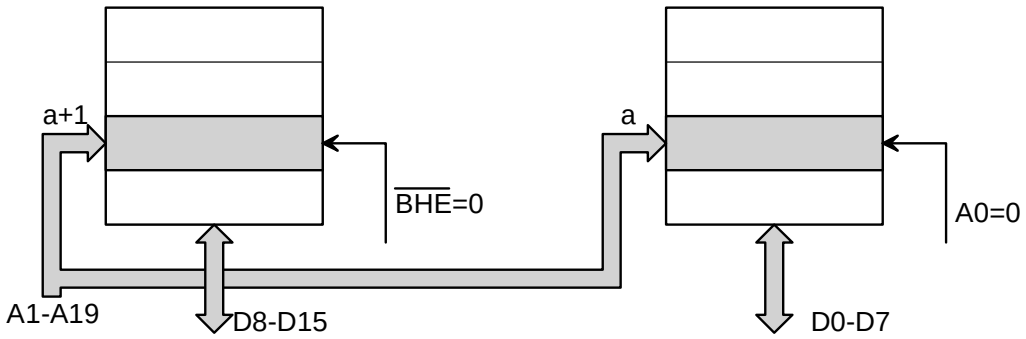


Fig.12 Transfer de cuvânt cu adresă pară

În cazul accesului la un cuvânt (16 biți) care începe la o adresă impară, se spune că operandul este *nealiniat* şi sunt necesare două cicluri maşină pentru transfer. În primul ciclu maşină se transferă octetul de la adresa impară (a+1) pe liniile D8 - D15, fiind considerat octet inferior al operandului de 16 biți. În al doilea ciclu maşină se transferă octetul de la adresa pară (a+2) pe liniile D0 - D7, fiind considerat octet superior al operandului de 16 biți. Toate operațiile, inclusiv direcționarea corectă a octeților, sunt coordonate de microprocesor, procesul de transfer fiind invizibil pentru utilizator. În acest caz, principalul dezavantaj este că procesul de transfer are o durată dublă față de cazul unui operand *aliniat*.

Realocarea dinamică

Structura segmentată a memoriei la 8086, 8088, 80186, 80188 face posibilă scrierea programelor în mod independent de zona de memorie din care vor fi executate. Sistemul de operare poate aloca segmentele de memorie care sunt libere în momentul intrării în execuție - *realocarea dinamică*, ceea ce permite modul de lucru *''multitasking''* (rularea simultană a mai multor programe).

Programele temporar inactive vor fi scrise pe discul magnetic iar spațiul ocupat de ele, alocat altor programe. Dacă programul rezident pe disc este necesar mai târziu, el poate fi reîncărcat în orice zonă disponibilă de memorie. Similiar, dacă este necesar un spațiu mare de memorie pentru blocuri de date, iar memoria disponibilă este formată din fragmente neadiacente, un program segmentat poate fi "compactat" cu toate segmentele dispuse consecutiv, unificând astfel zonele libere pentru blocuri masive de date (fig. 9).

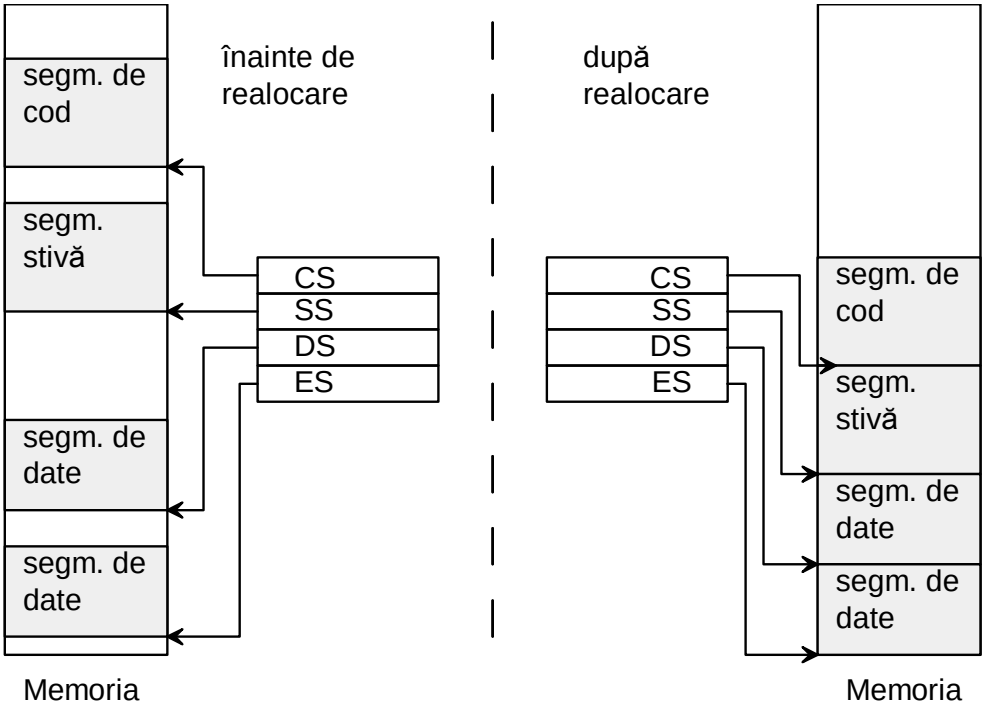


Fig. 9 Realocarea dinamică a memoriei

11.4.1. Locații de memorie dedicate şi locații rezervate

Două zone de memorie (fig. 10) de la adrese joase şi două zone de la adresele înalte sunt dedicate unor funcții specifice ale procesorului sau rezervate de Intel pentru utilizarea produselor sale soft sau hard. Acestea sunt:

- 00000 - 00007F (128 octeți), utilizată pentru memorarea tabelei de vectorilor de întrerupere;
- FFFF0 - FFFFF (16 octeți), zonă adresată la inițializarea sistemului prin activarea semnalului RESET.

Pentru asigurarea compatibilității sistemului cu produsele Intel, aplicațiile nu trebuie să utilizeze zonele de mai sus în alte scopuri.

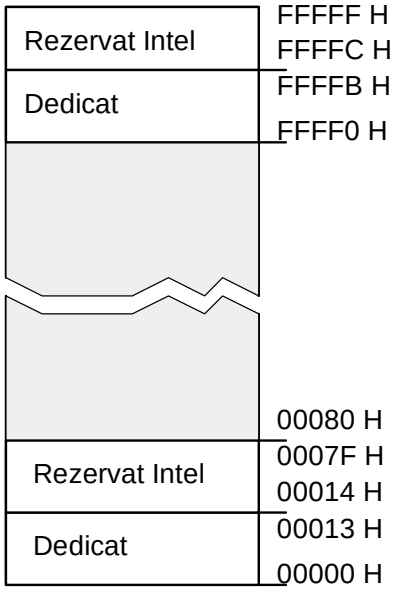


Fig. 10 Organizarea memoriei. Zone dedicate, zone rezervate.

11.5 Organizarea porturilor de intrare / ieşire

Microprocesoarele Intel 8086 / 88 permit utlizarea unui larg spațiu de adrese pentru dispozitive I/O (*Input/Output*), separat față de spațiul de memorie. Pentru transferuri rapide şi complexe se poate folosi accesul direct la memorie sau coprocesorul Intel 8089 specializat în operații I/O.

Porturile I/O pot fi adresate ca locații de memorie, ceea ce permite utilizarea instrucțiunilor de lucru cu memoria, eficiente în privința adresării şi vitezei de lucru.

Spațiul I/O este suprapus ca adrese peste primul segment de 64 ko, în zona 00000 H - 0FFFF H şi este accesibil prin instrucțiunile dedicate, IN şi OUT. Toate porturile sunt considerate în acelaşi segment; ele sunt adresate ca şi locațiile de memorie dar fără registru segment.

Un port poate fi de 8 biți (echivalent cu o locație de memorie de 8 biți), caz în care şi adresa lui este de 8 biți, sau poate fi de 16 biți (echivalent cu o locație de memorie de 16 biți), caz în care adresa lui este de 16 biți. Dacă portul este de 8 biți el se poate conecta fie la octetul

inferior al magistralei de date (D0 - D7), fie la octetul superior (D8 - D15). Adresele asociate porturilor trebuie să corespundă conectării la magistrala de date: dacă portul este conectat la D0 - D7, adresa trebuie să fie pară iar dacă este conectat la D8 - D15, adresa trebuie să fie impară.

Pentru un port de 16 biți, conectarea se face la D0 - D15 iar adresa lui trebuie să fie pară, pentru ca transferul datelor să se facă într-un singur ciclu maşină.

Pentru accesul la un port, unitatea BIU plasează adresa portului (0 - FFFF) pe liniile A0 - A15 ale magistralei de adrese. Adresa portului este specificată în instrucțiune (dacă este pe 8 biți) sau în registrul DX (dacă este pe 16 biți). Instrucțiunile IN (*Input* - intrare) şi OUT (*Output* - ieşire) transferă datele între acumulator şi port. Dacă portul este de 8 biți, transferul se face între port şi AL iar dacă portul este de 16 biți transferul se face între port şi registrul AX.

```
IN  AL, port      ; citire port de 8 biți - adresa = port, data = AL
IN  AX, DX        ; citire port de 16 biți - adresa = DX, data = AX
OUT port, AL      ; scriere port de 8 biți - adresa = port, data = AL
OUT DX, AX        ; scriere port de 16 biți - adresa = DX, data = AX
```

În adresarea indirectă, se utilizează registrul DX pentru adresa de 16 biți (registrul DX este încărcat anterior cu adresa). La execuția instrucțiunii, conținutul lui DX este încărcat pe magistrala de adrese (A0 - A15). Se poate adresa orice port în spațiul I/O din domeniul 0000 H - FFFF H (65 536 porturi).

În adresarea directă sau imediată, adresa portului apare în instrucțiune şi este în domeniul 00 H - FF H (256 porturi).

Dacă porturile sunt adresate ca locații de memorie, orice mod de adresare a operanzilor din memorie poate fi utilizat pentru acces la porturi (exemplu: un grup de terminale de date poate fi accesat ca un tablou).

11.6 Întreruperi

O *întrerupere* opreşte temporar execuția unui program şi transferă controlul unei rutine (subprogram) specifice de tratare ce corespunde cauzei ce a generat întreruperea. Mecanismul prin care se face acest transfer este în esență de tip apel de procedură, ceea ce implică revenirea în programul întrerupt după execuția rutinei de tratare.

Întreruperile hardware externe sunt activate de cereri de întrerupere generate de dispozitive periferice inteligente, sub forma unor semnale electrice, aplicate pe intrările INTR şi NMI ale procesorului; cele interne apar ca urmare a unor condiții speciale de funcționare a procesorului (de exemplu, modul de lucru pas cu pas).

Întreruperile mascabile pot fi dezactivate prin comenzi de program şi sunt cele produse de semnale aplicate pe intrarea INTR a procesorului; cele nemascabile nu pot fi dezactivate prin comenzi de program şi sunt cele produse de semnale aplicate pe intrarea NMI.

Într-un sistem cu procesor 8086 pot exista maxim 256 de întreruperi distincte. Fiecare din aceste nivele poate avea asociată o procedură de tip far, numită rutină de tratare. Adresele acestor rutine sunt înregistrate într-o tabelă de întreruperi aflată în zona de memorie 00000 - 003FFH, ocupând deci 1024 octeți. Fiecare nivel ocupă 4 octeți, primii 2 reprezentând offset-ul iar următorii 2 adresa de segment a procedurii.

Practic, un vector de întrerupere conține adresa completă a subrutinei de tratare întrerupere corespunzătoare.

Tabela vectorilor de întrerupere conține trei zone:

- o zonă dedicată, care este utilizată în mod automat de microprocesor pentru cererile de întrerupere predefinite (00000H-00013 H);
- o zonă rezervată, pentru păstrarea compatibilității cu produsele Intel;
- o zonă disponibilă, la dispoziția utilizatorului (00080H - 003FFH).

Adresa rutinei de tratare (în Hexa)	Tipul întreruperii
003FC	INT 255 - Disponibil
003F8	INT 254 - Disponibil
. .	. .
00080	INT 32 - Disponibil
0007C	INT 31 - Rezervat
. .	. .
00014	INT 5 - Rezervat
00010	INT 4 - Depăşire
0000C	INT 3 - Breakpoint
00008	INT 2 - Nemascabil
00004	INT 1 - Mod pas cu pas
00000	INT 0 - Eroare de divizare

La apariția unei întreruperi au loc următoarele acțiuni:

- se salvează în stivă registrele F, CS, IP;
- se pun în zero indicatorii IF şi TF;
- se furnizează procesorului un octet (0 - 255) numit *vector de întrerupere* care identifică nivelul asociat întreruperii curente;

- prin intermediul tabelei de întreruperi se execută salt intersegment la adresa rutinei de tratare;

Vectorul de întrerupere poate fi furnizat procesorului în unul din următoarele moduri:

- în cazul întreruperilor hard interne nivelul este implicit;
- în cazul întreruperilor hard externe, nivelul este transmis prin magistrala de date în cadrul ciclului maşină de tratare, de către dispozitivul care a generat întreruperea.
- în cazul întreruperilor soft, nivelul este conținut în instrucțiune.

11.6.1. Întreruperile externe

Sunt inițiate la intrările INTR şi NMI ale procesorului. O cerere pe linia NMI generează o întrerupere nemascabilă şi este predefinită de tipul INT 2, adică procesorul execută accesul la vectorul 2 fără alte informații suplimentare.

O cerere pe linia INTR este generată, de regulă, de un circuit extern specializat, care evaluează prioritatea în cazul apariției cererilor simultane din mai multe surse şi permite trecerea cererii cu prioritate maximă la momentul respectiv.

Intrarea INTR este testată de microprocesor la terminarea execuției fiecărei instrucțiuni. Singurele excepții de la regulă apar în cazul instrucțiunilor pentru şiruri de date, la care linia INTR este testată după prelucrarea fiecărui element din şir şi în cazul instrucțiunii WAIT, la care linia INTR este testată după fiecare verificare a liniei TEST.

Există câteva situații în care cererea de întrerupere este servită numai după execuția instrucțiunii următoare. Este cazul instrucțiunilor precedate de prefixe; nu este luată în considerație cererea de întrerupere între execuția prefixului şi a instrucțiunii. De asemenea este cazul instrucțiunilor MOV şi POP care încarcă registrele de segment; nu se recunoaşte nici o cerere de întrerupere decât după execuția instrucțiunii următoare din motive de protecție. Pentru schimbarea segmentului de memorie este necesară modificarea a două registre. Dacă cererea de întrerupere intervine între modificarea primului registru şi cea a celui de-al doilea, există riscul ca ultimul transfer să nu se mai execute corect ceea ce duce la erori grave în continuarea programului.

Pentru servirea cererilor INTR (mascabile) este necesar ca sistemul de întreruperi să fie activat prin setarea indicatorului IF (IF = 1); setarea se face cu instrucțiunea STI (*Set Interrupt*) iar resetarea (IF = 0) se face cu CLI (*Clear Interrupt*).

Dacă IF = 1, microprocesorul va lua în considerație cererea de întrerupere de pe linia INTR. După citirea codului întreruperii pe D0-D7, registrul F (al indicatorilor de condiții) este depus în stivă; se salvează în stivă, de asemenea, registrele CS şi IP şi se anulează IF şi TP pentru a invalida eventuale cereri INTR şi modul de lucru pas cu pas.

Se execută un ciclu maşină special, de tratare întrerupere, în care CS şi IP sunt încărcați din tabela vectorilor de întrerupere, în funcție de codul pe 8 biți al întreruperii.

Subrutina de tratare a întreruperii trebuie să se încheie cu instrucțiunea IRET care determină revenirea în programul principal şi refacerea din stivă a registrelor CS, IP şi F cu vechile valori.

Cererea de întrerupere de tip NMI are codul 2 şi este luată în considerație dacă are o durată de cel puțin două stări. Operațiile efectuate de microprocesor sunt aceleaşi cu cele de la cererea de tip INTR.

11.6.2. Întreruperile interne

Sursa de întrerupere nu este un eveniment extern. Întreruperea apare ca urmare a execuției unei instrucțiuni *INT nn* sau ca urmare a unui eveniment intern (întreruperi predefinite).

Instrucțiunea *INT nn* este pe doi octeți şi realizează legătura cu vectorul *nn* din tabela de întreruperi. Are acelaşi efect cu o întrerupere externă cu codul *nn*, dar nu este mascabilă şi nu se execută o secvență de acceptare, deci nu se vor genera semnale INTA.

Întreruperile predefinite sunt generate automat de procesor la detectarea unor evenimente interne; acestea sunt:

- împărțire la zero - INT 0;
- funcționare pas cu pas - INT 1;
- breakpoint (punct de oprire) - INT 3;
- depăşire - INT 4;

Întreruperea INT 0 este generată ca urmare a execuției unei împărțiri cu câtul mai mare decât valoarea maximă admisă. În cazul instrucțiunii DIV valoarea maximă a câtului este FF sau FFFF, după cum împărțitorul este octet, respectiv cuvânt. În cazul instrucțiunii IDIV, cele două valori sunt 7F sau 7FFF.

Întreruperea INT 1 este luată în considerație dacă indicatorul TF=1. Efectul ei este modul de lucru pas cu pas, în care după fiecare instrucțiune se poate afişa conținutul registrelor, al locațiilor de memorie, informații necesare verificării şi depanării programelor. Procesorul nu dispune de instrucțiuni pentru modificarea directă a indicatorului TF. Operația se poate realiza indirect, cu ajutorul stivei. Instrucțiunea PUSHF depune

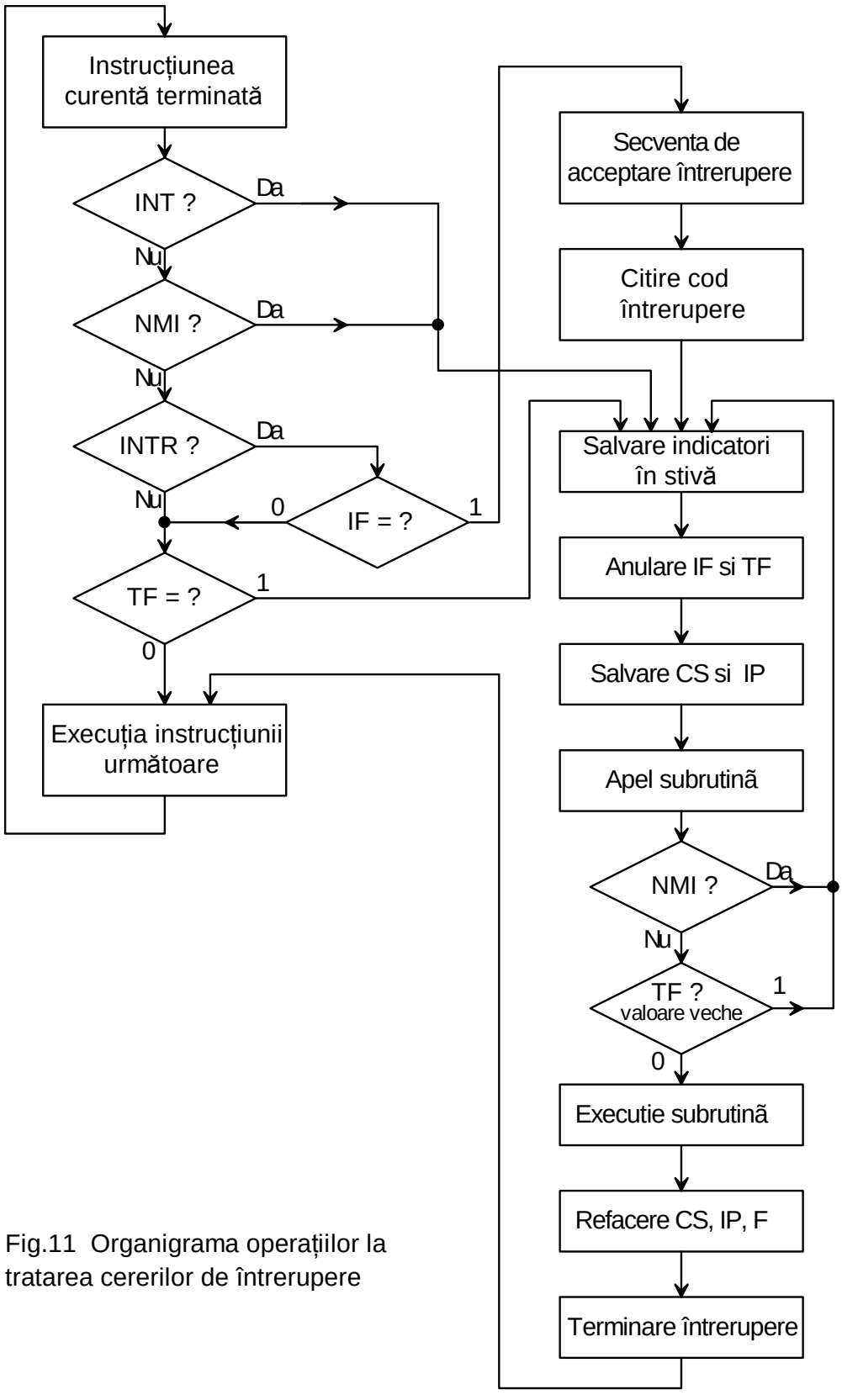


Fig.11 Organigrama operațiilor la tratarea cererilor de întrerupere

registrul F în stivă. Indicatorii pot fi modificați cu instrucțiunile logice şi

reîncărcați cu instrucțiunea POPF.

Întreruperea INT 3, de tip *breakpoint* - punct de oprire, are codul pe un singur octet şi are ca efect oprirea execuției în punctul din program în care apare INT 3. Serveşte testării şi depanării programelor prin examinarea stării procesorului (registre, indicatori etc.) în punctul de oprire.

Întreruperile INT 4 este generată la apariția depăşirii capacității registrelor la execuția unei operații aritmetice, depăşire ce determină setarea indicatorului OF (*Overflow Flag = 1*); întreruperea nu se generează automat ci numai dacă microprocesorul execută instrucțiunea INTO, care, în general, trebuie să urmeze după instrucțiunile aritmetice cu operanzi cu semn. Subrutina de tratare conține, de regulă, un mesaj către utilizator, care îl informează cu privire la apariția depăşirii.

În concluzie, instrucțiunile care generează întreruperi interne sunt:

- INT *nn* - determină generarea unei intreruperi interne de tip *nn*; microprocesorul extrage vectorul de întrerupere *nn* din tabela de întreruperi şi execută subrutina de tratare fără generarea unui ciclu extern de tip INTA;
- INTO - determină generarea unei întreruperi predefinite de tip 4, când indicatorul de depăşire OF=1, ca urmare a execuției unei instrucțiuni aritmetice cu operanzi cu semn;
- IRET - (*Interrupt Return*) determină revenirea în programul principal (din subrutină) prin restaurarea din stivă a registrelor IP, CS şi a indicatorilor de condiții F.

Timpul de procesare definit din momentul recunoaşterii unei cereri de întrerupere şi până când microprocesorul realizează accesul la începutul subrutinei de tratare, depinde de tipul întreruperii şi este dat în tabelul de mai jos.

Tip de întrerupere	Timp (în perioade de tact)
Întrerupere externă INTR	61
Întrerupere externă nemascabilă NMI	50
INT nn	51
INT 3	52
INTO	53
INT 1	50

În tabelul următor sunt prezentate nivelurile de prioritate ale cererilor de întrerupere interne şi externe, care determină ordinea de tratare când cererile apar simultan.

Tip de întrerupere	Prioritatea
Tip 0, INT nn	0 - maximă
INTO	1
NMI	2
INTR	3
Tip 1 (pas cu pas)	4 - minimă

În fig.11 sunt prezentate operațiile executate de microprocesor la acceptarea şi achitarea unei cereri de întrerupere. O cerere de întrerupere este luată în considerare numai la terminarea execuției instrucțiunii curente. Dacă indicatorul IF=1, procesorul execută secvența de acceptare cerere de întrerupere externă, preia codul întreruperii de pe magistrala de date şi execută operațiile comune tuturor tipurilor de întrerupere.

Se salvează în stivă IP, CS şi registrul indicatorilor de condiții, F, după care indicatorii IF şi TF sunt anulați. Dacă în timpul procesării unei cereri de întrerupere, apare o cerere pe linia NMI înaite de realizarea accesului la subrutina de tratare a primei întreruperi, cererea de pe linia NMI va fi servită cu prioritate. După execuția subrutinei de tratare întrerupere se refac din stivă registrele CS, IP, F şi se revine în programul principal. La încheierea instrucțiunii următoare se testează existența unei cereri de întrerupere şi dacă da, procesul se reia.

11.7 Unitate centrală cu microprocesorul 8086

După apariția lui Intel 8086, a fost lansată familia de circuite Intel necesară realizării unităților centrale cu 8086. Ea cuprinde:

- coprocesoarele 8087, 8089;
- 8284 - generator de tact;
- 8228 - controler de magistrală;
- 8282, 8283 - registre tampon de opt biți;
- 8286, 8287 - amplificatoare bidirecționale (8 biți) de magistrală.

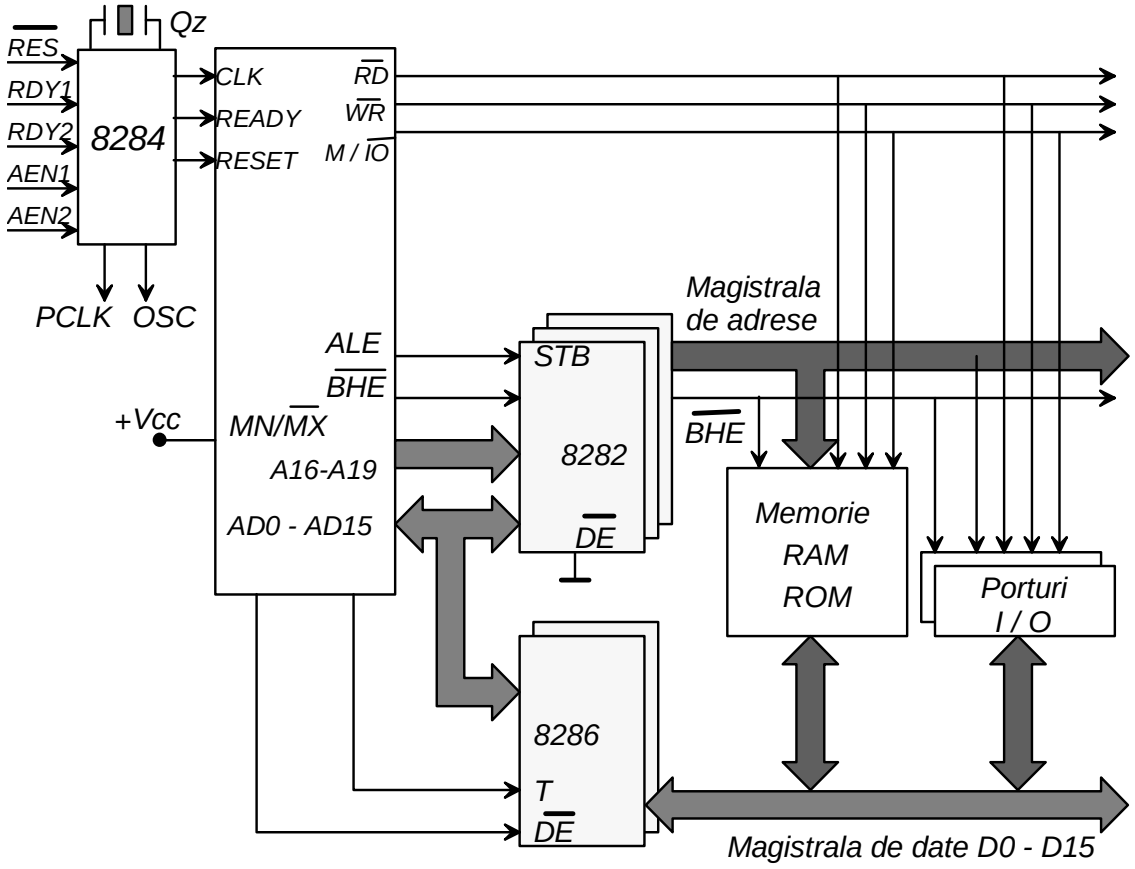


Fig. 12 Unitate centrală cu 8086 în modul minim

Generatorul de tact 8284 generează semnalul CLK pentru microprocesor şi PCLK pentru circuitele specializate cu diferite funcții în sistem. De asemenea generează semnalele RESET şi READY către microprocesor sincronizate cu semnalul de tact.

Pentru generarea semnalului CLK, se divizează semnalul generat de un oscilator cu cuarț sau un semnal exterior; semnalul PCLK are o frecvență de două ori mai mică şi un factor de umplere 1/2.

Registrul 8282 este utilizat ca tampon pentru demultiplexarea magistralei comune de date şi adrese. Are şi rol de amplificator de magistrală, asigurând un *fan - out* de 20 intrări TTL. Informația de la

intrările DI apare al ieşirile DO pe nivelul 1 logic al semnalului STB şi este memorată în cele 8 circuite basculante bistabile de tip D. Pentru ca informația să fie disponibilă la ieşiri, este necesar ca semnalul de validare $\overline{DE} = 0$.

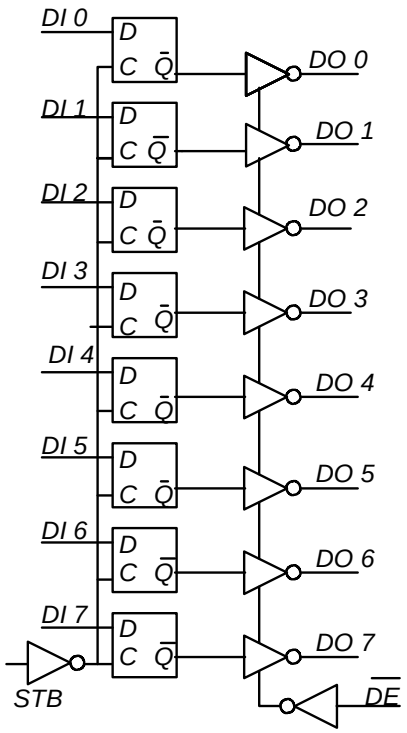


Fig. 13. Schema circuitului 8282

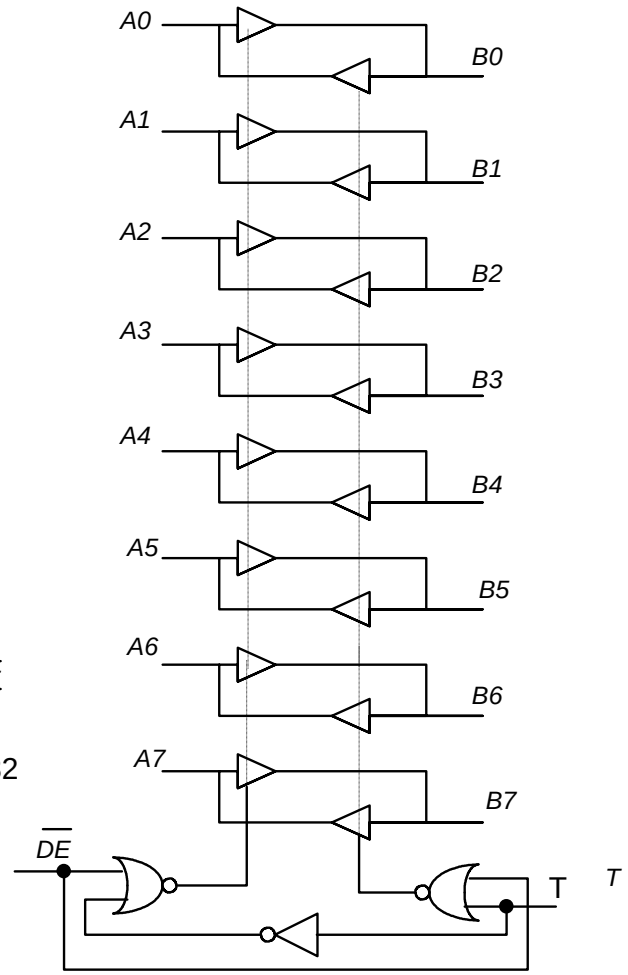


Fig. 14. Schema circuitului 8286

Amplificatorul de magistrală 8286 este bidirecțional, pe 8 biți, cu ieşiri cu trei stări şi *fan-out* 20 pentru ieşirile B, 6 pentru ieşirile A.

Circuitul realizează funcțiile de amplificare şi transfer atâta timp cât $\overline{DE} = 0$. Dacă $\overline{DE} = 1$, ieşirile trec în starea de înaltă impedanță. Sensul transferului este stabilit de intrarea T: dacă *T=1*, transferul se face de la A la B iar dacă *T=0*, de la B la A.

În modul maxim, 8086 poate fi utilizat în sisteme de complexitate mare, de tip multiprocesor. Generarea semnalelor de comandă către memorie şi porturi se face prin intermediul unui circuit specializat din familie, 8288.

Controlerul de magistrale, 8288, amplifică ieşirile de comandă şi date, realizând un *fan-out* 20 pentru ieşirile de comandă şi 10 pentru cele de date.

În funcție de combinația binară dată de liniile de stare, circuitul generează semnalele de comandă pentru citire/scriere memorie şi porturi precum şi pentru acceptarea unei cereri de întrerupere.

Circuitul generează câteva semnale de control: DEN (validarea transferului pe magistrala de date, DT/R (indică sensul transferului), ALE (Adress Latch Enable - pentru demultiplexarea magistralei de adrese şi date)

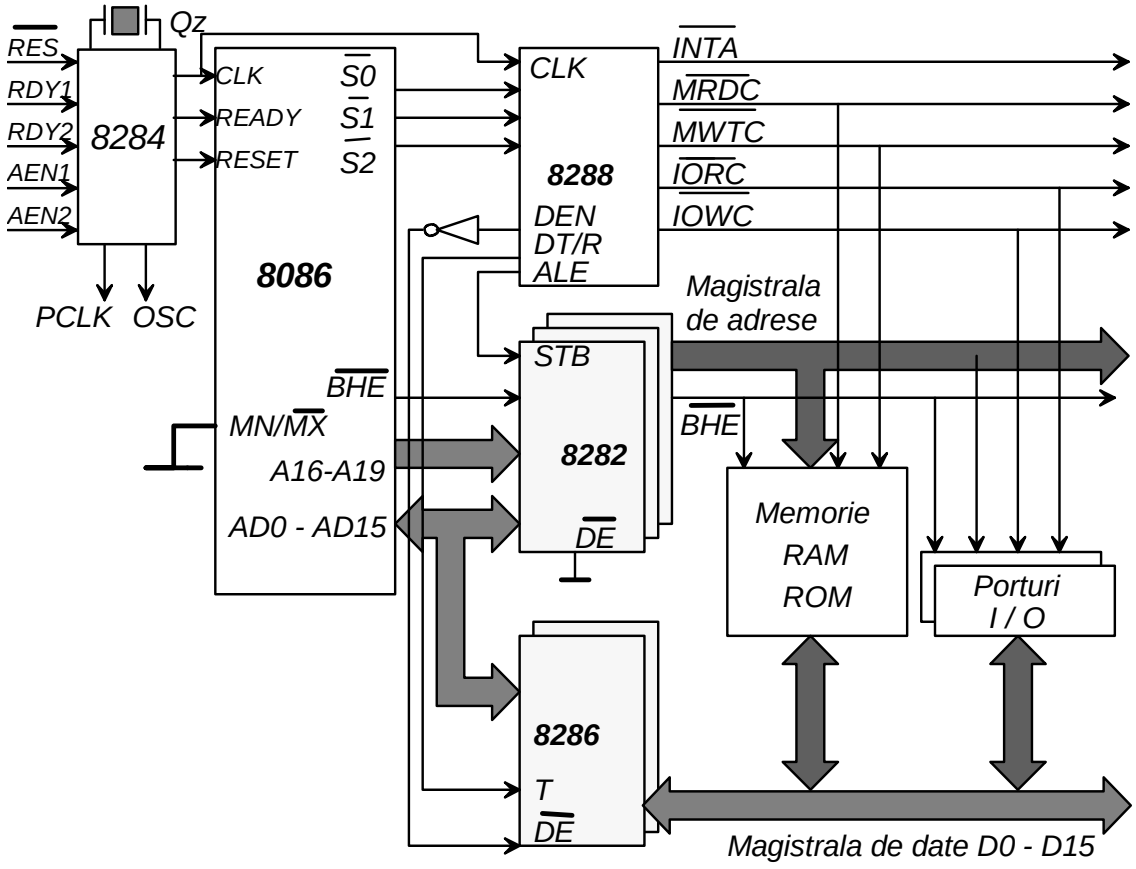


Fig. 15 Unitate centrală cu 8086 în modul maxim

Microprocesoare
Îndrumar de laborator

Dan NICULA [1]
Alexandru PIUKOVICI Radu GĂVRUŞ

August, 1999

[1]Universitatea TRANSILVANIA Braşov, Catedra de Electronică şi Calculatoare, Email: nicula@vega.unitbv.ro

Prefaţă

Prima parte a laboratorului cuprinde aplicaţii cu macheta didactică *MPF1-B Microprofessor*, echipată cu microprocesor Z80. Lucrările propuse au mai multe scopuri:

Exersarea unui limbaj de asamblare prin studierea tipurilor de instrucţiuni, modurilor de adresare şi a modului de transformare a instrucţiunilor de asamblare în cod maşină.

Studierea unei scheme minimale a unui sistem cu microprocesor şi a modului de interfaţare dintre microprocesor, memorie şi periferice.

Studierea microprocesorului ca sistem digital prin vizualizarea cu osciloscopul şi analizorul logic a semnalelor generate de acesta.

Partea a doua a laboratorului particularizează informaţiile generale dobîndite la cursul de "Microprocesoare" pentru familia de microprocesoare Intel 80x86. Sînt necesare două justificări:

De ce 80x86? Pentru că microprocesoarele compatibile Intel 80x86 echipează majoritatea calculatoarelor personale existente în prezent pe piaţă. Calculatoarele cu microprocesoare 80x86 pot rula trei sisteme de operare foarte răspîndite: DOS, Windows, Linux. Aplicaţiile care rulează pe aceste calculatoare necesită de multe ori module scrise în limbaj de asamblare. Totodată, un număr mare de alte sisteme digitale (plăci de achiziţie şi prelucrare de date, sisteme de automatizare şi control) conţin microprocesoare din această familie.

De ce 8086 şi nu ultima generaţie Intel? Pentru că aceste lucrări de laborator au scopul de a prezenta elementele de bază ale utilizării unui microprocesor şi de a oferi teme pentru exersarea programării în limbaj de asamblare. Procesorul 8086 reprezintă un "standard" al arhitecturilor 80x86. Folosirea facilităţilor existente la cei mai noi membrii ai familiei (gestiunea memoriei, multitasking, instrucţiuni multimedia) va fi tratată la disciplinele "Aplicaţii ale calculatoarelor", "Sisteme de operare" şi "Multimedia".

Ce este un sistem cu microprocesor?

Un sistem cu microprocesor, deseori numit "calculator", conţine trei mari blocuri constitutive, conectate aşa cum se prezintă în figura 1. Aceste blocuri sînt:

- Unitatea centrală de prelucrare, microprocesorul (Central Processing Unit - CPU);
- Memoria;
- Dispozitivele de intrare/ieşire (Input/Output = I/O).

Unitatea centrală de prelucrare implementată sub forma unui chip microprocesor, este piesa centrală a oricărui sistem de calcul. CPU realizează prelucrări numerice (adunări,

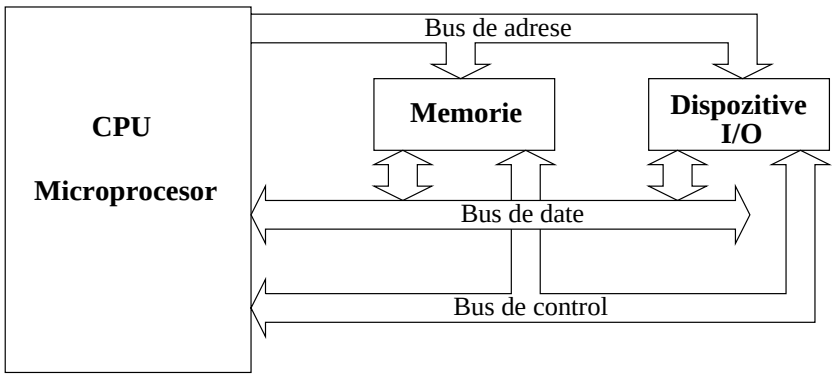


Figura 1: Schema bloc a unui sistem cu microprocesor.

scăderi, înmulţiri, etc.) şi operaţii logice asupra fluxului de date. Operaţiile realizate de CPU sînt controlate printr-o *secvenţă de instrucţiuni* grupate într-un *program*. Programele şi datele sînt înmagazinate în memorie. CPU citeşte cîte o instrucţiune din memorie, o decodifică şi apoi o execută. În procesul de execuţie, CPU poate citi/scrie date din/în memorie. Un microprocesor se structurează în două blocuri funcţionale: *calea de date* şi *calea de control*. Elementele esenţiale din calea de date sînt *registrele* şi *unitatea logico-aritmetică* (Arithmetic Logic Unit = ALU).

Registrele reprezintă locaţii de memorie temporare aflate în interiorul CPU. Registrele sînt fie *dedicate* (*program counter*, *status register*), fie *generale*.

Unitatea logico-aritmetică este unitatea care realizează prelucrarea efectivă a datelor. Operaţiile realizate de ALU sînt fie logice (operanzi interpretaţi ca o mulţime de biţi), fie aritmetice (operanzi interpretaţi ca numere exprimate în baza doi).

Calea de control coordonează activitatea microprocesorului şi realizează secvenţialitatea execuţiei programelor. Circuitele din calea de control decodifică instrucţiunea şi lansează comenzi pentru unităţile interne şi externe în scopul executării acesteia.

Memoria înmagazinează programele şi datele. Programul de iniţializare şi gestionare a resurselor sistemului (monitor, sistem de operare) este menţinut într-o memorie ROM. Restul spaţiului de memorie este ocupat de memorie RAM.

Dispozitivele de intrare/ieşire denumite şi *periferice*, reprezintă mijloacele de comunicare ale microprocesorului cu lumea exterioară. Tastatura, monitorul sau imprimanta sînt controlate de către CPU prin intermediul porturilor de intrare/ieşire.

Magistralele de adrese, date şi control interconectează unitatea centrală cu memoria şi dispozitivele I/O. Pe *bus-ul de date* se transferă bidirecţional informaţii codificate binar, interpretate ca date sau ca instrucţiuni. *Bus-ul de adrese* unidirecţional este folosit de CPU pentru a transmite adrese către memorie şi dispozitive I/O. Pe *bus-ul de control* se transmit comenzi de la CPU spre memorie şi spre dispozitivele I/O.

Întreruperile sînt situaţii în care microprocesorul îşi suspendă execuţia secvenţială a programului pentru a deservi apelul venit de la un periferic. De obicei, într-un sistem există mai multe dispozitive care pot lansa cereri de întrerupere. Pentru a putea fi servite toate, întreruperile trebuiesc ierarhizate prin asocierea unor priorităţi.

Accesul direct la memorie (Direct Memory Access - DMA) reprezintă o soluţie de transfer rapid a datelor de la un periferic în memorie fără ca acestea să mai treacă prin microprocesor. Prin utilizarea DMA, CPU predă controlul magistralelor către un dispozitiv periferic care controlează transferarea datelor direct în memoria sistemului.

Modurile de adresare reprezintă totalitatea modalităţilor de determinare a adreselor pentru accesarea memoriei externe.

Contribuţia autorilor este următoarea:

Dan Nicula (coordonator) - Partea a II-a, Lucrările 6, 7, 8, 9, 10, 11 şi anexele

Alexandru Piukovici - Partea I, Lucrările 1, 2 şi 3

Radu Găvruş - Partea I, Lucrările 4 şi 5

Fişierele cu programele propuse ca exemple în cadrul laboratoarelor pot fi accesate prin ftp anonim de la adresa ftp://vega.unitbv.ro/pub/microp. Orice fel de observaţie referitoare la acest îndrumar poate fi făcută prin e-mail nicula@vega.unitbv.ro.

Cuprins

Partea I

Microprocesorul Z80

Lucrarea 1

Prezentarea machetei cu microprocesor Z80

Această lucrare prezintă microprocesorul Z80 şi sistemul *MPF1-B Microprofessor*. Sînt prezentate funcţiile tastelor, facilităţile oferite de programul monitor, harta memoriei şi adresele porturilor de comandă.

1.1 Microprocesorul Z80

Microprocesorul Z80 este un procesor pe 8 biţi (unitatea logico-aritmetică acceptă operanzi reprezentaţi pe 8 biţi). Magistrala de date este de 8 biţi iar magistrala de adrese de 16 biţi (spaţiul de memorie este de $2^{16} = 64KB$). Setul de instrucţiuni conţine 158 de instrucţiuni.

Microprocesorul Z80 are următoarele caracteristici:

- generează semnalul de refresh pentru memoria DRAM;
- are un pin pentru primirea unei întreruperi nemascabile;
- implementează un mecanism de tratare a întreruperilor vectorizate;
- conţine un set dublu de registre;
- setul de instrucţiuni conţine instrucţiuni pentru adresarea indexată a memoriei;
- setul de instrucţiuni conţine instrucţiuni pentru prelucrarea unor blocuri de date din locaţii adiacente de memorie.

Din familia Z80 fac parte următoarele circuite specializate:

- controler de port paralel Z80-PIO;
- controler de port serial Z80-SIO;
- controler DMA;
- circuit timer Z80-CTC.

Z80 poate funcţiona şi cu porturi de la alte familii de microprocesoare:

- I8255 - interfaţă paralelă cu trei porturi;
- I8251 - interfaţă serială cu două porturi.

Structura internă a procesorului Z80 este prezentată în figura 1.1.

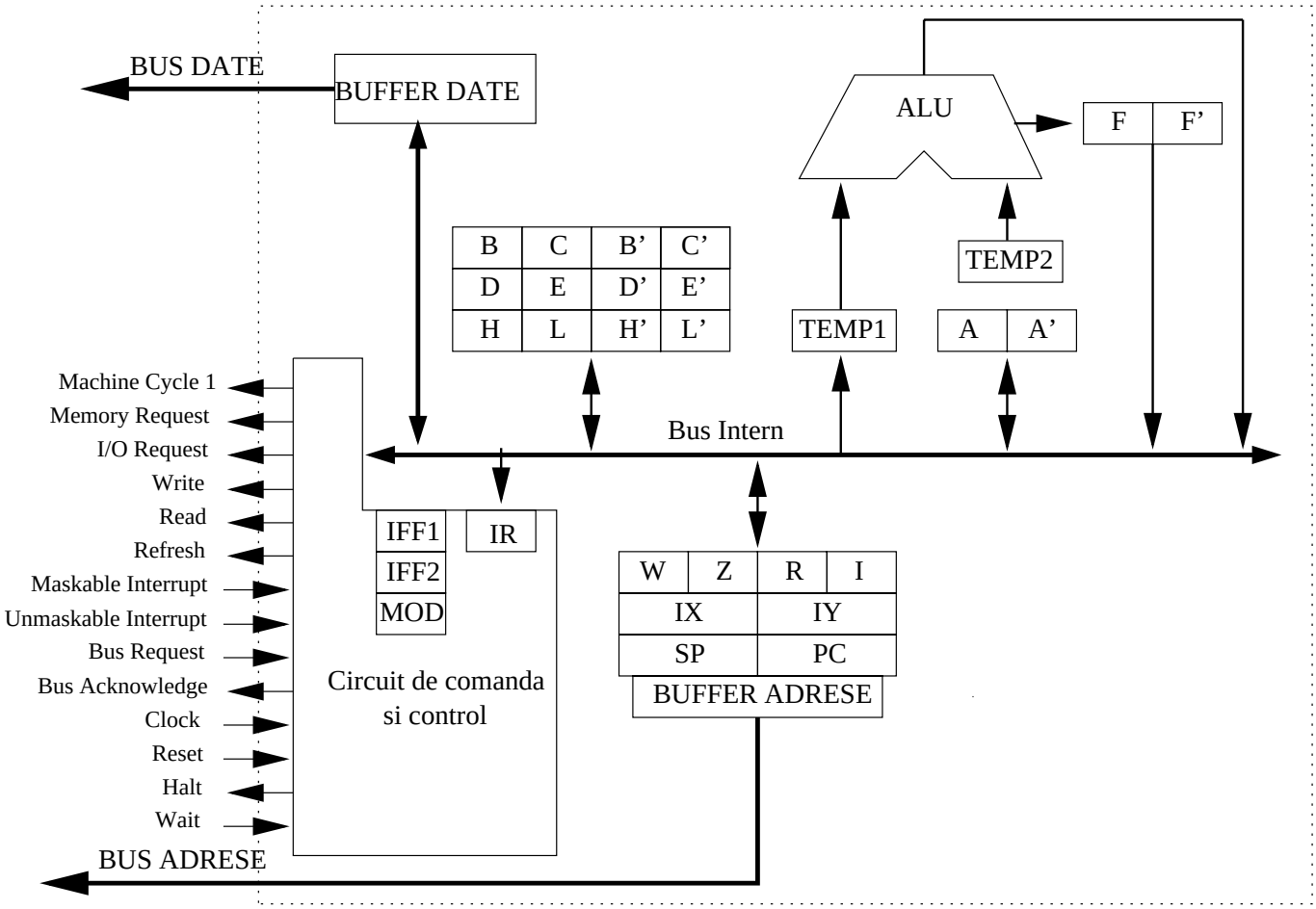


Figura 1.1: Structura internă a microprocesorului Z80.

1.1.1 Registrele procesorului

Registre specializate

Registrele din această categorie au roluri bine determinate în cadrul procesorului, avînd şi unele sarcini implicite:

- PC - *Program Counter* (16 biţi) utilizat pentru memorarea adresei instrucţiunii care urmează a fi executată;
- SP - *Stack Pointer* (16 biţi) utilizat pentru memorarea adresei vîrfului stivei. Stiva creşte spre adrese mici;

- IR - *Instruction Register* (8 biţi) registru invizibil pentru programator, folosit pentru memorarea codului instrucţiunii curente;
- A - *Accumulator* (8 biţi) utilizat ca registru implicit în multe operaţii aritmetice şi logice, A' - registru secundar;
- F - *Flag Register* (8 biţi) utilizat pentru memorarea indicatorilor de stare. Structura cuvîntului de stare este prezentată în tabelul 1.1:

7	6	5	4	3	2	1	0
S	Z	X	H	X	P	N	C

Tabelul 1.1: Structura registrului de stare şi indicatori.

  Semnificaţia biţilor din cuvîntul de stare este următoarea:

  - S - *Sign* este setat dacă rezultatul unei operaţii aritmetice este negativ;
  - Z - *Zero* este setat dacă rezultatul unei operaţii aritmetice este zero;
  - H - *Half Carry* este setat dacă există semitransport, de la bitul 3 la bitul 4 (se foloseşte la corecţia zecimală DAA);
  - P/V - *Parity/Overflow* este setat dacă numărul de biţi de valoare 1 din cuvînt este par (pentru paritate), sau dacă există o depăşire de domeniu (pentru operaţii aritmetice);
  - N - *Name of the last operation* este setat dacă ultima operaţie a fost adunare (indicator folosit pentru instrucţiunea DAA);
  - C - *Carry* este setat dacă există transport de la bitul cel mai semnificativ (MSB);
  - X - bit a cărui valoare nu contează.

- R - *Refresh Register* (7 biţi) utilizat pentru memorarea adresei de refresh;
- I - *Interrupt Register* (8 biţi) utilizat pentru identificarea sursei care a cauzat întreruperea (utilizat în modul 2 de întreruperi, IM2).

Registre de uz general

Registrele din această categorie au rolul de a păstra datele în imediata vecinătate a *ALU* pentru a putea fi accesate şi prelucrate rapid:

- B, C - registre generale de cîte 8 biţi care se pot accesa prin intermediul unor instrucţiuni ca un registru dublu de 16 biţi numit BC;
- D, E - registre generale de cîte 8 biţi care se pot accesa prin intermediul unor instrucţiuni ca un registru dublu de 16 biţi numit DE;
- H, L - registre generale de cîte 8 biţi care se pot accesa prin intermediul unor instrucţiuni ca un registru dublu de 16 biţi numit HL. Aceste registre se folosesc, în anumite instrucţiuni, pentru adresare indirectă;
- IX, IY - registre index de cîte 16 biţi utilizate la adresare indexată;

- B', C', D', E', H', L' - registre secundare al căror conţinut se poate interschimba cu conţinutul registrelor B, C, D, E, H, L prin executarea instrucţiunii EXX.

Registre de manevră

Registrele din această categorie sînt invizibile pentru programator, fiind folosite de către procesor ca locaţii de memorie temporare:

- W, Z - registre de cîte 8 biţi care se pot accesa şi ca un registru dublu de 16 biţi numit WZ, utilizat de către procesor pentru memorarea adreselor salturilor necondiţionate;
- T - *Temporar Register* utilizat de către procesor pentru diverse instrucţiuni;
- DB - *Data Buffer* registru tampon bidirecţional de 8 biţi utilizat la interfaţa dintre procesor şi lumea exterioară;
- AB - *Address Buffer* registru de 16 biţi care izolează magistrala de adrese internă de cea externă.

1.1.2 Circuite de comandă şi control

Circuitele de comandă şi control au funcţia de a genera în exterior semnalele de comandă de la procesor şi de a gestiona modul de lucru cu întreruperile. Alături de registrul IR în care se memorează codul instrucţiunii care se execută, circuitele de comandă mai conţin:

- IFF1, IFF2 - bistabile de validare/inhibare a întreruperilor;
- MOD - registru de 2 biţi utilizat pentru memorarea modului de întrerupere activ la un moment dat: IM0, IM1, IM2.

Semnalele de intrare şi ieşire ale circuitelor de comandă şi control sînt prezentate în figura 1.1.

1.1.3 Unitatea aritmetică şi logică

Unitatea aritmetică şi logică este de 8 biţi. Operaţiile aritmetice executate sînt: adunare, scădere, incrementare şi decrementare. Operaţiile logice executate sînt: OR, AND, XOR, NOT, comparare (la nivel de bit). Procesorul poate executa şi operaţii cu un singur operand, deplasări şi rotiri. Odată cu generarea rezultatului operaţiei aritmetice/logice se setează şi indicatorii din registrul de indicatori F.

1.1.4 Ciclurile procesorului

Definirea celor trei ciclii ce caracterizează funcţionarea procesorului este prezentată în figura 1.2. Aceşti ciclii sînt:

Ciclul de ceas (*Clock Cycle - CC*) are durata perioadei semnalului de ceas.

Ciclul maşină (*Machine Cycle - MC*) este format din unul sau mai mulţi ciclii de ceas. Pe durata unui ciclu maşină se desfăşoară o activitate intermediară, bine definită, cu o finalitate clară.

Ciclul instrucţiune (*Instruction Cycle - IC*) este format din unul sau mai mulţi ciclii maşină. Pe durata unui ciclu instrucţiune se desfăşoară toate acţiunile legate de execuţia unei instrucţiuni.

Ciclurile maşină ale procesorului Z80 sînt:

- citirea codului operaţiei (*Fetch - M1*);
- citirea unui operand din memorie (*Read*);
- scrierea unui operand în memorie (*Write*);
- citirea unui operand dintr-un port (*In*);
- scrierea unui operand într-un port (*Out*);
- ciclul intern;
- acceptarea cererii de întrerupere;
- acceptarea cererii de magistrale.

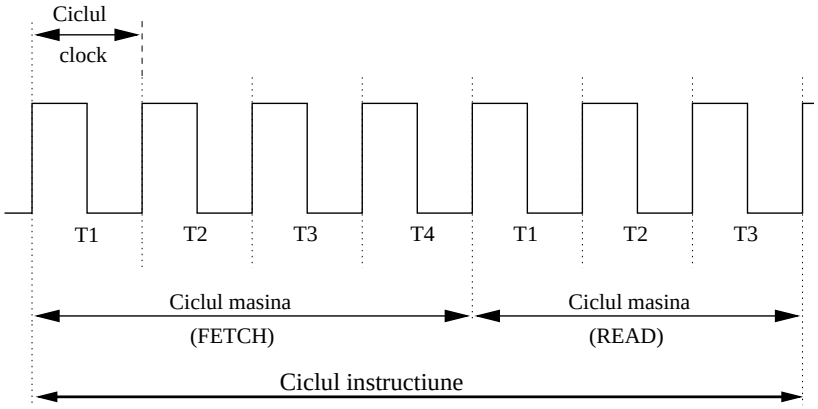


Figura 1.2: Ciclul instrucţiune pentru o operaţie de citire din memorie.

1.2 Macheta de laborator MPF1-B Microprofessor

1.2.1 Descriere hardware

Macheta didactică *MPF1-B Microprofessor* conţine un sistem cu microprocesor Z80. Frecvenţa semnalului de ceas este de 1.79 MHz. Sistemul dispune de două tipuri de memorie: o memorie ROM de capacitate 6 KB, în care sînt înscrise programele producătorului, şi o memorie RAM de capacitate 2 KB. Sistemul mai are posibilitatea de extindere a memoriei, avînd rezervat pentru aceasta un spaţiu de 8 KB. Zona de memorie ROM se întinde între adresele 0000H - 17FFH. Sistemul dispune de un program monitor care este înscris în EPROM-ul U6 şi ocupă spaţiul 0000H - 0FFFH. De la adresa 0800H este stocat un program de transfer între un calculator compatibil IBM PC şi macheta de laborator MPF1-B. Memoria RAM este de tip static (deci nu are nevoie de reîmprospătarea informaţiei) şi ocupă zona 1800H - 1FFFH. Conform

specificaţiilor tehnice ale MPF1-B, 80 de octeţi din memoria RAM (1FAFH - 1FFFH) sînt folosiţi de programul monitor. Se recomandă ca în acest spaţiu să nu se efectueze înscrieri. Spaţiul rezervat pentru extindere (2000H - 3FFFH) poate fi ocupat cu un alt EPROM sau cu o memorie RAM de tip static.

Harta memoriei poate fi configurată cu ajutorul unor jumperi (JP3, JP4, JP5) aflaţi pe machetă, conform tabelului 1.2.

U6	U7	JP3	JP4	JP5
4K (0000H - 0FFFH)	4K (2000H - 2FFFH)	2 - 3	2 - 1	2 - 3
4K (0000H - 0FFFH)	8K (2000H - 3FFFH)	2 - 3	2 - 3	2 - 1
6K (0000H - 1FFFH)	8K (2000H - 3FFFH)	2 - 1	2 - 3	2 - 1

Tabelul 1.2: Configurarea memoriei.

În figura 1.3 este prezentată harta memoriei sistemului MPF1-B.

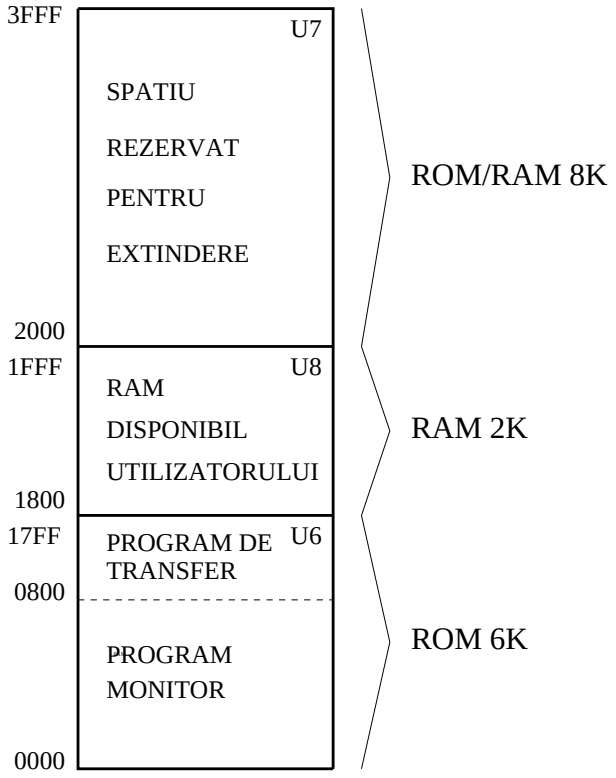


Figura 1.3: Harta memoriei sistemului MPF1-B.

Sistemul dispune de trei circuite specializate care folosesc porturi de intrare/ieşire. Interfaţa paralelă PIO dispune de două porturi paralele (16 biţi) şi ocupă următoarele adrese:

- 80H - portul A;
- 81H - portul B;
- 82H - controlul portului A;

- 80H - controlul portului B.

Circuitul counter/timer CTC ocupă următoarele adrese:

- 40H - canalul 0;
- 41H - canalul 1;
- 42H - canalul 2;
- 43H - registrul de control.

Al treilea circuit de care dispune sistemul este interfaţa paralelă programabilă 8255 din familia Intel. Aceasta are un total de 24 linii paralele care sînt folosite la scanarea tastaturii şi la controlul afişajului. Adresele ocupate de acest circuit sînt:

- 00H - portul A;
- 01H - portul B;
- 02H - portul C;
- 03H - registrul de control.

Portul A are biţii conectaţi astfel:

- bitul 7 - la intrarea de citire de la un dispozitiv de stocare magnetică;
- bitul 6 - la tasta *User Key* activă în stare *low*;
- biţii 5-0 - la şase rînduri ale tastaturii.

Portul B controlează cele 7 segmente şi punctele afişajului. Toţi biţii de ieşire sînt activi în stare *high*.

Portul C are biţii conectaţi astfel:

- bitul 7 - la ieşirea de scriere a unui dispozitiv de stocare pe bandă magnetică;
- bitul 6 - la tasta [MONI];
- biţii 5-0 - la şase coloane ale tastaturii şi ale afişajului. Bitul 0 corespunde primei cifre de la dreapta, iar bitul 5 primei cifre de la stînga. Toţi biţii sînt activi în stare *high*.

Portul C mai este conectat şi la difuzorul de pe machetă şi la led-ul *TONE-OUT*. Led-ul este aprins cînd ieşirea este în 0 logic. Afişajul este format din 6 celule cu 7 segmente. Tastatura conţine 36 taste din care 19 corespund anumitor funcţii, 16 reprezintă cifrele sistemului de numărare hexazecimal şi o tastă pe care poate fi definită de către utilizator. Ceasul sistemului este generat cu ajutorul unui oscilator cu cristal de cuarţ cu frecvenţa de $3,58MHz$. Acest semnal este divizat şi apoi aplicat la intrarea de ceas a microprocesorului Z80.

Iniţializarea sistemului se poate face în două moduri. Primul mod este punerea sub tensiune, caz în care au loc următoarele acţiuni:

1. Bistabilul de activare a întreruperilor este resetat ($IFF = 0$);
2. Registrul de întreruperi este resetat ($I = 0$);
3. Modul de întreruperi este setat cu valoarea zero ($MOD = 0$);

4. Registrul PC este încărcat cu valoarea 1800H;
5. Registrul SP este încărcat cu valoarea 1F9FH;
6. Se elimină punctele de oprire setate de utilizator;
7. Adresa rutinei de tratare a întreruperilor este setată la valoarea 0066H (această adresă se memorează la adresele 1FFEH şi 1FFFH);
8. Semnătura machetei, `uPF-1`, va defila pe afişaj.

Al doilea mod de iniţializare este prin activarea tastei de reset [RS]. În acest caz, au loc acţiunile de la punctele 1-5. Rutina de tratare a întreruperilor şi punctele de oprire rămîn neafectate, iar semnătura machetei nu mai defilează pe afişaj.

1.2.2 Descriere software

Sistemul dispune de un program monitor care permite introducerea programelor de la tastatură, verificarea şi rularea lor pas cu pas.

Tastele şi semnificaţia acestora este prezentată în tabelul 1.3.

Operaţii de bază

Iniţializarea sistemului se face prin apăsarea tastei de reset [RS]. Examinarea şi modificarea datelor din memorie se realizează folosind tastele adrese [ADDR] şi date [DATA]. Pentru verificarea unei date din memorie se tastează [ADDR] şi adresa la care se găseşte data respectivă. La apăsarea tastei [ADDR] vor apărea 4 puncte în dreptul primelor 4 cifre din stînga afişajului (cifrele care reprezintă adresa locaţiei de memorie). Cele 4 puncte semnifică faptul că este activ cîmpul de introducere a adresei. Cifrele care vor fi introduse, vor reprezenta adresa locaţiei de memorie ce va fi accesată. Primele două cifre din dreapta vor reprezenta data stocată la adresa afişată în stînga. Pentru vizualizarea datei de la adresa următoare se va tasta [+], iar pentru vizualizarea datei de la adresa anterioară se va tasta [-]. Pentru modificarea datei de la adresa afişată trebuie ca punctele să se afle în cîmpul de date. Acest lucru se obţine apăsînd tasta [DATA]. În momentul în care punctele se află în cîmpul de date, se pot modifica datele. Utilizatorul poate modifica numai datele dintre adresele 1800H şi 1FFFH (zona memoriei RAM). De asemenea, utilizatorul trebuie să evite modificarea datelor stocate în zona 1FA0H şi 1FFFH, deoarece această zonă este folosită de programul monitor.

Examinarea şi modificarea datelor stocate în registre se face cu tastele [REG] şi [DATA]. Pentru verificarea conţinutului unui registru se apasă tasta [REG] urmată de tasta pe care este înscris numele registrului (tastele de introducere a datelor). În partea dreptă se afişează numele registrului iar în partea stîngă datele memorate. Registrele de 8 biţi sînt grupate cîte două astfel: AF, BC, DE, HL şi AF', BC', DE', HL', reprezentate cu puncte lîngă nume. Registrul I şi bistabilul de validare a întreruperilor sînt grupate împreună. Pentru modificarea datelor, se apasă tasta [DATA], după care apar două puncte în zona corespunzătoare celui de-al doilea registru (poziţiile 3-4). După ce se modifică valoarea memorată, se apasă tasta [+] şi se trece la poziţiile 1-2, corespunzătoare primului registru.

Registrul de indicatori are o reprezentare particulară. Sistemul MPF1-B decodifică acest registru şi îl afişează în grupuri de 4 biţi astfel:

Tastă	*Semnificaţie*
[RS]	(ReSet) Semnal de iniţializare a sistemului
[ADDR]	(ADDRess) Introducerea unei valori în cîmpul de adresă
[REG]	(REGister) Selectarea unui registru al cărui conţinut este vizualizat
[DATA]	(DATA) Introducerea unei valori în cîmpul de date
[PC]	Tranferă controlul programului la valoarea curentă a registrului PC
[+]	Incrementare valoare afişată (dată sau adresă)
[-]	Decrementare valoare afişată (dată sau adresă)
[STEP]	(STEP) Rularea unei instrucţiuni din programul utilizatorului
[SBR]	(Set Break Point) Stabileşte un punct de oprire în programul utilizatorului
[CBR]	(Clear Brak Point) Anulează punctul de oprire din programul utilizatorului
[MONI]	(MONItor) Terminarea programului curent şi redarea controlului programului monitor
[GO]	(GO) Comandă de lansare în execuţie a programului aflat în memorie la adresa afişată pe display
[INS]	(INSert) Inserarea unui byte în memorie
[DEL]	(DELete) Ştergerea unui byte din memorie
[MOVE]	(MOVE) Copierea unui bloc de date dintr-un loc în altul
[RELA]	(RELAtive) Calcularea şi inserarea unei adrese relative pentru instrucţiuni de salt
[TAPE WR]	(TAPE WRite) Scriere pe bandă magnetică (ieşire serială)
[TAPE RD]	(TAPE ReaD) Citire de pe bandă magnetică (intrare serială)
[INTR]	(INTeRrupt) Semnal conectat la pinul de întreruperi mascabile al procesorului
[USER KEY]	Tastă a cărei semnificaţie poate fi modificată de către utilizator

Tabelul 1.3: Semnificaţia tastelor sistemului MPF1-B.

- grupul S, Z, H - [S Z x H];
- grupul P/V, N, C - [x P/V N C].

La apăsarea tastei [PC] în registrul PC se înscrie cea mai mică adresă din RAM (1800H).

Verificarea şi rularea programelor pas cu pas

Adresa de început a programului încărcat în memorie este stabilită prin apăsarea tastei [ADDR]. După apariţia adresei pe partea din stînga a afişajului, lansarea în execuţie a programului se face prin apăsarea tastei [GO]. Tasta [STEP] este similară tastei [GO], cu deosebirea că în acest caz procesorul va executa o singură instrucţiune, după care controlul este redat programului monitor. Programul monitor afişează noua valoare a registrului PC. Utilizatorul poate modifica şi verifica registrele sau conţinutul memoriei la fiecare pas. Tastele [GO] şi [STEP] sînt funcţionale numai cînd afişajul este în format standard: *Adresă-Dată*. Rularea unui program de dimensiuni mari pas cu pas consumă mult timp. Pentru evitarea acestui lucru se poate fixa un punct de oprire în program cu ajutorul tastei [SBR]. Astfel, programul se va executa

pînă la acest punct, după care se dă controlul programului monitor. În acest moment se pot vizualiza şi modifica date din memorie. Fixarea punctului de oprire se face apăsînd tasta [SBR], atunci cînd este afişată adresa la care se doreşte fixarea punctului de oprire. Într-un program se poate fixa un singur punct de oprire, care este simbolizat prin afişarea punctelor atît în zona de adrese, cît şi în zona de date a afişajului. Nu este permisă fixarea unui punct de oprire în zona memoriei ROM. Dacă o instrucţiune are mai mulţi octeţi, punctul de oprire trebuie fixat la primul octet al instrucţiunii, altfel vor apărea erori.

Eliminarea punctului de oprire se face apăsînd în orice moment tasta [CBR]. După apăsarea acestei taste, pe afişaj va apărea: [F.F.F.F.-F.F.]. În cazul în care un program se blochează sau intră în buclă infinită, se poate apăsa tasta [MONI] pentru ca sistemul să dea din nou controlul programului monitor. Pe display se va afişa conţinutul registrului PC. De asemenea, după execuţia unei instrucţiuni HALT, se poate apăsa tasta [MONI] pentru a reda controlul programului monitor, iar PC va lua valoarea adresei instrucţiunii următoare.

Funcţii pentru simplificarea utilizării machetei

Transferarea unui bloc de memorie se face cu tasta [MOVE]. Operaţia permite mutarea unui bloc de date dintr-o zonă în alta a memoriei. Succesiunea tastelor care trebuie apăsate pentru a muta blocul cuprins între adresele 1800H şi 18FFH la adresa 1810H este prezentată în tabelul 1.4.

Tastă apăsată	*Afişaj*	*Comentarii*
[MOVE]	`x.x.x.x.-s`	S semnifică adresa de început a blocului de date (Start)
[1][8][0][0]	`1.8.0.0.-s`	Adresa de început a blocului a fost fixată la adresa 1800H
[+]	`x.x.x.x.-e`	E semnifică adresa de sfîrşit a blocului de date (End)
[1][8][F][F]	`1.8.F.F.-e`	Adresa de sfîrşit a blocului a fost fixată la adresa 18FFH
[+]	`x.x.x.x.-d`	D semnifică adresa de destinaţie (Destination)
[1][8][1][0]	`1.8.1.0.-d`	Adresa de destinaţie a blocului a fost fixată la adresa 1810H
[GO]	`1810-x.x.`	Comanda executivă pentru mutarea blocului de date

Tabelul 1.4: Mutarea unui bloc de date în memorie.

Ştergerea unui byte de date se face cu ajutorul tastei [DEL]. Tasta este funcţională numai cînd afişajul este în formatul standard *Adresă-Dată*. Tasta se apasă în momentul în care este afişată data de la adresa de unde se doreşte ştergerea. Toate datele de deasupra sînt translatate în jos cu o poziţie (înspre adresele mici), iar la adresa 1DFFH se încarcă valoarea 00H. Zona de memorie în care se pot face ştergeri este 1800H şi 1DFFH.

Inserarea unui byte de date se face cu ajutorul tastei [INS]. Tasta este funcţională numai cînd afişajul este în format standard *Adresă-Dată*. Tasta se apasă în momentul în care este afişată adresa la care se doreşte inserarea datei respective. Succesiunea tastelor care trebuie apăsate pentru a insera un byte în memorie este prezentată în tabelul 1.5. Conţinutul locaţiilor de memorie înainte şi după operaţia de inserare este prezentat în tabelul 1.6.

Zona de memorie în care se pot face inserări este între 1800H şi 1DFFH. La inserare, baitul de la adresa 1DFFH se pierde.

Tastă apăsată	*Afişaj*	*Comentarii*
[ADDR][1802]	`1.8.0.2.-22`	Stabileşte adresa după care se doreşte inserarea unui byte
[INS]	`1803-0.0.`	La apăsarea tastei de inserare se înscrie 00 la adresa următoare
[3][3]	`1803-3.3.`	Se introduce valoare baitului inserat (în acest caz 33)

Tabelul 1.5: Inserarea unui byte în memorie.

Adrese	*Date înainte de inserare*	*Date după inserare*
1800	00	00
1801	11	11
1802	22	22
1803 inserare 33H	44	33
1804	55	44
1805	66	55

Tabelul 1.6: Conţinutul memoriei la inserarea unui byte.

Calculul adresei relative, necesare instrucţiunilor JR şi DJNZ, se poate face cu ajutorul tastei [RELA]. Se consideră cazul unei instrucţiuni JR aflată în memorie la adresa 1800H. Saltul trebuie făcut la adresa 1804H. Succesiunea tastelor care trebuie apăsate este prezentată în tabelul 1.7. Instrucţiunea JR de la adresa 1800H are doi baiţi. După execuţia acesteia, registrul PC are valoarea 1802H. Rezultă că pentru a face saltul la adresa 1804H, PC ar trebui însumat cu 02H.

Întreruperea unui program

Întreruperile nemascabile sînt folosite numai de programul monitor. Pinul 16 al procesorului (INT) este conectat la tasta [INTR]. La execuţia codului din programul monitor de la adresa 0038H controlul este transferat rutinei care începe la adresa stocată în locaţiile 1FFEH şi 1FFFH. În timpul acestui proces, starea procesorului este afectată. Conţinutul original de la adresele 1FFEH şi 1FFFH este 0066H. Instrucţiunea de la adresa 0038H este executată în următoarele situaţii:

1. Este acceptată o întrerupere de mod 1;
2. Se execută instrucţiunea RST 38H (cod FFH);
3. Magistrala de date este trecută în 1 logic. Dacă o întrerupere de mod 0 este acceptată fără vector de întrerupere, se execută instrucţiunea RST 38H;
4. Cînd un program încearcă să sară la o adresă inexistentă.

Dacă nu se modifică adresa memorată în locaţiile 1FFEH şi 1FFFH, efectul executării instrucţiunii de la adresa 0038H este acelaşi cu cel produs de apăsarea tastei [MONI], sau de atingerea

Tastă apăsată	*Afişaj*	*Comentarii*
[RELA]	`x.x.x.x.-s`	Se introduce adresa de start, adică adresa la care se află instrucţiunea JR
[1][8][0][0]	`1.8.0.0.-s`	Adresa instrucţiunii de salt este, în acest caz, 1800H
[+]	`x.x.x.x.-d`	Se introduce adresa de destinaţie, adică adresa la care se face saltul
[1][8][0][4]	`1.8.0.4.-d`	Adresa la care se face saltul este, în acest caz, 1804H
[GO]	`1801-0.2.`	Sistemul MPF1-B prelucrează valorile introduse şi introduce numărul relativ cu care se face saltul la adresa ce urmează instrucţiunii JR. În acest caz, de la 1801H la 1804H sînt 2 baiţi (JR ocupă 2 octeţi de la adresele 1800H şi 1801H)

Tabelul 1.7: Calculul unei adrese relative de salt.

unui punct de oprire în program. Utilizatorul poate să-şi definească propria rutină de servire modificînd conţinutul locaţiilor 1FFEH şi 1FFFH.

Instrucţiunea RST 30H are acelaşi efect cu cel al unui punct de oprire. Se numeşte *software break* deoarece nu implică partea hardware. De obicei, este folosită la sfîrsitul unei aplicaţii utilizator. De asemenea, se utilizează atunci cînd se doreşte stabilirea mai multor puncte de oprire într-un program.

La fiecare oprire a aplicaţiei utilizator, programul monitor verifică valoarea memorată în registrul SP. Dacă stiva programului utilizatorului se suprapune peste cea de sistem, se semnalizează o eroare prin apariţia pe afişaj a mesajului: [SYS-SP].

1.2.3 Program de transfer între PC şi MPF1-B

Editarea programului sursă

Fişierul sursă se poate edita cu orice editor de texte (NCEDIT, EDIT). Se recomandă folosirea editorului CWE.EXE deoarece acesta conţine meniuri specifice editării unui cod sursă Z80 şi oferă posibilitatea consultării interactive a unei documentaţii referitoare la setul de instrucţiuni şi directivele de asamblare Z80.

Codul sursă trebuie să înceapă obligatoriu cu directiva *ORG* care specifică adresa de memorie unde se va plasa programul transferat pe machetă. Zona de memorie RAM este 1800H - 1FFFH. Se recomandă încărcarea programului începînd cu adresa 1800H.

Prelucrarea programului sursă

Asamblarea fişierului sursă <*filename.s*> se face cu comanda:

asm800 <filename.s> -s asm816 -l-o <filename.o>

Rezultatul asamblării constă în obţinerea fişierelor obiect <*filename.o*> şi listing <*filename.l*>.

Link-editarea fişierului obiect <*filename.o*> se face cu comanda:

mlink <filename.o> -e 00000 -o <filename.bin>

Rezultatul link-editării constă în obţinerea fişierului binar *<filename.bin>*.

Conversia fişierului binar *<filename.bin>* în format Intel HEX se face cu comanda:

mload <filename.bin> -i-o <filename.hex>

Rezultatul conversiei constă în obţinerea fişierului în format Intel hex *<filename.hex>*. Acest fişier va fi încărcat în memoria RAM de pe machetă, la adresa specificată prin directiva ORG.

Prelucrarea automată a fişierului sursă, pînă la obţinerea formatului hex, poate fi făcută prin lansarea în execuţie a fişierului batch XASM.BAT cu comanda:

xasm < filename.s >

Ca rezultat, se obţin fişierele:

- *<filename.o>* - fişier obiect obţinut în urma asamblării;
- *<filename.l>* - fişier listing ce conţine adresele şi codul programului;
- *<filename.bin>* - fişier binar obţinut în urma link-editării;
- *<filename.hex>* - fişier în format Intel Hex care conţine imaginea programului în memorie.

Transferul programului de la PC la machetă

Transferul programului între PC şi macheta cu Z80 se face prin legătură serială între portul calculatorului PC COM1 şi un pin al portului paralel 8255 al machetei. Etapele transferării unui program între PC şi machetă sînt detaliate în continuare.

1. Se lansează în execuţie programul de recepţie de pe macheta cu Z80. Programul se află în memoria ROM la adresa 0800H. Aşteptarea transferului de la PC este semnalizată pe machetă prin apariţia pe afişaj a patru linii orizontale (_ _ _ _).
2. Se lansează în execuţie programul de transmisie de pe PC cu comanda:

   comm <filename.hex>

   Începerea transmisiunii este semnalizată prin apariţia pe afişajul machetei a patru linii verticale (| | | |).

   Dacă transferul s-a efectuat fără erori, pe afişajul machetei va apărea mesajul `Good`. Altfel v-a apărea un mesaj de eroare.

Execuţia programului utilizatorului

Se introduce adresa de început a programului utilizatorului prin apăsarea tastei [ADDR]. Lansarea în execuţie se realizeză prin apăsarea tastei [GO].

1.3 Experimente

I. Urmăriţi pe schema machetei blocul de selecţie a memoriei şi explicaţi conţinutul tabelului 1.2.

II. Încărcaţi următorul program în memoria RAM a machetei:

Adresă	Cod maşină	Instrucţiuni Z80
1800	3E00	LD A, 0
1802	3C	INC A
1803	47	LB B, A
1804	04	INC B
1805	48	LD C, B
1806	FB	EI

Justificaţi codul maşină plecînd de la instrucţiunile scrise în limbaj de asamblare.

III. Rulaţi programul. Ce valori vor avea registrele A, B, C, la terminarea programului?

IV. Executaţi programul pas cu pas. La fiecare pas, vizualizaţi conţinutul registrelor A, B, C.

V. Modificaţi programul astfel încît, după rulare, registrul C să conţină valoarea 6.

VI. Editaţi codul pe un PC, utilizînd editorul *CWE.EXE*. Asamblaţi programul şi transferaţi-l pe machetă. Rulaţi programul transferat.

Lucrarea 2

Vederea programatorului asupra procesorului Z80

Această lucrare prezintă setul de instrucţiuni al microprocesorului Z80 şi modul de utilizare a machetei de laborator. Se vor exersa programe simple, care pot fi asamblate manual. Introducerea programelor se va face atît manual cît şi prin transferul codului asamblat de la calculatorul PC unde este conectată macheta.

2.1 Moduri de adresare

Modul în care operanzii sînt aduşi pentru a fi procesaţi de către microprocesor, calea pe care i se comunică microprocesorului adresa la care se află stocat operandul căutat, se numeşte *mod de adresare*. Codul unei instrucţiuni trebuie să conţină şi informaţii asupra a ceea ce are de făcut instrucţiunea respectivă, nu numai adresa datelor implicate de aceasta. Adresarea registrelor, a celulelor de memorie şi a dispozitivelor de intrare/ieşire diferă esenţial. Microprocesorul Z80 are următoarele registre simpli (de 8 biţi): A, B, C, D, E, H, L. Adresele celor şapte registre pot fi codificate pe 3 biţi. Spaţiul de memorie adresabilă este $2^{16} = 64K$. Pentru a adresa o celulă de memorie, este necesară o adresă de 16 biţi. Pentru adresarea unui dispozitiv de intrare/ieşire este necesară o adresă de 8 biţi.

Microprocesorul Z80 are 6 moduri de adresare, prezentate în continuare.

- *Adresare imediată.* Acestă adresare este folosită ori de cîte ori corpul unei instrucţiuni înglobează şi data care reprezintă obiectul acelei instrucţiuni.

```
Limbaj de asamblare   Cod maşină
  LD A, 55H            3E55H       ;Încarcă registrul A
                                   ;cu valoarea 55H. Data (55H) este un
                                   ;octet de sine stătător în cîmpul de
                                   ;doi octeţi ai instrucţiunii.
  LD DE, 6AF5H         11F56AH     ;Încarcă registrul dublu DE cu
                                   ;valoarea hexazecimală 6AF5H.
                                   ;Data este un cuvînt de doi octeţi
                                   ;în cîmpul celor trei ai instrucţiunii.
  AND 8EH              E68EH       ;Se execută o operaţie logică între
```

```
                                    ;conţinutul registrului acumulator
                                    ;şi numărul 8EH.
```

- *Adresare implicită.* Această adresare este folosită dacă toate informaţiile necesare pentru localizarea datei care reprezintă obiectul unei instrucţiuni sînt încorporate în codul acesteia.

```
Limbaj de asamblare   Cod maşină
LD C, D                4AH          ;Transferă conţinutul registrului D în
                                    ;registrul C. Atît adresa iniţială a
                                    ;datei cît şi adresa ei de destinaţie
                                    ;sînt specificate în octetul de cod
ADD HL, BC             09H          ;Conţinutul registrului dublu HL este
                                    ;adunat cu cel al registrului dublu BC,
                                    ;ambii operanzi se specifică implicit,
                                    ;în octetul de cod.
LD DE, 6AF5H           11F56A       ;Operandul sursă este adresat imediat,
                                    ;iar operandul destinaţie este adresat
                                    ;implicit, ca fiind în registrul DE.
```

- *Adresare indirectă.* Această adresare este folosită dacă adresa unui operand este conţinută într-un registru sau o locaţie de memorie. Adresarea indirectă prin memorie se foloseşte într-un singur caz: atunci cînd Z80 acceptă o întrerupere în modul 2, adresa de început a subrutinei de tratare a întreruperii este determinată folosind această tehnică.

```
Limbaj de asamblare   Cod maşină
ADC (HL)               8EH          ;Adună la acumulatorul numărul stocat
                                    ;în memorie, la adresa conţinută în
                                    ;registrul dublu HL, iar apoi este
                                    ;adăugată şi valoarea bitului de
                                    ;transport.
PUSH DE                D5H          ;Conţinutul registrelor D şi E este
                                    ;salvat în memorie la adresa următoare
                                    ;din stivă, începînd cu registrul D.
                                    ;Adresa la care se face transferul este
                                    ;conţinută în indicatorul de stivă SP,
                                    ;al cărui conţinut este decrementat
                                    ;după fiecare transfer.
LD A, (BC)             0AH          ;Se încarcă conţinutul registrului A,
                                    ;de la adresa conţinută în registrul
                                    ;dublu BC.
```

- *Adresare directă.* Această adresare este folosită dacă în corpul unei instrucţiuni apare adresa efectivă a operandului ce constituie obiectul instrucţiunii. Adresarea directă se foloseşte la instrucţiunile de transfer între registre şi memorie, precum şi la instrucţiunile de salt.

```
Limbaj de asamblare   Cod maşină
LD (1784H), A          328417H      ;Transferă conţinutul acumulatorului
```

```
                                    ;în memorie, la adresa 1784H. Adresarea
                                    ;folosită este implicită pentru
                                    ;determinarea sursei (reg. A) şi directă
                                    ;pentru specificarea destinaţiei (1784H).
OUT (80H), A          D380H         ;Transferă conţinutul acumulatorului
                                    ;la dispozitivul de ieşire
                                    ;de la adresa 80H. Adresa sursei este
                                    ;specificată implicit iar cea a
                                    ;destinaţiei (portul cu adresa 80H)
                                    ;direct.
CALL 7778H            CD7877H       ;Apel de subrutină. Conţinutul registrului
                                    ;PC este salvat în stiva adresată de
                                    ;indicatorul de stivă SP, după care se
                                    ;execută un salt la adresa de început a
                                    ;subrutinei (7778H).
```

- *Adresare relativă.* Această adresare este folosită dacă adresa locaţiei de memorie în care se găseşte operandul se obţine adăugînd o valoare la conţinutul curent al registrului PC.

```
Limbaj de asamblare   Cod maşină
JR +05H                1803H        ;Execută un salt la adresa
                                    ;care se calculează însumînd valoarea
                                    ;curentă a registrului PC cu
                                    ;valoarea (deplasamentul) inclusă în
                                    ;cîmpul instrucţiunii.
```

- *Adresare indexată.* Această adresare este folosită dacă adresa unui operand se obţine adăugînd un deplasament (indice) la un registru de bază (index). Z80 are două registre de 16 biţi, IX şi IY, care pot fi folosite ca registre de bază. Tehnica de adresare indexată este eficientă în cazul în care datele sînt organizate într-un tabel. Considerînd că fiecare element din tabel ocupă un octet şi că adresa de început (adresa de bază) a tabelului se încarcă în registrul index IX sau IY, atunci regăsirea unei date dorite se poate face specificîndu-i doar indicele.

```
Limbaj de asamblare   Cod maşină
LD E, (IX+25H)         DD5E25H      ;Transferă un octet din memorie în
                                    ;registrul E. Adreasa celulei de memorie
                                    ;sursă se obţine însumînd conţinutul
                                    ;registrului de bază IX şi valoarea
                                    ;deplasamentului +25H din cîmpul
                                    ;instrucţiunii.
RES 7, (IY+69H)        FDCB69BEH    ;Şterge (înscrie 0) bitul
                                    ;cel mai semnificativ al octetului din
                                    ;memorie, octet a cărui adresă se obţine
                                    ;însumînd conţinutul registrului de bază
                                    ;(index) IY şi valoarea numerică (indice)
                                    ;69H conţinută în cîmpul instrucţiunii.
                                    ;Instrucţiunea ocupă 4 octeţi,
                                    ;din care trei octeţi sînt rezervaţi
                                    ;pentru cod.
```

2.2 Instrucţiuni de transfer

Majoritatea operaţiilor de tranfer de date se realizează cu ajutorul instrucţiunii LD. Operaţiile permit ca operanzii să fie de 8 sau 16 biţii. Procesorul poate să execute două instrucţiuni pentru interschimbarea datelor din registre: EX şi EXX. Pentru operaţii cu stiva se pot folosi instrucţiunile PUSH şi POP, iar pentru transferul blocurilor de date, instrucţiunile LDx şi LDxR.

Instrucţiunile de transfer pot avea unul, doi sau nici un operand. De asemenea, aceste instrucţiuni mai pot fi clasificate după sensul de transmitere a datelor şi tipul operanzilor:

```
  Tipul transferului              Exemple
registru <- registru          LD A, B     ; LD   HL, BC
registru <- memorie           LD A, (HL)  ; POP  AF
registru <- val. imediata     LD A, 25H   ; LD   HL, 125BH
memorie  <- registru          LD (HL), A  ; PUSH BC
memorie  <- memorie           LDD         ; LDIR
memorie  <- val. imediata     LD (HL), 5BH
```

Modul de stocare a unei date pe 16 biţi în memorie este "little endian" (octetul mai puţin semnificativ este stocat la adresa mai mică).

```
Adresa    Limbaj maşinã   Limbaj de asamblare   Comentarii
                           ORG 1800H             ;directivã
                                                 ;necesarã dacã programul
                                                 ;se transferã pe machetã
 1800      3E88           LD   A, 88H
 1802      011000         LD   BC, 10H          ;contorul operaţiei LDIR
 1805      112018         LD   DE, 1820H        ;adresa destinaţiei
 1808      216600         LD   HL, 0066H        ;adresa sursei
 180B      C5             PUSH BC               ;se salveazã BC în stivã
 180C      EDB0           LDIR
 180E      C1             POP  BC               ;se reface registrul BC
 180F      77             LD   (HL), A
 1810      FF             RST  38H              ;întoarcere la programul
                                                 ;monitor
```

Instrucţiunea RST 38H este necesară la sfîrşitul codului deoarece, după terminarea programului, controlul sistemului este preluat de programul monitor. În acest fel, se pot verifica datele obţinute.

Dacă se vizualizează un registru care conţine o dată pe 16 biţi (adresă) şi se apasă tasta [ADDR], sistemul va afişa conţinutul memoriei de la adresa memorată în tregistru. Reîntoarcerea în programul utilizatorului se va face apăsînd tasta [PC]. Instrucţiunea LDIR, fiind o instrucţiune repetitivă, va fi executată în mai mulţi paşi, astfel încît utilizatorul poate urmări ce se întîmplă la fiecare etapă a instrucţiunii.

2.3 Instrucţiuni aritmetice şi logice

Operaţiile aritmetice pot avea operanzi pe 8 sau 16 biţi. Operaţiile aritmetice cu date de 8 biţi au ca prim operand obligatoriu registrul acumulator (A), iar rezultatul este stocat tot în registrul A. Operaţiile cu date pe 16 biţi au ca surse unul din registrele HL, IX sau IY. Instrucţiunile logice sînt operaţii pe 8 biţi şi au ce registru sursă întotdeauna acumulatorul. Operaţiile aritmetice şi logice afectează indicatorii în funcţie de rezultatele acestora:

- *Carry* este setat în urma operaţiilor cu semn sau fără semn dacă rezultatul este un număr ce nu poate fi reprezentat pe 8 biţi. Indicatorul este setat şi atunci cînd se generează împrumuturi la operaţia de scădere. Acest indicator poate fi folosit drept condiţie la instrucţiunile de salt sau ca element de legătură la adunarea numerelor mari, ce se reprezintă pe 24 sau 32 de biţi.
- *Parity/Overflow* este setat la depăşirea domeniului de reprezentare a numerelor cu semn în complement faţă de 2 (între -128 şi +127) la operaţii aritmetice, sau în cazul în care rezultatul unei operaţii logice are un număr par de biţi 1.
- *Zero* este setat atunci cînd acumulatorul devine zero în urma unei operaţii aritmetice sau logice.
- *Sign* este setat atunci cînd cel mai semnificativ bit al acumulatorului este 1 (reprezentînd faptul că numărul din acumulator este interpretat drept un număr negativ).

```
Adresa   Limbaj maşină   Limbaj de asamblare   Comentarii
                          ORG  1800H
 1800     AF              XOR  A                ;A=0, Z=1
 1801     3D              DEC  A                ;A=0FFH, C=1
 1802     06FF            LD   B, 0FFH
 1804     90              SUB  A                ;A=0, Z=1
 1805     05              DEC  B                ;B=0FEH
 1806     88              ADC  B                ;A=FFH
 1807     0E0F            LD   C, 0FH
 1809     A1              AND  C                ;A=0FH, P/V=0
 180A     06F5            LD   B, 0F5H
 180C     80              ADD  B                ;A=4, C=1
 180D     FF              RST  38H
                          END
```

2.4 Instrucţiuni de salt şi de ciclare

Instrucţiunile de salt se pot clasifica, după adresa la care se face saltul, în salturi:

- *Absolute* - caz în care argumentul instrucţiunii este chiar adresa de memorie a destinaţiei.
- *Relative* - se calculează un deplasament (pozitiv sau negativ) faţă de adresa curentă. Aceste instrucţiuni se folosesc în cazul salturilor mici deoarece ocupă mai puţin spaţiu în memorie (2 octeţi, faţă de 3 cît ocupă un salt absolut).

Acest tip de instrucţiuni se mai pot clasifica şi după felul în care se face saltul:

- *Condiţionate* - se efectuează în funcţie de valorile unor indicatori, dacă aceştia respectă sau nu condiţia de salt.
- *Necondiţionate.*

Uneori, în programe este nevoie ca anumite părţi din cod să fie executate de mai multe ori. Pentru realizarea acestui lucru se stabileşte mai întîi un contor, în care se va încărca numărul de execuţii ale buclei, şi apoi se foloseşte o instrucţiune de salt condiţionat. În programele mai complexe, pot să apară chiar şi bucle imbricate.

În continuare, este prezentat un program care poate fi folosit la calcularea sumei dintre operanzii aflaţi într-un bloc de memorie (1900-190FH). Rezultatul va fi memorat în registrul DE.

```
Adresa   Limbaj     Eticheta   Limbaj de          Comentarii
         maşină                 asamblare
 1800     0E10                   LD  C,10         ;C-contor
 1802     AF                     XOR A            ;A=0
 1803     210019                 LD  HL, 1900H    ;adresa de început a
                                                  ;datelor; HL pointer
 1806     57                     LD  D, A         ;registrul D va memora
                                                  ;transporturile, D=0
 1807     86         XXX:        ADD A, (HL)
 1808     23                     INC HL           ;se obţine următoarea
                                                  ;dată
 1809     3001                   JR  NC, YYY      ;dacă nu s-a generat
                                                  ;transport, se efectuează
                                                  ;un salt relativ la
                                                  ;adresa YYY
 180B     14                     INC D            ;dacă s-a generat
                                                  ;transport, registrul D
                                                  ;se incrementează
 180C     0D         YYY:        DEC C            ;decrementare contor
 180D     20F8                   JR  NZ, XXX      ;repetă pînă se adună
                                                  ;toate datele
 180F     5F                     LD  E, A         ;se încarcă A în E,
                                                  ;rezultatul este în DE
 1810     FF                     RST 38H          ;întoarcere în programul
                                                  ;monitor
```

2.5 Stiva

În cadrul proiectării programelor, stiva este o secţiune a memoriei care un singur port pentru intrări şi ieşiri. Datele sînt înscrise şi citite în stivă cu ajutorul acestui port. Prima dată care se salvează în stivă se plasează la baza ei (*bottom of stack*). Data înscrisă cel mai recent în stivă este plasată în vîrful acesteia (*top of stack*). Din aceste motive, stiva este considerată o memorie LIFO (Last-In First-Out). Pentru a defini o stivă la începutul zonei de memorie

RAM, cea mai mare adresă este incrementată cu 1 şi după aceea este stocată în registrul stivă (SP) al microprocesorului. Următorul program ilustrează operaţiile cu stiva:

```
Limbaj de asamblare       Comentarii
 LD  SP, 1FAFH             ;secţiunea RAM cu adrese mai mici sau egale cu
                           ;1FAFH este consideratã stivă
 DEC SP                    ;SP este decrementat, deci adresa de bază a
                           ;stivei este 1FAEH
 LD  (SP), H               ;încarcă conţinutul registrului H în memorie
                           ;(RAM), la adresa 1FAEH
 DEC SP                    ;decrementare SP
 LD  (SP), L               ;plaseaza conţinutul registrului L în vîrful
                           ;stivei
 DEC SP
 LD  (SP), A               ;registrul A este stocat în vîrful stivei
 DEC SP
 LD  (SP), F               ;plasează conţinutul registrului F în vîrful
                           ;stivei
 ...
 LD  C, (SP)               ;extrage un octet din vîrful stivei, acesta este
                           ;stocat în registrul C
 INC SP                    ;SP incrementat cu 1. SP este deplasat spre
                           ;vîrful stivei
 LD  B, (SP)               ;extrage un octet din vîrful stivei
 INC SP
 LD  E, (SP)               ;încarcă în E octetul din vîrful stivei
 INC SP
 LD  D,(SP)                ;încarcă în D octetul din vîrful stivei.
                           ;Acesta dată a fost prima salvată în stivă
 INC SP                    ;SP are valoarea iniţială
```

În cadrul operaţiilor cu stiva, din programului anterior prezentat, datele pot fi stocate în memoria RAM folosind SP ca şi pointer. SP este decrementat cu 1 ori de cîte ori un octet este salvat în stivă, deci dimensiunea stivei creşte. Similar, SP este incrementat cu 1 ori de cîte ori un octet este citit din stivă, deci dimensiunea stivei scade. Stiva poate fi folosită pentru a stoca temporar adrese sau date pe 16 biţi. Microprocesorul Z80 dispune de instrucţiuni care permit salvarea/recuperarea regiştrilor dubli în/din stivă (*PUSH/POP*). În timpul acestor operaţii SP este decrementat/incrementat cu 2. Următorul program este echivalent cu cel prezentat anterior.

```
Limbaj de asamblare       Comentarii
 LD   SP, 1FAFH            ;identică cu prima instrucţiune
 PUSH HL                   ;identică cu instrucţiunile 2, 3, 4, 5
 PUSH AF                   ;identică cu instrucţiunile 6, 7, 8, 9
 POP  BC                   ;identică cu instrucţiunile 10, 11, 12, 13
 POP  DE                   ;identică cu instrucţiunile 14, 15, 16, 17
```

În cadrul unui program, este foarte important ca numărul instrucţiunilor de tip *PUSH* să fie egal cu cel al instrucţiunilor de tip *POP*.

2.6 Subrutine

Programele de tip aritmetic (adunări, scăderi, înmulţiri sau împărţiri), de control a tastaturii şi a afişajului sau altele, sînt des folosite ca părţi ale unor aplicaţii de dimensiuni mari. Pentru a economisi memorie şi a reduce posibilitatea de apariţie a erorilor, subrutinele sînt des folosite în cadrul programelor. Pentru manipularea subrutinelor se folosesc instrucţiunile *CALL* şi *RET*. Subrutinele pot fi executate necondiţionat, în funcţie de anumite condiţii sau indicatori.

În cazul apelului unei subrutine dintr-un program principal (*CALL*), se execută operaţiile prezentate în exemplul următor:

```
CALL   1A38H            ;apelul subrutinei de la adresa 1A38H
```

Apelul de procedură este echivalent cu:

```
PUSH   PC               ;salvează contorul program PC în stivă
JP     1A38H            ;salt la adresa 1A38H şi continuă execuţia
```

Spre deosebire de instrucţiunle de salt, după executarea unei subrutine, controlul programului este transferat instrucţiunii care urmează după apelul subrutinei. Instrucţiunea *RET* nu are nevoie de nici un operand, este codificată pe un octet şi are acelaşi efect ca şi instrucţiunea POP PC.

```
RET                     ;reîntoarcere la programul principal şi continuă
                        ;execuţia
```

Instrucţiunea de revenire din procedură este echivalentă cu:

```
POP PC                  ;se reface conţinutul PC, din stivă, după care
                        ;programul se execută conform valorii PC-ului
```

În figura 2.1 este prezentată forma generală a unui program care conţine apeluri de subrutine.

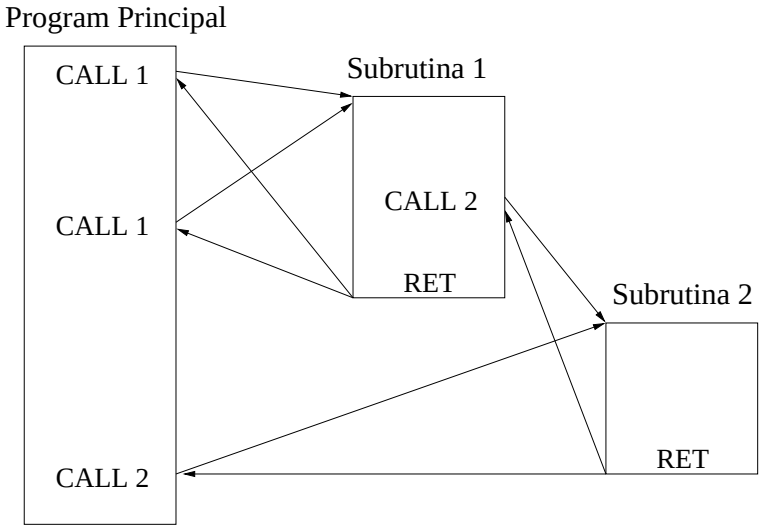


Figura 2.1: Apeluri de subrutine.

În cazul unui apel de subrutină din programul principal, trebuie considerate următoarele observaţii:

- Registrele care nu trebuie să fie afectate de către subrutină vor fi salvate în stivă de către programul apelant, înainte de apelul subrutinei.
- Modalitatea prin care rezultatele, obţinute în urma execuţiei subrutinei, se transferă în programul principal este stabilită de programator.

2.7 Experimente

I. Scrieţi un program care să încarce în registrele microprocesorului următoarele valori, folosind instrucţiuni de transfer pe 16 biţi: B=12, C=34, D=56, E=78, H=9, L=A. Asamblaţi-l şi apoi verificaţi-l pe macheta MPF1-B.

II. Transferaţi pe machetă codul executabil de la exemplul secţiunii 2.2. Executaţi programul. Verificaţi dacă s-au copiat cei 16 octeţi de la adresa 0066H la adresa 1820H. Rulaţi programul pas cu pas şi urmăriţi acţiunile fiecărei instrucţiuni. Verificaţi conţinutul registrelor afectate înainte şi după instrucţiuni.

III. Scrieţi un program în limbaj de asamblare care să şteargă conţinutul memoriei între adresele 1850H - 186FH. Rulaţi-l pe machetă şi verificaţi rezultatele.

IV. Scrieţi un program în limbaj de asamblare care setează conţinutul memoriei între adresele 1840H - 184FH cu următoarele valori: 0, 1, ..., F. Asamblaţi-l şi apoi verificaţi-l pe macheta MPF1-B.

V. Următorul program este folosit pentru a aduna un operand de 16 biţi aflat în memorie la adresele 1A00H - 1A01H cu conţinutul registrului dublu DE. Rezultatul va fi stocat în registrul dublu HL.

```
ORG 1800H
LD  A, (1A00H)
ADD A, E
LD  L, A
LD  A, (1A01H)
ADC A, D
LD  H, A
RST 38H
```

Rulaţi programul pe machetă şi urmăriţi rezultatele. Modificaţi programul de mai sus pentru o operaţie de scădere.

VI. Următorele linii de cod pot fi folosite pentru a împărţi 256 de octeţi din memorie în 16 blocuri. Adresa de început este 1900H.

```
       LD    HL, 19FFH
       LD    C,  0FH
LOOP2: LD    B,  10H
LOOP1: LD    (HL), C
       DEC   HL
       DJNZ  LOOP1
       DEC   C
```

```
JP    NZ, LOOP2
RST   38H
```

Setaţi valoarea fiecărui bloc de date în modul următor: 1H pentru blocul 1, 2H pentru blocul 2, ... , 0FH pentru blocul 16.

Lucrarea 3

Afişajul şi tastatura MPF1-B

3.1 Afişajul

Macheta *Microprofessor MPF1-B* conţine un display format din şase afişoare cu 7 segmente (plus punctul zecimal). Figura 3.1 prezintă asocierea dintre cele 7 segmente şi literele cu care sînt denumite acestea. Fiecare afişor conţine 8 LED-uri (7 asociate segmentelor şi unul asociat punctului zecimal). Cele 8 LED-uri sînt conectate cu anodul în comun.

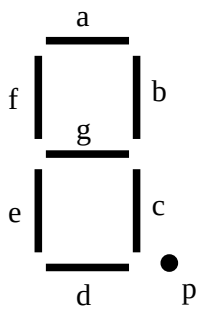


Figura 3.1: Denumirea celor 7 segmente ale unui afişor.

Comandarea concurentă a celor şase afişaje ar necesita un număr de 48 semnale, determinat astfel: (6 afişaje) × (8 semnale de date) = 48 semnale.

Comandarea secvenţială a celor şase afişaje ar necesita un număr de 14 semnale, determinat astfel: (6 semnale de selecţie a afişajului) + (8 semnale de date) = 14 semnale.

Ţinînd cont de inerţia ochiului uman, comandarea secvenţială a afişajelor cu o frecvenţă mare (cel puţin de 40 de ori pe secundă), poate crea impresia că afişajele sînt aprinse simultan. Liniile de date pentru segmente sînt notate cu Sa, Sb, Sc, Sd, Se, Sf, Sg şi Sp. Liniile de selecţie a afişajului comandat la un moment dat sînt notate cu D0, D1, D2, D3, D4, D5.

Pentru a comanda afişajul machetei sînt folosite porturile A şi B ale circuitului port paralel I8255, aşa cum se este prezentat în figura 3.3. Portul B (liniile PB0 - PB7) este folosit ca port de ieşire pentru liniile de date ale afişoarelor (Sa, Sb, Sc, Sd, Se, Sf, Sg). Portul C (liniile PC0 - PC5) este folosit ca port de ieşire pentru semnalele de comandă (D0, D1, D2, D3, D4, D5). Prin portul C se selectează afişorul care se luminează. Prin portul B se specifică segmentele luminate din cadrul afişajului selectat prin portul C. Toate segmentele sînt controlate de semnale active în $'1'$.

3.2 Tastatura

Deoarece reacţia unui calculator este mult mai rapidă decît cea a utilizatorului, tastatura trebuie scanată repetat pînă în momentul în care se detectează o tastă apăsată. O tastă oscilează pentru un timp scurt în momentul în care este apăsată sau eliberată. În figura 3.2 este prezentată diagrama răspunsului, în timp, în cazul operaţiilor de apăsare, respectiv de eliberare, a unei taste.

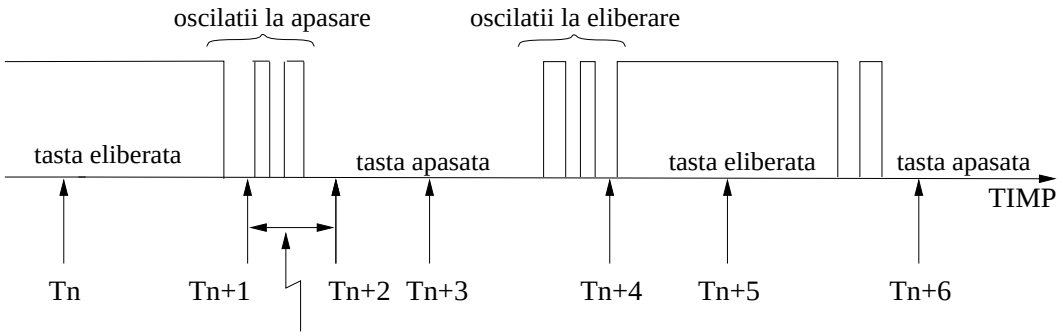


Figura 3.2: Răspuns în timp la scanarea tastaturii.

Datorită oscilaţiilor mecanice, dacă viteza de scanare a tastaturii este prea mare,apăsarea unei taste poate fi interpretată ca două sau mai multe apăsări succesive. Pentru a evita aceast lucru, perioada de scanare trebuie să fie mai mare decît perioada oscilaţiilor. Perioada oscilaţiilor nu este mai mare de 10 ms. Perioada de scanare trebuie aleasă între 10 ms şi 50 ms. În figura 3.2, săgeţile indică momentele de timp cînd este examinată tastatura. La momentul T_{n+2}, programul monitor detectează o tastă apăsată şi identifică codul acesteia. La momentul T_{n+3} tasta este, din nou, găsită apăsată. Deoarece tasta a fost detectată apăsată în timpul precedentei scanări, monitorul nu consideră această acţiune ca fiind o nouă apăsare (tasta nu a fost eliberată în acest interval). Doar dacă tasta este detectată ca fiind eliberată la momentul T_{n+4} sau T_{n+5}, atunci se consideră o nouă apăsare a acesteia. Un program care obţine date de la tastatură în acest mod, este lipsit de erori. Practic, nu contează intervalul de timp cît o tastă este apăsată.

Structura tastaturii, prezentată în figura 3.3, constă dintr-un număr de linii dispuse în formă matricială. În fiecare nod de intersecţie este poziţionată o tastă. Cele 6 linii orizontale şi cele 6 linii verticale formează, 36 de puncte de contact pentru tastatură. În momentul în care este acţionată o tastă, se va crea un contact electric între o linie şi o coloană a matricei. Cele şase linii orizontale (PA0 - PA5), sînt conectate la portul de intrare A al circuitului port paralel I8255. Dacă nici o tastă nu este apăsată, atunci cele 6 linii sînt conectate la tensiunea de alimentare (+5V) prin 6 rezistoare. Coloanele matricii sînt conectate la portul de ieşire C (PC0 - PC5), care la rîndul lui este conectat şi la afişaj.

Microprocesorul selectează cea mai din drepta coloană cu ajutorul liniei PC0. Tensiunile celor 6 linii ale matricei sînt evaluate secvenţial. La începutul procesului de scanare a tastaturii, un numărător este setat la zero, portul C va avea valoarea "11000001", deci PC5 - PC0 se vor găsi în starea logică "000001". În timpul cît se scanează tastatura, PC6 şi PC7 trebuie să fie în stare 1, deoarece PC6 este conectat la semnalul BREAK, iar PC7 la ieşirea de difuzor. Tensiunile liniilor tastaturii se citesc succesiv. Dacă o tastă este apăsată (pe linia respectivă se detectează o tensiune nulă), ea poate fi identificată cu ajutorul poziţiei liniei în cadrul portului. Dacă nici o tastă din prima coloană nu este apăsată, atunci microprocesorul va forţa pe portul C următoarea valoare a numărătorului (11000010), selectînd a doua coloană.

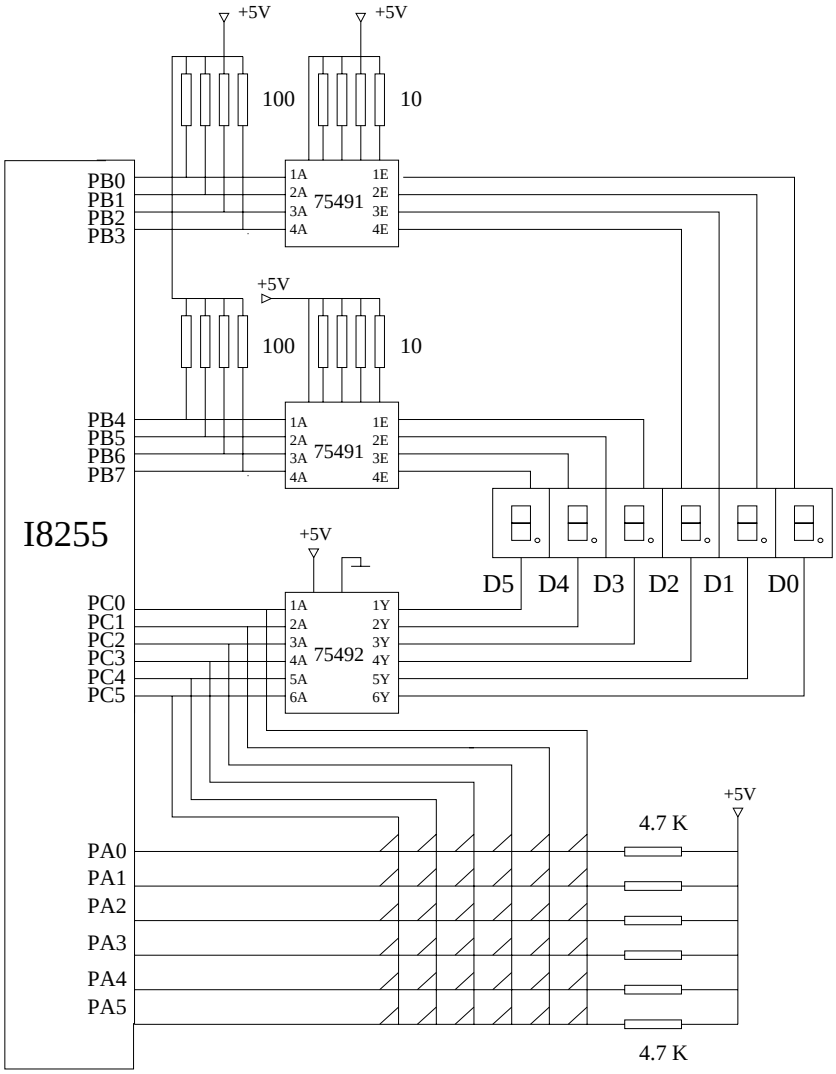


Figura 3.3: Structura display-ului şi a tastaturii machetei MPF1-B.

Procesul de scanare al tastaturii se desfăşoară succesiv din partea dreaptă spre stînga, şi de sus în jos. Fiecare tastă este codată: de cîte ori o tastă examinată este găsită neapăsată, se incrementează valoarea numărătorului. În momentul în care se identifică o tastă apăsată, codul poziţiei acelei taste este chiar valoarea numărătorului.

În tabelele 3.1 şi 3.2 sînt descrise codul poziţiei şi codul intern al fiecărei taste.

3.3 Subrutinele programului monitor

Programul monitor conţine 8 subrutine ce pot fi apelate şi din programele utilizatorului.

Adresele, descrierea şi parametrii subrutinelor SCAN1, SCAN, HEX7, HEX7SG, RAMCHK şi TONE sînt prezentate în tabelele 3.3, 3.4, 3.5, 3.6, 3.7, 3.8.

1E SBR	18 CBR	12 ′0′	0C ′1′	06 ′2′	00 ′3′
1F ′ − ′	19 PC	13 ′4′	0D ′5′	07 ′6′	01 ′7′
20 DATA	1A REG	14 ′8′	0E ′9′	0B ′A′	02 ′B′
21 ′ + ′	1B ADDR	15 ′C′	0F ′E′	09 ′E′	03 ′F′
22 INS	1C DEL	16 GO	10 STEP	0A	04
23 MOVE	1D RELA	17 TPWR	11 TPRD	0B	05

Tabelul 3.1: Codul poziţiei tastelor.

15 SBR	1A CBR	00 ′0′	01 ′1′	02 ′2′	03 ′3′
11 ′ − ′	18 PC	04 ′4′	05 ′5′	06 ′6′	07 ′7′
14 DATA	1B REG	08 ′8′	09 ′9′	0A ′A′	0B ′B′
10 ′ + ′	19 ADDR	0C ′C′	0D ′E′	0E ′E′	0F ′F′
16 INS	17 DEL	12 GO	13 STEP	22	20
1C MOVE	1D RELA	1E TPWR	1F TPRD	23	21

Tabelul 3.2: Codul intern al tastelor.

3.4 Exemple

Exemplul 1:

Afişarea mesajului `HELP US` pînă cînd se apasă tasta *STEP*.

```
        ORG  1800H
        LD   IX, HELP
DISP:   CALL SCAN
        CP   13H                ;codul intern al tastei STEP
        JR   NZ, DISP
        HALT

        ORG  2000H
HELP:   DEFB 0AEH               ; "S"
        DEFB 0B5H               ; "U"
```

SCAN1	
Adresă	0624H
Funcţie	Scanează tastatura şi afişajul timp de 1 ciclu, de la dreapta la stînga. Timpul de execuţie este de 9.97ms.
Intrare	IX este un pointer la buffer-ul de afişare.
Ieşire	(1) Indicatorul carry este setat dacă nu s-a apăsat nici o tastă; (2) Dacă a fost apăsată o tastă, indicatorul carry este resetat şi codul poziţiei tastei este memorat în registrul A.
Registre afectate	AF, AF', BC, BC', DE'.
Observaţii	(1) Sînt necesari 6 octeţi pentru memorarea celor 6 pattern-uri; (2) IX este un pointer spre cuvîntul ce corespunde rîndului din dreapta. IX+5 indică cuvîntul ce corespunde rîndului din stînga.

Tabelul 3.3: Subrutina SCAN1.

SCAN	
Adresă	05FEH
Funcţie	Similară cu cea a rutinei SCAN1 cu 2 excepţii: (1) SCAN1 scanează un ciclu, pe cînd SCAN scanează pînă se apasă o tastă; (2) SCAN1 întoarce poziţia tastei apăsate, în timp ce SCAN întoarce codul tastei apăsate.
Intrare	IX este un pointer la buffer-ul de afişare.
Ieşire	Registrul A conţine codul intern al tastei apăsate.
Registre afectate	AF, AF', B, BC', DE', HL.

Tabelul 3.4: Subrutina SCAN.

```
        DEFB 01FH           ; "P"
        DEFB 085H           ; "L"
        DEFB 08FH           ; "E"
        DEFB 037H           ; "H"

SCAN    EQU  05FEH
        END
```

Exemplul 2:

Afişarea cu intermitenţă a mesajului `HELP US`, folosind rutina SCAN1. Fiecare pattern este afişat timp de 500 ms prin executarea rutinei SCAN de 50 de ori. Valoarea registrului B determină frecvenţa de afişare.

```
        ORG  1800H
        LD   HL, BLANK
        PUSH HL
```

HEX7	
Adresă	0689H
Funcţie	Converteşte o cifră în baza 16 în formatul de afişare cu 7 segmente.
Intrare	Cei mai puţin semnificativi 4 biţi ai registrului A conţin cifra, exprimată în baza 16.
Ieşire	Rezultatul este memorat în registrul A.
Registre afectate	AF.

Tabelul 3.5: Subrutina HEX7.

HEX7SG	
Adresă	0678H
Funcţie	Converteşte două cifre din baza 16 în formatul de afişare cu 7 segmente.
Intrare	Cei mai puţin semnificativi 4 biţi ai registrului A conţin prima cifră, iar cei mai semnificativi 4 biţi ai registrului A conţin a doua cifră.
Ieşire	Primul pattern de afişat este memorat la adresa din registrul HL, iar al doilea este memorat la adresa următoare (conţinutul registrului HL, plus 1).
Registre afectate	AF, HL.

Tabelul 3.6: Subrutina HEX7SG.

```
         LD   IX, HELP
LOOP:    EX   (SP), IX
         LD   B, 50
HALFSEC: CALL SCAN1
         DJNZ HALFSEC
         JR   LOOP

         ORG  1820H
HELP:    DEFB 0AEH              ; "S"
         DEFB 0B5H              ; "U"
         DEFB 01FH              ; "P"
         DEFB 085H              ; "L"
         DEFB 08FH              ; "E"
         DEFB 037H              ; "H"
BLANK:   DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0
```

RAMCHK	
Adresă	05F6H
Funcţie	Verifică dacă o anumită adresă este în RAM.
Intrare	Adresa care trebuie memorată este stocată în HL.
Ieşire	Dacă adresa este în RAM, indicatorul de zero este setat, altfel este resetat.
Registre afectate	AF.

Tabelul 3.7: Subrutina RAMCHK.

TONE	
Adresă	05E4H
Funcţie	Generează un sunet.
Intrare	Registrul C controlează frecvenţa sunetului. Perioada este aproximativ egală cu $(44 + C \cdot 13) \cdot 2 \cdot 0.56\mu s$, iar frecvenţa este $200/(10 + 3 \cdot C)kHz$.
Registre afectate	AF.

Tabelul 3.8: Subrutina TONE.

```
SCAN1    EQU  0624H
         END
```

Exemplul 3:

Afişarea codului intern al tastei apăsate.

```
         ORG  1800H
         LD   IX, OUTBUF
LOOP:    CALL SCAN
         LD   HL, OUTBUF
         CALL HEX7SEG
         JR   LOOP

         ORG  1900H
OUTBUF:  DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0
         DEFB 0

SCAN     EQU  05FEH
HEX7SG   EQU  0678H
         END
```

Pentru a afişa codul poziţiei tastei apăsate, programul trebuie modificat după cum urmează:

```
         ORG  1800H
         LD   IX, OUTBUF
LOOP:    CALL SCAN1
         JR   C, LOOP
         LD   HL, OUTBUF
         CALL HEX7SEG
         JR   LOOP
```

Exemplul 4:

Se convertesc trei octeţi din memorie în formatul şapte segmente. Rezultatul este stocat în memorie la adresele 1903H - 1908H, după care este afişat.

```
         ORG  1800H
         LD   DE, BYTE0
         LD   HL, OUTBUF
         LD   B, 3
LOOP:    LD   A, (DE)
         CALL HEX7SEG
         INC  DE
         DJNZ LOOP

         LD   IX, OUTBUF
         CALL SCAN
         HALT

         ORG  1900H
BYTE0:   DEFB 10H
         DEFB 32H
         DEFB 45H

OUTBUF:  DEFS 6

SCAN     EQU  05FEH
HEX7SEG  EQU  0678H
         END
```

Cei trei octeţi de date sînt stocaţi la adresele 1900H - 1902H.

3.5 Experimente

I. Transferaţi codul executabil de la *Exemplul 1* pe machetă, după care executaţi programul. Încărcaţi în memorie la adresa 1808H valoarea 1AH. Apăsînd tasta *CBR*, mesajul nu va mai apărea pe afişaj. De ce? Setaţi conţinutul memoriei între adresele 1820H - 1822H cu valorile 3FH, BDH, 85H. Ce se va afişa pe display? Scrieţi un program care să afişeze `SYS-SP` pînă cînd se apasă tasta *PC*.

II. Transferaţi codul executabil de la *Exemplul 2* pe machetă, după care executaţi programul. Setaţi conţinutul locaţiei de memorie 180BH cu valoarea 01. Ce se va afişa pe display? Dar pentru valoarea 05?

III. Pentru *Exemplul 3* setaţi conţinutul zonei de memorie 1900H - 1905H la valoarea FFH. Ce se va afişa?

IV. Modificaţi programul de la *Exemplul 4*, astfel încît să se afişeze secvenţa `333446`.

Lucrarea 4

Aplicaţii cu circuitul Z80-PIO

Această lucrare prezintă modul de lucru al circuitului port paralel Z80-PIO, comanda cu acest circuit a unor dispozitive aflate pe placa de aplicaţii, precum şi cîteva programe care să exemplifice această problematică.

4.1 Interfaţa paralelă programabilă Z80-PIO

4.1.1 Arhitectura internă a circuitului Z80-PIO

Z80-PIO (Parallel Input/Output - engl.) este o interfaţă paralelă programabilă prevăzută cu o unitate de comandă şi două porturi paralele de 8 biţi de date şi 2 semnale de conversaţie (Ready şi Strob) cu ajutorul cărora se controlează transferul de date. Cele două porturi furnizează o interfaţă compatibilă TTL între procesor şi dispozitivele periferice. Porturile, denumite A şi B, pot fi programate ca porturi de intrare sau porturi de ieşire, la nivel de octet sau de bit. Portul A poate fi programat şi pentru a lucra bidirecţional. În funcţie de indicatorii de stare ai echipamentelor periferice, se pot genera întreruperi programabile. Figura 4.1 prezintă arhitectura internă a circuitului Z80-PIO. Simbolul bloc asociat acestui circuit este prezentat în figura 4.2.

Pinii circuitului au următoarea semnificaţie:

Semnale generale:

CLK (System Clock) - Semnal de ceas comun tuturor circuitelor din sistemul cu microprocesor Z80.

/RESET (Reset) - Semnal de iniţializare.

Magistrala de date:

D0-D7 (System Data Bus) - Magistrală bidirecţionlă conectată la magistrala de date a procesorului.

Semnalele de control (primite de la procesor):

SEL.PB/NPA (Port B or A Select) [intrare, High=B, Low=A] - Semnal de selecţie a portului accesat în timpul unui transfer de date între procesor şi PIO. Pentru această selecţie se foloseşte bitul A0 al magistralei de adrese a procesorului.

SEL.CTRL/NDATA (Control or Data Select) [intrare, High=C, Low=D] - Semnal care de-

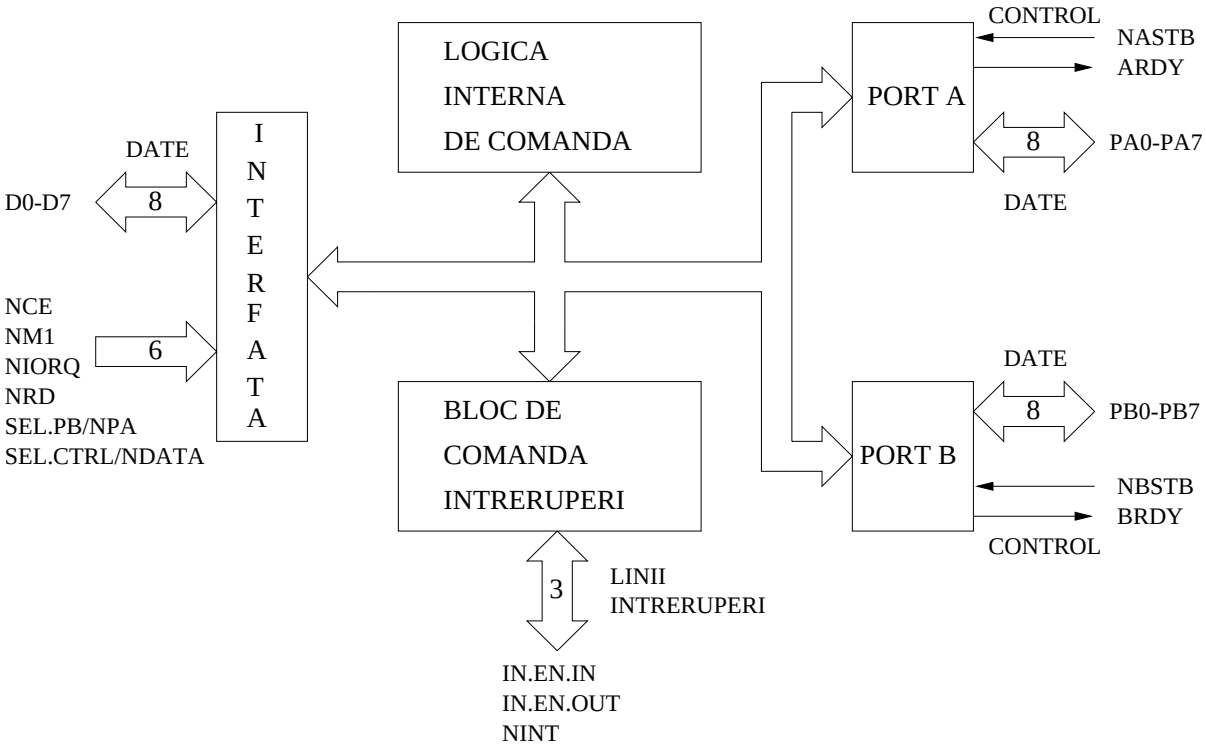


Figura 4.1: Arhitectura circuitului Z80-PIO.

fineşte tipul de date care se transferă între procesor şi PIO. Dacă semnalul este în starea 1, cuvîntul este interpretat ca o comandă, altfel ca o dată. Pentru această selecţie se foloseşte bitul A1 al magistralei de adrese a procesorului.

/CE (Chip Enable) [intrare, activă în stare Low] - Semnal de validare a circuitului PIO. Se obţine prin decodificarea magistralei de adrese.

/M1 (Machine Cycle 1) [intrare de la procesor, activă în stare Low] - Semnal de la procesor utilizat ca impuls de sincronizare pentru a controla mai multe operaţii interne ale circuitului PIO. Cînd semnalele */M1* şi */RD* sînt active simultan, procesorul încarcă o dată din memorie. Semnalul */M1* mai are încă două funcţii în cadrul circuitului PIO: sincronizează logica de întreruperi din PIO şi iniţializează circuitul PIO în momentul apariţiei semnalului */M1*, fără ca unul din semnalele */RD* sau */IORQ* să fie active.

/IORQ (Input/Output Request) [intrare de la procesor, activ în stare Low] - Semnal de la procesor utilizat împreună cu *SEL.PB/NPA*, *SEL.CTRL/NDATA*, */CE* şi */RD* pentru a transfera comenzi şi date între procesor şi circuitul PIO. Cînd */CE*, */RD* şi */IORQ* sînt active portul adresat de *SEL.PB/NPA* scrie date în procesor (operaţie de citire). Cînd */RD* nu este activ portul adresat de *SEL.PB/NPA* este înscris cu date sau informaţii de control de la procesor, în funcţie de starea semnalului *SEL.CTRL/NDATA*.

Dacă */IORQ* şi */M1* sînt simultan active procesorul anunţă acceptarea unei întreruperi. Portul care a cerut întreruperea pune în mod automat vectorul lui de întrerupere pe magistrala de date a procesorului, în cazul în care dispozitivul periferic care a cerut întreruperea are prioritatea cea mai mare.

/RD (Read Cycle Status) [intrare de la procesor, activ în stare Low] - Dacă semnalul */RD* este activ sau o operaţie de intrare/ieşire este în curs de desfăşurare,*/RD* este folosit împreună cu semnalele SEL.PB/NPA, SEL.CTRL/NDATA, /CE şi /IORQ pentru a se transfera date de la

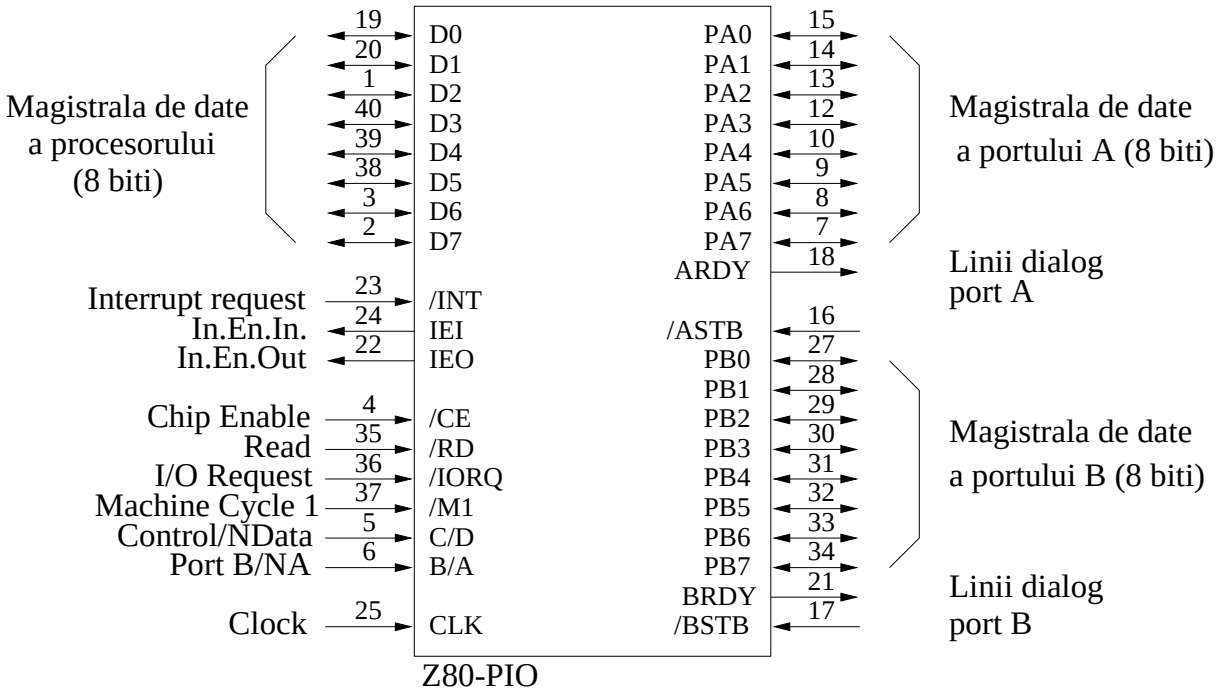


Figura 4.2: Simbolul bloc al circuitului Z80-PIO.

circuitul PIO spre procesor.

Semnalele de întrerupere:

IN.EN.IN (IEI) (Interrupt Enable In) [intrare, activ în stare High] - Semnal de validare a întreruperilor. Este folosit pentru a forma un lanţ de priorităţi la cererile de întrerupere, cînd se utilizează mai multe dispozitive periferice comandate prin întreruperi. Starea logică 1 a aceastei linii semnifică faptul că nici un alt dispozitiv cu prioritate mai mare nu este servit de procesor printr-o rutină de tratare a întreruperii.

IN.EN.OUT (IEO) (Interrupt Enable Out) [ieşire, activ în stare High] - Semnal de validare a întreruperilor. Este cel de-al doilea semnal necesar pentru a forma lanţul de priorităţi la întreruperi. Starea logică a acestui semnal este 1 numai dacă şi IEI este în 1 logic şi dacă procesorul nu deserveşte o întrerupere de la acest circuit PIO. Altfel acest semnal blochează cererile de întrerupere pentru dispozitivele mai puţin prioritare, în timp ce un dispozitiv cu prioritate mai mare este deservit de procesor printr-o rutină specifică.

/INT (Interrupt Request) [ieşire, activ în stare Low] - Cerere de întrerupere adresată procesorului.

Semnalele porturilor:

PA0-PA7 (Port A Bus) [intrări/ieşiri, active în stare High, 3-state] - Magistrală bidirecţională de date. Pe această magistrală se realizează transferul de date între portul A al PIO şi dispozitivul periferic.

/ASTB (Port A Strobe Pulse From Peripheral Device) [intrare, activ în stare Low] - Semnificaţia semnalului depinde de modul de funcţionare ales pentru portul A după cum urmează:

Mod de ieşire: frontul crescător al semnalului emis de către dispozitivul periferic semnalează că acesta a primit data furnizată de circuitul PIO.

Mod de intrare: pulsul emis de către dispozitivul periferic semnalează că acesta este gata să scrie date de la portul A. Datele sînt încărcate în PIO numai cînd acest semnal este activ.

Mod bidirecţional: cînd semnalul este activ datele din registrul de ieşire al portului A sînt transferate pe magistrala de date a portului A (bidirecţională). Frontul crescător al semnalului emis de către dispozitivul periferic semnifică faptul că acesta a primit data furnizată de circuitul PIO.

Mod bit: semnalul este dezactivat intern (întrucît nu este necesar).

ARDY (Register A Ready) [ieşire, activ în stare High] - Semnificaţia semnalului depinde de modul de funcţionare selectat după cum urmează:

Mod de ieşire: semnalul activ indică faptul că registrul de ieşire al portului A a fost încărcat şi datele sînt valide pentru citire.

Mod de intrare: semnalul este activ cînd registrul de intrare al portului A este gol şi poate să preia datele de la dispozitivul periferic.

Mod bidirecţional: semnalul este activ cînd datele pentru dispozitivul periferic sînt disponibile în registrul de ieşire al portului A. În acest mod datele nu sînt puse pe magistrala de date a portului A numai cînd semnalul /ASTB este activ.

Mod bit: semnalul este dezactivat intern (întrucît nu este necesar).

PB0-PB7 (Port B Bus) [intrări/ieşiri, active în stare High, 3-state] - Magistrală bidirecţională de date. Pe această magistrală se realizează transferul de date între portul B al PIO şi dispozitivul periferic. Portul B poate furniza pe fiecare linie 1.5 mA la 1.5V pentru a comanda tranzistoare montate în conexiune Darlington.

em /BSTB (Port A Strobe Pulse From Peripheral Device) (intrare, activ în stareLow) - Similar cu semnalul */ASTB*, cu excepţia faptului că în modul bidirecţional semnalul încarcă datele de la dispozitivul periferic în registrul de intrare al portului A.

BRDY (Register B Ready) [ieşire, activ în stare High] - Similar cu semnalul *ARDY*, cu excepţia faptului că în mod bidirecţional semnalul este în 1 logic cînd registrul de intrare al portului A este gol şi poate să preia datele de la dispozitivul periferic.

4.1.2 Modurile de lucru ale circuitului Z80-PIO

Modul 0, de ieşire Ambele porturi (A sau B) pot fi programate în acest mod. Un ciclu de ieşire este întotdeauna pornit de execuţia unei instrucţiuni de ieşire de către procesor. La semnalul */WR* furnizat de către procesor datele de pe magistrala de date sînt înscrise în registrul de ieşire al portului PIO selectat. Impulsul de scriere poziţionează indicatorul *READY* după frontul descrescător al semnalului de ceas, indicînd astfel disponibilitatea informaţiei pentru dispozitivul periferic. Linia *READY* rămîne activă pînă cînd PIO recepţionează de la dispozitivul periferic semnalul de *STROB*, care semnifică faptul că perifericul a preluat infomarţia. Frontul crescător al semnalului de STROB generează o întrerupere */INT*, dacă bistabilul de activare a întreruperilor a fost setat şi dacă dispozitivul periferic are prioritatea cea mai mare.

Modul 1, de intrare Ambele porturi (A sau B) pot fi programate în acest mod, fiecare avînd un registru de intrare adresabil de către procesor. Data de la echipamentul periferic este înscrisă în registrul de intrare al porturilor PIO pe frontul descrescător al semnalului de *STROB*. Frontul crescător al aceluiaşi semnal activează */INT*, dacă bistabilul de activare a întreruperilor a fost setat şi dacă echipamentul periferic respectiv are prioritatea cea mai mare. Următorul front descrescător (activ) al semnalului de ceas inactivează semnalul *READY*, prin care echipamentul periferic este anunţat că portul de intrare conţine o informaţie care nu a fost preluată de către procesor, şi deci nu mai poate fi încărcat cu altă informaţie pînă la citirea celei existente de către procesor. După frontul crescător al semnalului */RD* de la procesor, pe următorul front descrescător al semnalului de ceas, se reactivează semnalul *READY*.

Modul 2, bidirecţional Acest mod reprezintă o combinaţie a modurilor 0 şi 1. Portul A va fi port bidirecţional. Liniile de coversaţie ale portului A se folosesc drept linii de dialog pentru ieşire, iar liniile de conversaţie ale portului B se folosesc drept linii de dialog pentru intrare. Portul B va fi programat, în acest caz, în modul bit (care nu necesită linii de dialog). Dacă apare o întrerupere va fi folosit vectorul de întrerupere al portului A în cazul unei operaţii de ieşire sau cel al portului B în cazul unei operaţii de intrare. Datele de ieşire sînt disponibile perifericului numai cînd semnalul */ASTB* este în 0 logic.

Modul 3, bit Ambele porturi pot fi programate în acest mod. Nu se folosesc semnale de dialog. O operaţie normală de scriere poate avea loc în orice moment. Un semnal de întrerupere se generează dacă starea unei intrări sau starea tuturor intrărilor se modifică. Condiţiile de generare a întreruperii sînt definite în timpul programării circuitului. Nivelul activ poate fi ales 1 sau 0 logic, iar condiţia logică este fie pentru o intrare activă (SAU logic), fie pentru toate intrările active (SI logic). Dacă portul A este programat în mod bidirecţional, portul B nu mai are un semnal de întrerupere şi pentru acest motiv va trebui să fie interogat. Acest mod se foloseşte pentru aplicaţiile în care se generează semnale de comandă sau se monitorizează stări.

4.1.3 Structura unui port

Portul are un registru de intrare şi unul de ieşire, în felul acesta putînd funcţiona în orice mod. Conţinutul acestor registre se modifică numai atunci cînd sînt înscrise, în rest ele păstrînd datele scrise în ele. Portul mai conţine un registru de comandă a modului de lucru (astfel fiecare port se poate programa independent de celălalt - cu excepţia modului 2 de lucru), logică de comandă a registrului mască (utilizat în modul 3 de lucru) şi logică de comandă a liniilor de dialog, fiind astfel capabil să controleze un sistem de întreruperi ierarhizate. În figura 4.3 este prezentată structura unui port al circuitului PIO.

4.1.4 Blocul de comandă a întreruperilor

Blocul de comandă a întreruperilor se ocupă de întregul protocol de întreruperi spre procesor. Poziţia fizică a unui dispozitiv într-un lanţ de priorităţi determină prioritatea lui. Fiecare circuit Z80-PIO are două semnale (*IEI* şi *IEO*) pentru a forma un lanţ de priorităţi, aşa cum se prezintă în figura 5.4. Dispozitivul care este cel mai apropiat de procesor are prioritatea cea mai mare. În cadrul unui circuit PIO, întreruperile portului A au prioritate mai mare decît cele ale portului B. În modurile de intrare, ieşire şi bidirecţional o cerere de întrerupere se poate genera

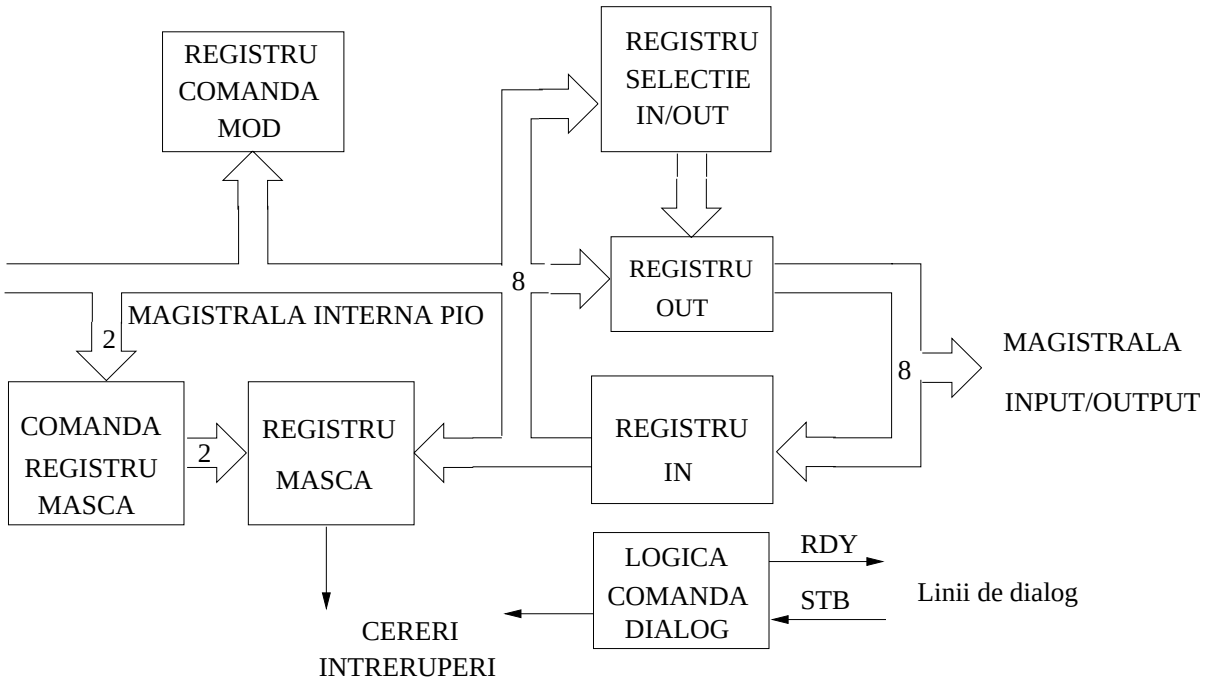


Figura 4.3: Structura unui port al circuitului Z80-PIO.

oricînd perifericul cere transferul unui nou octet. Circuitul PIO permite controlul complet al întreruperilor ierarhizate. Astfel, dispozitivele cu prioritate mai mică nu le pot întrerupe pe cele cu prioritate mai mare, ale căror subrutine de deservire nu au fost executate de procesor. Cele cu prioritate mai mare pot întrerupe deservirea celor mai puţin prioritare.

Dacă procesorul aflat în modul 2 de întrerupere acceptă o întrerupere, circuitul PIO care a cerut întreruperea trebuie să furnizeze unităţii centrale *un vector de întrerupere*. Acest vector indică o locaţie de memorie unde se află adresa rutinei de servire a întreruperii. Cei 8 biţi furnizaţi de dispozitivul care a cerut întreruperea reprezintă cei mai puţin semnificativi 8 biţi ai indicatorului, în timp ce registrul I din procesor asigură cei mai semnificativi 8 biţi.

Fiecare port are un vector de întrerupere independent. Cel mai puţin semnificativ bit al vectorului este fixat în mod automat în 0 în interiorul circuitului PIO, pentru că indicatorul trebuie să identifice două locaţii de memorie succesive, pentru a forma o adresă completă de 16 biţi. Spre deosebire de alte periferice din sistemul Z80, circuitul PIO nu acceptă întreruperi imediat după programare, ci aşteaptă până cînd */M1* este în 0 logic (de exemplu în timpul aducerii unui cod de operaţie).

Circuitul PIO decodifică instrucţiunea de revenire din întrerupere RETI direct de pe magistrala de date a sistemului, astfel încît fiecare circuit PIO din sistem ştie în orice moment dacă este deservit de procesor printr-o rutină de tratare a întreruperii, nefiind astfel necesară nici o comunicaţie în plus cu procesorul.

4.1.5 Iniţializarea circuitului Z80-PIO

Circuitul Z80-PIO intră în mod automat în starea iniţială (de reset) cînd este pus sub tensiune. În acest caz au loc următoarele acţiuni:

- Ambele registre de mascare a porturilor sînt iniţializate pentru a inhiba toţi biţii de date ai porturilor;
- Liniile de date ale magistralelor porturilor trec în starea de înaltă impedanţă şi semnalele de conversaţie sînt inactivate. Modul 1 este selectat automat;
- Registrele vectorilor de adresă nu sînt iniţializate;
- Ambele bistabile de validare a intreruperilor din port sînt initializate;
- Ambele registre de ieşire ale porturilor sînt iniţializate.

Circuitul PIO poate fi iniţializat aplicînd un semnal */M1* în absenţa unui semnal */RD* sau */IORQ*, rezultatul fiind iniţializarea circuitului imediat după ce */M1* devine inactiv. După ce intră în starea iniţială, circuitul PIO rămîne în această stare pînă la primirea unui cuvînt de control de la procesor.

4.1.6 Programarea circuitului Z80-PIO

Programarea unui port în modurile 0, 1 sau 2 necesită două cuvinte pentru fiecare port. Al treilea cuvînt este trimis numai atunci cînd se doreşte validarea/invalidarea întreruperilor.

Primul cuvînt este *cuvîntul de selectare a modului de operare*. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0
M1	M2	X	X	1	1	1	1

unde:

- D0-D3 identifică cuvîntul de selectare a modului de operare;
- D4, D5 nu contează;
- D6, D7 determină modul de operare după cum urmează:

M0	*M1*	*MOD*
0	0	ieşire
0	1	intrare
1	0	bidirecţional
1	1	bit

Al doilea cuvînt este *vectorul de întrerupere*, cuvînt care trebuie furnizat de circuitul PIO care a cerut întreruperea, în cazul în care aceasta a fost acceptată. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0
V7	V6	V5	V4	V3	V2	V1	0

unde:

- D0 identifică vectorul de întrerupere;
- D1-D7 reprezintă vectorul de întrerupere fixat de utilizator.

Programarea unui port în modul 3 necesită un cuvînt de control, vector de întrerupere (dacă întreruperile sînt activate) şi încă trei cuvinte care vor fi descrise în continuare.

Cuvîntul registrului de control intrare/ieşire defineşte care linii ale portului sînt intrări şi care sînt ieşiri. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0

unde:

- un bit 0 defineşte o ieşire;
- un bit 1 defineşte o intrare.

În modul 3 semnalele conversaţionale nu sînt folosite. Întreruperile sînt generate ca funcţii logice aplicate liniilor considerate intrări.

Cuvîntul de control al întreruperii fixează condiţiile şi nivelele logice necesare generării semnalului de întrerupere. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0
I3	I2	I1	I0	0	1	1	1

unde:

- D3 - D0 identifică cuvîntul de control al întreruperii
- D4 = 0 - nu urmează cuvănt mască, 1 - urmează cuvînt mască
- D5 = 0 - semnale active în stare Low, 1 - semnale active în stare High
- D6 = 0 - întrerupere la funcţia SAU logic, 1 - întrerupere la funcţia ŞI logic
- D7 = 0 - dezactivare întreruperi, 1 - activare întreruperi.

Cuvîntul mască permite ca orice bit nefolosit din port să fie mascat. Dacă se doreşte acest lucru, atunci bitul D4 din cuvîntul de control al întreruperii trebuie setat, iar următorul cuvînt scris în port trebuie să fie cuvîntul mască. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0

unde un bit este monitorizat dacă este definit ca ieşire, iar bitul mască este pus în 0 logic.

Pentru invalidarea întreruperilor unui port se poate folosi *cuvîntul de dezactivare întreruperi*. Se poate utiliza fără a schimba restul cuvîntului de control al întreruperilor şi în acest mod conţinutul bistabilului de validare a întreruperilor. Structura cuvîntului este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0
I	X	X	X	0	0	1	1

unde:

- D3-D0 identifică cuvîntul de dezactivare întreruperi
- D4-D6 pot lua orice valori
- D7 = 0 - invalidare întreruperi, 1 - validare întreruperi.

Dacă apare o cerere de întrerupere, în timp ce procesorul înscrie cuvîntul de dezactivare a întreruperilor în PIO (03H), poate să apară o problemă în sistem. Dacă întreruperile sînt validate de procesor, acesta va accepta întreruperea cerută de PIO. Totuşi în acest timp circuitul

PIO, primind cuvîntul de dezactivare a întreruperilor nu va trimite vectorul de întrerupere în timpul ciclului de recunoaştere a întreruperilor. Ca urmare, procesorul va prelua de pe magistrala de date, un vector eronat. Soluţia pentru evitarea acestor erori este să se dezactiveze întreruperile în procesor cu o instrucţiune *DI*, chiar înainte de dezactivarea întreruperilor circuitului PIO, şi să se valideze din nou întreruperile cu o instrucţiune *EI*, după aceea. Aceasta determină procesorul să ignore eventualele cereri de întrerupere de la circuitul PIO în timpul dezactivării lui.

4.1.7 Întrebări

I. Descrieţi modurile de funcţionare ale circuitului PIO.

II. Daţi un exemplu de programare al circuitului PIO în modul 3 cu portul B ca port de ieşire.

Răspuns:

```
LD  A, 0FFH
OUT (83H), A
INC A
OUT (83H), A
```

III. Cum se programează circuitul Z80-PIO?

IV. Cum se poate realiza un transfer între PIO şi un periferic, fără întreruperi?

4.1.8 Aplicaţie: Comanda motorului de curent continuu

Programul va conţine trei părţi, prezentate în figura 4.4:

- programarea circuitului PIO,
- comanda motorului,
- procedura de întîrziere.

Programarea circuitului PIO: Portul B al circuitului este legat la portul de intrare al plăcii de aplicaţii. Acest port va trebui comandat pentru a trimite date plăcii. Deoarece nu este nevoie de semnalele de conversaţie ale portului, şi nici de un vector de întrerupere, portul B al circuitului PIO va fi programat în mod bit, cu numai două cuvinte de comandă.

Primul cuvînt, cuvîntul de selectare a modului de operare, va fi 0FFH. Cel de-al doilea cuvînt, cuvîntul registrului de control intrare/ieşire, va fi 00H (adică toţi biţii portului sînt biţi de ieşire).

Comanda motorului: Motorul de curent continuu de pe placa de aplicaţii poate fi comandat cu ajutorul biţilor 6 şi 7 ai portului de intrare, care este legat la portul B al circuitului PIO. Motorul poate fi pornit, caz în care accelerează pînă la viteza maximă, sau oprit. Semnificaţia biţilor de comandă este prezentată în tabelul 4.1.

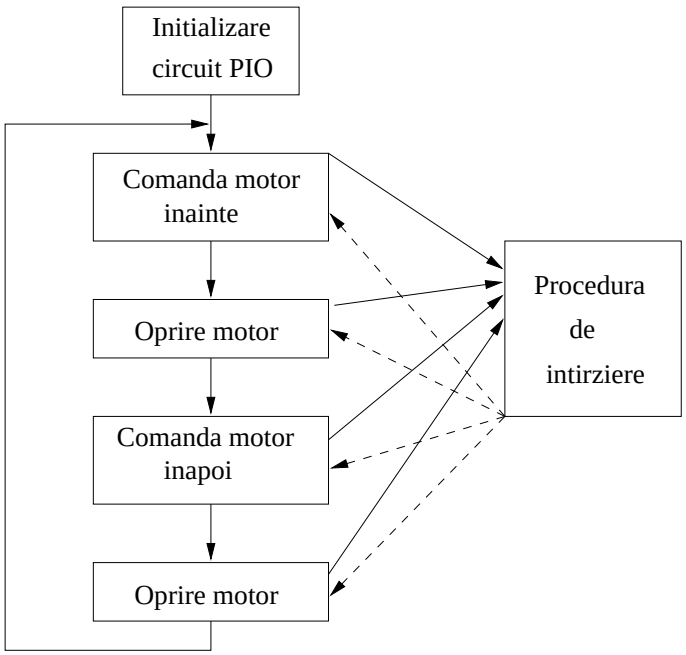


Figura 4.4: Etapele de programare

Bit 7	*Bit 6*	*Actiune*
0	0	Motor oprit
0	1	Mişcare înainte
1	0	Mişcare înapoi
1	1	Motor oprit

Tabelul 4.1: Semnificaţia biţiilor de comandă.

Procedura de întîrziere: Durata buclei de întîrziere se calculează cunoscînd frecvenţa de ceas a sistemului şi numărul de perioade de ceas necesare instrucţiunilor. Frecvenţa sistemului este 1.79 MHz, de unde rezultă că perioada ceasului este 0.5586 ms (în calcule se consideră 0.56 ms). În continuare se va programa o buclă de întîrziere de 1 s. Instrucţiunea `DEC` consumă 4 perioade de ceas, iar instrucţiunea `JP NZ, nn` consumă 10 perioade de ceas, deci este nevoie de $14 \times 255 = 3570$ perioade de ceas pentru a decrementa un registru care conţine data 0FFH pînă la 0. Deci sînt necesare $255 \times 3570 = 910350$ perioade de ceas pentru a decrementa un registru care conţine data 0FFFFH pînă la 0. Această operaţie durează $0.56ms \times 910350 = 0.509796s$. Prin urmare, realizînd această operaţie de două ori, se obţine o întîrziere de aproximativ 1 s.

Programul scris în limbaj de asamblare

```
****;  comanda motorului 1s înainte, 1s stop, 1s înapoi

1800                        ORG  1800H
1800    3E FF               LD   A,0FFH   ; programare circuit PIO
1802    D3 83               OUT  (83H),A  ; mod 3
```

```
1804    3C                  INC  A         ; A=0
1805    D3 83               OUT  (83H),A   ; B port de ieşire
1807    3E 40   START:      LD   A,40H     ; bit 6 = 1, mişcare înainte
1809    D3 81               OUT  (81H),A   ; trimis la port B
180B    CD 24 18            CALL DELAY     ; apel procedură întîrziere (1 s)
180E    AF                  XOR  A         ; A=0
180F    D3 81               OUT  (81H),A   ; oprire motor
1811    CD 24 18            CALL DELAY     ; repaus 1 s
1814    3E 80               LD   A,80H     ; bit 7 = 1, mişcare înapoi
1816    D3 81               OUT  (81H),A
1818    CD 24 18            CALL DELAY     ; timp de 1 s
181B    AF                  XOR  A         ; A=0
181C    D3 81               OUT  (81H),A   ; oprire motor
181E    CD 24 18            CALL DELAY     ; repaus 1 s
1821    C3 07 18            JP   START     ; reia ciclul
1824    2E 02    DELAY:     LD   L,2
1826    01 FF FF LOOP2:     LD BC,0FFFFH   ; aproximativ 1/2 s
1829    0D       LOOP1:     DEC C
182A    C2 29 18            JP   NZ,LOOP1 ; repetă pînă C=0
182D    05                  DEC  B
182E    C2 29 18            JP   NZ,LOOP1 ; repetă pînă B=0
1831    2D                  DEC  L
1832    C2 26 18            JP   NZ,LOOP2 ; întîrzie încă 1/2 s
1835    C9                  RET           ; întoarcere în program
                            END
```

Înainte de a executa programul pe machetă, comutatorul SW2-2 trebuie fixat pe poziţia "MOTOR".

4.1.9 Experimente:

I. Comandaţi motorul de curent continuu de pe placa de aplicaţii conform graficului din figura 4.5.

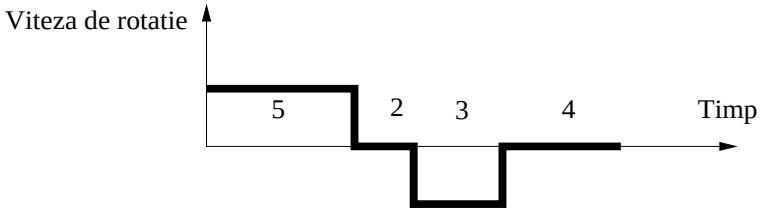


Figura 4.5: Grafic de funcţionare a motorului de curent continuu.

II. Cum se poate obţine comanda motorului de curent continuu astfel încît să se obţină o pantă de viteză? Realizaţi un program care să comande motorul conform graficului din figura 4.6. Figura prezintă atit profilul de viteză ideal cît şi cel real pentru motorul de curent continuu.

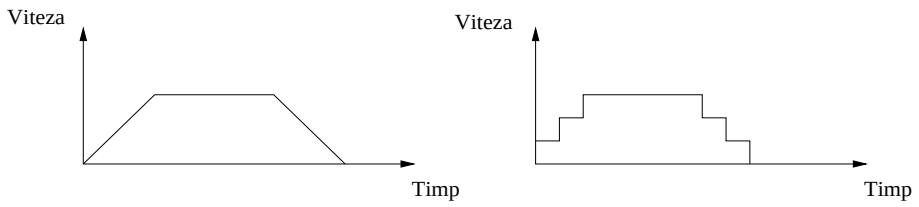


Figura 4.6: Graficul ideal (a) şi real (b) în regim accelerat/decelerat.

Lucrarea 5

Aplicaţii cu circuitul Z80-CTC

Această lucrare prezintă circuitul numărător/temporizator Z80-CTC şi modul de utilizare a acestuia.

5.1 Prezentare generală şi arhitectura internă a circuitului Z80-CTC

Z80-CTC (Counter/Timer Circuit - engl.) este un circuit cu patru canale ce pot funcţiona în mod numărător sau temporizator. Acest circuit poate fi folosit pentru o gamă largă de aplicaţii de numărare: numărare de evenimente, cronometrări de intervale de timp, generare de întreruperi şi generarea unui semnal de ceas. Cele patru canale sînt programabile independent în două moduri de lucru. Circuitul Z80-CTC se conectează direct (pin la pin) la circuitul microprocesor Z80-CPU. Fiecare canal se programează cu doi octeţi de comandă. Cînd se activează întreruperile, este necesar încă un octet suplimentar ce semnifică vectorul de întrerupere. După programare, circuitul numără descrescător pînă la zero. Apoi, se reîncarcă automat (dintr-un registru) şi reia procesul de numărare descrescătoare. Prin utilizarea circuitului CTC se pot elimina buclele de întîrziere implementate prin program. Lucrul cu întreruperile este simplificat, deoarece circuitului i se trimite un singur vector de întrerupere, iar acesta generează intern cîte un vector pentru fiecare canal. Semnalul de ceas monofazic este primit de la procesor. Figura 5.1 prezintă structura internă a circuitului Z80-CTC. Simbolul circuitului este prezentat în figura 5.2.

5.1.1 Semnificaţia pinilor circuitului Z80-CTC

Semnale generale:

CLK (System Clock) - Semnal de ceas comun tuturor circuitelor din sistemul cu microprocesor Z80.

/RESET (Reset) - Semnal de iniţializare. Activarea acestui semnal conduce la terminarea tuturor acţiunilor de numărare descrescătoare şi dezactivarea tuturor întreruperilor. Biţii de întrerupere din registrele de control ale canalelor sînt resetaţi. Ieşirile de întreruperi şi *ZC/TO* devin inactive. *IEO* ia valoarea lui *IEI*. Magistrala de date este trecută în starea de înaltă impedanţă.

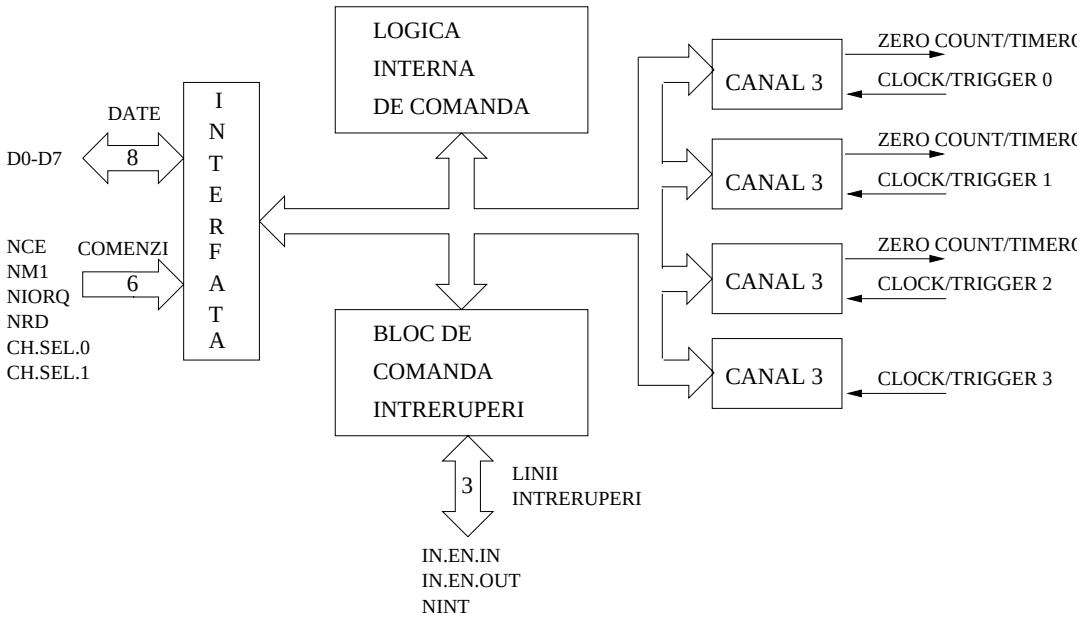


Figura 5.1: Structura internă a circuitului Z80-CTC.

Magistrala de date:

D0-D7 (System Data Bus) - Magistrală bidirecţională, conectată la magistrala de date a procesorului.

Semnalele de control (primite de la procesor):

CS0-CS1 (Channel Select) - Semnale care formează o adresă de doi biţi cu care se selectează unul din cele patru canale ale circuitului pentru o operaţie de scriere sau citire. De obicei, aceşti pini se leagă la pinii *A0-A1* ai magistralei de adrese a procesorului. Modul de selecţie a canalelor este prezentat în tabelul 5.1.

	CS1	*CS0*
Ch0	0	0
Ch1	0	1
Ch2	1	0
Ch3	1	1

Tabelul 5.1: Selecţia canalelor cu biţii CS0 şi CS1.

/CE (Chip Enable) - Semnal de validare a chip-ului. Semnalul este activat cînd circuitul acceptă cuvinte de control, vectori de întrerupere sau constante de timp de pe magistrala de date, în timpul unui ciclu de scriere la dispozitivele de intrare/ieşire sau cînd se transmite procesorului conţinutul unui numărător în timpul unui ciclu de citire de la dispozitivele de intrare/ieşire. În majoritatea aplicaţiilor, acest semnal este decodificat din cei mai puţin semnificativi opt biţi ai magistralei de adrese pentru oricare din cele patru adrese de intrare/ieşire care sînt asociate celor patru canale ale circuitului.

/M1 (Machine Cycle 1) - Semnal provenit de la pinul */M1* al procesorului. Cînd */M1* şi */IORQ* sînt active, procesorul Z80 acceptă o întrerupere. Apoi, dacă are prioritatea cea mai mare şi dacă unul din canale a cerut o întrerupere (prin activarea semnalului */INT*), circuitul CTC

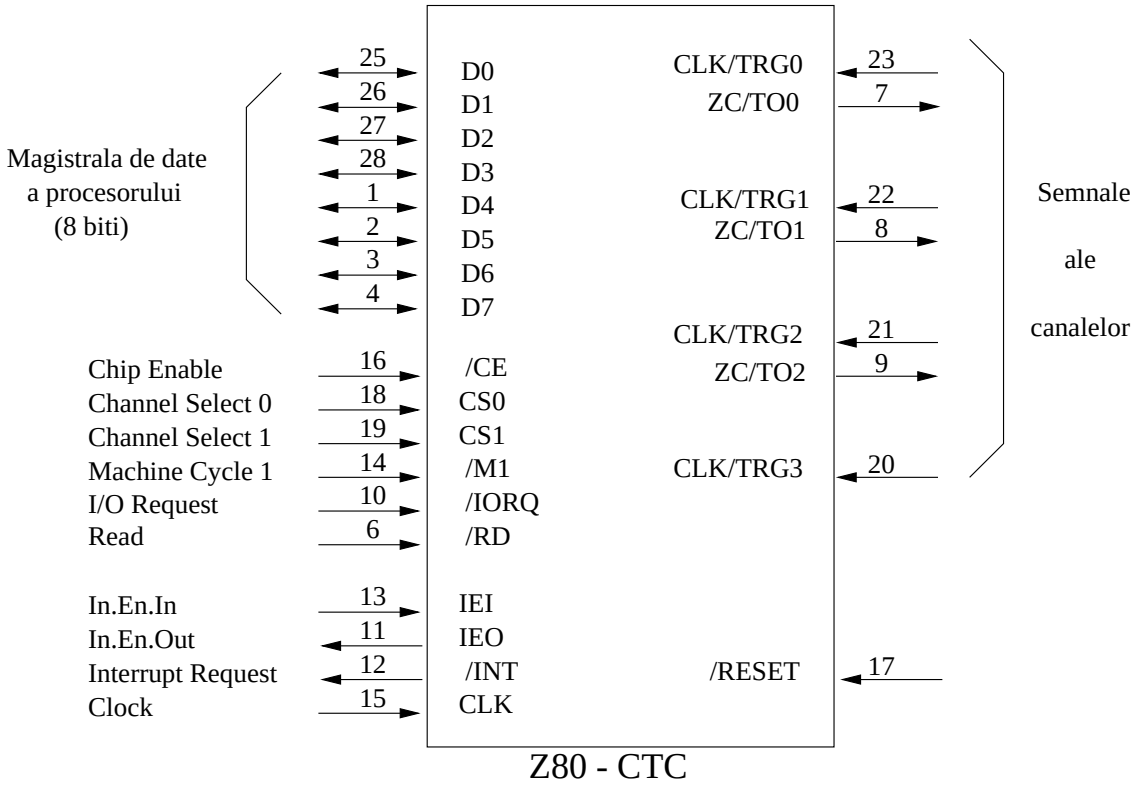


Figura 5.2: Simbolul circuitului Z80-CTC.

plasează vectorul de întrerupere pe magistrala de date.

/IORQ (Input/Output Request) - Semnal provenit de la pinul */IORQ* al procesorului. Semnalul este folosit în conjuncţie cu */CE* şi */RD* pentru a transfera date şi cuvinte de control pentru canale între procesor şi CTC. În timpul unui ciclu de scriere, semnalele */IORQ* şi */CE* trebuie să fie active, iar semnalul */RD* trebuie să fie inactiv. Circuitul CTC nu primeşte un semnal specific de scriere, ci îşi generează unul intern din inversul unui semnal */RD*. Într-un ciclu de citire, */IORQ*, */RD* şi */CE* trebuie să fie active pentru ca procesorul să poată citi conţinutul unui numărător.

/RD - (Read Cycle Status) - Semnal provenit de la pinul */RD* al procesorului. Semnalul este folosit în conjuncţie cu */IORQ* şi */CE* pentru a transfera date şi cuvinte de control între CTC şi procesor.

Semnale de întrerupere:

IN.EN.IN (IEI) - (Interrupt Enable In) - Un semnal cu valoare logică 1 pe această linie semnifică faptul că nici un alt dispozitiv periferic cu prioritate mai mare în lanţul de întreruperi nu este deservit de către procesor.

IN.EN.OUT (IEO) - (Interrupt Enable Out) - Semnal folosit în conjuncţie cu *IEI* pentru a forma un sistem de întreruperi ierarhizat. Linia este în 1 logic numai dacă linia *IEI* este în aceeaşi stare şi procesorul nu deserveşte o întrerupere de la unul din canalele circuitului. Semnalul blochează dispozitivele cu prioritate mai mică pentru ca acestea să nu poată întrerupe un dispozitiv cu prioritate mai mare în timp ce este deservit de procesor.

/INT - (Interrupt Request) - Semnal activ cînd numărătorul unui canal al circuitului CTC, programat să activeze semnalul de întrerupere, a ajuns la zero.

Semnalele canalelor:

CLK/TRG0-CLK/TRG3 - (External Clock/Timer Trigger) - Patru semnale ce corespund celor patru canale ale circuitului. În mod *numărător*, fiecare front activ pe acest pin decrementează numărătorul. În mod *timer*, un front activ al semnalului porneşte timerul. Utilizatorul poate selecta frontul activ fie crescător, fie descrescător.

ZC/TO0-ZC/TO2 - (Zero Count/Timeout) - Trei semnale ce corespund canalelor 0-2 ale circuitului. În ambele moduri, ieşirea prezintă un impuls cu valoare 1 logic cînd numărătorul ajunge la zero.

Funcţiile semnalelor de la pinii circuitului Z80-CTC sînt prezentate centralizat în tabelul 5.2.

Nume	*Funcţie*	*Tip*
Semnale generale		
CLK	Tact sistem	Intrare
/RESET	Reset sistem	Intrare
	Magistrală de date	
D0-D7	Bus date	Bidir. 3-stări
Semnale de control		
CS0-CS2	Selecţie canal	Intrare
/CE	Validare chip	Intrare
/M1	Ciclu maşină 1	Intrare
/IORQ	Cerere I/O	Intrare
/RD	Citire	Intrare
Semnale de întrerupere		
IEI	Activare întreruperi	Intrare
IEO	Inactivare întreruperi	Ieşire
/INT	Cerere de întrerupere	Intrare
Semnale ale canalelor		
CLK/TRG0-CLK/TRG3	Ceas extern	Intrare
ZC/TO0-ZC/TO2	Sfîrşit numărare	Intrare

Tabelul 5.2: Funcţiile pinilor circuitului Z80-CTC.

5.1.2 Structura unui canal

Un canal este compus din două registre de cîte opt biţi, două număratoare şi logică de control. Un registru este folosit pentru a memora constanta de timp, iar celălalt pentru a memora modul de lucru şi parametrii canalului. Un numărător descrescător pe 8 biţi poate fi citit de către procesor. Un alt numărător de opt biţi implementează un divizor de frecvenţă al semnalului de ceas. Figura 5.3 prezintă structura unui canal al circuitului Z80-CTC.

Registrul de control al canalului este înscris de către procesor pentru a selecta modul de lucru şi parametrii canalului. În circuit sînt patru asemenea registre, corespunzînd celor patru canale Selectarea registrului în care se scrie se face cu pinii *CS1* şi *CS0*.

Divizorul este folosit numai în modul *timer*. Acesta este un dispozitiv de numărare pe 8 biţi care este programat de procesor prin registrul de control al canalului, el divizînd semnalul de

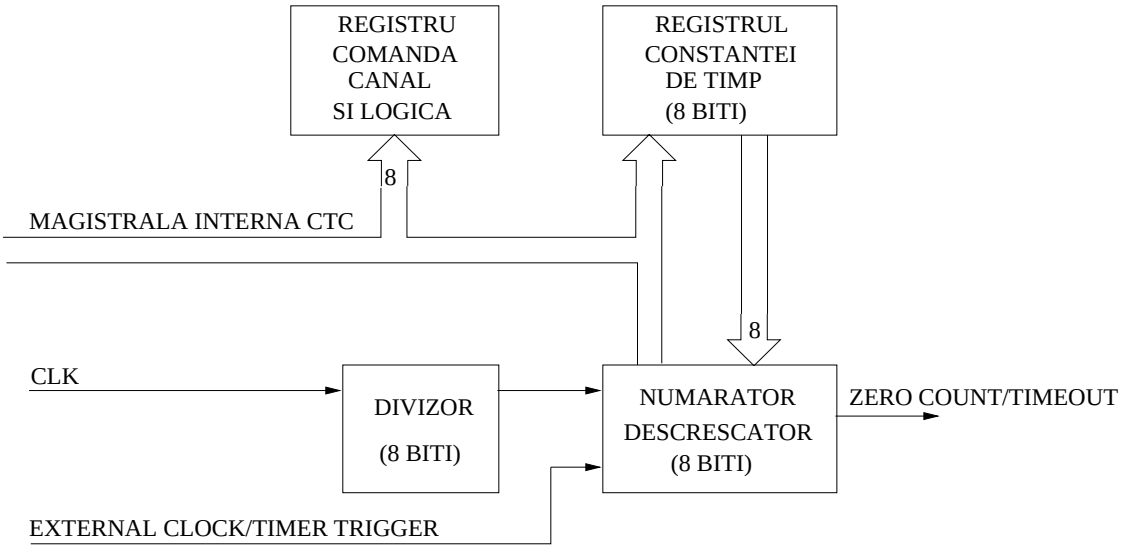


Figura 5.3: Schema bloc a unui canal a circuitului Z80-CTC.

intrare (ceasul sistemului). Ieşirea divizorului este folosită ca intrare de ceas pentru numărătorul descrescător. Registrul constantei de timp este un registru de 8 biţi folosit în ambele moduri de funcţionare. Acest registru este înscris imediat după registrul de control al canalului. Constanta de timp este o valoare întreagă între 1 şi 256 (256 este codificat cu 8 biţi de zero). Această constantă este automat încărcată în numărătorul descrescător atunci cînd canalul este iniţializat sau de fiecare dată cînd numărătorul ajunge la zero.

Numărătorul descrescător este un dispozitiv de numărare pe 8 biţi folosit în ambele moduri de funcţionare. Înainte de fiecare ciclu de numărare el este încărcat cu valoarea conţinută în registrul constantei de timp. Numărătorul este decrementat pe frontul activ al ceasului extern în modul numărător sau pe cel al ieşirii de ceas dată de divizor. Valoarea conţinută în numărător poate fi citită de către procesor în orice moment printr-o operaţie de citire de la adresa portului ce a fost asociat canalului respectiv. Canalele 0, 1 şi 2 pot fi programate ca atunci cînd ajung la zero să genereze o întrerupere. Datorită limitărilor de pini, canalul 3 nu are această posibilitate. Canalul 3 poate fi folosit numai în aplicaţiile care nu trebuie să genereze semnal de întrerupere.

5.2 Modurile de lucru ale circuitului Z80-CTC

La punerea sub tensiune, starea circuitului Z80-CTC este necunoscută. Prin activarea semnalului */RESET* se aduce circuitul într-o stare iniţială, cunoscută. Pentru a putea folosi un canal pentru numărare, acesta trebuie programat cu un cuvînt de control şi o constantă de timp. Dacă un canal a fost programat să activeze semnalul de întrerupere, trebuie programat şi un vector de întrerupere. După programarea unui canal prin trimiterea cuvintelor de control, acesta va începe să funcţioneze conform modului programat: *numărător* sau *timer*.

Modul numărător

În modul *numărător* circuitul numără fronturile active ale intrării de ceas extern *CLK/TRG*. Circuitul numărător este încărcat cu constanta de timp şi la fiecare eveniment extern este decrementat pînă cînd ajunge la zero. Numărătoarele 0, 1 şi 2 pot fi programate să genereze o întrerupere în acel moment. În acelaşi timp, sînt încărcate automat cu valoarea conţinută

în registrul constantei de timp, fără să se întrerupă procesul de numărare. Dacă în registrul constantei de timp se înscrie o nouă valoare în timp ce numărătorul funcţionează, se termină mai întîi numărătoarea curentă şi abia apoi se va încărca noua valoare în numărător.

Modul timer

În modul *timer* circuitul generează semnale cu perioada multiplu de perioada ceasului sistem. Pentru a realiza acest lucru sînt folosite divizoarele aferente fiecărui canal. Divizarea ceasului sistem se face în două etape: prima în divizor (cu 16 sau 256), iar a doua în numărător (cu valoarea înscrisă în registrul constantei de timp). Şi în acest mod numărătoarele 0, 1 şi 2 pot genera o întrerupere atunci cînd ajung la zero. Se obţine un semnal cu perioada:

$$t_C \times D \times C_T$$

unde:
t_C este perioada ceasului sistem,
D este factorul de divizare programat,
C_T este constanta de timp programată.

Circuitul poate fi programat să numere imediat după ce a fost iniţializat (numărătoarea porneşte odată cu ciclul procesor ce urmează celui în care a fost înscris registrul constantei de timp) sau la frontul activ al semnalului de triggerare *CLK/TRG* (numărătoarea începe la al doilea front activ al semnalului de trigerare, după ce a fost încărcată constanta de timp).

5.3 Blocul de comandă a întreruperilor

Blocul de comandă a înteruperilor asigură interfaţarea întreruperilor circuitului CTC cu sistemul de întreruperi ierarhizate al procesorului Z80. Semnalele cu care se asigură corelarea cu celelalte dispozitive periferice sînt *IEI* şi *IEO*. În timp ce o cerere de întrerupere a circuitului CTC este deservită de procesor, blocul de comandă a întreruperilor ţine semnalul *IEO* în 0 logic, inhibînd astfel întreruperile dispozitivelor mai puţin prioritare. Cînd *IEI* devine 0, blocul de comandă a întreruperilor poate genera o întrerupere. Figura 5.4 prezintă strucura unui lanţ de întreruperi cu priorităţi ierarhizate.

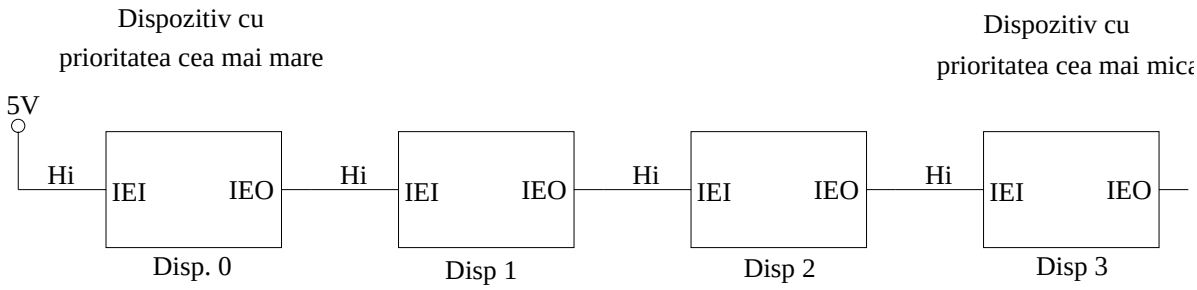


Figura 5.4: Lanţ de dispozitive cu priorităţi ierarhizate.

Dacă un canal este programat să genereze o întrerupere, blocul de comandă a întreruperilor pune linia *IEO* în 0 logic atunci cînd numărătorul canalului respectiv ajunge la zero. Simultan, se activează semnalul */INT*. La răspunsul procesorului (*/M1* şi */IORQ*), acest bloc pune pe magistrala de date vectorul de întrerupere corespunzător canalului care a generat întreruperea. Totodată, acest bloc arbitrează priorităţile întreruperilor în circuitul CTC. Sistemul este identic cu cel al procesorului Z80, canalul 0 avînd prioritatea cea mai ridicată. Blocul de comandă

a întreruperilor monitorizează magistrala de date şi decodifică instrucţiunea de întoarcere din întrerupere (RETI). Dacă o întrerupere este în aşteptare, blocul ţine semnalul *IEO* în 0 logic. Cînd apare instrucţiunea RETI (2 octeţi), dispozitivul comută linia *IEO* în 1 logic pe durata unui ciclu maşină (*M1*) ca să se asigure că dispozitivele cu prioritate mai mică vor decodifica întreaga instrucţiune RETI şi se vor iniţializa corespunzător.

5.4 Iniţializarea circuitului Z80-CTC

Circuitul Z80-CTC are două moduri de iniţializare: *hardware* şi *software*.

Iniţializarea *hardware* încheie toate operaţiile de numărare. În acest caz, au loc următoarele acţiuni:

- dezactivarea tuturor întreruperilor circuitului CTC şi resetarea biţilor de întrerupere din registrele de control ale canalelor;
- inactivarea semnalelor *ZC/TO* şi *INT*;
- semnalul *IEO* ia valoarea semnalului *IEI*;
- magistrala de date este trecută în starea de înaltă impedanţă.

Toate canalele trebuie să fie complet reprogramate după acest tip de iniţializare.

Iniţializarea *software* este controlată de bitul 1 din registrul de control al canalului. Cînd un canal este iniţializat software se opreşte numărarea. În momentul iniţializării software, ceilalţi biţi din cuvîntul de control modifică parametrii canalului. După o astfel de iniţializare trebuie înscrisă o nouă constantă de timp în registrul corespunzător al aceluiaşi canal.

5.5 Programarea circuitului Z80-CTC

Fiecare canal al circuitului trebuie programat înainte de funcţionare. Programarea constă în înscrierea a două cuvinte de control la adresa portului ce corespunde canalului dorit. Primul cuvînt de control selectează modul de operare şi parametrii canalului. Al doilea cuvînt reprezintă constanta de timp, care este o dată binară cu valoare între 1 şi 256. Constanta de timp trebuie să fie precedată de cuvîntul de control al canalului. După iniţializare, canalele pot fi reprogramate la orice moment. Dacă pentru un canal sînt activate întreruperile, atunci mai este nevoie de un cuvînt de comandă ce reprezintă vectorul de întrerupere. Este necesar un singur vector de întrerupere, deoarece circuitul generează intern vectori diferiţi pentru fiecare canal. Structura cuvîntului de control pentru un canal este următoarea:

D7	D6	D5	D4	D3	D2	D1	1

Biţii au următoarea semnificaţie:

- D0 = 1 identificator pentru cuvîntul de control;
- D1 = 0 - continuarea funcţionării, 1 - iniţializare software;

- D2 = 0 - nu urmează constanta de timp, 1 - urmează constanta de timp;
- D3 = 0 - triggerare automată după încărcarea constantei de timp, 1 - triggerare externă cu semnalul CLK/TRG;
- D4 = 0 - frontul activ este descrescător, 1 - frontul activ este crescător;
- D5 = 0 - factorul de divizare are valoarea 16, 1 - factorul de divizare are valoarea 256;
- D6 = 0 - funcţionare în mod 'timer', 1 - funcţionare în mod 'numărător';
- D7 = 0 - dezactivare întreruperi, 1 - activare întreruperi.

Întreruperile pot fi programate în orice mod de funcţionare şi pot fi activate sau dezactivate în orice moment. Reprogramarea frontului activ al semnalului *CLK/TRG* în timpul funcţionării este echivalentă cu inserarea unui front activ în semnal. Dacă numărătorul aşteaptă un eveniment în timpul reprogramării (în ambele moduri), aceasta nu va interveni în procesul de numărare. Odată pornit, numărătorul funcţionează neîntrerupt pînă este oprit prin iniţializare. Numărătorul nu poate funcţiona fără o constantă de timp. Aceasta se înscrie în registrul corespunzător în urma unui cuvînt de control care are bitul 2 setat. Structura cuvîntului de cod care stabileşte constanta de timp este următoarea: Cei 8 biţi codifică un număr binar între 1 şi 256.

CT7	CT6	CT5	CT4	CT3	CT2	CT1	CT0

Valoarea 0 pe toţi cei 8 biţi semnifică o constantă de timp egală cu 256. Dacă Z80-CTC are una sau mai multe întreruperi activate, atunci trebuie să i se furnizeze un vector de întrerupere. Din acest cuvînt trebuie programaţi numai cei mai semnificativi 5 biţi, deoarece ceilalţi 3 biţi sînt completaţi de circuitul CTC. Structura cuvîntului de stabilire a vectorului de întrerupere este următoarea:

D7	D6	D5	D4	D3	D2	D1	D0
V7	V6	V5	V4	V3	CS1	CS0	0

unde:

- D0 = 0 identificatorul vectorului de întrerupere,
- CS1-CS0 reprezintă semnalele de selecţie a canalelor (automat introduşi de circuitul CTC),
- V7-V3 cei 5 biţi programaţi de către utilizator.

5.5.1 Realizarea unui ceas folosind circuitul Z80-CTC

Realizarea ceasului implică trei etape:

- programarea circuitului Z80-CTC,
- scrierea rutinei de servire a întreruperii,
- actualizarea şi afişarea orei (numerele binare vor fi convertite în cod BCD pentru a uşura citirea orei.

Programul este conceput astfel încît să se înceapă măsurarea timpului de la o ora prestabilită. Această ora poate fi stabilită modificînd valorile iniţiale din buffer-ul de timp.

Programarea circuitului Z80-CTC

Pentru acest program este nevoie de un singur canal al circuitului CTC, folosit în modul timer. Ceasul timer-ului va fi ceasul sistemului. Problema principală va fi măsurarea secundei. Vom considera factorul de divizare 256. Ramîne problema stabilirii constantei de timp.

$256 \times 0.56\mu s = 0.00014336s$

$1/0.00014336 = 6975.446428571$

$6975.446428571/218 = 31.99746068152$

Deci, $32 \times 218 \times 256 \times 0.56\mu s = 1.00007936s$, o aproximare destul de bună a secundei. Eroarea este de $0.08ms$ la o secundă, adică aproape 8 secunde la 24 ore. Prin urmare programarea circuitului CTC se va face cu trei cuvinte astfel:

cuvîntul de control canalului 101101012 = 0B5H

- b7=1 întreruperi activate,
- b6=0 mod timer,
- b5=1 factor de divizare 256,
- b4=1 numărare la front crescător,
- b3=0 folosirea ceasului intern,
- b2=1 urmează constanta de timp,
- b1=0 nu se resetează numărătorul,
- b0=1 identificarea cuvîntului de control

constanta de timp 21810 = 0DAH Înmulţirea cu al treilea factor - 32 - se va face în rutină de servire a întreruperii, imcrementînd contorul secundelor numai cînd contorul întreruperilor ajunge la 32.

vectorul de întrerupere 0A8H În prealabil registrul I va fi încărcat cu valoarea 18H, deoarece memoria disponibilă utilizatorului se află între adresele 1800-1F9FH.

Actualizarea şi afişarea orei

În rutina de servire a întreruperii se va testa condiţia de scurgere a unei secunde (contorul întreruperilor are valoarea 32). După trecerea unei secunde se va incrementa contorul secundelor şi se va testa depăşirea valorii maxime pentru secunde, minute şi ore (60, 60 respecţiv 12 sau 24). După incrementarea unui contor, se va face şi ajustarea zecimală a numerelor. În cazul atingerii valorii maxime pentru un contor, acesta va lua valoarea zero şi va fi incrementat contorul unităţii de măsură superioare (dacă este posibil). Afişarea orei se face convertind mai întîi numerele din buffer-ul de afişare în formatul de afişare cu 7 segmente, după care se apelează procedura SCAN. Cifrele care reprezintă orele, minutele şi secundele vor fi despărţite de puncte zecimale. În figura 5.5 este prezentată schema logică a programului.

5.5.2 Întrebări:

I. Care sînt condiţiile de scriere/citire pentru circuitul CTC-Z80?

Răspuns

Din punct de vedere hardware condiţiile pentru scrierea/citirea circuitului CTC-Z80 sînt următoarele:

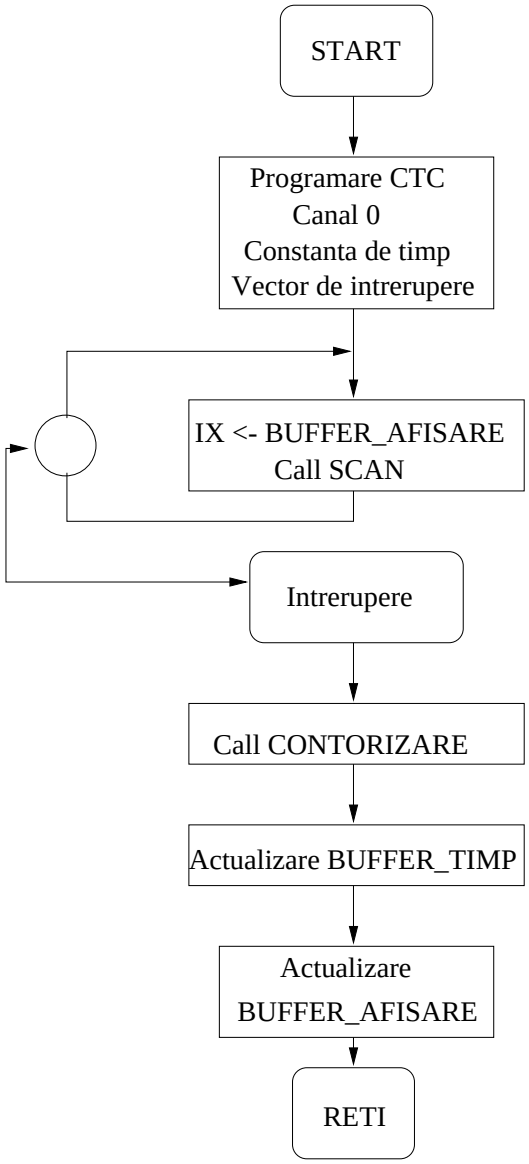


Figura 5.5: Schema logică a programului.

- pentru scriere: semnalele /IORQ şi /CE active, /RD inactiv. Circuitul CTC-Z80 generează intern semnalul de citire din inversul semnalului /RD.

- pentru citire: /IORQ, /CE, /RD active.

II. Care este structura unui canal? Pot fi folosite toate cele patru canale în sistem de întreruperii?

Răspuns

Fiecare canal este compus din două registre de 8 biţi, două numărătoare şi logică de control. Un numărător este folosit ca numărătoar descrescător iar altul ca divizor. Registrele folosesc pentru memorarea constantei de timp şi pentru setarea modului de lucru.

Datorită limitării numărului de pinii, canalul 3 nu se poate folosi în sistemul de întreruperi.

III. Cînd se activează semnalul de întrerupere */INT*?

Răspuns

Semnalul de întreruperi este activat de fiecare dată cînd numărătorul canalului respectiv ajunge la zero.

IV. Care sînt modurile de lucru ale CTC-Z80 şi care sînt condiţiile de programare?

Răspuns

Modurile de lucru ale CTC-Z80 sînt:

- mod numărător şi
- mod timer.

Programarea circuitului pentru canalul 0 se face în următorul mod:

```
CTC0 EQU 40H

LD  A,18H
LD  I,A
LD  A,10110101B
OUT (CTC0),A
LD  A,020H
OUT (CTC0),A
LD  A,0A8H
OUT (CTC0),A
IM  2
EI
```

Adresele celor 4 canale sînt 40H, 41H, 42H şi 43H. Cuvîntul de control fiind 10110101B, canalul 0 este programat în mod timer, factorul de divizare este 256. CTC va genera o întrerupere după 8192 (256×32) tacte, constanta de timp fiind 020H.

5.5.3 Experimente

I. Programaţi canalului 2 al circuitului CTC-Z80 în mod numărător?

II. Studiaţi următorul program ce implementează un ceas, conform schemei logice prezentate în figura 5.5. Numărarea secundelor se face cu ajutorul circuitului CTC.

```
*****; Program care implementează un ceas
*****; Pentru măsurarea unei secunde se foloseşte circuitul CTC

ORG             1800H
CTC0    EQU     40H   ; adresa portului la care se găseşte canalul 0
SCAN    EQU     05FEH ; adresa rutinei de afişare
HEX7SG  EQU     0678H ; adresa rutinei de conversie pentru afişare

START:
        LD A,18H
        LD I,A

*****; se completează cu programarea CTC, canalul 0
```

```
....
*****;

MAIN:
        LD   IX,BUFFER_AFISARE
        CALL SCAN             ; apel procedură SCAN
        JR   MAIN

*****; rutina de contorizare
CONTORIZARE:
        LD   DE,BUFFER_TIMP
        LD   A,(DE)           ; se citeşte contorul întreruperilor din buffer
        INC  A                ; se numără întreruperile
        LD   (DE),A           ; se salvează contorul actualizat
        CP   20H              ; într-o secundă apar 32 (20H) întreruperi
        LD   B,4              ; contor şi indicator pentru scurgerea secundei
        RET  NZ
        XOR  A                ; A=0
        DEC  B                ; B=3 (contor)
        LD   (DE),A           ; resetarea contorului de întreruperi
        INC  DE
        LD   HL,VALORI_MAXIME; se încarcă adresa tabelei cu
                              ; valori maxime în HL
LOOP:
        LD   A,(DE)           ; se încarcă în A numărul de secunde,
                              ; minute sau ore
        INC  A
        DAA                   ; ajustare zecimală a acumulatorului
        LD   (DE),A           ; actualizare buffer de timp
        SUB  (HL)             ; verificarea depăşirii valorilor maxime
        RET  C                ; ieşire în caz de nedepăşire a limitelor
        LD   (DE),A
        INC  HL               ; dacă se depăşesc limitele, se creşte
                              ; unitatea următoare
        INC  DE               ; şi se aduce la 0 cea curentă
        DJNZ LOOP
        RET

*****; procedura de conversie a bufferului de afişare
CONVERSIE_7SEG:
        LD   HL,BUFFER_AFISARE ; se pregăteşte rutina de conversie
        LD   DE,SECUNDE
        LD   B,3                 ; contor
LOOP2:  LD   A,(DE)              ; conversie buffer de timp
        CALL HEX7SG
        INC  DE
        DJNZ LOOP2
        DEC  HL
```

```
        DEC  HL
        SET  6,(HL)            ; se setează punctul zecimal pt. ore
        DEC  HL
        DEC  HL
        SET  6,(HL)            ; se setează punctul zecimal pt. minute
        RET

*****; rutina de servire a întreruperii

ORG     18A8H
DEFW    INTRERUPERE    ; la această adresă se află adresa de întrerupere

*****; început a rutinei de întrerupere
INTRERUPERE:
*****; se salvează registrele care vor fi afectate în stivă

        CALL CONTORIZARE       ; apelul rutinei care măsoară timpul
        LD   A,B
        CP   4                 ; se verifică dacă a trecut 1 s
        CALL NZ,CONVERSIE_7SEG ; conversia buffer-ului de timp

*****; rutina pentru afişare

*****; restaurare registre, activare întreruperi, revenire din întreruperi

*****; tabela de valori maxime

*****; se completează cu valorile pentru secunde, minute şi ore
*****; valorile sînt hexazecimale pt. că şi cele cu care se compară sînt
*****; ajustate BCD pt. afişare

*****; buffer-ul de afişare
BUFFER_AFISARE:
        DEFS       6      ; pt. afişare este nevoie de spaţiu pt. 6 caractere
        END
```

III. Completaţi programul cu părţiile comentate.

IV. Ce se va afişa pe display dacă se modifică valorile din tabela de valori maxime?

V. Cîte tacte de ceas sînt necesare pentru a contoriza o secundă?

Partea II

Microprocesorul 8086

Lucrarea 6

Arhitectura şi organizarea microprocesorului 8086

Această lucrare prezintă aspectele fundamentale ale microprocesorului 8086. Se regăsesc aici detalii necesare atît programatorului cît şi proiectantului de sisteme cu microprocesor. Expunerea din această lucrare este un material de referinţă şi pentru lucrările de laborator ce vor urma.

6.1 Arhitectura microprocesorului

Arhitectura microprocesorului 8086 (figura 6.1) prezintă două mari unităţi funcţionale:

- unitatea de interfaţare cu magistrala externă (*Bus Interface Unit* = BIU) şi
- unitatea de execuţie a instrucţiunilor (*Execution Unit* = EU).

Aceste două unităţi operează asincron şi formează un mecanism unitar de aducere şi execuţie a unei instrucţiuni.

În esenţă, procesarea paralelă asigurată de BIU şi EU elimină timpul necesar pentru aducerea multor instrucţiuni prin suprapunerea etapei de aducere a unei instrucţiuni (*fetch*) cu etapa de execuţie a altei instrucţiuni. Ca o consecinţă, bus-ul sistem este utilizat mai eficient şi se obţine o creştere a performanţelor sistemului.

6.1.1 Unitatea de interfaţă cu busul

BIU asigură toate semnalele necesare desfăşurării ciclurilor de magistrală. Această unitate realizează legătura dintre microprocesor şi lumea exterioară. Sarcinile acestei unităţi sînt:

- aducerea instrucţiunilor din memorie şi plasarea acestora în coada de instrucţiuni;
- gestionarea cozii de instrucţiuni;
- realocarea adreselor;
- controlul semnalelor de comandă.

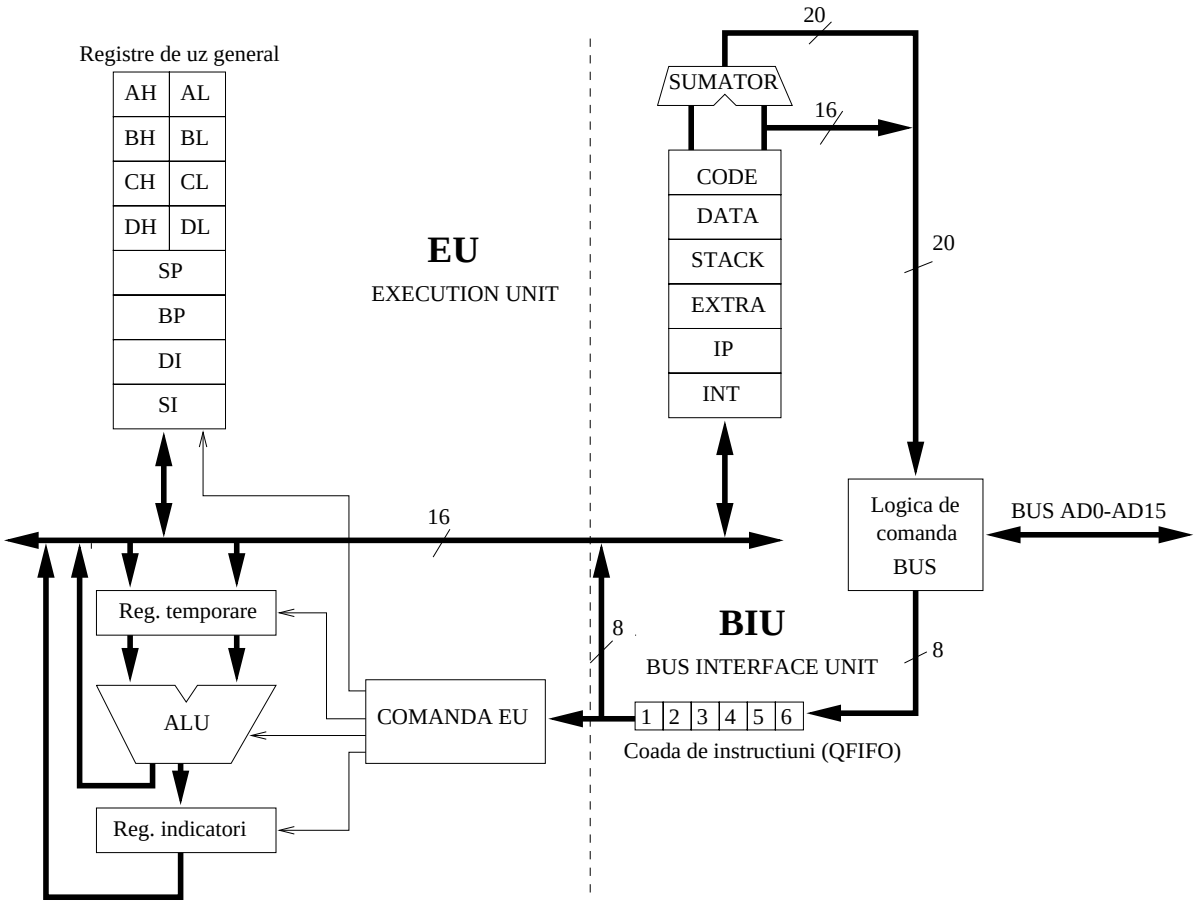


Figura 6.1: Arhitectura microprocesorului 8086.

Pentru implementarea acestor funcţii, *BIU* conţine:

- registre de comunicaţie internă;
- registre pentru segmentarea memoriei;
- sumator de adrese;
- registru indicator de program;
- memorie QFIFO pentru păstrarea instrucţiunilor în aşteptarea execuţiei;
- un bloc pentru controlul semnalelor de comandă.

Suportul fizic al *BIU* permite implementarea unei structuri "pipeline". Dacă la un moment dat în memoria QFIFO există cel puţin doi octeţi liberi şi dacă *EU* nu cere un acces la magistrală, atunci *BIU* face un acces la memorie, aducînd doi octeţi. Este posibilă astfel aducerea în avans (*prefetch*) a maximum şase octeţi din programul executat. Octeţii extraşi din memorie sînt introduşi, octet cu octet, în memoria QFIFO prin capătul de intrare. Conţinutul memoriei QFIFO este utilizat de *EU* prin capătul de ieşire, ceea ce determină deplasarea automată a octeţilor spre ieşire, cu o poziţie după citirea fiecărui octet. Dacă memoria QFIFO nu are la un moment dat nici un bloc liber şi nici *EU* nu cere un acces prin magistrala externă, atunci *BIU* nu iniţiază nici un ciclu al magistralei, ceea ce implică o stare de inactivitate (*idle state*)

a acesteia. Dacă *BIU* a iniţiat un ciclu iar *EU* cere un acces, va fi încheiat ciclul de *BIU* după care se va ceda accesul pentru *EU*.

Pentru generarea adresei de acces la o resursă externă, *BIU* trimite pe magistrala externă o adresă fizică de 20 biţi. Adresa fizică se formează prin însumarea conţinutul unui registru segment de 16 biţi, deplasat la stînga cu patru poziţii, cu un deplasament de 16 biţi, primit de la *EU*. De exemplu, dacă CS = 7100H, iar IP = 9002H, atunci următoarea instrucţiune va fi citită de la adresa: 71000H + 9002H = 7A002H. *BIU* generează şi semnale de comandă (scriere/citire memorie, scriere/citire port, etc.) necesare pentru desfăşurarea unui acces la o resursă externă.

6.1.2 Unitatea de execuţie

EU citeşte din capătul de ieşire al QFIFO octeţii care aparţin unei instrucţiuni. Instrucţiunea adusă în *EU* este decodificată de o unitate specială. Dacă instrucţiunea necesită acces la memorie, se generează către *BIU* un deplasament. Simultan, se transmit şi informaţii care identifică tipul accesului la magistrala externă (citire/scriere din/în memorie/port). După execuţia instrucţiunii, *EU* actualizează starea indicatorilor şi aşteaptă ca următoarea instrucţiune să fie disponibilă în memoria QFIFO. Datorită faptului că *BIU* foloseşte fiecare moment în care magistrala este liberă pentru a încărca memoria QFIFO, este foarte probabil ca în această memorie să se găsească instrucţiuni care urmează a fi executate. Astfel, viteza de prelucrare a informaţiilor de către microprocesorul 8086 este crescută pe baza unui hardware intern suplimentar şi nu prin intermediul creşterii frecvenţei de tact. După execuţia unei instrucţiuni de salt sau ramificaţie, memoria QFIFO este descărcată, pierzînd instrucţiunile care s-ar fi executat dacă n-ar fi fost executată instrucţiunea de salt. Rezultă o concluzie importantă: pentru ca programele să fie executate cît mai rapid este necesar să fie eliminate, pe cît posibil, instrucţiunile ce implică saltul la altă secvenţă de instrucţiuni.

6.1.3 Registrele microprocesorului

Procesorul 8086 are trei grupe de registre interne accesibile utilizatorului:

- *8 registre generale de date*,
- *4 registre segment*,
- *2 registre de stare şi control*.

Registrele generale de date sînt în număr de 8 şi sînt folosite pentru memorarea temporară a unor informaţii. Cele 8 registre fac parte din două categorii:

- *registre de date* şi
- *registre pointer şi index*.

Fiecare *registru de date* (tabelul 6.1) poate fi referit ca registru cuvînt (16 biţi) sau ca două registre semicuvînt (1 byte, 8 biţi). Sufixele H şi L desemnează partea superioară (High) sau inferioară (Low) a registrului de 16 biţi.

Denumire registru general de date	*16 biţi*	*8 biţi*	*Denumire în limba engleză*	*Semnificaţie*
A	AX	AH, AL	Accumulator	acumulator
B	BX	BH, BL	Base	bază
C	CX	CH, CL	Counter	numărător
D	DX	DH, DL	Data	date

Tabelul 6.1: Denumirile registrelor generale de date.

Registrul	*Operaţii*
AX	Înmulţire cuvînt, împărţire cuvînt, I/O cuvînt
AL	Înmulţire byte, împărţire byte, I/O byte, conversie aritmetică zecimală
AH	Înmulţire byte, împărţire byte
BX	Conversie
CX	Operaţii cu şiruri; cicluri
CL	Deplasare şi rotire variabilă
DX	Înmulţire cuvînt, împărţire cuvînt, I/O indirectă

Tabelul 6.2: Utilizarea implicită a registrelor generale de date.

Registrele de date pot fi utilizate pentru operaţii aritmetice şi logice, dar există instrucţiuni care presupun implicit utilizarea anumitor registre (tabelul 6.2).

Registrele indicator (pointer) şi index au funcţii dedicate. Cele două registre pointer se pot accesa numai ca registre de 16 biţi. Aceste registre sînt folosite pentru a păstra deplasamentul (offset) relativ la registrele de segment, în operaţiile de acces la memorie. *Registrul indicator de stivă* (Stack Pointer = SP) asigură accesul la segmentul de memorie definit ca stivă de SS şi indică deplasamentul ultimei locaţii de memorie care a fost implicată într-o operaţie cu stiva. După o asemenea operaţie, registrul SP este automat modificat (nefiind necesară intervenţia programatorului) indicînd în permanenţă vîrful stivei relativ la baza segmentului de stivă. *Registrul indicator de bază* (Base Pointer = BP) conţine un deplasament faţă de originea stivei şi se foloseşte pentru a accesa date din cadrul stivei.

Registrele index conţin deplasamentul faţă de originea segmentului de date definit de DS. *Registru index al sursei* (Source Index = SI) şi *registrul index al destinaţiei* (Destination Index = DI) sînt folosite implicit pentru calculul adresei operandului sursă, respectiv destinaţie, în instrucţiunile care apelează la adresarea indexată.

Spre deosebire de registrele de uz general, registrele pointer şi index, se accesează numai la nivel de cuvînt, fiind imposibil accesul la nivel de octet. Aceste registre pot fi implicate în operaţii aritmetice şi logice.

Registrele de segment oferă suport pentru implementarea unei memorii cu un spaţiu de adresare de 1 MB. Spaţiul de adresare este împărţit în segmente de cîte 64 KB, din care, la un moment dat, numai 4 segmente pot fi active. Segmentele active sînt definite prin registrele de segment:

- CS (*Code segment*) segmentul de cod/program,

- DS (*Data segment*) segmentul de date,
- SS (*Stack segment*) segmentul de stivă,
- ES (*Extra segment*) segmentul de date suplimentar.

Fiecare din aceste registre conţine adresa de început a segmentului de 64 KB pe care îl defineşte, adresă care poate fi modificată sub acţiunea programului rulat. Încărcarea registrului de cod CS este echivalentă cu transferul controlului programului în alt segment de memorie. Pentru cazuri excepţionale, în care este necesară referirea la un operand aflat în afara segmentului implicit, se poate include în instrucţiune un prefix care desemnează explicit segmentul în care se găseşte operandul.

Registrele de stare şi control constau din registrul indicator de instrucţiuni şi registrul de stare (flag-uri).

Indicatorul de instrucţiuni (Instruction Pointer = IP) este un registru de 16 biţi care identifică locaţia următoarei instrucţiuni ce va fi executată, în segmentul de cod curent. IP este similar unui registru contor program (Program Counter = PC). Spre deosebire de un registru PC care conţine adresa fizică a următoarei instrucţiuni, IP conţine deplasamentul de 16 biţi al următoarei instrucţiuni. Pentru determinarea adresei fizice, deplasamentul trebuie combinat cu conţinutul unui registrului segment de cod (CS), pentru a genera adresa fizică a instrucţiunii. Fizic, acest registru se află în BIU. La fiecare aducere a unei instrucţiuni din memorie, BIU actualizează valoarea registrului IP pentru a indica următoarea instrucţiune.

Registrul de stare şi indicatori este un registru de 16 biţi aflat în EU. Din cei 16 biţi, numai 9 biţi au o semnificaţie: 3 biţi de control şi 6 biţi de stare.

Structura registrului de stare şi indicatori este prezentată în tabelul 6.3.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
*	*	*	*	OF	DF	IF	TF	SF	ZF	*	AF	*	PF	*	CF

Tabelul 6.3: Structura registrului de stare şi indicatori.

Indicatorii de stare sînt modificaţi ca urmare a execuţiei unor anumite instrucţiuni (aritmetice, logice sau special destinate), iar conţinutul lor nu poate fi citit direct, ci numai testat indirect, prin transferul controlului programului la ramuri diferite, în funcţie de valoarea indicatorului testat.

Indicatorii de stare sînt:

Indicator de transport (Carry Flag = CF), care este setat după o operaţie aritmetică în care a apărut transport sau împrumut; în rest valoarea acestui indicator este "0";

Indicator de paritate (Parity Flag = PF), care este setat dacă rezultatul unei operaţii aritmetice conţine un număr par de biţi de valoare "1"; dacă numărul este impar, atunci PF=0;

Indicator de transport auxiliar (Auxiliary Carry Flag = AF), care este setat dacă după o operaţie aritmetică pe 16 biţi a apărut transport/împrumut în/din tetrada mai semnificativă a octetului mai puţin semnificativ dintr-un cuvînt (16 biţi);

Indicator de zero (Zero Flag = ZF), care este setat dacă în urma unei operaţii logice sau aritmetice toţi biţii rezultatului sînt egali cu zero;

Indicator de semn (Sign Flag = SF), care în urma unei operaţii logice sau aritmetice ia valoarea celui mai semnificativ bit (MSB); astfel, dacă rezultatul este negativ, atunci SF = MSB = 1. SF = 0 pentru restul situaţiilor;

Indicator de depăşire (Overflow Flag = OF), care este setat dacă, în cazul a doi operanzi de acelaşi semn, semnul rezultatului este diferit de cel al operanzilor, ceea ce este echivalent cu faptul că rezultatul este în afara domeniului de reprezentare.

Exemplul 1:

În urma operaţiei de adunare pe octet a numerelor fără semn $199|_{10} = C7H$ şi $90|_{10} = 5AH$, rezultatul operaţiei este $199|_{10} + 90|_{10} = C7H + 5AH = 21H = 33|_{10}$.

```
  1100 0111 +
  0101 1010
-----------
1 0010 0001
```

CF=1 (pentru că a apărut transport), PF=1 (pentru că rezultatul 21H = 0010 0001 are un număr par de biţi egali cu unu), AF=1 (pentru că a apărut un transport în tetrada mai semnificativă), ZF=0 (pentru că rezultatul este diferit de zero), SF = MSB = 0, OF = 0. Numerele fiind considerate fără semn, SF şi OF nu au nici o semnificaţie în acest caz.

Exemplul 2:

Considerînd numerele cu semn $-57|_{10} = C7H$ şi $+90|_{10} = 5AH$, rezultatul adunării şi valoarea biţilor de stare sînt aceleaşi cu deosebirea că CF şi AF nu au nici o semnificaţie, iar $SF = 0$ indică faptul că rezultatul este corect, deci nu s-a produs depăşire.

Indicatorii de control sînt:

Indicatorul de întrerupere (Interrupt Flag = IF), care este utilizat pentru determinarea modului în care microprocesorul 8086 reacţionează la cererile de întrerupere mascabilă aplicate pe pinul INT (IF = 1 - cererile de întrerupere sînt validate). Starea acestui indicator poate fi setată/resetată prin program;

Indicatorul de direcţie (Direction Flag = DF), care este utilizat în operaţiile cu şiruri de octeţi cînd este necesar ca şirul să fie definit prin adresa de bază şi prin sensul descrescător (DF=0) sau crescător (DF=1) al adreselor care definesc octeţii din şir faţă de adresa de bază;

Indicatorul de mod "pas cu pas" (Trap Flag = TF), care permite implementarea unui mod de funcţionare util în depanarea programelor. Astfel, dacă TF = 1, după execuţia fiecărei instrucţiuni se iniţiază o întrerupere care determină transferul controlului într-o zonă de memorie definită de utilizator, unde, sub acţiunea unui program adecvat, se poate face analiza stării interne a microprocesorului.

6.1.4 Structura memoriei

Spaţiul de adresare al microprocesorului 8086 este de 1 MB. Datele aflate la o adresă de memorie au 8 biţi. Un cuvînt de 16 biţi poate fi stocat în memorie la două adrese succesive. Ordinea de

stocare a celor doi baiţi aparţinînd unui cuvînt este cu cel mai semnificativ la adresa superioară ("little endian"). Spaţiul de adresare de 1 MB al microprocesorului 8086 necesită 20 de biţi de adresă. Pentru a rezolva accesibilitatea resurselor de 1 MB utilizînd cuvinte de cîte 16 biţi s-a adoptat segmentarea memoriei în pachete a cîte 2^{16} B = 64 KB. Memoria poate fi interpretată ca un număr oarecare de segmente avînd fiecare maximum 64 KB. Adresa de început a unui segment este divizibilă cu 16 (ultimii patru biţi sînt 0). Un program poate avea acces imediat numai în patru segmente:

- segmentul de cod/program,
- segmentul de date,
- segmentul de stivă,
- segmentul de date suplimentar.

Selecţia acestor patru segmente a fost motivul pentru care unitatea *BIU* a lui 8086 a fost prevăzută cu registre segment de 16 biţi.

Pentru generarea unei adrese fizice în spaţiul de 1 MB, conţinutul unui registru de segment este deplasat la stînga cu 4 poziţii şi adunat cu conţinutul unuia din registrele pointer sau index. Rămîne la latitudinea programatorului segmentarea memoriei disponibile apelînd la segmente disjuncte, parţial disjuncte sau suprapuse, neexistînd nici o restricţie în acest sens. Determinarea adresei fizice din adresa logică este realizată de *BIU* aşa ca în figura 6.2.

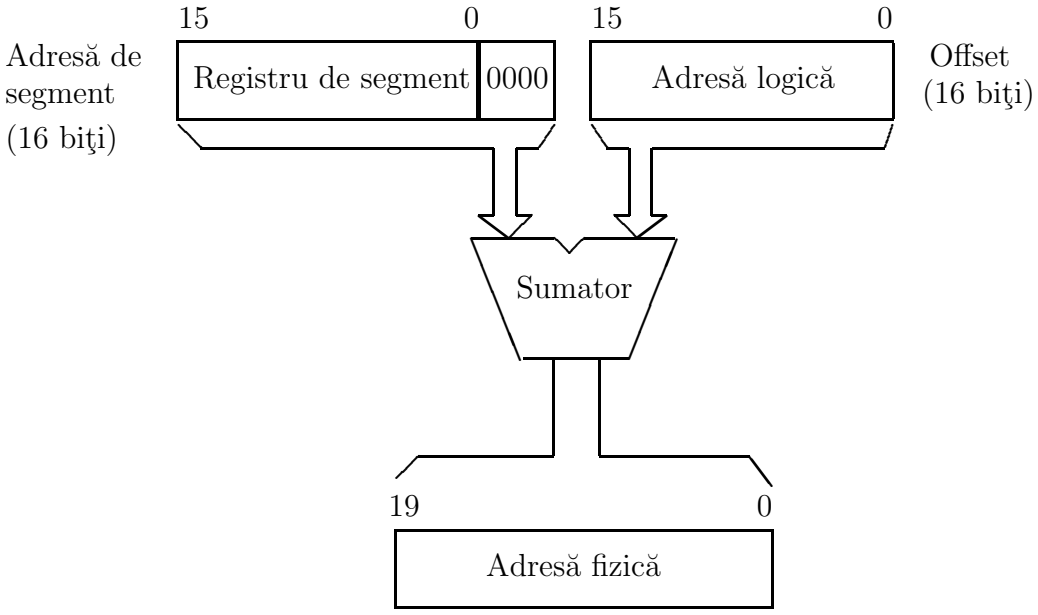


Figura 6.2: Determinarea adresei fizice.

BIU obţine adresele logice şi adresele de segment din diferite surse, în funcţie de tipul referinţei la memorie. Tabelul 6.4 prezintă sursele implicite pentru diferite tipuri de acces la memorie.

Segmentele de cod (CS) ale memoriei conţin instrucţiuni ale unuia sau mai multor programe. Conţinutul lui CS identifică locaţia de început a unei instrucţiuni în cadrul segmentului de cod activ. Pentru a face acces la locaţia unde se memorează instrucţiunea în segmentul de cod

Tipul referinţei la memorie	*Segment implicit*	*Segment alternativ*	*Deplasament (Offset)*
Fetch instrucţiune	CS	-	IP
Operaţie cu stiva	SS	-	SP
Accesare de variabilă	DS	CS, ES, SS	Adresă efectivă
Sursă de operaţie cu şiruri	DS	CS, ES, SS	SI
Destinaţie de operaţie cu şiruri	ES	-	DI
Registrul BP utilizat ca registru de bază	SS	CS, DS, ES	Adresă efectivă
Registrul BX utilizat ca registru de bază	DS	CS, ES, SS	Adresă efectivă

Tabelul 6.4: Sursele implicite pentru diferite tipuri de acces la memorie.

activ, 8086 trebuie să-şi poziţioneze toţi cei 20 biţi de adresă fizică. Pentru a realiza acest lucru, microprocesorul 8086 combină conţinutul registrului pointer de instrucţiuni (IP) cu valoarea din CS, rezultînd o adresă fizică ce va fi furnizată pe ieşirile de adresă ale microprocesorului. Segmentul de cod activ poate fi schimbat prin execuţia unei instrucţiuni care încarcă o nouă valoare în registrul CS. Din acest motiv, se poate utiliza pentru memorarea codului oricare din cele 16 segmente independente de memorie a cîte 64 KB.

Conţinutul registrului segment de date (DS) identifică locaţia de început a segmentului de date curent în memorie. Acesta este cel de-al doilea segment activ de 64 KB, care se constituie într-un spaţiu de memorie tip citire/scriere în care datele pot fi stocate. Majoritatea operanzilor instrucţiunilor sînt preluaţi din acest segment. Valorile din registrele index sursă (SI) sau destinaţie (DI) sînt combinate cu valoarea din (DS) pentru a forma o adresă fizică pe 20 de biţi (adresa operandului sursă sau destinaţie în segmentul de date).

Registrul segment stivă (SS) conţine adresa logică ce identifică locaţia de început a stivei curente în memorie. Este un segment de 64 KB în care sînt depuse valorile pointerului de instrucţiuni (IP), indicatorilor de stare şi ale altor registre (printr-o instrucţiune PUSH) la orice întrerupere hardware, întrerupere software sau execuţia unui apel de subrutină. După ce s-a încheiat execuţia subrutinei, starea originală a sistemului este restaurată prin executia instrucţiunii POP sau prin execuţia instrucţiunii RETURN. Următoarea locaţie în care va fi depus un cuvînt sau din care va fi preluat un cuvînt se obţine prin combinarea valorii curente din SS cu pointerul de stivă (SP). Stiva creşte spre valori de adrese descrescătoare. Cuvintele din stivă au 16 biţi (2 bytes).

Ultimul registru segment identifică cel de-al patrulea segment activ de 64 KB în spaţiul de adresare a memoriei. Acest spaţiu este denumit segment suplimentar (ES). Acest segment este folosit uzual pentru memorarea datelor. Instrucţiunile cu şiruri utilizează valoarea din ES cu conţinutul din DI drept un deplasament pentru a identifica adresa destinaţie.

6.1.5 Organizarea porturilor I/O

Spaţiul de adresare al porturilor I/O este disjunct faţă de spaţiul de adresare al memoriei. Adresarea porturilor se face cu instrucţiuni specifice (IN sau OUT).

Spaţiul de adresare al porturilor poate fi văzut ca fiind de dimensiune $64K \times 8$ biţi sau $32K \times 16$ biţi. Adresarea primelor 256 de porturi se poate face direct (adresa portului inclusă în codul

instrucţiunii). Toate porturile pot fi adresate indirect, adresa de 16 biţi fiind plasată în registrul implicit DX.

Instrucţiunile IN şi OUT transferă date între acumulator (AL pentru transferuri pe 8 biţi şi AX pentru transferuri pe 16 biţi) şi portul localizat în spaţiul I/O.

6.1.6 Modurile de adresare

Operanzii instrucţiunilor microprocesorului 8086 pot fi conţinuţi în registre, într-un cîmp al instrucţiunii, în memorie sau la porturile I/O. Adresele la care se găsesc operanzii în memorie sau la porturile I/O se pot determina în mai multe feluri.

Operanzii imediaţi sînt constante de 8 sau 16 biţi incluse în codul instrucţiunii. Accesarea acestora se face în faza de aducere a instrucţiunii în memorie (fetch) şi nu necesită un ciclu suplimentar de citire a memoriei.

Operanzi imediaţi (operandul este în corpul instrucţiunii)

```
CR   EQU  13
MOV  AL, CR         ; iniţializează registrul AL cu 13
MOV  AX, 0FFFFH     ; iniţializează registrul AX cu o dată imediată
```

Instrucţiunile cu *operanzi din registre* sînt executate mai rapid deoarece aceste operaţii nu fac acces la resurse din afara EU.

Operanzi în registre (corpul instrucţiunii conţine codul unui registru)

```
MOV  AL, BL         ; copiază conţinutul registrului BL în registrul AL
MOV  AX, BX         ; copiază conţinutul registrului BX în registrul AX
```

Spre deosebire de operanzii imediaţi şi de cei din registre, care sînt direct accesaţi de către EU, *operanzii din memorie* trebuie transferaţi între CPU şi memorie prin intermediul BIU şi al bus-ului extern. EU calculează *adresa efectivă* (Effective Address = EA) a operandului şi o transmite către BIU. EA este de tipul întreg reprezentat pe 16 biţi şi reprezintă offsetul operandului în segmentul în care se face adresarea. Pentru calcularea adresei efective, EU foloseşte informaţia din cel de-al doilea byte al instrucţiunii. Codul acestui byte este dedus de către compilator sau asamblor pe baza instrucţiunii scrise de către programator. În limbaj de asamblare se poate accesa memoria în toate modurile de adresare.

Adresare directă EA = $deplasament_{8|16}$

```
LOC1 DB 13          ; defineşte o variabilă de tip byte
                    ; şi o iniţializează cu 13 (baza 10)
MOV AL, LOC1        ; transferă în AL data de 8 biţi aflată
                    ; în memorie la adresa LOC1
```

Adresare indirectă la registru EA = $[BX|BP|SI|DI]$

```
MOV AX, [BX]        ; transferă în AX data de 16 biţi aflată
                    ; în memorie la adresa conţinută în registrul BX
```

Adresare la bază EA $= [BX|BP] + deplasament_{8|16}$

```
MOV AX, [BX+2]     ; transferă în AX data de 16 biţi aflată în
                   ; memorie la adresa obţinută prin însumarea
                   ; conţinutului registrului BX cu 2
```

Adresare indexată EA $= [SI|DI] + deplasament_{8|16}$

```
MOV AX, [SI+2]     ; transferă în AX data de 16 biţi aflată
                   ; în memorie la adresa obţinută prin însumarea
                   ; conţinutului registrului SI cu 2
```

Adresare la bază indexată EA $= [BX|BP] + [SI|DI] + deplasament_{8|16}$

```
MOV AX, [BX+SI+2] ; transferă în AX data de 16 biţi aflată
                   ; în memorie la adresa obţinută prin însumarea
                   ; conţinutului registrului BX cu conţinutul
                   ; registrului SI şi cu 2
```

Adresarea şirurilor EAsursă $= [SI]$, EAdestinaţie $= [DI]$

```
s1  DB 'şir1'      ; defineşte două variabile de tip
s2  DB 'şir2'      ; byte şi le iniţializează cu cîte patru caractere
MOV SI, offset s1 ; pregăteşte registrele implicite ale
MOV DI, offset s2 ; instrucţiunii care operează asupra
MOV CX, 4          ; şirurilor de caractere

REP MOVSB          ; instrucţiune care mută un şir de
                   ; caractere adresat implicit, de la DS:SI la ES:DI
```

Adresarea porturilor I/O Adresă $\text{Port}_8 = data_8$, Adresă $\text{Port}_{16} = [DX]$

```
IN AL, 0EAH        ; citeşte de la portul adresat direct un cuvînt de
                   ; 8 biţi şi îl transmite în registrul AL

MOV DX, 0ABCDH     ; pregăteşte în registrul DX adresa
                   ; portului reprezentată pe 16 biţi
IN AX, DX          ; citeşte de la portul a cărui adresă este
                   ; conţinută în registrul DX un cuvînt de 16 biţi
                   ; şi îl transmite în registrul AX
```

6.2 Conexiunile externe ale microprocesorului

Firma Intel a redus numărul de conexiuni externe, prezentînd microprocesorul 8086 într-o capsulă cu 40 de pini.

6.2.1 Modurile de lucru ale microprocesorului

Soluţia de încapsulare aleasă a impus multiplexarea în timp a funcţiilor mai multor pini (figura 6.3) cu implicaţii directe în realizarea circuitelor (schemelor) tipice cu acest microprocesor. Întrevăzînd faptul că 8086 va fi înglobat atît în configuraţii simple cît şi foarte complexe, firma Intel a prevăzut două moduri de lucru:

- modul *minim* şi
- modul *maxim*.

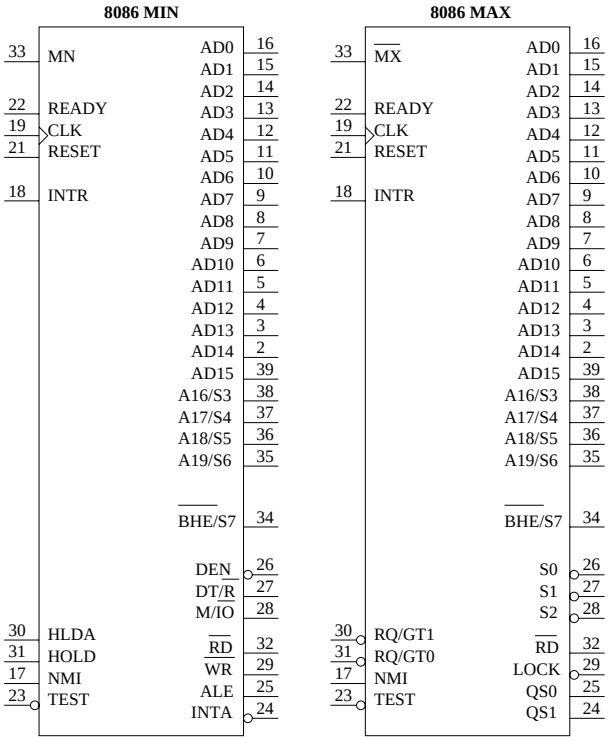


Figura 6.3: Configuraţia pinilor microprocesorului 8086.

Stabilirea modului de lucru se face prin impunerea unei stări logice pinului MN/MX, astfel:

- MN/MX = 1 pentru modul minim, care este potrivit sistemelor simple, cu un singur microprocesor;
- MN/MX = 0 pentru modul maxim, care se pretează la configuraţii multiprocesor.

În acord cu modul de lucru ales, destinaţia unor pini este diferită, ceea ce impune împărţirea pinilor în două categorii: *pini cu funcţii comune ambelor moduri de lucru* şi *pini cu funcţii specifice modurilor de lucru minim şi maxim*. Tabelul 6.5 prezintă semnificaţia pinilor microprocesorului 8086.

6.2.2 Modul minim

În modul minim, microprocesorul 8086 generează şi acceptă toate semnalele necesare pentru a realiza un sistem simplu, avînd totuşi facilităţi importante (lucru cu memoria sau cu porturile, în mod DMA sau prin întreruperi).

Semnale comune		
Nume	*Funcţie*	*Tip*
AD15-AD0	Bus adrese/date	Bidir. 3-Stări
A19/S6-A16/S3	Adrese/Stări	Ieşire, 3-Stări
NBHE/S7	Validare bus superior/stare	Ieşire, 3-Stări
MN/NMX	Control mod minim/maxim	Intrare
NRD	Control citire	Intrare, 3-Stări
NTEST	Wait pe control test	Intrare
READY	Control stare wait	Intrare
RESET	Reset sistem	Intrare
NMI	Cerere întrerupere nemascabilă	Intrare
INTR	Cerere întrerupere mascabilă	Intrare
CLK	Tact sistem	Intrare
VCC	+5V	Intrare
GND	Masa	Intrare
Semnalele modului minim (MN/NMX = VCC)		
Nume	*Funcţie*	*Tip*
HOLD	Cerere hold	Intrare
HLDA	Acceptare hold	Ieşire
NWR	Control scriere	Ieşire, 3-Stări
M/NIO	Control memorie/IO	Ieşire, 3-Stări
DT/NR	Emisie/recepţie date	Ieşire, 3-Stări
NDEN	Validare date	Ieşire, 3-Stări
ALE	Validare preluare adrese	Ieşire
NINTA	Recunoaştere întrerupere	Ieşire
Semnalele modului maxim (MN/NMX = GND)		
Nume	*Funcţie*	*Tip*
NRQ/NGT1,0	Control acces bus Cerere/cedare	Bidirecţional
NLOCK	Control lock prioritate bus	Ieşire, 3-Stări
NS2-S0	Stare ciclu de bus	Ieşire, 3-Stări
NQS1, NQS0	Starea cozii de instrucţiuni	Ieşire

Tabelul 6.5: Semnificaţia pinilor microprocesorului 8086.

S0	*S1*	*S2*	*Tip ciclu de bus*
0	0	0	Confirmare întrerupere (INTA)
0	0	1	Citire port I/O
0	1	0	Scriere port I/O
0	1	1	Stare de "halt"
1	0	0	Fetch instrucţiune
1	0	1	Citire memorie
1	1	0	Scriere memorie
1	1	1	Pasiv, nici un ciclu de bus

Tabelul 6.6: Tipul ciclului de bus, în funcţie de biţii de stare S0, S1 şi S2.

Magistrala de adrese şi date - AD0..AD15, A16..A19
Magistrala de adrese îndeplineşte, prin multiplexare în timp, două funcţii.

Magistrală de adrese de 20 biţi. În ciclurile I/O, A16=A17=A18=A19='0' ceea ce limitează la $2^{16} = 65536$ adresele disponibile pentru dispozitivele periferice.

Magistrală de date de 16 biţi (D15-D0). În acest caz, magistrala de date D0..D15 este bidirecţională.

Procesorul 8086 are posibilitatea de a trece liniile AD0..AD15, A16..A19 în starea de înaltă impedanţă (3-state).

Semnale de stare
Semnalele S0, S1 şi S2 prezintă informaţia referitoare la tipul ciclului de magistrală curent. Starea prezentată de aceşti biţi este validă la începutul fiecărui ciclu de bus. Tabelul 6.6 prezintă modul de interpretare a informaţiei furnizate de aceste semnale.

Prin multiplexare, liniile A16..A19 îndeplinesc şi o funcţie de stare. Prin prisma acestui dublu rol, denumirea corectă a magistralei este A16/S3..A19/S6. Informaţia de stare S3..S6 este prezentă simultan cu situaţia cînd informaţia de pe liniile de date este stabilă. Semnalele de pe magistrală au următoarea semnificaţie:

- S3 şi S4 formează un cod ce reprezintă registrul segment utilizat pentru generarea adresei din ciclul curent de magistrală (tabelul 6.7);
- S5 reprezintă valoarea flagului IF (Interrupt Flag) de autorizare a întreruperilor mascabile;
- S6 este neutilizat, fiind întotdeauna egal cu "0".

Semnale de control

Semnalele de control permit interconectarea cu memoria şi dispozitivele periferice. Semnificaţia biţilor de control este dependentă de tipul informaţiei existente pe magistrala D0..D15:

- adresă validă (ieşire);
- date de intrare, într-un ciclul de citire;

S4	*S3*	*Registru segment*
0	0	ES (segment de date auxiliar)
0	1	SS (segment de stivă)
1	0	CS sau nimic (I/O sau vector de întrerupere)
1	1	DS (segment de date)

Tabelul 6.7: Informaţia codificată cu biţii de stare S3 şi S4.

- date de ieşire, într-un ciclu de scriere.

ALE (Address Latch Enable) este un semnal activ în "1", care indică faptul că magistrala AD0..AD15 conţine o informaţie de adresă validă referitoare la ciclul aflat în execuţie. Întrucît această adresă este validă numai pe perioada cît ALE="1", ea trebuie memorată pe frontul căzător al semnalului ALE (în circuite "latch") pentru a fi folosită ulterior.

BHE (Bank High Enable) este un semnal activ în "0", folosit pentru a activa modulul de memorie legat de magistrala de date, în jumătatea mai semnificativă (D8..D15). De asemenea, BHE poate fi folosit ca semnal de stare S7.

M/IO (Memory/Input-Output) este un semnal care se formează pe baza tipului ciclului aflat în execuţie şi anume:

- M/IO="1" pentru acces la memorie;
- M/IO="0" pentru acces la port.

DT/R (Data Transmit/Receiver) indică dacă ciclul curent este unul de scriere, cînd datele sînt transmise de către microprocesor (DT/R="1"), sau unul de citire, cînd datele sînt citite de către microprocesor (DT/R="0"). Citirea şi scrierea datelor se referă atît la accesul la memorie cît şi la porturi. Semnalul DT/R poate fi folosit pentru schimbarea sensului de transfer al unui circuit bidirecţional de amplificare a magistralei de date.

RD (Read) este generat într-un ciclu de citire, fiind folosit pentru selectarea dispozitivelor logice, care oferă informaţia citită de către microprocesor.

WR (Write) este generat într-un ciclu de scriere semnalizînd că pe AD0..AD15 se află o dată validă, care trebuie înscrisă în memorie sau port. Semnalele RD şi WR pot fi prelungite nelimitat, oferind astfel posibilitatea ca memorii sau porturi mai lente să poată fi folosite cu microprocesorul 8086.

DEN (Data Enable) este generat atît în ciclurile de citire cît şi în cele de scriere, cînd este necesară marcarea momentului în care magistrala comună îndeplineşte funcţia de date (D0..D15). Semnalul DEN poate fi folosit pentru selecţia circuitelor 3-state de amplificare a magistralei de date.

READY este un semnal primit de microprocesor, la activarea căruia microprocesorul îşi prelungeşte ciclurile de magistrală şi implicit semnalele de comandă pentru resursele hardware lente.

Semnale de întrerupere

Întreruperea, în sens general, este devierea unui program principal prin comanda unui semnal exterior (întrerupere hardware), ca urmare a unei stări interne (întrerupere internă) sau ca

Registru	*Conţinut*
Registru de stare şi indicatori	0000H
IP	0000H
CS	FFFFH
DS	0000H
SS	0000H
ES	0000H
Coadă de instrucţiuni	Goală

Tabelul 6.8: Starea registrelor după resetare.

urmare a execuţiei unor instrucţiuni specifice (întrerupere software). Devierea este urmată de o tratare specifică şi revenirea în starea iniţială, în care a fost făcută întreruperea.

INTR (Interrupt) este un semnal prin care un dispozitiv periferic cere microprocesorului să îşi întrerupă temporar activitatea pentru a-l deservi. Acest semnal este testat de către microprocesor în ultima perioadă de ceas a fiecărui ciclu de aducere a instrucţiunii (fetch). Dacă semnalul are valoarea "1" şi întreruperile sînt validate, se iniţiază un ciclu de recunoaştere a întreruperii, cînd INTA devine activ ("0").

TEST poate determina, în anumite condiţii, suspendarea unui program. După execuţia instrucţiunii WAIT, microprocesorul 8086 testează starea liniei TEST. Dacă aceasta are valoarea "1" se trece magistrala în inactivitate şi se aşteaptă revenirea liniei TEST la valoarea "0". Programul este reluat din punctul de întrerupere. Prin utilizarea semnalului TEST se poate implementa un mod simplu de a sincroniza activitatea microprocesorului cu evenimente externe.

NMI (Non Maskable Interrupt) poate determina, pe frontul crescător, declanşarea unei întreruperi care nu poate fi invalidată. Activarea NMI determină necondiţionat execuţia unei rutine speciale de tratare a întreruperii. NMI poate fi folosit pentru iniţierea unor comenzi în cazul scăderii tensiunii de alimentare sau pentru tratarea unor erori de paritate la memorie (ca la calculatorul IBM PC).

RESET este inclus tot în categoria semnalelor de întrerupere. Activarea sa (pe valoare "0"), determină aducerea registrelor şi a flag-urilor interne într-o stare iniţială, urmînd ca la inactivarea acestuia (revenirea la valoarea "1") să fie reluată activitatea microprocesorului din această stare. După resetarea microprocesorului registrele sînt iniţializate în stările prezentate în tabelul 6.8.

Semnale de interfaţă DMA

HOLD poate fi adus în "1" de către un dispozitiv care intenţionează să preia controlul asupra magistralelor de adrese, date şi comenzi. La activarea semnalului, după terminarea ciclului curent, microprocesorul 8086 trece în starea de înaltă impedanţă semnalele: AD0 .. AD15, A16/S3 .. A19/S6, BHE, DEN, DTR, MI/0, RD, WR şi INTR. Totodată, această stare (*hold state*) este semnalizată în exterior prin HLDA="1" (Hold Acknowledge), perioadă în care *EU* utilizează informaţia din QFIFO, pînă cînd aceasta rămîne vidă.

6.2.3 Modul maxim

Dacă microprocesorul 8086 este pus să lucreze în modul maxim, (prin fixarea pinului MN/MX la "0") el generează semnale pentru implementarea structurilor multiprocesor/coprocesor. Prin acest termen se definesc acele structuri hardware care presupun existenţa mai multor microprocesoare în cadrul aceleiaşi configuraţii sistem, în condiţiile în care fiecare microprocesor îşi execută, în cea mai mare parte a timpului, propriul său program. De obicei, asemenea structuri conţin anumite resurse globale, comune tuturor microprocesoarelor. Fiecare microprocesor poate avea şi resurse locale (private).

În sistemele cu coprocesor există un al doilea microprocesor în sistem. În acest caz, cele două microprocesoare nu pot face acces la bus în acelaşi timp. Unul dintre ele trebuie să transmită celuilalt controlul bus-ului sistem şi să-şi suspende pentru un timp operaţiile. În sisteme cu microprocesor 8086 cuplat în modul maxim se întîlnesc facilităţi suplimentare, în ceea ce priveşte implementarea alocării resurselor globale şi transmiterea controlului bus-ului altor microprocesoare sau coprocesoare.

6.3 Instrucţiunile microprocesorului

Instrucţiunile microprocesorului 8086 pot fi împărţite în următoarele categorii:

- Instrucţiuni pentru transfer de date;
- Instrucţiuni aritmetice;
- Instrucţiuni pentru manipulare de biţi;
- Instrucţiuni pentru manipulare de şiruri;
- Instrucţiuni pentru transferul controlului programului;
- Instrucţiuni pentru controlul microprocesorului.

În continuare se face o prezentare sumară a categoriilor de instrucţiuni. La fiecare categorie se va prezenta un tabel cu mnemonicile asociate instrucţiunilor şi cu definiţia instrucţiunii, aşa cum este dată de producător (în limba engleză). Anexa **??** prezintă mai detaliat fiecare instrucţiune în parte.

6.3.1 Instrucţiuni pentru transfer de date

Instrucţiunile pentru transfer de date (tabelul 6.9) mută (copiază) date reprezentate pe 8 sau 16 biţi între memorie şi registre sau între registre şi porturi I/O. Acest grup include şi instrucţiunile de manipulare a stivei şi instrucţiunile de încărcare a registrelor de segment.

6.3.2 Instrucţiuni aritmetice

Instrucţiunile aritmetice (tabelul 6.10) se pot executa asupra patru tipuri de date binare:

Transferuri generale	
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
PUSHA	Push all registers on stack
POPA	Pop all registers from stack
XCHG	Exchange byte or word
XLAT	Translate byte
Intrare/Ieşire	
IN	Input byte or word
OUT	Output byte or word
Adresare de obiecte	
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
Transferuri de indicatori	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags from stack

Tabelul 6.9: Instrucţiuni pentru transfer de date.

- numere binare întregi, fără semn (pozitive);
- numere binare întregi, cu semn;
- numere în baza 10, fără semn, neîmpachetate;
- numere în baza 10, fără semn, împachetate.

Numerele binare pot fi reprezentate pe 8 sau 16 biţi. Totdeauna microprocesorul presupune că operanzii specificaţi conţin date valide. Pentru numere binare fără semn există instrucţiuni de adunare, scădere, înmulţire şi împărţire.

6.3.3 Instrucţiuni pentru manipulare de biţi

Instrucţiunile pentru manipulare de biţi (tabelul 6.11) cuprind instrucţiunile de prelucrare logică, instrucţiunile de deplasare şi instrucţiunile de rotire.

6.3.4 Instrucţiuni pentru manipulare de şiruri

Instrucţiunile elementare pentru manipulare de şiruri (tabelul 6.12) operează asupra unui singur element de şir (8 sau 16 biţi). Prin utilizarea prefixelor de repetare, se pot realiza operaţii asupra unui număr de maxim 128 K elemente.

Instrucţiunile de manipulare de şiruri folosesc implicit următoarele registre şi indicatori:

Adunare	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Scădere	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
Înmulţire	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
Împărţire	
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte or word
CWD	Convert word to doubleword

Tabelul 6.10: Instrucţiuni aritmetice.

SI - offset şir sursă;

DI - offset şir destinaţie;

DX - adresă de port;

CX - numărător de repetiţii;

AL/AX - destinaţie/sursă pentru LODS/STOS;

DF - '0' - autoincrement SI, DI sau '1' - autodecrement SI, DI;

ZF - indicator folosit pentru determinarea sfîrşitului de şir.

6.3.5 Instrucţiuni pentru transferul controlului programului

Locul de unde se execută programul este determinat de conţinutul registrelor CS şi IP. Abaterea de la prelucrarea secvenţială a instrucţiunilor se realizează prin execuţia unor instrucţiuni care modifică valorile stocate în aceste registre (tabelul 6.13).

Logice	
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
Deplasare	
SHL/SAR	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
Rotire	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

Tabelul 6.11: Instrucţiuni pentru manipulare de biţi.

6.3.6 Instrucţiuni pentru controlul microprocesorului

Instrucţiunile pentru controlul microprocesorului (tabelul 6.14) permit controlarea prin program a diferitelor funcţii ale CPU. Această categorie include instrucţiunile de manipulare a indicatorilor şi cele pentru sincronizare externă.

6.4 Întrebări

I. Justificaţi prezenţa diferitelor grupe de registre în EU sau BIU.

II. Prezentaţi suportul oferit de *microprocesorul 8086* (hardware) pentru realizarea unor programe relocatabile dinamic (programe care pot fi încărcate în memorie la adrese

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
INS	Move byte or word string from I/O
OUTS	Move byte or word string to I/O
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

Tabelul 6.12: Instrucţiuni pentru manipulare de şiruri.

Transferuri necondiţionate	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
Transferuri condiţionate	
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
Controlul iteraţiilor	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX=0
Întreruperi	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

Tabelul 6.13: Instrucţiuni pentru transferul controlului programului.

Operaţii cu indicatori	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt flag
CLI	Clear interrupt flag
Sincronizări externe	
HLT	Halt until interrupt or reset
WAIT	Wait until TEST pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
Operaţie nulă	
NOP	No operation

Tabelul 6.14: Instrucţiuni pentru controlul microprocesorului.

aleatoare). Care sînt condiţiile pe care trebuie să le îndeplinească un *program* (software) pentru a fi dinamic relocatabil?

III. Se poate ca o procedură să implementeze o stivă diferită de cea a programului care o apelează? Daţi explicaţii referitoare la suportul *microprocesorului 8086* (hardware) şi la particularităţile *programelor* (software).

IV. Care este adresa fizică a primei instrucţiuni executate de microprocesorul 8086, după resetare? Decodificaţi datele existente în memorie la acea adresă şi deduceţi ce instrucţiuni execută un calculator PC imediat după resetare. Folosiţi utilitarul *debug*. Dacă se tastează '?' la prompterul '-' se obţine lista comenzilor, aşa cum este prezentată în continuare:

```
assemble     A [address]
compare      C range address
dump         D [range]
enter        E address [list]
fill         F range list
go           G [=address] [addresses]
hex          H value1 value2
input        I port
load         L [address] [drive] [firstsector] [number]
move         M range address
name         N [pathname] [arglist]
output       O port byte
proceed      P [=address] [number]
quit         Q
register     R [register]
search       S range list
trace        T [=address] [value]
unassemble   U [range]
```

```
write        W [address] [drive] [firstsector] [number]
allocate expanded memory        XA [#pages]
deallocate expanded memory      XD [handle]
map expanded memory pages       XM [Lpage] [Ppage] [handle]
display expanded memory status  XS
```

V. Realizaţi o schemă de calcul a *adresei efective* pentru fiecare mod de adresare al microprocesorului 8086. De exemplu, calculul adresei efective la adresarea bazată se face conform schemei din figura 6.4.

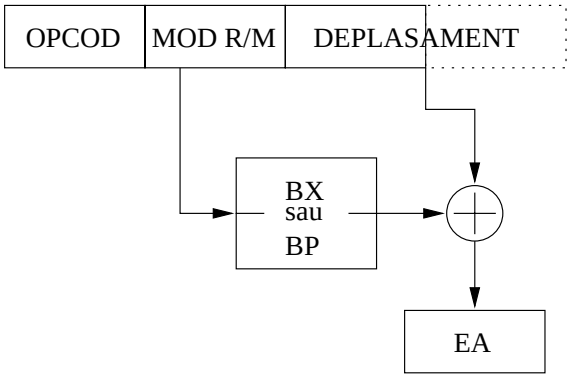


Figura 6.4: Calculul adresei efective la adresarea bazată.

VI. Faceţi o paralelă între modurile de adresare ale microprocesorului 8086 şi cele ale microprocesoarelor RISC.

VII. Referitor la setul de instrucţiuni al microprocesorului 8086 justificaţi atributele: *simplitate*, *ortogonalitate*, *completitudine*.

Lucrarea 7

Programarea în limbaj de asamblare 8086

Această lucrare prezintă etapele care trebuie urmate pentru realizarea unui program executabil pornind de la un cod sursă scris în limbaj de asamblare.

Există un sfat care zice că pentru a te face înţeles trebuie să vorbeşti fiecăruia pe limba lui. Dar care este "limba microprocesorului"? Microprocesorul codifică atît datele cît şi instrucţiunile în cifre binare $\{0, 1\}$. Deşi teoretic este posibil, astăzi nimeni nu mai concepe să "vorbească" (programeze) la un nivel atît de scăzut. Programarea într-un limbaj de nivel înalt (C, Pascal, Fortran) a devenit ceva natural. Limbajele de acest nivel au specificaţii foarte apropiate de modul de gîndire uman: decizii, iteraţii, prelucrări aritmetice şi logice cu scalari sau matrici, etc. Din păcate, cu cît limbajul este mai apropiat de limbajul uman, cu atît este mai depărtat de limbajul microprocesorului. Limbajele de nivel înalt permit gestionarea unor programe mari dar nu totdeauna permit gestionarea eficientă a resurselor hardware interne microprocesorului.

Limbajul de asamblare, spre deosebire de limbajul cifrelor binare, este un limbaj accesibil programatorului uman. Programul scris în limbaj de asamblare (*cod sursă*) nu este direct executabil de către microprocesor. Însă, fiecărei instrucţiuni în limbaj de asamblare îi corespunde o instrucţiune binară ce este "înţeleasă" (decodificată) şi executată de către microprocesor. Codul binar al operaţiei (*opcode*) este asociat cu un grup de litere sugestiv pentru efectul instrucţiunii respective (*mnemonică*).

Se poate face o analogie între limbajele vorbitorilor umani şi limbajele de asamblare ale microprocesoarelor. Limbajul de asamblare este propriu fiecărui tip de microprocesor în parte. Familii diferite de microprocesoare au seturi diferite de instrucţiuni. În cadrul unei familii, o generaţie mai nouă de microprocesoare preia setul de instrucţiuni al generaţiei vechi şi îl sporeşte, incluzînd instrucţiuni specifice gestionării noilor resurse hardware apărute. Dar, în general, un microprocesor de generaţie nouă poate rula şi aplicaţii scrise pentru microprocesoarele din generaţia anterioară. Există elemente comune ale seturilor de instrucţiuni ale diferitelor familii de microprocesoare.

7.1 Ciclul de dezvoltare al programelor scrise în limbaj de asamblare

Pentru a fi executat de către microprocesor, un program scris de un programator uman trebuie să parcurgă cîteva transformări. Reprezentarea grafică a acestor transformări este prezentată în figura 7.1.

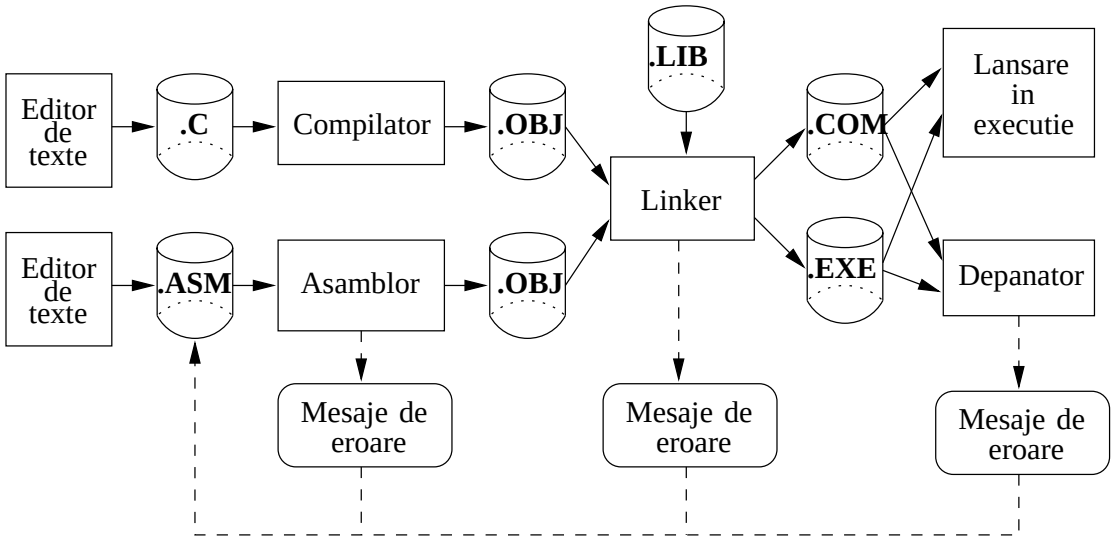


Figura 7.1: Reprezentarea grafică a ciclului de dezvoltare a programelor scrise în limbaj de asamblare

Pentru înţelegerea termenilor care apar în figura 7.1, se prezintă cîteva definiţii.

Limbaj de asamblare - limbaj de programare folosit de programatorul uman pentru a descrie un algoritm folosind setul de instrucţiuni propriu unui anumit microprocesor. Fiecare microprocesor are un limbaj de asamblare propriu, deşi se pot găsi multe elemente comune mai multor microprocesoare. Formatul instrucţiunilor în limbaj de asamblare 8086 este:

[<etichetă>:] <mnemonică> [<destinaţie>[, <sursă>]] [; comentariu]

Program (cod) sursă - program scris în limbaj de asamblare. Pe lîngă instrucţiunile în limbaj de asamblare care au corespondent în setul de instrucţiuni al microprocesorului, programul sursă mai poate conţine şi *directive de asamblare*. Directivele de asamblare nu generează cod executabil ci furnizează asamblorului indicaţii despre modul de transformare a codului sursă în cod executabil.

Asamblor - program (software) care translatează programul sursă scris în limbaj de asamblare într-un limbaj maşină posibil de executat de către microprocesor.

Asamblare - procesul de conversie de cod efectuat de asamblor.

Cod obiect - program aflat între limbajul de asamblare şi limbajul maşină. Codul obiect nu este direct executabil de către microprocesor ci trebuie să treacă prin procesul de editare a legăturilor.

Linker (link editor) - program (software) care realizează procesul de editare de legături. Linker-ul primeşte la intrare unul sau mai multe fişiere obiect (eventual şi fişiere ce conţin biblioteci de module) şi generează un fişier direct executabil.

Link-editare (editare de legături) - procesul de rezolvare a referinţelor externe modulului şi a referinţelor la adrese în cadrul unui modul obiect. Rezultatul link-editării este un program direct executabil de către microprocesor. Editarea de legături realizează trei acţiuni principale:

- Combină modulele separate de cod obiect într-un singur fişier executabil;
- Rezolvă referinţele la variabile externe;
- Opţional, produce un fişier ce conţine informaţia despre modul în care au fost legate fişierele obiect.

Limbaj maşină - limbaj binar interpretat de către microprocesor ca instrucţiuni executabile.

Program executabil - transpunerea unui algoritm în limbaj maşină.

7.2 Asamblarea

Asamblorul firmei *Borland* se numeşte *Turbo Assembler* ($TASM$). Asamblorul acceptă la intrare un *fişier sursă* care conţine instrucţiuni în limbaj de asamblare (specifice microprocesorului) şi un set de directive (specifice asamblorului). Prin directive de asamblare, programatorul poate transmite asamblorului indicaţii despre modul de asamblare a codului sursă. Acest fişier este un fişier ASCII, care are extensia implicită $.ASM$. Ca punct de plecare, se poate spune că un asamblor este un program care translatează un program sursă scris în limbaj de asamblare în cod maşină. În principiu, sarcina asamblorului este de a converti mnemonicile limbajului de asamblare în echivalentele numerice care reprezintă codul maşină.

Pentru a translata codul, asamblorul trebuie să parcurgă programul sursă cel puţin o dată. Fiecare citire a codului sursă se numeşte *trecere* (*pass*, în limba engleză). Cele mai multe asambloare necesită două treceri, deşi există asambloare cu o singură trecere sau cu mai mult de două treceri.

În prima trecere, asamblorul execută următoarele acţiuni:

- face analiza sintactică a codului şi determină offset-ul pentru fiecare linie (adresa la care se va asambla);
- face ipoteze asupra valorilor nedefinite;
- face verificările elementare de corectitudine a codului şi afişează mesaje de eroare;
- opţional, generează un fişier listing intermediar.

În a doua trecere, asamblorul execută următoarele acţiuni:

- încearcă reconcilierea valorilor presupuse în trecerile anterioare;
- generează *fişierul obiect*, care are extensia implicită $.OBJ$.

- opţional, generează *fişierul listing*, care are extensia implicită *.LST*. Fişierul listing poate fi considerat ca un raport al asamblorului. Acest fişier conţine atît specificaţile în limbaj de asamblare cît şi instrucţiunile în limbaj maşină rezultate după asamblare;

- opţional, generează *fişierul de referinţe încrucişate*, care are extensia implicită *.XRF*. Acest fişier conţine informaţii pe care alte programe (specifice asamblorului) le folosesc pentru a crea o listă de referinţe încrucişate ale simbolurilor utilizate în fişierul sursă. Pentru crearea concretă a acestui fişier se utilizează un alt program, care în cazul *TASM* se numeşte *TCREF*.

Turbo Assembler este un program care acceptă parametrii transmişi prin linia de comandă. Sintaxa liniei de comandă a asamblorului, aşa cum este ea specificată de firma producătoare este descrisă în continuare..

```
Turbo Assembler  Version 2.51  Copyright (c) 1988, 1991 Borland International

Syntax:TASM [options] source [,object] [,listing] [,xref]

/a,/s          Alphabetic or Source-code segment ordering
/c             Generate cross-reference in listing
/dSYM[=VAL]    Define symbol SYM = 0, or = value VAL
/e,/r          Emulated or Real floating-point instructions
/h,/?          Display this help screen
/iPATH         Search PATH for include files
/jCMD          Jam in an assembler directive CMD (eg. /jIDEAL)
/kh#           Hash table capacity # symbols
/l,/la         Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu    Case sensitivity on symbols: ml=all, mx=globals, mu=none
/mv#           Set maximum valid length for symbols
/m#            Allow # multiple passes to resolve forward references
/n             Suppress symbol tables in listing
/o,/op         Generate overlay object code, Phar Lap-style 32-bit fixups
/p             Check for code segment overrides in protected mode
/q             Suppress OBJ records not needed for linking
/t             Suppress messages if successful assembly
/w0,/w1,/w2    Set warning level: w0=none, w1=w2=warnings on
/w-xxx,/w+xxx Disable (-) or enable (+) warning xxx
/x             Include false conditionals in listing
/z             Display source line with error message
/zi,/zd        Debug info: zi=full, zd=line numbers only
```

Sintaxa lansării în execuţie a utilitarului TCREF este următoarea:

```
Turbo CREF  Version 2.0  Copyright (c) 1988, 1991 Borland International

Syntax:  TCREF xrffiles, reffile [options]

@xxxx indicates use response file xxxx
Options:
```

```
  /c  = lower case significant in symbols
  /r  = full module level report
  /p# = page length (# in lines)
  /w# = page width  (# in characters)
```

7.3 Editarea de legături

Denumirile de *Link – Editor* şi de *Linker* sînt absolut echivalente. Linker-ul este un program care translatează un cod obiect relocatabil (produs de un asamblor sau un compilator) în cod maşină executabil. Linker-ul realizează trei funcţii principale:

- combină module obiect separate într-un singur fişier executabil;
- încearcă rezolvarea referinţelor la variabile externe;
- opţional, produce un fişier listing în care se prezintă modul de legare a fişierelor obiect.

Link editorul firmei *Borland* se numeşte *Turbo Link* ($TLINK$).

Link editorul acceptă la intrare un *fişier obiect*, cu extensia implicită $.OBJ$. Ca rezultat al link-editării se va genera un *fişier de cod direct executabil*, cu extensia $.EXE$ sau $.COM$. Opţional, poate fi creat un *fişier listing*, cu extensia implicită $.MAP$, care conţine informaţii despre modul de legare a modulelor în fişierul executabil (adrese de început şi sfîrşit ale modulelor etc.).

Turbo Link este un program care acceptă parametrii transmişi prin linia de comandă. Sintaxa liniei de comandă a linkerului, aşa cum este ea specificată de firma producătoare este descrisă în continuare.:

```
Turbo Link  Version 4.0 Copyright (c) 1991 Borland International

Syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile

@xxxx indicates use response file xxxx
Options:
/m = map file with publics
/x = no map file at all
/i = initialize all segments
/l = include source line numbers
/L = specify library search paths
/s = detailed map of segments
/n = no default libraries
/d = warn if duplicate symbols in libraries
/c = lower case significant in symbols
/3 = enable 32-bit processing
/v = include full symbolic debug information
/e = ignore Extended Dictionary
/t = create COM file (same as /Tc)
/o = overlay switch
/P[=NNNNN] = pack code segments
```

```
/A=NNNN = set NewExe segment alignment factor
/ye = expanded memory swapping
/yx = extended memory swapping
/C  = case sensitive exports and imports
/Txx = specify output file type
  /Tdx = DOS image (default)
  /Twx = Windows image (third letter can be c=COM, e=EXE, d=DLL)
```

7.4 Depanarea

Depanarea (*debugging*, în limba engleză) constă în depistarea şi eliminarea greşelilor din programe. Firma *Borland* produce un program numit *Turbo Debugger* (*TD*) care facilitează depanarea programelor, la nivel de limbaj de asamblare, în mod interactiv, prin intermediul unei interfeţe cu meniuri. Sintaxa lansării în execuţie a programului TD este descrisă în continuare.

```
Turbo Debugger  Version 3.1  Copyright (c) 1988, 92 Borland International

Syntax: TD [options] [program [arguments]]-x

-           turn option x off
-c<file>    Use configuration file <file>
-do,-dp,-ds Screen updating: do=Other display, dp=Page flip, ds=Screen swap
-h,-?       Display this help screen
-i          Allow process id switching
-k          Allow keystroke recording
-l          Assembler startup
-m<#>       Set heap size to # kbytes
-p          Use mouse
-r          Use remote debugging
-rp<#>      Set COM # port for remote link
-rs<#>      Remote link speed: 1=slowest, 2=slow, 3=medium, 4=fast
-sc         No case checking on symbols
-sd<dir>    Source file directory <dir>
-sm<#>      Set spare symbol memory to # Kbytes (max 256Kb)
-vg         Complete graphics screen save
-vn         43/50 line display not allowed
-vp         Enable EGA/VGA palette save
-w          Debug remote Windows program (must use -r as well)
-y<#>       Set overlay area size in Kb
-ye<#>      Set EMS overlay area size to # 16Kb pages
```

Modul de utilizare a programului *TD* pentru depanarea programelor scrise în limbaj de asamblare este prezentat în anexa **??**.

7.5 Apelul procedurilor în limbaj de asamblare

Un fişier ce conţine codul sursă a unei proceduri descrisă în limbaj de asamblare are formatul următor:

```
code  SEGMENT PUBLIC          ; declaraţie de segment de cod
ASSUME CS:code, DS:code

<nume_proc> PROC  NEAR        ; declaraţia procedurii (de tip NEAR)

  <salvarea pe stivă a registrelor ce vor fi folosite în cadrul procedurii>
  <implementarea algoritmului propriu-zis al procedurii>
  <refacerea din stivă a registrelor folosite în cadrul procedurii>
  RET                         ; instrucţiune de întoarcere din procedură

<nume_proc> ENDP              ; terminarea procedurii

code   ENDS                   ; încheierea segmentului de cod

PUBLIC <nume_proc>            ; declararea publică a numelui de procedură
                              ; pentru a fi posibilă apelarea acesteia
                              ; dintr-un alt fişier de cod
END
```

Un fişier ce conţine în codul sursă apelul unei proceduri externe (corpul procedurii se află într-un alt fişier sursă) are formatul următor:

```
EXTRN  <nume_proc> : NEAR     ; declaraţie de procedură externă

code  SEGMENT PUBLIC          ; declaraţie de segment de cod
ASSUME CS:code, DS:code

<declaraţii de date şi directive EQU>

start:
  MOV  AX, CS
  MOV  DS, AX                 ; suprapune segmentul de date peste cel de cod

  <pregăteşte parametrii de intrare ai procedurii>
  CALL <nume_proc>            ; apel de procedură
  <preia parametrii de ieşire ai procedurii>

  MOV  AX, 4C00H              ; terminare program prin apelul funcţiei DOS 4CH
  INT  21H

code ENDS

; lansează în execuţie programul la adresa etichetei start:
END     start
```

7.6 Experimente

I. Faceţi o comparaţie între limbajul de asamblare şi limbajul de nivel înalt C. Ca exemplificare, se consideră problema afişării mesajului *"Hello, world!"* pe ecran. Acest lucru se poate descrie în C cu instrucţiunea:

```
printf("Hello, world!\n");
```

Obţinerea aceluiaşi efect se descrie în limbaj de asamblare cu mai multe instrucţiuni:

```
mov  dx, offset HelloMessage
mov  ah, 9
int  21H
HelloMessage  DB   'Hello, world!',13,10,'$'
```

Editaţi în fişierul $HELLOA.ASM$ următorul program în limbaj de asamblare:

```
code SEGMENT
assume cs:code, ds:code
org 100H

start: mov  ax, cs
       mov  ds, ax
       mov  dx, offset HelloMessage     ; pointer la şirul "Hello, world!"
       mov  ah, 9                       ; funcţie DOS de afişare şir
       int  21H                         ; afişează mesajul "Hello, world!"
       mov  ah, 4CH
       int  21H                         ; terminarea programului

HelloMessage DB 'Hello, world!',13,10,'$'

code ends

end start
```

Asamblaţi fişierul $HELLOA.ASM$ lansînd comanda:

```
TASM HELLOA.ASM
```

Ca efect, se va genera fişierul obiect $HELLOA.OBJ$. Consultaţi cu un editor de texte conţinutul acestui fişier.

Link-editaţi fişierul $HELLOA.OBJ$ lansînd comanda:

```
TLINK  HELLOA.OBJ
```

Ca efect, se va genera fişierul executabil $HELLOA.EXE$ şi fişierul de asocieri $HELLOA.MAP$. Lansaţi în execuţie fişierul $HELLOA.EXE$. Consultaţi cu un editor de texte conţinutul fişierului $HELLOA.MAP$.

Link-editaţi fişierul $HELLOA.OBJ$ lansînd comanda:

```
TLINK  /t HELLOA.OBJ
```

Opţiunea `/t` determină link-editorul să genereze fişierul executabil $HELLOA.COM$, în format $.COM$.

Lansaţi în execuţie fişierul $HELLOA.COM$.

Editaţi în fişierul $HELLOC.C$ următorul program în limbajul C:

```
#include <stdio.h>
void main()
{
  printf("Hello, world!\n");
}
```

Compilaţi fişierul sursă $HELLOC.C$ în mediul $BorlandC$ ($Alt - C, C$ sau $Alt - F9$). Ca efect, se va genera fişierul obiect $HELLOC.OBJ$. Consultaţi cu un editor de texte conţinutul acestui fişier şi comparaţi-l cu cel al fişierului $HELLOA.OBJ$.

Link-editaţi fişierul $HELLOC.C$ ($Alt-C, L$) şi obţineţi fişierul executabil $HELLOC.EXE$ şi fişierul de asocieri $HELLOC.MAP$. Lansaţi în execuţie fişierul $HELLOC.EXE$. Consultaţi cu un editor de texte conţinutul fişierului $HELLOC.MAP$ şi comparaţi-l cu cel al fişierului $HELLOA.MAP$.

Revenind la prompterul sistem, convertiţi fişierul $HELLOC.EXE$ din format $.EXE$ în fişierul $HELLOC.COM$ în format $.COM$, folosind utilitarul $EXE2BIN.EXE$. Comanda este:

```
EXE2BIN HELLOC.EXE HELLOC.COM
```

Studiaţi diferenţele între descrierea unui program în limbaj de asamblare şi descrierea aceluiaşi program în limbajul C urmărind chestiunile enumerate în continuare.

- Comparaţi fişierele $HELLOA.ASM$ şi $HELLOC.C$ din punct de vedere al numărului de linii şi din punct de vedere al dimensiunii acestora;
- Comparaţi conţinutul fişierele $HELLOA.MAP$ şi $HELLOC.MAP$;
- Comparaţi dimensiunile fişierelor $HELLOA.COM$ şi $HELLOC.COM$;
- Comparaţi dimensiunile fişierelor $HELLOA.EXE$ şi $HELLOC.EXE$;
- Comparaţi dimensiunile fişierelor $HELLOA.EXE$ şi $HELLOA.COM$, respectiv $HELLOC.EXE$ şi $HELLOC.COM$;
- Rulaţi "pas cu pas" fişierul $HELLOC.C$ în mediul BorlandC. Rulaţi "pas cu pas" fişierul $HELLOC.COM$ în *Turbo Debugger*. Linia de comandă este:

  ```
  TD HELLOC.COM
  ```

- Rulaţi "pas cu pas" fişierul $HELLOC.EXE$ în *Turbo Debugger*. Linia de comandă este:

  ```
  TD HELLOC.EXE
  ```

- Rulaţi "pas cu pas" fişierul $HELLOA.COM$ în *Turbo Debugger*.

- Comparaţi rularea "pas cu pas" în *Turbo Debugger* a fişierului $HELLOA.COM$ cu cea a fişierului $HELLOC.COM$. Faceţi corespondenţa între modul în care s-a descris programul în limbaj de asamblare şi în limbaj C, cu instrucţiunile efectiv executate de către microprocesor.
- Asamblaţi fişierul $HELLOA.ASM$ cu opţiunea de includere a informaţiei pentru depanare, lansînd comanda:

  ```
  TASM /zi HELLOA
  ```

  Link-editaţi fişierul $HELLOA.OBJ$ cu opţiunea de includere a informaţiei pentru depanare, lansînd comanda:

  ```
  TLINK /v HELLOA, HELLOA_D
  ```

  Se va genera fişierul $HELLOA\_D.EXE$. Comparaţi imaginea în *Turbo Debugger* a fişierului $HELLOA.EXE$ cu cea a fişierului $HELLOA\_D.EXE$.
- Generaţi un listing la asamblarea fişierului $HELLOA.ASM$ cu comanda:

  ```
  TASM /la HELLOA
  ```

  Consultaţi cu un editor de texte fişierul $HELLOA.LST$.

II. Scrieţi un program în limbaj de asamblare care să apeleze o procedură scrisă în limbaj de asamblare, dar care se află într-un alt fişier. Procedura trebuie să înscrie într-un bloc de memorie un octet şi să genereze suma de control asociată blocului. Parametrii de intrare ai procedurii sînt transmişi în următoarele registre:
AX - adresa de început a blocului de memorie;
BX - numărul de locaţii de memorie din bloc (dimensiunea blocului);
DL - octetul care trebuie înscris în fiecare locaţie a blocului de memorie.

La revenirea din procedură, registrul DL conţine suma de control. Suma tuturor locaţiilor blocului, plus cea a registrului DL este zero.

Editaţi în fişierul $PROC.ASM$ următorul cod sursă în limbaj de asamblare:

```
code SEGMENT PUBLIC
ASSUME CS:code, DS:code

WriteDL PROC   NEAR
        PUSH   SI
        PUSH   BX

        MOV    SI, BX   ; Offset-ul byte-ului curent
        MOV    BX, AX
        DEC    BX       ; Adresa începutului de bloc în BX
        XOR    DH,DH    ; Suma de control
muta:   MOV    [BX+SI], DL
        SUB    DH,DL    ; Calculează suma de control
        DEC    SI       ; Pointează la următorul byte
        JNZ    muta

        POP    BX
        POP    SI
```

```
        RET
WriteDL ENDP

code    ENDS
PUBLIC WriteDL

END
```

Editaţi în fişierul $APEL.ASM$ următorul cod sursă în limbaj de asamblare:

```
EXTRN WriteDL : NEAR

code    SEGMENT
ASSUME CS:code, DS:code

Dimens  EQU     3H
DataB   EQU     1H

start:  MOV     AX, CS
        MOV     DS, AX

        MOV     AX, offset Bloc
        MOV     BX, Dimens
        MOV     DL, DataB
        CALL    WriteDL

        MOV     AX, 4C00H
        INT     21H

Bloc DB  Dimens DUP(?)

code ENDS

END start
```

Din fişierele sursă, realizaţi fişierul executabil $TEST.EXE$, lansînd comenzile:

```
TASM PROC
TASM APEL
TLINK APEL PROC, TEST
```

Urmăriţi execuţia programului prin rulare "pas cu pas" cu *Turbo Debugger*:

```
TD TEST
```

Lucrarea 8

Declararea datelor şi a segmentelor

Această lucrare prezintă modul în care se declară în limbaj de asamblare structurile de date necesare oricărui program. La microprocesoarele familiei 8086, memoria este "segmentată". Gestionarea memoriei în limbaj de asamblare revine exclusiv programatorului. Exerciţiile propuse în finalul lucrării demonstrează că nu este suficientă exprimarea algoritmului în limbajul de asamblare (atît de sărac faţă de C...), ci este necesară înţelegerea modului de administrare a memoriei segmentate.

8.1 Declararea datelor

Sintaxa declaraţiei unei date (variabile) este:

[<nume>] <tip> <listă expresii de iniţializare>

sau

[<nume>] <tip> <factor> DUP (<listă expresii de iniţializare>)

<nume> este numele asociat variabilei, prin care se va putea face o referire la aceasta. Numele este opţional. Dacă într-o declaraţie de date nu apare nici un nume, sînt alocate date, dar adresa de început nu poate fi referită printr-un nume simbolic. Numele poate fi considerat ca fiind un pointer (adresa începutului zonei de memorie) la data asociată.

<tip> reprezintă unul din cuvintele rezervate ce desemnează un tip de date. Tipurile de date existente şi cuvintele rezervate pentru desemnarea acestora sînt prezentate în tabelul 8.1.

<listă expresii de iniţializare> specifică valorile cu care este iniţializată zona de memorie asociată datelor definite. În lipsa unor valori concrete, prin caracterul '?', se indică faptul că zona de date este rezervată dar nu şi iniţializată. În lista de iniţializare pot apare constante sau expresii evaluate de către asamblor (static).

Operanzii expresiilor pot fi constante numerice, alfanumerice şi nume pentru care asamblorul asociază valori: nume de date, etichete de instrucţiuni, nume de segmente, etc. Constantele întregi pot reprezenta numere scrise în bazele 2, 8, 10 sau 16. Baza în care

<tip>	*Semnificaţie (în limba engleză)*	*Tip*
DB	Define Byte (1 byte)	byte
DW	Define Word (2 bytes)	word
DD	Define Doubleword (4 bytes)	dword
DQ	Define Quadword (8 bytes)	qword
DT	Define Ten bytes (10 bytes)	tbyte

Tabelul 8.1: Tipuri de date şi cuvinte rezervate pentru desemnarea acestora.

este reprezentat numărul este desemnată de o literă aflată la sfîrşitul şirului de cifre ce formează constanta $\{B, Q, D, H\}$. Implicit, dacă nu apare nici una din literele ce desemnează baza de numeraţie, se consideră că numărul este exprimat în baza 10. Constantele în virgulă flotantă se pot exprima în formatul cu mantisă şi exponent. Constantele alfanumerice, individuale sau în şiruri, sînt incluse între caractere apostrof simple (') sau duble ("). O constantă alfanumerică este asociată de către asamblor cu codul ASCII al caracterului corespunzător.

O *dată* iniţializată cu un *nume de dată* poate fi considerată similară cu o variabilă de tip *pointer*. În acest caz, data conţine adresa unei date ce are o valoare şi nu o valoare propriu-zisă. Pointerii de dimensiune 2 baiţi conţin adrese cu *referinţe apropiate* (în acelaşi segment de memorie). Pointerii de dimensiune 4 baiţi conţin adrese cu *referinţe îndepărtate* (în segmente de memorie disjuncte).

<factor> specifică de cîte ori se repetă *<lista de expresii de iniţializare>* ce este inclusă între paranteze rotunde. Valoarea iniţială poate fi caracterul '?' (neiniţializat), orice expresie constantă, orice valoare evaluată la o constantă de către asamblor sau un alt operator DUP. Prin utilizarea operatorului DUP se pot defini structuri de date similare cu matricile uni sau multidimensionale.

Datele reprezentate pe mai mult de un byte sînt stocate în memorie conform regulii 'little-endian' (cel mai puţin semnificativ byte este stocat la cea mai mică adresă de memorie).

Adresa locaţiei de memorie curente (unde se alocă data) poate să fie referită prin simbolul $ sau prin expresia *this <tip>*.

Adresa logică (offset-ul) a unei date poate fi obţinută prin operatorul *OFFSET*. Adresa de bază a segmentului în care a fost definită data poate fi obţinută prin operatorul *SEG*.

Adresarea unei date ca fiind de alt tip decît a fost declarată este posibilă prin utilizarea operatorului de conversie *PTR*. Sintaxa unei expresii construite cu acest operator este:

<tip> PTR <expresie>

Ca efect, *<expresia>* va fi interpretată prin intermediul atributului *<tip>*. În continuare se prezintă, ca exemplu, o porţiune de cod.

```
. . .
DataW   dw    10 dup (?)
DataB   db    10 dup (?)
. . .
```

```
mov al, byte ptr DataW  ; mută în AL primul byte din cei 10 x 2 rezervaţi
                        ; pentru DataW
mov ax, word ptr DataB  ; mută în AX primii doi baiţi din cei 10 rezervaţi
                        ; pentru DataB
```

În continuare se prezintă listingul unor declaraţii de date în limbaj de asamblare. Listingul a fost obţinut cu TASM (cu opţiunea `/la`). Listingul prezintă, în partea din stînga, adresa de memorie şi datele rezultate în urma asamblării, iar în partea din dreapta codul, aşa cum a apărut în fişierul sursă.

```
 1   0000                          data segment
 2   0000  10                      D1    db   16
 3   0001  000C                    D2    dw   4*3
 4   0003  FFFFFFFF                D3    dd   4294967295
 5   0007  ????????????????        D4    dq   ?
 6   000F  000000000002DFDC1C35    D5    dt   12345678901D
 7   0019  12345678901234567890    D6    dt   12345678901234567890
 8   0023  00000000012345678901    D7    dt   12345678901
 9   002D  05 17 1D AF             D8    db   0101B,27O, 29, 0AFH
10   0031  33                      D9    db   3+"0"
11   0032  01 02 03                      db   1, 2, 3
12   0035  0032r                   D10   dw   D9+1
13   0037  61 62 63                D11   db   97, 98, 99
14   003A  61 62 63                D12   db   'a', 'b', 'c'
15   003D  61 62 63                D13   db   'abc'
16   0040  61 62 63                D14   db   "abc"
17   0043  4D 65 73 61 6A 3A 0D+   D15   db   'Mesaj:', 13, 10, '$'
18         0A 24
19   004C  53 61 6C 75 74 21       D16   db   "Salut!"
20   0052  06                      D17   db   $-D16
21   0053  07                      D18   db   this byte-D16
22   0054  004Cr                   D19   dw   OFFSET D16
23   0056  0000s                   D20   dw   SEG D16
24   0058  61 62                   D21   db   "ab"
25   005A  00006162                D22   dd   "ab"
26   005E  00000061                D23   dd   "a"
27   0062  54 65 78 74 00          D24   db   "Text",0
28   0067  0062r                   D25   dw   D24
29   0069  00000062sr              D26   dd   D24
30   006D  0A*(00000001)           D27   dd   10 dup (1)
31   0095  10*(??)                 D28   db   16 dup (?)
32   00A5  02*(02*(02*     +       D29   dd   2 dup (2 dup (2 dup (0)))
33         (00000000)))
34   00C5  05*(03*(01) 02)         D30   db   5 dup (3 dup (1), 2)
35   00D9                          data ends
36                 end
```

- Pe linia 2 se declară o dată de tip *byte* cu numele `D1`, iniţializată cu constanta `16`.

- Pe linia 3 se declară o dată de tip *word* cu numele `D2`, iniţializată cu valoarea `12`. Valoarea este determinată în momentul asamblării prin evaluarea expresiei `4*3`.

- Pe linia 4 se declară o dată de tip *dword* cu numele `D3`, iniţializată cu cea mai mare valoare posibilă, exprimată în baza 10. Exprimarea în baza 16 este `FFFFFFFF`.

- Pe linia 5 se declară o dată de tip *qword* cu numele `D4`, lăsată neiniţializată.

- Pe liniile 6, 7 şi 8 se declară date de tip *tbyte* cu numele `D5`, `D6` şi `D7`, iniţializate cu diverse valori. De remarcat corespondenţa între valorile din codul sursă şi cele din memorie.

- Pe linia 9 se declară o dată de tip *byte* cu numele `D8`, iniţializată cu trei valori constante exprimate în bazele de numeraţie 2, 8, 10 şi 16.

- Pe linia 10 se declară o dată de tip *byte* cu numele `D9`, iniţializată cu codul ASCII al caracterului "3", obţinut prin însumarea valorii întregi 3 la codul ASCII al caracterului "0".

- Pe linia 11 se declară o dată de tip *byte* căreia nu i se asociază nici un nume. Aceste date pot fi referite prin numele altor date.

- Pe linia 12 se declară o dată de tip *word* cu numele `D10`, iniţializată cu o valoare constantă obţinută prin evaluarea unei expresii în care intră şi numele unei date. Valoarea obţinută este adresa de început a datelor declarate pe linia 11. `D10` poate fi privit ca un pointer la datele declarate pe linia 11.

- Pe liniile 13, 14, 15 şi 16 se declară date de tip *byte* cu numele `D11`, `D12`, `D13` şi `D14`, iniţializate cu aceleaşi valori exprimate în moduri diferite: întregi, caractere, şir delimitat de caractere apostrof simplu, şir delimitat de caractere apostrof dublu.

- Pe linia 17 se declară o dată de tip *byte* cu numele `D15`, iniţializată cu o listă de expresii de diferite tipuri (şir de caractere, întregi, caracter). Caracterele ASCII asociate codurilor `13` şi `10` sînt caracterele de control `CR` şi `LF`. Şirul astfel definit poate fi transmis ca argument al funcţiei DOS 9H de tipărire şir de caractere.

- Pe linia 19 se declară o dată de tip *byte* cu numele `D16`, iniţializată cu un şir de caractere ce conţine şi un semn de punctuaţie.

- Pe linia 20 se declară o dată de tip *byte* cu numele `D17`, iniţializată cu dimensiunea datelor `D16`. Adăugarea unor caractere şirului `D16` determină actualizarea automată a valorii din `D17`, datorită modului în care a fost scrisă expresia de iniţializare.

- Pe linia 21 se declară o dată de tip *byte* cu numele `D18`, iniţializată cu o valoare provenind dintr-o expresie similară cu cea de pe linia 20.

- Pe liniile 22 şi 23 se declară date de tip *word* cu numele `D19` şi `D20`, iniţializate cu valorile offset-ului şi adresei de bază a segmentului în care a fost definită data `D16`. Adresa datei `D16` poate fi considerată `D20:D19`.

- Pe liniile 24, 25 şi 26 se observă modul în care se ordonează datele de mai mulţi baiţi, conform regulii "little-endian".

- Pe linia 27 se declară o dată de tip *byte* cu numele `D24`, iniţializată cu un şir de caractere urmat de un întreg.

- Pe linia 28 se declară o dată de tip *word* cu numele `D25`, ce poate fi interpretată ca pointer apropiat la data `D24`. `D25` are valoarea offset-ului datei `D24`.

- Pe linia 29 se declară o dată de tip *dword* cu numele `D26`, ce poate fi interpretată ca pointer îndepărtat la data `D24`. `D25` are ca valoare doi baiţi segmentul şi doi baiţi offset-ul datei `D24`.

- Pe linia 30 se declară o dată de tip *dword* cu numele `D27`, inițializată cu un vector de 10 valori întregi.

- Pe linia 31 se declară o dată de tip *byte* cu numele `D28`, de dimensiune 16 baiți. Zona de memorie asociată se lasă neinițializată.

- Pe linia 32 se declară o dată de tip *dword* cu numele `D29`, de dimensiune $2 \times 2 \times 2 = 8$. Zona de memorie asociată este în întregime inițializată cu 0.

- Pe linia 34 se declară o dată de tip *byte* cu numele `D30`. Zona de memorie asociată este inițializată cu 5 de pattern-uri: 1, 1, 1, 2. În total, dimensiunea datei `D30` este de 20 de baiți.

Zona de memorie rezervată pentru declarațiile anterioare este prezentată în figura 8.1.

```
ds:0000 10 0C 00 FF FF FF FF 00 00 00 00 00 00 00 00 35
ds:0010 1C DC DF 02 00 00 00 00 00 90 78 56 34 12 90 78
ds:0020 56 34 12 01 89 67 45 23 01 00 00 00 00 05 17 1D
ds:0030 AF 33 01 02 03 32 00 61 62 63 61 62 63 61 62 63
ds:0040 61 62 63 4D 65 73 61 6A 3A 0D 0A 24 53 61 6C 75
ds:0050 74 21 06 07 4C 00 56 5A 61 62 62 61 00 00 61 00
ds:0060 00 00 54 65 78 74 00 62 00 62 00 56 5A 01 00 00
ds:0070 00 01 00 00 00 01 00 00 00 01 00 00 00 01 00 00
ds:0080 00 01 00 00 00 01 00 00 00 01 00 00 00 01 00 00
ds:0090 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:00C0 00 00 00 00 00 01 01 01 02 01 01 01 02 01 01 01
ds:00D0 02 01 01 01 02 01 01 01 02 00 00 00 00 00 00 00
```

Figura 8.1: Zonă de memorie rezervată prin declarații de date.

8.2 Declararea segmentelor

Microprocesoarele familiei 8086 implementează un mecanism de segmentare a memoriei. Se pot defini oricîte segmente logice, de dimensiune maximă 64 KB. Segmentele pot fi disjuncte sau suprapuse parțial sau total. La un moment dat, microprocesorul poate avea acces numai la patru segmente. Adresele de bază ale acestor segmente se găsesc în cele patru registre segment: *CS*, *DS*, *SS* şi *ES*.

8.2.1 Sintaxa declarației de segment

Sintaxa declarației unui segment este următoarea:

<nume> SEGMENT [<aliniere>] [<combinare>] [<clasa>]
<corpul segmentului>
<nume> ENDS

<nume> definește numele segmentului. Acest nume poate fi unic sau poate fi același nume

atribuit şi altor segmente din program. Segmentele cu nume identic sînt tratate ca şi cum ar fi acelaşi segment. De exemplu, dacă este necesar să se plaseze diferite porţiuni ale unui singur segment în module sursă diferite, atunci segmentului îi este atribuit acelaşi nume în ambele module.

<*aliniere*> este un cîmp opţional ce defineşte tipul adresei de început a segmentului. Prin valorile cîmpului aliniere: PARA/BYTE/WORD/PAGE se specifică faptul că adresa de început a zonei de memorie rezervată segmentului este divizibilă cu 16/1/2/256. Valoarea implicită a acestui cîmp este *PARA*.

<*combinare*> este un cîmp opţional care defineşte modul de combinare a segmentelor ce au acelaşi nume. Informaţia transmisă de programator prin intermediul acestui cîmp este folosită de către link-editor. Cîmpul combinare poate avea cinci valori, cuvinte rezervate.

- PUBLIC - concatenează toate segmentele cu acelaşi nume pentru a forma un segment unic, contiguu. Toate instrucţiunile şi adresele de date din noul segment se referă la un registru de segment unic, iar toate deplasamentele sînt ajustate pentru a reprezenta distanţa de la începutul segmentului.
- STACK - concatenează toate segmentele cu acelaşi nume pentru a forma un segment unic, contiguu. Acest tip de combinare este identic cu tipul de combinare PUBLIC, cu excepţia faptului că toate adresele din noul segment se referă la registrul SS al segmentului. Registrul indicator de stivă (SP) este iniţializat la lungimea segmentului. Segmentul stivă din programul utilizatorului va folosi în mod normal STACK întrucît acesta iniţializează automat registrul SS. Dacă se creează un segment de stivă şi nu se utilizează tipul STACK, se vor da implicit instrucţiuni pentru iniţializarea registrelor SS şi SP.
- COMMON - creează segmente suprapuse prin plasarea începutului tuturor segmentelor cu acelaşi nume la aceeaşi adresă. Lungimea zonei rezultate este lungimea celui mai lung segment. Toate adresele din segmente sînt raportate la aceeaşi adresă de bază. Dacă variabilele sînt iniţializate în mai multe segmente ce au acelaşi nume şi tipul COMMON, datele cel mai recent iniţializate înlocuiesc toate datele iniţializate anterior.
- MEMORY - concatenează toate segmentele cu aceleaşi nume pentru a forma un segment unic, contiguu. Link-editorul tratează segmentele MEMORY în aceleaşi mod ca şi segmentele PUBLIC. Segmentul curent va fi plasat în memorie în spaţiul rămas disponibil după plasarea în memorie a celorlalte segmente.
- AT <*adresă*> - determină ca toate adresele de etichete şi variabile definite în segment să fie relative la o adresă. Adresa poate fi orice expresie corectă, dar nu poate conţine o referinţă anticipată (o referinţă la un simbol definit mai tîrziu în fişierul sursă). Un segment AT nu conţine, de obicei, cod sau date iniţializate. În schimb, el reprezintă un model de adresă ce poate fi plasat peste codul sau datele aflate deja în memorie, cum ar fi o zonă tampon sau o altă locaţie de memorie absolută definită de partea hardware. Editorul de legături nu va genera cod sau date pentru segmentele AT, dar datele sau codul existent pot fi accesate prin nume dacă se specifică o etichetă într-un segment AT.

Dacă nu apare nici un tip combinare, segmentul va avea un tip special. Segmentele cu acelaşi nume nu vor fi combinate. În schimb, fiecare segment va primi propriul segment

fizic atunci cînd este încărcat în memorie. Deşi un nume de segment dat poate fi utilizat de mai multe ori într-un fişier sursă, fiecare definiţie de segment ce utilizează acel nume trebuie să aibe exact aceleaşi atribute. Dacă sînt precizate tipuri de combinare într-o definiţie iniţială de segment, definiţile ulterioare pentru acel segment nu necesită specificarea nici unui alt tip de combinare.

<*clasa*> este un cîmp opţional care specifică modul de asociere a segmentelor ce au nume diferite, dar scopuri similare. Tipul se foloseşte pentru a controla ordinea segmentelor şi pentru a identifica segmentul de cod. Numele de clasă va fi inclus între caractere apostrofuri simple (’). Segmentele pentru care nu este stabilit explicit un nume de clasă vor avea numele de clasă nul. Link-editorul nu impune restricţii asupra numărului sau dimensiunii segmentelor dintr-o clasă.

<*corpul segmentului*> conţine instrucţiunile care se vor asambla în spaţiul de memorie al segmentului şi declaraţiile de date prin care se vor rezerva zone de memorie.

8.2.2 Asocierea segmentelor cu registre

Deşi la un moment dat doar patru segmente pot fi accesate, într-un program se pot defini mult mai multe segmente. De cele mai multe ori, instrucţiunile în limbaj de asamblare apelează datele din memorie prin adresa lor logică, segmentul fiind determinat implicit de către procesor. În tabelul 6.4 sînt prezentate segmentele considerate implicite în cazul unor apeluri la memorie. De exemplu, instrucţiunea JMP consideră că segmentul implicit este cel asociat cu registrul CS. Instrucţiunile MOV consideră că segmentul este asociat cu registrul DS.

Determinarea segmentului în care se face referirea la memorie se face de către microprocesor, în momentul execuţiei efective a instrucţiunii. În limbaj de asamblare se poate apela o dată care nu se află în segmentul implicit prin plasarea înainte de numele datei a numelui registrului de segment.

```
mov  ax, Data1     ; apel la Data1 aflatã în segmentul DS (implicit)
mov  ax, ES:Data2  ; apel la Data2 aflatã în segmentul ES (explicit)
mov  ax, CS:Data3  ; apel la Data3 aflatã în segmentul CS (explicit)
```

Prin utilizarea directivei $ASSUME$, programatorul poate transfera asamblorului sarcina stabilirii registrelor explicite în cazul apelării unor date declarate în segmente diferite.

Directiva $ASSUME$ se foloseşte pentru a indica asamblorului asocierea dintre un segment şi un registru segment. Această directivă nu controlează activitatea procesorului. Programatorul trebuie să scrie în program instrucţiuni explicite de încărcare a registrelor de segment cu valorile declarate în directiva $ASSUME$. Directiva ASSUME afectează numai consideraţiile referitoare la momentul asamblării. Există şi situaţii cînd se pot folosi instrucţiuni pentru modificarea consideraţiilor referitoare la momentul execuţiei.

Sintaxa directivei $ASSUME$ este:

ASSUME <registru segment> : <nume>
ASSUME <registru segment> : NOTHING
ASSUME NOTHING

<*registru segment*> poate fi oricare din cele patru nume de registre segment: CS, DS, ES sau SS.

<*nume*> trebuie să fie numele segmentului ce se va asocia cu <*registru segment*>. Instrucţiunile ulterioare directivei, care consideră un registru implicit pentru referirea în mod automat a etichetelor sau a variabilelor, presupun că dacă segmentul implicit este <*registru segment*>, atunci eticheta sau variabila accesată se află în segmentul <*nume*>.

Cuvîntul cheie *NOTHING* anulează selecţia de segmente curentă, pentru unul sau toate registrele de segment.

În mod normal, o singură instrucţiune *ASSUME* defineşte toate cele patru registre segment la începutul registrului sursă. Totuşi, directiva *ASSUME* poate fi folosită în orice punct pentru a modifica supoziţiile referitoare la segment.

În continuare este prezentat un program pentru exemplificarea efectului directivei *ASSUME*. Pe coloana din stînga apare codul sursă. Pe coloana din dreapta apare codul dezasamblat cu *TurboDebbuger*.

```
A  segment
A1   dw  1
A2   dw  2
A  ends

B  segment
B1   dw  1
B2   dw  2
B  ends

code segment
assume cs:code, ds:A, es:B
C1  dw 1
C2  dw 2

start:
; inițializează registrele de segment
; conform directivelor "assume"
  mov  ax, A                              mov    ax,5A56
  mov  ds, ax                             mov    ds,ax
  mov  ax, B                              mov    ax,5A57
  mov  es, ax                             mov    es,ax

; referiri la date
  mov  ax, A1                             mov    ax,[0000]
  mov  B1, ax                             mov    es:[0000],ax
  mov  es:B2, ax                          mov    es:[0002],ax
  mov  dx, A2                             mov    dx,[0002]
  mov  B2, dx                             mov    es:[0002],dx

; suprapune segmentul DS peste CS
; segmentul A nu mai poate fi referit
  mov  ax, cs                             mov    ax,cs
  mov  ds, ax                             mov    ds,ax
```

```
  assume ds:code

; referiri la date în segmentul DS
; care, în acest caz,
; coincide cu segmentul CS

  mov  ax, B1                                   mov    ax,es:[0000]
  mov  C1, ax                                   mov    [0000],ax
  mov  ds:C2, ax                                mov    [0002],ax
  mov  cs:C1, ax                                mov    cs:[0000],ax

  mov  B2, ax                                   mov    es:[0002],ax

; asamblorul semnalează o eroare
; la linia următoare
; mov  A2, ax
code ends

end start
```

Au fost definite două segmente: A şi B. În fiecare segment s-au definit două date de tip *word*. La începutul segmentului de cod, prin directiva $ASSUME$, s-au asociat cele două segmente cu registrele DS şi ES. La începutul programului efectiv, imediat după eticheta `start:`, s-au inserat instrucţiuni de transfer care să iniţializeze registrele de segment cu valorile presupuse. Ca efect al directivei $ASSUME$, asamblorul a asamblat instrucţiunea `mov B1, ax` sub forma: `mov es:[0000],ax`. Offset-ul datei `B1` (`0000`) a fost precedat de numele de segment `es`.

Prin schimbarea ipotezei asupra conţinutului registrului DS (prin directiva: `assume ds:code`) segmentul A nu mai poate fi accesat. Din acest motiv, în cazul existenţei instrucţiunii de pe ultimul rînd (`mov A2, ax`), asamblorul va genera o eroare. Mesajul de eroare este:

```
**Error** assume.ASM(50) Can't address with currently ASSUMEd segment registers
```

8.2.3 Iniţializarea registrelor

Registrele CS şi IP sînt iniţializate prin specificarea unei adrese de început cu directiva END. Sintaxa directivei este:

END [<adresa de început>]

Cîmpul *<adresa de început>* este o etichetă sau o expresie ce identifică adresa la care utilizatorul doreşte începerea execuţiei atunci cînd programul este încărcat în memorie. Dacă un program este format dintr-un singur modul sursă, este necesară ca adresa de început să se precizeze în acel modul. Dacă un program are mai multe module, toate modulele se vor termina cu directiva END. Numai o directivă END defineşte adresa de început. Link-editorul nu generează eroare la omiterea adresei de început, dar execuţia va începe probabil la o adresă greşită.

Registrul DS trebuie iniţializat la adresa segmentului ce va fi folosit pentru date. Adresa segmentului va fi încărcată în registrul DS. Deoarece o valoare din memorie nu poate

fi încărcată direct într-un registru segment, iniţializarea registrului DS necesită două instrucţiuni de transfer. Instrucţiunile de iniţializare a registrului DS apar, de obicei, la începutul sau foarte aproape de începutul segmentului de cod. Un exemplu de iniţializare a registrului de segment DS este prezentat în continuare:

```
date1 SEGMENT
       ...
date1 ENDS

code1 SEGMENT
ASSUME cs:code1, ds:date1
start:
  mov  ax, date1  ; AX <= adresa de bază a segmentului date1
  mov  ds, ax     ; DS <= AX
  ...
code1 ENDS
END start
```

Registrele SS şi SP sînt iniţializate automat la lansarea în execuţie a unui program. Registrul de segment SS este iniţializat automat la valoarea ultimului segment din codul sursă cu tipul de combinare STACK. Registrul SP este iniţializat automat la dimensiunea segmentului de stivă. Astfel, SS:SP indică iniţial adresa de sfirşit a segmentului de stivă. Segmentul de stivă poate fi iniţializat sau reiniţializat direct prin schimbarea valorilor SS şi SP, prin program. Întrucît întreruperile hardware folosesc aceeaşi stivă ca şi programul, acestea trebuie dezactivate în timpul schimbării stivei.

Registrul ES nu este iniţializat automat. Dacă programul foloseşte segmentul de date auxiliar (folosit implicit în operaţiile cu şiruri), programatorul trebuie să iniţializeze explicit registrul ES. Iniţializarea se face prin transferarea în registrul ES a valorii corespunzătoare adresei de început a segmentului auxiliar, folosind două instrucţiuni de transfer.

8.2.4 Definirea simplificată a segmentelor

Versiunile actuale de asambloare au introdus un nou mecanism de definire simplificată a segmentelor. Acest mecanism este mai uşor de utilizat mai ales cînd se doreşte legarea mai multor module scrise în diferite limbaje de programare. Pentru utilizarea acestui mod de definire trebuie specificat mai întîi modelul de memorie pentru care este scris programul. Modelul de memorie are o semnificaţie software, precizînd modul în care se utilizează segmentele în cadrul programului respectiv. Caracteristicile celor 6 modele de memorie sînt sintetizate în tabelul 8.2.

Declararea modelului trebuie să preceadă alte pseudo-instrucţiuni sau instrucţiuni care implică referiri la segmente. Declararea modelului se face cu ajutorul unei pseudoinstrucţiuni de forma:

.MODEL <tip model>

8.3 Experimente

I. Editaţi într-un fişier programul care urmează. Denumiţi fişierul *DEBUG.ASM*.

Model de memorie	*Volum de date*	*Volum de cod*	*Observaţii*
TINY	< 64 KB - adrese relative	< 64 KB - apeluri "near"	- date+cod+stivă < 64 KB - toate segmentele fac parte din acelaşi grup - programele sînt de tip .COM
SMALL	< 64 KB - adrese relative	< 64 KB - apeluri "near"	- segmente de date şi cod distincte
MEDIUM	< 64 KB - adrese relative	> 64 KB - apeluri "far"	
COMPACT	> 64 KB - adrese fizice	< 64 KB - apeluri "near"	- deşi datele pot avea un volum mai mare de 64 KB, există o restricţie: o structură de date nu poate depăşi 64 KB
LARGE	> 64 KB - adrese fizice	> 64 KB - apeluri "far"	- deşi datele pot avea un volum mai mare de 64 KB, există o restricţie: o structură de date nu poate depăşi 64 KB
HUGE	> 64 KB - adrese fizice	> 64 KB - apeluri "far"	- o structură de date poate avea un volum mai mare de 64 KB

Tabelul 8.2: Caracteristici ale modelelor de memorie.

```
code SEGMENT
      assume cs:code, ds:code, es:code
      org 100H

start:
      mov  ax, cs
      mov  ds, cx
      mov  si, OFFSET Sir_sursa
      mov  di, OFFSET Sir_dest
      mov  cx, 17
rep   movs Sir_dest, Sir_sursa
      mov  ah, 4ch
      int  21h

Sir_sursa DB 'Acesta este sirul'
Sir_dest  DB 17 dup (?)

code ends
end start
```

Scopul programului este de a copia un şir de caractere definit cu numele `Sir_sursa` într-o altă zonă de memorie unde i s-a rezervat un spaţiu de aceeaşi dimensiune sub numele `Sir_dest`. Programul conţine cîteva greşeli deşi nici asamblorul şi nici link-editorul nu furnizează vreun mesaj de eroare. Realizaţi următoarele:

- Studiaţi efectul fiecărei instrucţiuni sau directive de asamblare care apare în codul sursă. Studiaţi instrucţiunea de transfer de şiruri `movs` şi efectul prefixului `rep`.

- Studiaţi modul de definire a datelor şi a segmentelor. Ţinînd cont de directivele `assume` şi `org`, estimaţi offset-ul la care se vor găsi datele definite.
- Asamblaţi şi link-editaţi fişierul *DEBUG.ASM*.
- Utilizînd *Turbo Debugger* corectaţi greşelile depistate în program.
- Modificaţi definirile de date şi programul pentru a minimiza modificările ulterioare determinate de redefinirea şirului sursă, ca lungime şi conţinut.
- Studiaţi diferenţele dintre instrucţiunile:

```
mov  si, OFFSET Sir_sursa
mov  si, word PTR Sir_sursa
mov  al, Sir_sursa
```

II. Studiaţi conţinutul fişierului *AFISARE.ASM*. Codul sursă descrie un program de iniţializare a registrului AX şi ulterior afişează conţinutul acestuia pe ecran, exprimat în baza 10.

- Studiaţi modul de definire a datelor şi segmentelor.
- Cum explicaţi că programul utilizează stiva (prin instrucţiuni `PUSH` şi `POP`) dar nu are un segment de stivă definit explicit? Verificaţi conţinutul stivei de-a lungul execuţiei programului.
- Studiaţi modul în care este descrisă şi apelată o procedură din acelaşi segment de cod. Apelaţi la *Turbo Debugger* pentru a intra în detalii referitoare la modul de asamblare a instrucţiunilor implicate în apelul şi revenirea din procedură.

III. Studiaţi conţinutul fişierului *MEDIE.ASM*. Codul sursă descrie un program care însumează valorile dintr-un vector de date de tip *word* şi ulterior determină valoarea medie a elementelor.

- Studiaţi modul în care s-a parametrizat definirea datelor. Modificaţi programul pentru a însuma un număr dublu de date. Cîte linii de cod trebuie modificate?
- Studiaţi instrucţiunile de împărţire (`div`).
- Plecînd de la codul considerat, scrieţi o procedură care primeşte ca parametrii un pointer la un vector de date şi dimensiunea acestuia şi întoarce într-un registru suma elementelor vectorului de date. Scrieţi un program care apelează procedura de două ori, cu parametrii diferiţi.

Lucrarea 9

Programarea cu întreruperi software

Un sistem de calcul cu microprocesor, aşa cum este prezentat în figura 1, nu poate fi imaginat fără dispozitive periferice de intrare/ieşire. De obicei, aceste periferice sînt foarte lente faţă de microprocesor şi sînt (mai mult sau mai puţin) controlate de acesta. Microprocesorul poate trata perifericele în două moduri, fiecare cu avantaje şi dezavantaje.

Tratarea perifericelor prin polling (prin program) presupune suspendarea rulării programului curent de către microprocesor şi interogarea perifericelor. Avantajul constă în faptul că programatorul controlează precis momentele cînd programul principal poate fi suspendat. Dezavantajul constă în faptul că un periferic "grăbit" (de exemplu o placă de achiziţie de date) nu îşi poate permite să aştepte un timp nedefinit pentru a fi servit. În plus, interogarea perifericelor se face, şi consumă timp, chiar dacă nici un periferic nu are nimic de comunicat.

Tratarea perifericelor prin întreruperi presupune existenţa unui semnal hardware exterior microprocesorului prin activarea căruia perifericele cer să fie deservite de către acesta. La activarea semnalului de întrerupere, microprocesorul îşi suspendă execuţia programului curent, depistează perifericul care a lansat cererea de întrerupere şi lansează procedura specifică de tratare a acestuia.

Această lucrare prezintă sistemul de întreruperi al microprocesorului 8086 şi apelarea din limbaj de asamblare a funcţiilor BIOS şi DOS, prin întreruperi software.

9.1 Sistemul de întreruperi

De obicei, setul de instrucţiuni al microprocesorului conţine instrucţiuni pentru dezactivarea (mascarea) întreruperilor. Acestea sînt folosite în cazul în care microprocesorul urmează să execute o porţiune critică de program, ce nu trebuie perturbată de evenimente externe. Pentru cazuri deosebite, CPU are un pin dedicat pentru primirea întreruperilor nemascabile.

Microprocesoarele familiei 8086 recunosc 256 de întreruperi. Există posibilitatea apelării acestor întreruperi şi prin program, utilizînd instrucţiuni specifice. Aceste întreruperi sînt denumite *întreruperi software* pentru a fi deosebite de *întreruperile hardware* ce sînt determinate de activarea unui semnal provenit din exteriorul microprocesorului. Întreruperile software sînt tratate la fel ca şi cele hardware cu deosebirea că acestea nu pot fi mascate. Întreruperile software

sînt sincrone cu ceasul sistem şi au loc la momente de timp predictibile, specificate prin program. Întreruperile hardware sînt în general asincrone şi inpredictibile în timp. Instrucţiunea de întrerupere software la microprocesorul 8086 are sintaxa:

INT <număr întrerupere>

Pe baza *<numărului întreruperii>* (între 0 şi 255), microprocesorul determină adresa procedurii de tratare a întreruperii.

9.1.1 Întreruperi externe

Microprocesorul 8086 are doi pini dedicaţi pentru primirea întreruperilor din exterior: *INTR* şi *NMI*. Pinul *INTR* (INTerrupt Request) este comandat de un *controler programabil de întreruperi* (8259A) care, la rîndul său, este conectat la perifericele care pot lansa întreruperi. Circuitul 8259A este comandat de microprocesor prin software, fiind văzut de acesta ca un set de porturi I/O. Sarcina controlerului de întreruperi este de a primi şi ierarhiza întreruperile de la periferice şi de a activa pinul *INTR*.

CPU verifică starea pinului *INTR* la terminarea fiecărei instrucţiuni. Întreruperea este ignorată dacă indicatorul *IF* (Interrupt-enable Flag) este resetat. Starea indicatorului *IF* poate fi controlată prin program cu instrucţiunile *CLI* (Clear IF) şi *STI* (SeT IF). CPU semnalizează acceptarea întreruperi prin executarea a doi cicli de confirmare a întreruperii (INTA - INTerrupt Acknowledge). Primul ciclu INTA semnalizează controlerului de întreruperi că întreruperea a fost acceptată. În al doilea ciclu INTA, controlerul de întreruperi răspunde prin plasarea pe bus-ul de date a numărului întreruperii asociat cu dispozitivul care a lansat întreruperea. Asocierea între periferic şi numărul de întrerupere este făcută prin software, la iniţializarea controlerului de întreruperi. CPU citeşte numărul întreruperii şi îl utilizează pentru a determina adresa procedurii de tratare a întreruperii.

O întrerupere externă poate sosi şi pe pinul denumit *NMI* (Non-Maskable Interrupt). Întreruperile venite pe această linie nu pot fi mascate şi sînt prioritare faţă de întreruperile venite pe pinul *INTR*. Întreruperile nemascabile sînt predefinit asociate cu numărul 2. Din acest motiv, nu mai este necesar ca CPU să execute ciclii INTA şi poate apela imediat procedura de tratare a întreruperii nemascabile.

9.1.2 Întreruperi software

Execuţia instrucţiunilor *INT* (INTerrupt) generează imediat o întrerupere. Numărul întreruperii este inclus în codul instrucţiunii şi permite CPU determinarea imediată a adresei procedurii de tratare a întreruperii. Prin utilizarea întreruperilor software se poate testa procedura de tratare a întreruperii provenite de la un dispozitiv extern.

Întreruperea cu numărul 0 (*Divide error*) este generată de CPU cînd, după o instrucţiune de împărţire, cîtul are dimensiune mai mare decît locaţia specificată ca destinaţie.

Întreruperea cu numărul 1 (*Single Step*) este generată de CPU cînd indicatorul TF (Trap Flag) este setat. În acest caz, microprocesorul intră în modul de lucru de depanare ("pas cu pas").

Dacă indicatorul OF (Overflow Flag) este setat, la terminarea execuţiei unei instrucţiuni INTO (INTerrupt on Overflow) se generează o întrerupere cu numărul 4.

Toate întreruperile interne (INT n, INTO, eroare la împărţire, pas cu pas) au următoarele caracterisitici:

- numărul întreruperii este fie inclus în instrucţiune, fie predefinit;
- nu se execută ciclii INTA;
- nu pot fi mascate, cu excepţia întreruperii de lucru pas cu pas;
- cu excepţia întreruperii de lucru pas cu pas, sînt prioritare faţă de orice întrerupere externă.

9.1.3 Tabela vectorilor de întrerupere

Tabela vectorilor de întrerupere reprezintă legătura dintre numărul întreruperii şi procedura de tratare a întreruperii. Tabela vectorilor de întrerupere ocupă primul 1 KB din memoria sistem. Fiecare din cele 256 de întreruperi are cîte o intrare în această tabelă. Fiecare intrare în tabelă conţine un dublu cuvînt (4 baiţi). Cei mai semnificativi doi baiţi conţin adresa de bază a segmentului în care se găseşte procedura de tratare a întreruperii. Cei mai puţin semnificativi doi baiţi conţin offset-ul procedurii de tratare a întreruperii. CPU calculează adresa intrării în tabel prin înmulţirea numărului întreruperii cu 4.

Figura 9.1 prezintă structura tabelei vectorilor de întrerupere.

9.1.4 Acţiuni executate după acceptarea unei întreruperi

După acceptarea unei întreruperi şi determinarea numărului întreruperii, CPU execută cîteva acţiuni specifice, enumerate în continuare:

- Plasează în stivă registrul de indicatori;
- Execută acţiuni similare unei instrucţiuni *CALL* intersegment indirect. Adresa procedurii este conţinută de elementul aflat la adresa (numărul întreruperii $\times$ 4), în tabela vectorilor de întrerupere;
- Plasează în stivă registrele CS şi IP pentru a se putea continua programul abandonat;
- Resetează indicatorii TF şi IF;
- Înlocuieşte registrele CS şi IP cu al doilea şi primul cuvînt din elementul selectat din tabela vectorilor de întrerupere.

După executarea acestor acţiuni, se dă controlul procedurii de tratare a întreruperii. În cadrul acesteia, întreruperile externe pot fi din nou permise dacă se setează IF cu instrucţiunea STI. Sarcina salvării şi restaurării registrelor folosite în procedură revine programatorului.

Procedura trebuie să se încheie cu instrucţiunea *IRET* (Interrupt RETurn). Ca efect, se restaurează din stivă registrele IP, CS şi de indicatori, controlul revenind în programul întrerupt.

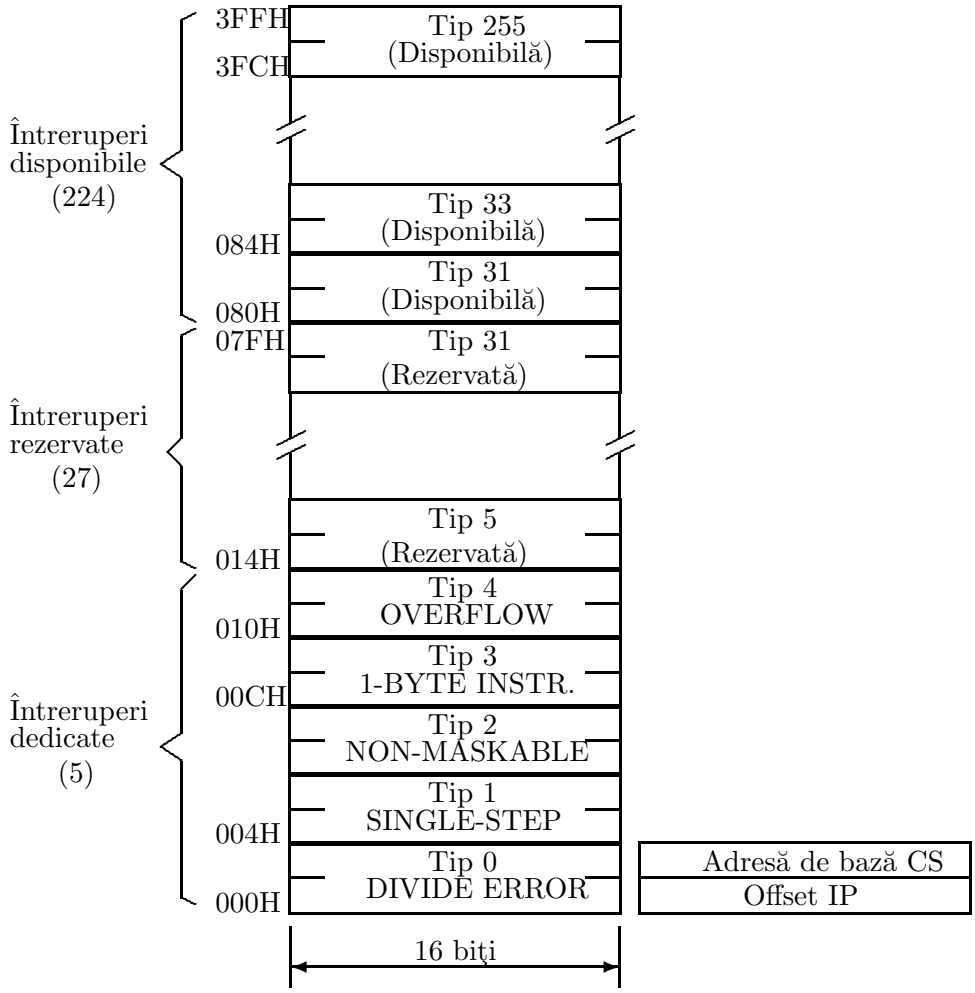


Figura 9.1: Structura tabelei vectorilor de întrerupere.

9.2 Funcţii DOS

Sistemul de operare *DOS* (Disk Operating System) pune la dispoziţia aplicaţiilor un set de rutine (funcţii) pentru gestionarea resurselor sistemului. Utilizarea acestor rutine uşurează munca programatorului permiţindu-i o viziune de nivel înalt asupra aplicaţiei dezvoltate. Mecanismul prin care o aplicaţie poate utiliza un servicu pus la dispoziţie de sistemul de operare este cel al întreruperilor software.

Funcţiile DOS pot fi apelate prin INT 21H atît din limbaj de asamblare cît şi din limbaje de nivel înalt, aşa cum este C. Pentru apelul funcţiilor DOS este rezervată întreruperea 21H. Numărul funcţiei DOS apelate este transmis ca parametru în registrul AH. Parametrii de intrare şi ieşire ai funcţiilor DOS sînt transmişi prin registre.

Pentru a apela o funcţie DOS, programul trebuie să realizeze următoarele acţiuni:

- încarcă în registrele corespunzătoare funcţiei parametrii de intrare;
- încarcă în registrul AL codul subfuncţiei (dacă este necesar);
- încarcă în registrul AH codul funcţiei;

- lansează instrucţiunea *INT 21H*.

La terminarea procedurii, programul trebuie să preia eventualii parametrii de ieşire din registrele corespunzătoare funcţiei.

Lista principalelor funcţii DOS este prezentată în tabelul 9.1

Funcţie DOS	*Denumire*
01H	Character input with echo
02H	Output character
08H	Character input without echo
09H	Output character string
25H	Set interrupt vector
2AH	Get system date
2BH	Set system date
2CH	Get system time
2DH	Set system time
31H	Terminate and stay resident
35H	Get interrupt vector
39H	Create subdirectory (MKDIR)
3AH	Remove directory entry (RMDIR)
3BH	Set directory (CHDIR)
3CH	Create file (CREAT)
3DH	Open file (OPEN)
3EH	Close file (CLOSE)
3FH	Read file (READ)
40H	Write file (WRITE)
41H	Delete file (UNLINK)
43H	Get/set file attributes
4BH	Load or execute program (EXEC)
4CH	Process terminate (EXIT)
4DH	Get return code of a subprocess
56H	Rename file
57H	Get/set file date and time
62H	Get program segment prefix (PSP) address

Tabelul 9.1: Funcţii DOS.

9.2.1 Exemplul 1: Preluare caracter de la tastatură, cu ecou/scriere caracter pe ecran

Acest exemplu prezintă un program în limbaj de asamblare care citeşte de la tastatură un număr de două cifre, cu ecou. Cele două cifre sînt interpretate ca fiind un număr reprezentat în baza 10. Pe o linie nouă, se va afişa restul împărţirii cu 9 a numărului introdus.

Pseudocodul algoritmului implementat de program este următorul:
- preia primul caracter, cu ecou, prin utilizarea funcţiei DOS 01H (character input with echo);

- converteşte primul caracter la valoarea numerică prin scăderea codului ASCII al caracterului '0';
- preia al doilea caracter, cu ecou;
- converteşte al doilea caracter la valoarea numerică;
- compune numărul introdus (prima cifră × 10 + a doua cifră);
- împarte numărul la 9 şi păstrează restul;
- converteşte numărul la codul ASCII al caracterului corespunzător prin adunarea codului ASCII al caracterului '0';
- poziţionează cursorul la începutul liniei următoare;
- afişează caracterul prin utilizarea funcţiei DOS 02H (character output);
- termină programul prin utilizarea funcţiei DOS 4CH (process terminate).

Programul face apel la trei funcţii DOS:

Funcţie DOS 01H (character input with echo)

Registre la intrare:
AH: 1

Registre la ieşire:
AL: caracter

Descriere: Aşteaptă introducerea unui caracter de la tastatură. În momentul introducerii unui caracter, codul ASCII asociat acestuia este returnat în registrul AL iar simbolul caracterului este afişat pe ecran la poziţia curentă a cursorului. Dacă tastatura generează un cod ASCII extins (cazul apăsării une taste funcţionale sau taste de control), atunci funcţia returnează 0 în registrul AL. Invocarea ulterioară a funcţiei returnează codul tastei (*scan code*) fără a mai aştepta apăsarea unei taste.

Funcţie DOS 02H (character output)

Registre la intrare:
AH: 2
DL: cod ASCII de caracter

Registre la ieşire:
nemodificate

Descriere: Conţinutul registrului DL este trimis spre dispozitivul standard de ieşire (ecran). Simbolul asociat codului ASCII al caracterului este afişat la poziţia curentă a cursorului.

Funcţie DOS 4CH (process terminate)

Registre la intrare:
AH: 4CH
AL: cod returnat

Registre la ieşire:
nemodificate

Descriere: Termină programul care a apelat funcţia şi întoarce controlul programului părinte (care a apelat programul) sau sistemului de operare. Codul returnat poate fi determinat de programul părinte prin apelul funcţiei DOS 4DH. Dacă programul a fost lansat din DOS, codul de retur este disponibil prin intermediul variabilei ERRORLEVEL, în fişiere de tip *.BAT*.

Codul sursă al programului ce rezolvă problema enunţată este prezentat în continuare.

```
code segment
  assume cs:code, ds:code
  CR equ 0AH
  LF equ 0DH

start:
  mov ax, code
  mov ds, ax      ; suprapune segmentul de date peste cel de cod
  mov bh, 0

  mov ah, 1       ; citire tastatură prin serviciu DOS 01H
  int 21h

  sub al, '0'     ; prima cifră în AL
  mov cl, 10
  mul cl          ; AL <- AL * 10
  mov bl, al      ; prima cifră în BL înmulţită cu 10

  mov ah, 1       ; citire a doua cifră prin apel DOS
  int 21h

  sub al, '0'
  add al, bl
  mov cl, 9
  mov ah, 0
  div cl
  mov bl, ah      ; restul este în BL
  add bl, '0'
  call afisare    ; afişare şir de caractere

  mov ax, 4C00h   ; terminare program (apel funcţie DOS 4CH)
  int 21h

afisare PROC      ; afişare şir de caractere prin întrerupere
  mov dl, CR
  mov ah, 2
  int 21h
  mov dl, LF
  int 21h
  mov dl, bl
  int 21h
  ret
afisare ENDP

code ENDS
END start
```

9.2.2 Exemplul 2: Citire caracter fără ecou

Acest exemplu prezintă un program care primeşte de la tastatură un şir de cuvinte pe care le afişează pe ecran. Toate cuvintele vor avea iniţiala majusculă, indiferent de modul în care a fost tastată (majusculă sau minusculă). Terminarea programului se va face la apăsarea tastei ESC.

Programul necesită prelucrarea fiecărui caracter primit de la tastatură înainte de a fi afişat pe ecran. Soluţia constă în preluarea caracterului cu funcţia DOS 8 (character input without echo), analizarea şi eventuala prelucrare a acestuia şi, ulterior, afişarea caracterului cu funcţia DOS 2 (character output).

Codul sursă al programului ce rezolvă problema enunţată este prezentat în continuare.

```
code SEGMENT
assume cs:code, ds:code
tasta_ESC equ 27

start:
  mov   ax, cs
  mov   ds, ax
  mov   bl, 1           ; indicator: BL=1 - converteşte caracterul în majusculă
                        ;            BL=0 - lasă caracterul neschimbat

caracter_in:
  mov   ah, 8
  int   21h             ; apel funcţie DOS 8 (character input without echo)
                        ; returnează în AL codul tastei apăsate
  cmp   al, tasta_ESC
  jz    final           ; s-a tastat ESC

  cmp   al, ' '
  jz    schimb_maj      ; s-a tastat ' ' (spaţiu=delimitator de cuvinte)
                        ; următorul caracter este transformat în majusculă
  cmp   bl, 0           ;
  jz    scrie_al        ; caracterul se lasă neschimbat

  mov   bl, 0           ; este o iniţială...
  cmp   al, 'a'
  jl    scrie_al        ; cu cod ASCII mai mic decît 'a'...
  cmp   al, 'z'
  jg    scrie_al        ; sau mai mare decît 'z' (în afara domeniului
                        ; minusculelor)
  sub   al, 20H         ; transformă minuscula în majusculă
  jmp   scrie_al

schimb_maj:
   mov  bl, 1

scrie_al:               ; afişează caracterul
```

```
   mov  ah, 2
   mov  dl, al
   int  21h
   jmp  caracter_in

final:
   mov  ah, 4ch
   int  21h
code ends
end start
```

9.2.3 Exemplul 3: Preluare dată sistem

Acest exemplu prezintă un program care afişează pe ecran data sistemului sub forma:

<*AA*> <*Luna*>

unde:
<*AA*> este anul (ultimele două cifre);
<*Luna*> este denumirea lunii în limba română.

Codul sursă al programului ce rezolvă problema enunţată este prezentat în continuare.

```
code SEGMENT
assume cs:code, ds:code
Denumire_luna db 'Ianuarie$  Februarie$ Martie$    Aprilie$   '
              db 'Mai$       Iunie$     Iulie$     August$    '
              db 'Septembrie$Octombrie$ Noiembrie$ Decembrie$ '

start:
  mov  ax, cs
  mov  ds, ax

  mov  ah, 2Ah          ; preia data sistem, funcţie DOS 2AH
  int  21h
                        ; preluare parametrii, prelucrare şi afişare
  sub  cx, 1900         ; CX=anul (parametru de ieşire din funcţia DOS 2AH)
  mov  ah, 0
  mov  al, cl
  mov  bl, 10
  div  bl               ; (AH:AX) / BL => AL=cît, AH = rest
                        ; AL=cifra zecilor, AH=cifra unităţilor (AN)
  mov  bx, ax
  mov  dl, bl
  add  dl, '0'
  mov  ah, 2
  int  21h              ; scrie cifra zecilor din reprezentarea anului

  mov  dl, bh
  add  dl, '0'
```

```
  mov  ah, 2
  int  21h              ; scrie cifra unităţilor din reprezentarea anului

  mov  dl, ' '
  mov  ah, 2
  int  21h              ; scrie ' '
                        ; DH=luna (parametru de ieşire din funcţia DOS 2AH)

  dec  dh               ; DH=index în tabelul Denumire_luna
  mov  al, 11           ; numărul de baiţi alocaţi în tabelul Denumire_luna
                        ; pentru o intrare (o denumire de lună)
  mul  dh               ; AL x DH = (AH:AL), index sir
  mov  dx, offset Denumire_luna
  add  dx, ax           ; DX=pointer la denumirea lunii curente, şir terminat
                        ; cu '$'

  mov  ah, 9            ; afişează şir cu funcţia DOS 9
  int  21h
  mov  ah, 4ch
  int  21h
code ends
end start
```

Programul face apel la două funcţii DOS:

Funcţie DOS 2AH (get system date)

Registre la intrare:
AH: 2AH

Registre la ieşire:
AL: ziua din săptămînă
CX: anul
DH: luna
DL: ziua

Descriere: Se returnează ziua (DL), luna (DH) şi anul (CX) datei sistem. În registrul AL se returnează codul zilei din săptămînă: 0 = duminică, 1 = luni, 2 = marţi, etc.

Funcţie DOS 9 (output character string)

Registre la intrare:
AH: 9
DX: offset adresă şir
DS: segment adresă şir

Registre la ieşire:
nemodificate

Descriere: Se afişează pe ecran şirul de caractere care începe la adresa DS:DX. Terminatorul de şir, care nu se afişează, este caracterul '$' (cod ASCII 36, 24H). Prin această funcţie nu se poate afişa caracterul '$'.

Categorie	*Întrerupere*
Servicii video	10H
Servicii tastatură	16H
Servicii disk	13H
Servicii imprimantă	17H
Servicii port comunicaţii	14H
Servicii dată/timp	1AH
Servicii unitate de bandă	15H
Servicii sistem	11H, 12H, 19H

Tabelul 9.2: Categorii de servicii BIOS.

Denumirea în limba română a lunilor anului a fost specificată sub forma unei definiţii de date de tip byte. Şirul a fost conceput astfel încît să se aloce fiecărei luni acelaşi număr de caractere (11). Adresa care trebuie transmisă ca parametru funcţiei DOS 9 este calculată după formula: *Denumire_luna + 11 × (luna curentă - 1).*

Funcţia DOS 2AH (get system date) se apelează chiar la începutul programului. Restul programului se ocupă cu prelucrarea şi afişarea parametrilor returnaţi de această funcţie. Anul este preluat în registrul CX şi este prelucrat pentru a se obţine cele două caractere asociate cifrelor zecilor şi unităţilor. Apoi este afişat un caracter spaţiu. Pe baza numărului lunii, returnat în registrul DH, se calculează adresa care trebuie transmisă funcţiei DOS 9 (output character string).

9.3 Funcţii BIOS

BIOS (Basic Input/Output System) este cel mai de jos nivel de software care interacţionează cu structura hardware a calculatorului. BIOS reprezintă un set de proceduri conţinute în memoria ROM a sistemului. Funcţiile BIOS sînt disponibile pentru a fi apelate din programe, indiferent de sistemul de operare.

Categoriile de servicii BIOS şi numerele întreruperilor asociate sînt prezentate în tabelul 9.2.

Lista principalelor funcţii BIOS este prezentată în tabelul 9.3.

9.3.1 Exemplul 4: Servicii video BIOS

Acest exemplu prezintă un program în limbaj de asamblare care citeşte de la tastatură cinci cifre cu ecou. Ulterior, programul afişează cu intermitenţă pe cea mai mică. Dacă cifra minimă apare de mai multe ori, se va afişa cu intermitenţă prima apariţie (cea mai din stînga).

Pseudocodul algoritmului implementat de program este următorul:
- citeşte (BIOS 10h, serviciul 3) şi memorează (pe stivă) poziţia cursorului la începutul programului;
- pregăteşte registre pentru codul ASCII (DL) şi coloana caracterului (DH) minim;
- citeşte cîte un caracter (DOS 1) şi, dacă este necesar, actualizează minimul;
- determină şi plasează cursorul (BIOS 10h, serviciul 2) pe poziţia caracterului minim;

Serviciu BIOS - Întrerupere	*Funcţie*	*Denumire*
Servicii video - INT 10H	00H	Set video mode
	01H	Set cursor size
	02H	Set cursor position
	03H	Read cursor position
	06H	Scroll window up
	07H	Scroll window down
	08H	Read character and attribute
	09H	Write character and attribute
	0CH	Write pixel dot
	0DH	Read pixel dot
	0EH	TTY character output
	0FH	Get current video state
	13H	Write string
Servicii tastatură - INT 16H	00H	Read keyboard character
	01H	Read keyboard status
	02H	Read keyboard shift status
Servicii disk - INT 13H	01H	Get floppy disk status
	02H	Read disk sectors
	03H	Write disk sectors
	04H	Verify disk sectors
	08H	Return drive parameters
Servicii imprimantă - INT 17H	00H	Print character
	01H	Initialize printer
	02H	Get printer status
Servicii port comunicaţii - INT 14H	00H	Initialize communication port
	01H	Transmit character
	02H	Receive character
	03H	Get communication port status
Servicii dată/timp - INT 1AH	00H	Get clock counter
	01H	Set clock counter
	06H	Set alarm
	07H	Disable alarm
	09H	Read alarm
Servicii sistem - INT 12H		Get memory size
Servicii sistem - INT 19H		Warm boot

Tabelul 9.3: Funcţii BIOS.

- afişează caracterul minim pe poziţia cursorului, cu intermitenţă (BIOS 10h, serviciul 9).

Programul face apel la trei funcţii BIOS din categoria video (INT 10h):

Serviciul 3 (read cursor position and size)

Registre la intrare:
AH: 3
BH: număr pagină video

Registre la ieşire:
BH: număr pagină video
CH: început linie cursor
CL: sfîrşit linie cursor
DH: rînd cursor
DL: coloană cursor

Descriere: Returnează caracteristicile cursorului, în funcţie de modul grafic curent.

Serviciul 2 (set cursor position)

Registre la intrare:
AH: 2
BH: număr pagină video
DH: rînd cursor
DL: coloană cursor

Registre la ieşire:
nemodificate

Descriere: Poziţionează cursorul pe ecran pe rîndul şi coloana transmise prin DH şi DL.

Serviciul 9 (write character and attribute)

Registre la intrare:
AH: 9
AL: cod ASCII caracter
BH: număr pagină video
BL: atribut video al caracterului din AL
CX: număr de caractere afişate

Registre la ieşire:
nemodificate

Descriere: Afişează unul sau mai multe caractere pe ecran. Atributul de culoare este transmis prin BL iar numărul de caractere prin CX. Poziţia cursorului nu se modifică nici dacă se afişează mai mult de un caracter. Semnificaţia biţilor registrului BL este prezentată în figura 9.2.

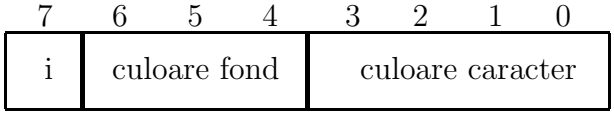


Figura 9.2: Semnificaţia biţilor registrului de atribut video BL.

Tabelul 9.4 prezintă codificarea culorilor caracterelor. Culoarea de fond este reprezentată numai pe 3 biţi. În consecinţă, sînt posibile numai primele 8 culori (culorile închise). Dacă bitul 7 (MSB) al registrului BL este 1, afişarea caracterului se va face cu intermitenţă.

Cod	*Culoare*	*Cod*	*Culoare*
00H	negru	08H	gri
01H	albastru	09H	albastru intens
02H	verde	0aH	verde intens
03H	cyan	0bH	cyan intens
04H	roşu	0cH	roşu intens
05H	magenta	0dH	magenta intens
06H	maro	0eH	galben
07H	alb	0fH	alb intens

Tabelul 9.4: Codificarea culorilor caracterelor.

Codul sursă al programului ce rezolvă problema enunţată este prezentat în continuare.

```
code SEGMENT
assume cs:code, ds:code

start:
  mov  ax, cs
  mov  ds, ax

  mov  ah, 3
  mov  bh, 0
  int  10h          ; citeşte poziţia şi dimensiunea cursorului
                    ; (funcţie BIOS 10h, serviciu 3)

  push dx           ; poziţie cursor la intrare în program
                    ; DH - rînd, DL - coloană
  mov  cx, 5        ; contor număr de caractere citite
  mov  dl, '9'      ; caracterul cel mai mare posibil
  mov  ah, 1        ; DL va conţine codul ASCII al caracterul minim
                    ; DH va conţine coloana caracterului minim

caracter_in:
  int  21h          ; DOS 1 (citeşte caracter în AL)
  cmp  dl, al
  jg   schimb       ; DL>AL implică înlocuire caracter minim din DL

intorc:
  dec  cx           ; actualizează contor caractere
  jnz  caracter_in
  jmp  afisare      ; au intrat 5 caractere, se afişează minimul
```

```
schimb:
  mov  dl, al
  mov  dh, 5
  sub  dh, cl
  jmp  intorc

afisare:
  ; DL conţine codul ASCII al caracterului minim
  ; DH conţine coloana caracterului minim

  ; pregăteşte parametrii de intrare pentru întrerupere BIOS 10h, serviciu 2
  ; (set cursor position)
  pop  cx           ; CX <= poziţie cursor la intrare în program
  push dx           ; pune în stivă DL, DH
  push cx           ; repune în stivă poziţia iniţială a cursorului
  mov  ah, 2        ; număr serviciu BIOS video
  mov  bh, 0        ; număr pagină video
  xor  cx, cx       ; CX=(CH:CL) <= 0
  mov  cl, dh       ; CL <= coloana caracterului minim
  pop  dx           ; DX <= poziţie cursor la intrare în program
  add  dx, cx       ; DH=rînd şi DL=coloană caracter minim
  int  10h          ; apelul întreruperii software

  ; pregăteşte parametrii de intrare pentru întrerupere BIOS 10h, serviciu 9
  ; (write character and attribute)
  mov  ah, 9        ; număr serviciu BIOS video
  pop  dx
  mov  al, dl       ; cod ASCII caracter
  mov  bh, 0        ; număr pagină video
  mov  bl, 128+7    ; atributul caracterului (alb pe fond negru, cu intermitenţă)
  mov  cx, 1        ; numărul de caractere
  int  10h          ; apelul întreruperii software

  mov  ah, 4ch
  int  21h
code ends
end start
```

9.4 Experimente

I. Studiaţi programul *CICLU_A.ASM*. Programul şterge ecranul (BIOS video, int 10h, serviciul 6), şi citeşte ciclic un caracter de la tastatură prin funcţii BIOS, int 16h, serviciul 0. În cazul în care se apasă tasta 'A' (caracter 'A' sau 'a'), programul afişează un mesaj în centrul ecranului.

II. Rezolvarea propusă de exemplul 3 nu este "compatibilă cu anul 2000". Modificaţi programul astfel încît să realizaţi compatibilitatea. Estimaţi costul operaţiei considerînd următoarele metrici:

- număr de linii de cod modificate;
- număr de iteraţii (asamblare, link-editare, depanare) necesare;
- număr de variante testate prin simulare;
- timp de execuţie.

Considerînd costul manoperei şi al amortizării calculatorului folosit, estimaţi costul pe care ar trebui să-l plătească beneficiarul.

III. Folosind bibliografia şi programul de documentare hipertext $TECHhelp$, studiaţi parametrii de intrare, parametrii de ieşire şi acţiunile funcţiilor DOS specificate în tabelul 9.1 şi a funcţiilor BIOS specificate în tabelul 9.3.

Lucrarea 10

Noţiuni avansate de programare în limbaj de asamblare

Această lucrare prezintă modul de utilizare a limbajului de asamblare în sistem de operare DOS pentru:

- Accesarea fişierelor prin funcţii DOS;
- Scrierea programelor rezidente.

10.1 Accesarea fişierelor din limbaj de asamblare

Deşi limbajele de nivel înalt pun la dispoziţia programatorului un set larg de funcţii pentru accesarea fişierelor, uneori este nevoie de a accesa fişiere direct din limbaj de asamblare. Pentru aceasta, sistemul de operare DOS furnizează funcţii sistem dedicate, apelabile prin întreruperi software.

10.1.1 Formate de fişiere executabile

DOS este un sistem de operare capabil să încarce în memorie şi să execute două tipuri de fişiere program, avînd extensiile *.COM* şi *.EXE*. Ambele tipuri de programe sînt relocatabile (pot fi încărcate la orice adresă fizică). Acest lucru este asigurat prin mecanismul de segmentare a memoriei şi prin faptul că toate salturile în program sînt realizate cu instrucţiuni de salt relativ.

Fişierul în formatul COM conţine o imagine binară a codului şi datelor programului. Se poate considera că programele *COM* sînt scrise într-un model de memorie *"tiny"*, conform tabelului 8.2. Codul, datele şi stiva împreună nu pot avea o dimensiune mai mare de un segment (64 KB).

Fişierul în formatul EXE conţine, în plus faţă de fişierul în format *COM*, un antet care informează sistemul de operare despre modul de gestiune a segmentelor.

Înainte de a încărca un program în memorie, DOS selectează o adresă de segment la care va încărca programul. Ca adresă de bază, DOS alege totdeauna cea mai mică adresă de memorie

liberă. Indiferent de formatul fişierului (*COM* sau *EXE*), la începutul zonei de memorie rezervate pentru program se încarcă un prefix de 256 de baiţi numit *PSP* (*Program Segment Prefix*). PSP conţine informaţii pe care sistemul de operare le pune la dispoziţia programului. Structura PSP-ului este prezentată în tabelul 10.1.

După ce controlul este predat programului încărcat în memorie, registrele *DS* şi *ES* conţin adresa PSP. Informaţia din PSP poate fi utilizată în cadrul programului pentru a prelua argumentele din linia de comandă, pentru a determina cîtă memorie este disponibilă programului sau a pentru a accesa variabilele de sistem. Adresa PSP poate fi accesată ulterior prin funcţiile DOS 62*H* (*query current PSP*).

Offset	*Dimensiune*	*Semnificaţie*
0	2	Codul instrucţiunii `INT 20H` (`CD 20`). Printr-o instrucţiune de salt la această adresă se poate termina programul.
2	2	Adresa de sfîrşit a memoriei alocate programului, exprimată în blocuri de 16 baiţi.
4	1	Rezervat.
5	5	Codul instrucţiunii de apel de procedură de tip FAR spre punctul de intrare al funcţiilor DOS.
0AH	4	Adresă de revenire în sistemul de operare la execuţia instrucţiunilor `INT 22H` sau `INT 21H` funcţia `4CH`.
0EH	4	Adresă de tratare a întreruperii `INT 23H` (Ctrl-Break).
12H	4	Adresă de tratare a întreruperii `INT 24H` (erori critice DOS).
16H	22	Rezervat.
2CH	2	Adresă de segment a variabilelor DOS.
2EH	46	Rezervat.
5CH	16	FCB pentru primul parametru.
6CH	20	FCB pentru al doilea parametru.
80H	1	Numărul de baiţi din linia de comandă (fără numele programului).
81H	127	Linia de comandă a programului (fără numele programului şi fără directive de redirectare).

Tabelul 10.1: Structura PSP.

Imediat după încărcarea unui program *EXE* în memorie:

- registrele *DS* şi *ES* conţin adresa PSP;
- registrele *CS*, *IP*, *SS* şi *SP* conţin valorile indicate în antetul fişierului *EXE*;
- cîmpul al doilea din PSP (adresa sfîrşitului memoriei disponibile) conţine valoarea din antetul fişierului *EXE*.

Imediat după încărcarea unui program *COM* în memorie:

- registrele *CS*, *DS*, *ES* şi *SS* conţin adresa PSP;
- registrul *SS* conţine adresa de sfîrşit a segmentului (de obicei `0FFFEH`);

- registrul IP este iniţializat cu valoarea `0100H`.

Utilizarea formatului COM presupune respectarea următoarelor constrîngeri:

- Programul trebuie să fie format dintr-un singur segment;
- Codul sursă trebuie să înceapă cu o pseudoinstrucţiune `ORG 100H`. Programul trebuie să înceapă cu o instrucţiune executabilă a cărei etichetă trebuie să apară în pseudoinstrucţiunea `END`, care încheie fizic textul programului sursă. Această instrucţiune este plasată la adresa 100H şi poate fi o instrucţiune de salt la adresa la care începe efectiv programul;
- Datele pot fi plasate oriunde în codul sursă dar este de preferat să se plaseze la începutul acestuia deoarece asamblorul poate semnala o eroare în cazul unor referinţe la date nedeclarate încă;
- Nu este necesară iniţializarea registrelor segment, toate avînd aceeaşi valoare ca şi registrul CS;
- Stiva este iniţializată automat la sfîrşitul segmentului ($SS = CS$ iar $SP =$ `0FFFE0H`);
- Încheierea execuţiei programului se face prin execuţia secvenţei de instrucţiuni:
  ```
  mov  ax, 4c00H
  int 21H
  ```

Formatul unui fişier sursă în limbaj de asamblare pentru generarea unui fişier executabil în format COM este prezentat în continuare. Pentru a genera acest format de fişier executabil, link-editorul trebuie lansat cu opţiunea */t*.

```
code  SEGMENT
assume cs:code, ds:code
org  100H

start:
  jmp  inceput

;
; Definiţii de date
;

data1 db  'Sir de caractere'   ; exemple
a1    dw  10

;
; Codul programului
;

start:
  ...                          ; instrucţiuni în limbaj de asamblare
  ...
```

```
;
; Sfîrşit execuţie program
;

  mov  ax, 4c00H
  int  21H

code  ENDS

end start
```

10.1.2 Funcţii DOS pentru lucru cu fişiere

Începînd cu versiunea 2.0, sistemul de operare DOS pune la dispoziţia programelor utilizatorului cîteva funcţii apelabile prin întreruperea software $21H$, similare cu funcţiile UNIX. Prin aceste funcţii se pot realiza atît operaţii elementare asupra fişierelor (citire, scriere) cît şi gestionarea acestora (creare, deschidere, închidere, ştergere).

Un fişier pe disc este identificat pe baza numelui său şi a directorului în care se află. Numele fişierului este transmis ca parametru funcţiilor DOS sub forma unui pointer la un şir de caractere aflate în memorie. În cadrul unui program, un fişier este identificat pe baza unui *identificator de fişier* (*file handle*, în limba engleză). Identificatorul fişierului este un număr de 16 biţi returnat de funcţiile de creare sau deschidere de fişiere. Operaţiile ulterioare de accesare a fişierului (căutare, citire, scriere) se fac pe baza identificatorului de fişier. Funcţiile DOS returnează identificatorul unui fişier în registrul de 16 biţi BX.

Există cîţiva identificatori rezervaţi de către sistemul de operare DOS pentru anumite dispozitive din calculator. În acest fel, aceste dispozitive pot fi accesate cu aceleaşi funcţii ca şi fişierele. Tabelul 10.2 prezintă identificatorii rezervaţi pentru dispozitivele standard dintr-un calculator PC. Identificatorii standard sînt automat iniţializaţi la începutul programului.

Identificator	*Descriere*
0000H	Dispozitiv standard de intrare (tastatură)
0001H	Dispozitiv standard de ieşire (monitor)
0002H	Dispozitiv standard pentru afişarea erorilor (monitor)
0003H	Dispozitiv standard auxiliar (COM1)
0004H	Imprimantă standard (LPT1)

Tabelul 10.2: Identificatorii rezervaţi pentru dispozitivele standard dintr-un calculator PC.

Tabelul 10.3 prezintă funcţiile DOS din categoria *serviciilor de disc*. Toate funcţiile returnează în registrul AX un cod de eroare asociat operaţiei realizate de funcţie.

Funcţie DOS	*Denumire funcţie*
3CH	Creare fişier
3DH	Deschidere fişier
3EH	Închidere fişier
3FH	Citire din fişier
40H	Scriere în fişier
41H	Ştergere fişier
5BH	Creare fişier
44H	Controlul dispozitivului I/O

Tabelul 10.3: Funcţii DOS din categoria serviciilor de disc.

10.1.3 Exemplul 1: Program în limbaj de asamblare pentru copierea unui fişier

Acest exemplu prezintă un program pentru copierea unui fişier sub un alt nume. Dacă există deja un fişier cu numele celui de destinaţie, copierea se abandonează şi se returnează un mesaj de atenţionare.

Pseudocodul programului este prezentat în continuare.
- Identifică argumentele liniei de comandă, aflate în PSP la offset 81h;
- În şirul de argumente, caută delimitatorii (caractere spaţiu ' ') şi identifică argumentele;
- Dacă linia de comandă are două argumentele, continuă. Altfel, termină programul cu afişarea mesajului cu sintaxa liniei de comandă:

```
Sintaxă: copiere <sursă> <destinaţie>
```

- Deschide fişierul sursă în citire:

```
mov ax, 3d00h
mov dx, offset cale_sursa
int 21h
```

- Dacă operaţia anterioară nu a generat eroare, continuă. Altfel, termină programul cu afişarea mesajului:

```
Eroare deschidere fişier sursă
```

- Salvează identificatorul fişierului sursă;
- Crează fişierul destinaţie:

```
mov ah, 5bh
mov cx, 0
int 21h
```

- Dacă operaţia anterioară nu a generat eroare, continuă. Altfel, înseamnă că numele fişierului destinaţie nu este valid sau există un fişier cu acelaşi nume. În acest caz, termină programul cu afişarea mesajului:

```
Eroare deschidere fişier destinaţie
```

- Salvează identificatorul fişierului destinaţie;
- Citeşte cîte un buffer din fişierul sursă şi scrie bufferul în fişierul destinaţie. Cînd ultimul buffer nu este plin, înseamnă că s-a terminat fişierul sursă:

```
copiere:
```

```
  mov ah, 3fh                 ; citire din fişier <sursă>
  mov bx, [handle1]
  mov dx, offset buff_rw      ; adresa bufferului unde se vor depune octeţii
  mov cx, 1024                ; CX conţine numărul de octeţi citiţi
  int 21h

  cmp ax, cx                  ; cînd numărul de octeţi citiţi este mai mic
  je buffer_intreg            ; decît capacitatea bufferului, va fi ultima
  mov [cond], 1               ; citire
buffer_intreg:

  mov cx, ax
  mov ah, 40h                 ; scriere în fişier <destinaţie>
  mov bx, [handle2]
  int 21h

  cmp [cond], 1
  je inchidere
  jmp copiere
```

- Închidere fişiere sursă şi destinaţie:

```
inchidere:
  mov ah, 3eh
  mov bx, [handle1]
  int 21h
  mov ah, 3eh
  mov bx, [handle2]
  int 21h
```

10.2 Programe rezidente

Sistemul de operare DOS permite unui program să rămînă în memorie şi să predea controlul programului ce l-a lansat (programului părinte) sau sistemului de operare. Programul rămas rezident poate fi reactivat de către un eveniment extern (apăsarea unei taste - *INT 09H*, sau orice întrerupere software sau hardware). În acest mod, deşi la un moment dat se execută un singur porgram, prin comutarea rapidă între programe, se poate crea impresia de 'multitasking'. Terminarea unui program care să rămînă rezident în memorie se face prin apelul funcţiei DOS $31H$. Apelul funcţiei de rămînere rezident trebuie precedat de stabilirea condiţiilor de reactivare. Condiţiile de reactivare sînt stabilite prin modificarea ('deturnarea') unor întreruperi. Programele de acest fel sînt cunoscute ca programe TSR (*Terminate and Stay Resident*).

Descrierea funcţiei DOS 31H este prezentată în tabelul 10.4.

Structura generală a programelor TSR este prezentată în continuare.

- Se verifică dacă programul lansat este deja rezident: se verifică o locaţie de memorie aflată la o adresă fixă, stabilită prin programul TSR;

Registre la intrare	*Descriere*
AH: 31H AL: cod returnat DX: dimensiune memorie rezervată	Termină programul dar îl păstrează rezident în memorie. Codul de retur din AL este întors programului părinte sau sistemului de operare DOS prin variabila ERRORLEVEL. Codul returnat poate fi determinat cu funcţia DOS 4DH.

Tabelul 10.4: Descrierea funcţiei DOS 31H (terminare program prin rămînere rezident în memorie).

- Dacă programul nu este deja rezident (este la prima rulare) atunci se pregăteşte rămînerea rezidentă:
  - Se deturnează întreruperile folosite la reactivare (tastatură 09H, hard-disk 13H, etc.);
  - Se eliberează memoria alocată programului TSR (funcţie DOS 49H);
  - Se termină programul cu rămînere rezidentă în memorie prin apelul funcţiei DOS 31H (*Terminate and Stay Resident*).
- Dacă programul este deja rezident (nu este la prima rulare) se realizează următoarele acţiuni:
  - Se verifică dacă este o comandă de dezinstalare prin evaluarea argumentelor liniei de comandă care se obţine din PSP (adresa de intrare în program ES:0081);
  - Dacă este o comandă de dezinstalare, se refac vectorii întreruperilor deturnate.
- Se termină programul prin funcţia DOS 4CH.

10.3 Experimente şi întrebări

I. Care sînt diferenţele între cele două formatele de fişiere executabile în sistem de operare DOS (EXE şi COM)?

II. Enumeraţi acţiunile care au loc de la lansarea în execuţie a unui program (după tastarea numelui programului la linia de comandă) şi pînă la începerea execuţiei efective a acestuia.

III. Utilizînd *Turbo Debugger*, verificaţi conţinutul registrelor la începerea programului lansat (format EXE şi COM).

IV. Folosind programul de documentare hipertext $TECHhelp$, studiaţi antetul fişierelor EXE (*DOS kernel/EXE File Header Layout*). Folosind un editor de texte hexa (*nc.exe*, $F3$, $F4$) vizualizaţi antetul unui fişier executabil. Ulterior, folosind *Turbo Debugger*, vizualizaţi PSP-ul aceluiaşi program, după ce a fost încărcat în memorie. Faceţi corespondenţa între datele existente în antetul fişierului (date statice) şi cele existente în PSP (date dinamice).

V. Folosind programul de documentare hipertext $TECHhelp$ realizaţi un tabel cu descrierea funcţiilor DOS pentru lucru cu fişiere. Pentru fiecare din funcţiile prezentate în tabelul 10.3, realizaţi un tabel similar celui din figura 10.5, corespunzător funcţiei 3FH (citire din fişier).

Registre		*Descriere*
intrare	*ieşire*	
AX: 3FH BX: identificator fişier CX: nr. baiţi citiţi DX: offset buffer DS: segment buffer	AX: cod eroare	Citeşte informaţie dintr-un fişier. Identificatorul de fişier este în BX, iar bufferul în care se trimit datele citite este la adresa DS:DX. CX conţine numărul de baiţi care se citesc din fişier. Prima citire se face de la începutul fişierului. Citirile ulterioare se fac din poziţia de unde s-a terminat citirea anterioară. Dacă a apărut o eroare la citire se setează CF iar în registrul AX se returnează codul de eroare. Dacă citirea s-a făcut fără eroare, CF este resetat iar în registrul AX se returnează numărul de baiţi citiţi.

Tabelul 10.5: Descrierea funcţiei DOS 3FH, citire din fişier.

VI. Un fişier poate fi creat fie cu funcţia DOS 3CH, fie cu funcţia DOS 5BH. Care este diferenţa între cele două funcţii DOS?

VII. Studiaţi fişierul *COPIERE.ASM*, ce conţine un program în limbaj de asamblare pentru copierea unui fişier. Testaţi programul cu diferite tipuri de argumente şi aduceţi-i îmbunătăţiri.

VIII. Studiaţi fişierul *ALTR.ASM*, ce conţine un program în limbaj de asamblare capabil să rămînă rezident în memorie. Programul se reactivează la apăsarea combinaţiei de taste 'Alt-R'. La reactivare, programul scrie un mesaj pe ecran. Comanda de dezinstalare a programului rezident este `altr /u`. Modificaţi programul pentru a se reactiva şi la alte combinaţii de taste şi pentru a avea o interfaţă mai 'prietenoasă'. De exemplu, sintaxa linie de comandă să fie:

altr [/u][/h]

Dacă se lansează programul cu opţiunea */h*, apar pe ecran informaţii care descriu sintaxa liniei de comandă.

Lucrarea 11

Teme de programare în limbaj de asamblare

Această lucrare prezintă o parte din temele propuse la colocviul de laborator, în perioada 1995-1998, de Catedra de Electronică şi Calculatoare.

I. Scrieţi o funcţie care să caute un caracter într-un şir de caractere. Funcţia returnează numărul primei poziţii a caracterului în şir. Dacă respectivul caracter nu apare în şir, funcţia returnează 0. Modul de transmitere al parametrilor este la latitudinea programatorului.

II. Scrieţi un program care să calculeze produsul scalar a doi vectori. Vectorii se află în segmentul de date, au dimensiuni egale şi elemente de tip byte. Rezultatul va fi un cuvînt.

III. Scrieţi o funcţie care transformă minusculele în majuscule lăsînd orice alt caracter neschimbat. Se operează pe un şir ASCII ('A' - 'Z', 41h - 51h, 'a' - 'z', 61h - 7Ah).

IV. Scrieţi o funcţie care să aibe ca parametru de intrare un pointer la un şir de caractere terminat cu 0. Funcţia returnează un pointer la alt şir, cu caracterele inversate. Toate caracterele de control (cod ASCII $< 32h$) se vor înlocui cu constanta $20h$.

V. Scrieţi un program care preia de la tastatură două şiruri de maxim 10 caractere terminate cu caracterul CR (Carrige Return, cod ASCII 0DH) şi le compară. În caz că sînt identice, se afişează pe ecran mesajul 'ADEVARAT', altfel se afişează mesajul 'FALS'. Primul şir, al doilea şir şi mesajul vor apare pe linii diferite, aşa ca în exemplele următoare:

```
Exemplul 1        Exemplul 2        Exemplul 3
----------        ----------        ----------
abdf              abcde             abc
cdfrer            abcde             abcde
FALS              ADEVARAT          FALS
```

VI. Scrieţi un program care să primească de la tastatură două numere binare reprezentate pe cîte 8 biţi şi să afişeze rezultatul produsului lor sub forma:
`00001100*00000010=0000000000011000`

VII. Scrieţi un program care să citească de la tastatură trei litere majuscule $\{A, B, ..., Z\}$, să scrie pe ecran un spaţiu (cod ASCII 20H) şi apoi să le scrie în ordine alfabetică. Păstraţi cele trei litere tastate în registrele BH, BL şi CL.

VIII. Scrieţi un program care să utilizeze funcţii DOS şi să funcţioneze după cum urmează. Utilizatorul tastează două litere. Dacă una dintre ele este 'D', programul afişează în continuare cuvîntul DA urmat de o linie nouă. Altfel, este afişat cuvîntul NU, urmat de o linie nouă. Secvenţa de evenimente se repetă de două ori, după care controlul este întors în DOS.

IX. Scrieţi un program care să primească de la tastatură un număr N, între 1 şi 9. Programul va afişa pe ecran un pătrat cu latura N, umplut cu caractere '*', colţul stînga sus fiind pe rîndul 10, coloana 10.

X. Scrieţi un program care să şteargă ecranul şi să afişeze în partea din dreapta sus ora sub forma:

```
HH:MM
```

XI. Scrieţi un program care să afişeze pe ecran unul din mesajele de salut care urmează, în funcţie de ora sistem:

`Buna dimineata!`, între orele 5:00 şi 10:00,
`Buna ziua!`, între orele 10:00 şi 20:00,
`Noapte buna!`, între orele 20:00 şi 5:00.

De exemplu, la ora 7:28p (oră furnizată de comanda DOS `time`), pe ecran va apare salutul:

```
Buna ziua!
```

XII. Scrieţi un program care să şteargă ecranul şi să afişeze în partea de stînga jos ora sistem sub forma:

```
Ora: HH
Minutul: MM
```

XIII. Pentru fiecare din următoarele perechi de adrese date sub forma *segment:offset*, decideţi dacă cele două adrese corespund aceleiaşi locaţii de memorie:

a) 1234:1234 şi 1358:0040

b) 0500:ABCD şi 0EB0:10CD

XIV. Scrieţi un program care să citescă de la tastatură un număr de două cifre şi apoi să afişeze pe display, pe o linie nouă, restul împărţirii cu 9 a numărului introdus.

XV. Scrieţi un program care la apăsarea unei taste să afişeze pe ecran semnificaţia tastei, urmată de un spaţiu şi de codul ASCII al tastei apăsate (exprimat în baza zece). Programul se va termina la apăsarea tastei ESC.

Exemplu:

```
A 65
O 79
g 103
5 53
H 72
```

XVI. Scrieţi un program care să preia de la tastatură un număr de o cifră (1 - 9) şi să afişeze pe ecran toţi divizorii acestui număr (despărţiţi de virgule). Terminarea programului se face dacă se tastează 0. În cazul tastării altui caracter, în afara celor de la 0 la 9, programul nu reacţionează.

Exemplu:

```
4 1,2,4
a
7 1,7
8 1,2,4,8
0
```

XVII. Scrieţi un program care să citescă de la tastatură un număr de două cifre şi apoi să afişeze pe ecran, în continuarea cifrelor, clasa de resturi modulo 5 căreia îi aparţine numărul. Terminarea programului se face tastînd 00.

Exemplu:

```
47 = 2 mod 5
23 = 3 mod 5
16 = 1 mod 5
15 = 0 mod 5
00
```

XVIII. Scrieţi un program care să afişeze pe ecran data sistemului sub forma:

`ZZ Luna`,

unde:

`ZZ` este numărul zilei (pe două cifre) iar
`Luna` este denumirea lunii în limba română.

De exemplu, în data de 05-13-1999 (dată furnizata de comanda DOS `date`), pe ecran va apare:

```
13 Mai
```

XIX. Scrieţi un program care să primească de la tastatură un număr reprezentat pe două cifre în baza zece. Programul va afişa pe ecran, în continuarea numărului tastat, caracterele '`->`' urmate de reprezentarea numărului în baza 16.

De exemplu, dacă se tastează `3` şi `5`, pe ecran va apare:

```
35->23
```

XX. Scrieţi un program care să primească de la tastatură un număr reprezentat pe două cifre în baza 16. Programul va afişa pe ecran, în continuarea numărului tastat, caracterele '`=>`' urmate de reprezentarea numărului în baza 10.

De exemplu, dacă se tastează `1` şi `A`, pe ecran va apare:

```
1A=>26
```

Partea III

Anexe

Anexa A

Utilizarea Turbo Debugger

Această anexă prezintă utilitarul *Turbo Debugger* al firmei *Borland*. Interfaţa grafică este aceeaşi atît pentru DOS cît şi pentru Windows. *Turbo Debugger* este un dezasamblor de programe folosit pentru depanarea şi depistarea erorilor acestora.

A.1 Descrierea interfeţei grafice

Interfaţa grafică a *Turbo Debugger* este compusă din:

- *meniului principal* - aflat pe linia superioră a ecranului;
- *meniul general sau local* - aflat pe linia inferioară ecranului;
- *ferestre* dedicate afişării diferitelor date utilizate pentru depanarea programelor.

Fereastra principală utilizată pentru depanarea codului este fereastra CPU. Această fereastră este împărţită în cinci subferestre în care sînt prezentate următoarele date:

- segmentul de cod cu instrucţiuni dezasamblate;
- setul de registre;
- indicatorii;
- stiva;
- zona de memorie din segmentul de date.

Figura A.1 prezintă interfaţa grafică a programului *Turbo Debugger*, cu fereastra CPU maximizată.

A.2 Meniul principal

Meniul principal se află în partea superioră a ferestrei şi poate fi accesat în două moduri:

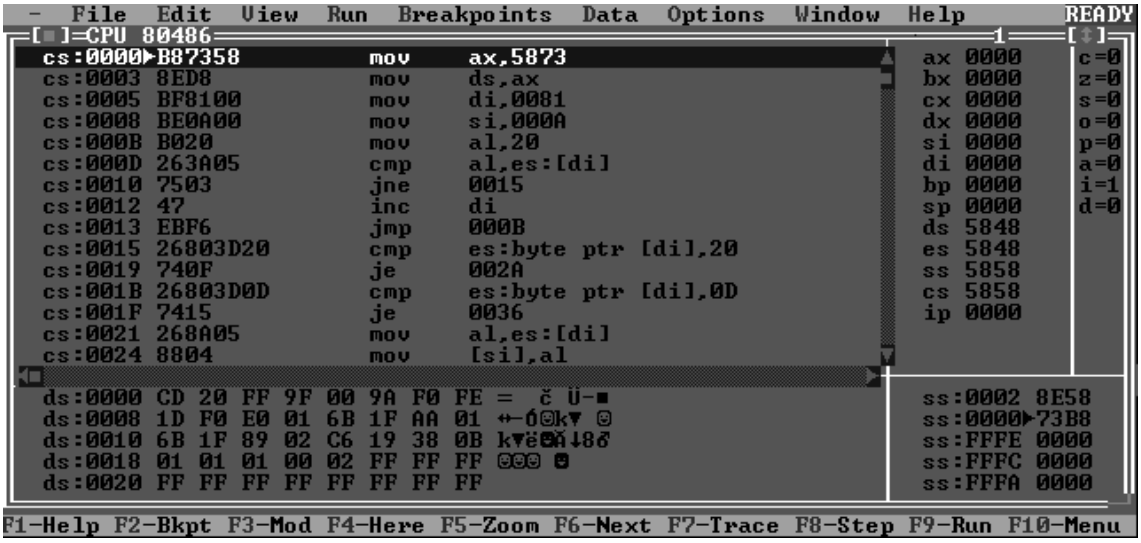


Figura A.1: Interfaţa grafică a *Turbo Debugger*.

- apăsarea tastei funcţionale F10;
- apăsarea tastei *ALT* simultan cu prima literă a unui meniu (F - *File*, E - *Edit*, V - *View*, R - *Run*, B - *Breakpoints*, D - *Data*, O - *Options*, W - *Window*, H - *Help*).

A.2.1 File

Meniul *File*, obţinut prin tastarea $ALT - F$, conţine opţiunile ce permit utilizatorului operaţii externe dezasamblorului. Cînteva dintre acestea sînt:

Open - comanda permite deschiderea un fişier .exe, .com sau .dll pentru a fi dezasamblat. În funcţie de opţiunea de asamblare (tasm /zi şi tlink/v), codul dezasamblat poate fi însoţit de informaţii de depanare complete sau nu. Programul depanat poate fi specificat şi ca argument în linia de lansare a programului *TD*.

DOS Shell - comanda permite suspendarea temporară a *TurboDebugger* pentru lansarea unor comenzi DOS. Revenirea se face prin tastarea la prompterul sistem a comenzii "exit".

Resident - comanda termină *TD* cu rămînerea sa rezidentă. Această comandă se utilizează pentru depanarea programelor *TSR*.

Quit ($ALT - X$) - comanda termină programul *TD* şi întoarce controlul în DOS.

A.2.2 Edit

Meniul *Edit*, obţinut prin tastarea $ALT - E$, permite comenzi de copiere a unor articole prin intermediul *clipboard*.

A.2.3 View

Meniul *View*, obţinut prin tastarea $ALT - V$, conţine comenzi pentru afişarea separată a tuturor tipurilor de ferestre. Cîteva din comenzile meniului sînt prezentate în continuare.

Breakpoints (F2) - comandă de deschidere a ferestrei cu descrierea punctelor de oprire în program. Fereastra prezintă o listă a adreselor de oprire şi a condiţiilor asociate acestora.

Stack - comandă de deschidere a ferestrei care prezintă starea actuală a stivei, adresele de întoarcere ale instrucţiunilor CALL precum şi variabile memorate în stivă. Informaţia din această fereastră este disponibilă şi într-o subfereastră a ferestrei CPU.

Watches - comandă de deschidere ferestrei în care se poate urmări modificarea unei variabile sau expresii în timpul rulării unei secvenţe de cod. Prin comanda *Data/Add Watch*, variabila sau expresia este adăugată în fereastra $Watches$ de la poziţia cursorului sau prin specificare explicită.

Variable - comandă de deschidere a ferestrei în care se prezintă o listă cu simboluri şi valorile asociate lor în modulul de cod curent. Se pot prezenta şi simboluri globale şi valorile lor asociate.

Module... (F3) - comandă de deschidere a unei ferestre cu codul sursă al modulului curent. Fereastra $Module$ este fereastra deschisă implicit la lansarea în execuţie a TD, în cazul în care programul depanat include informaţii suplimentare de depanare.

File... - comandă de deschidere a unei ferestre unde se poate încărca un program. Conţinutul programului poate fi afişat în hexazecimal sau ASCII, folosind comanda $Ctrl - D$ (Display).

CPU - comandă de deschidere a ferestrei CPU, compusă din cele cinci subferestre prezentate în paragraful A.1. În fereastra principală este afişat codul dezasamblat al programului. Datele sînt structurate pe trei coloane:

- coloana cu adresa sub forma *segment:offset*, exprimate în baza 16;
- coloana cu codurile instrucţiunilor, exprimate în baza 16; numărul de octeţi din această coloană este dependent de instrucţiunea respectivă;
- coloana cu instrucţiunile în limbaj de asamblare (mnemonici şi operanzi).

Celelalte subferestre sînt similare cu ferestrele $Registers$, $Dump$ şi $Stack$. Fereastra CPU este fereastra deschisă implicit la lansarea în execuţie a TD, în cazul în care programul depanat nu conţine informaţii suplimentare de depanare.

Dump - comandă de deschidere a ferestrei în care se afişează conţinutul unei zone de memorie. Formatul de reprezentare implicit este în hexazecimal. Modificarea formatului se face prin comanda $Ctrl - D$ (Display), sau din meniul ce apare prin apăsarea butonului din dreapta a mouse-ului. Modificarea conţinutul se face direct prin scrierea noii valori la adresa dorită, sau prin comanda $Ctrl - C$ (Change). Informaţia din această fereastră este disponibilă şi într-o subfereastră a ferestrei CPU.

Registers - comandă de deschidere a ferestrei de regiştrii şi indicatori. Modificarea se face direct prin scrierea noii valori în registrul selectat sau prin comanda $Ctrl - C$ (Change). Modificarea indicatorilor se face prin tasta ENTER sau cu comanda $Ctrl - T$ (Toggle). Informaţia din această fereastră este disponibilă şi într-o subfereastră a ferestrei CPU.

A.2.4 Run

Meniul Run, obţinut prin tastarea $Alt - R$, conţine comenzi pentru execuţia programului.

Run (F9) - comandă ce determină execuţia continuă a programului pînă la întîlnirea unui punct de oprire, pînă la întreruperea de către utilizator prin tastarea $CTRL - Break$ sau pînă la teminarea acestuia.

Go to cursor (F4) - comandă ce determină execuţia programului pînă la atingerea instrucţiunii pe care se află cursorul.

Trace Into (F7) - comandă ce determină execuţia unei singure instrucţiuni. Dacă instrucţiunea este de apel de procedură (CALL), atunci execuţia se opreşte la prima instrucţiune din procedură.

Step over (F8) - comandă ce determină execuţia unei singure instrucţiuni. Dacă instrucţiunea este de apel de procedură (CALL), se execută întreaga procedură şi execuţia se opreşte la instrucţiunea din programul principal, care urmează instrucţiunii CALL.

Execute to... (Alt-F9) - comandă ce determină execuţia programului pînă la o adresă specificată. Specificarea adresei poate face printr-o constantă, la care TD adaugă valoarea de segment, sau o expresie ce reprezintă o locaţie de memorie.

Until Return (Alt-F8) - comandă ce determină execuţia programului pînă cînd funcţia sau procedura curentă se reîntoarce în programul care a apelat-o.

Back trace (Alt-F4) - comandă ce determină anularea rezultatului ultimei instrucţiuni executate. Starea procesorului şi a memorie este refăcută la valoarea anterioară execuţiei instrucţiunii.

Instruction trace (Alt-F7) - comandă ce determină execuţia unei singure instrucţiuni maşină în cazul depanării unui program scris într-un limbaj de nivel înalt. Se foloseşte pentru a urmări o procedură de tratare a întreruperii sau o funcţie într-un modul compilat fară includerea informaţiilor pentru depanare.

Arguments... - comandă ce permite setarea sau modificarea argumentelor liniei de comandă a programului depanat.

Program reset (Ctrl-F2) - comandă ce determină reîncărcarea programului iniţial de pe disc, cu reiniţializarea procesorului.

A.2.5 Breakpoints

Meniul *Breakpoints*, obţinut prin tastarea $ALT - B$, conţine comenzi pentru plasarea, setarea sau anularea punctelor de oprire în programul depanat.

Toggle (F2) - comandă ce setează sau anulează un punct de oprire la o adresă sau linie de cod indicată de cursor. Programul se va opri de fiecare dată cînd va ajunge la acel punct. Aceelaşi efect îl are şi apăsarea butonului din stînga al mouse-ului pe primele două coloane ale liniei la care se doreşte setarea unui punct de oprire. Programul poate fi rulat şi între două puncte de oprire specificîndu-se primul punct ca pornire în execuţie. Acţiunea executată la întîlnirea unui punct de oprire poate fi stabilită prin fereastra *Breakpoints*. La atingerea unui punct de oprire TD poate declanşa următoarele acţiuni:

- *Break* - oprire program, controlul este preluat de către TD putîndu-se examina starea procesorului şi a memoriei;
- *Log* - memorarea valorii unei variabile sau expresii în fereastra *Log*;

- *Execute* - executarea unei expresii şi evaluarea ei. Expresia poate fi în orice limbaj suportat de *TD* (C, Pascal, Assembler) prin alegerea limbajului de interpretare. *TD* evaluează implicit expresia de la adresa cerută în limbajul în care a fost scris codul;
- *Enable* - setarea unui alt punct de oprire;
- *Disable* - anularea unui alt punct de oprire.

At...(Alt-F2) - comandă ce setează un punct de oprire la adresa specificată de utilizator.

Changed memory global... - comandă ce setează un punct de oprire în cazul în care conţinutul unui bloc de memorie se schimbă.

Expression true global... - comandă ce setează un punct de oprire în cazul în care o expresie devine adevărată.

Delete all - comandă ce anulează toate punctele de oprire definite anterior.

A.2.6 Data

Meniul *Data*, obţinut prin tastarea $ALT - D$, conţine comenzi pentru evaluarea, inspectarea şi urmărirea unor variabile, porţiuni de memorie sau expresii în cadrul programului.

Inspect - comandă ce deschide fereastra *Inspector* ce prezintă valoarea unei variabile sau a unei expresii de referinţă la memorie. Dacă cursorul se află în fereastra cu cod sursă, sub numele unei variabile, atunci aceasta este adăugată în fereastra de inspecţii.

Evaluate/Modify... (Ctrl-F4) - comandă pentru evaluarea unei expresii arbitrare introdusă de utilizator.

Add Watch... (Ctrl-F7) - comandă ce plasează o variabilă sau o expresie în fereastra de urmărire (*Watches*). Variabila de sub cursor, dacă acesta este în zona de cod sursă, este adăugată în mod automat în fereastra de urmărire.

A.2.7 Options

Meniul *Options*, obţinut prin tastarea $ALT - O$, conţine comenzi pentru configurarea mediului de depanare şi a opţiunilor care au efect global asupra comportării *Turbo Debugger*. Opţiunile pot fi salvate într-un fişier cu denumirea implicită $tdconfig.td$.

A.2.8 Window

Meniul *Window*, obţinut prin tastarea $ALT - W$, conţine comenzi pentru manipularea ferestrelor. Deşi manipularea ferestrelor se poate face mai uşor cu mouse-ul, în lipsa acestuia, comenziile meniului *Window* rămîn singura soluţie de organizare a afişării mai multor ferestre pe ecran.

- *Zoom (F5)* - comandă ce maximizează fereastra curentă pe tot ecranul. Revenirea la dimensiunea iniţială a ferestrei se face prin aceeaşi comandă. Acelaşi efect se obţine dacă se acţionează butonul din stînga al mouse-ului cînd cursorul este poziţionat pe simbolul [↑] aflat în partea din dreapta sus al ferestrei.

- *Next (F6)* - comandă care determină schimbareea ferestrei curente. Fereastra curentă are un chenar cu linie dublă. Comutarea între ferestre se poate face prin acţionarea butonului din stînga al mouse-ului pe o fereastră diferită de cea curentă.

- *Next pare (Tab)* - comandă care determină mutarea cursorului între subferestrele ferestrei curente.

- *Size/More (Ctrl -F5)* - comandă ce permite redimensionarea şi mutarea ferestrei curente. Cu mouse-ul, redimensionarea se face dacă se acţionează asupra laturilor din dreapta sau de jos ale ferestrei. Mutarea ferestrei se face prin acţionarea mouse-ului asupra laturilor din stînga sau de sus ale ferestrei.

- *Iconize/Restore* - comandă ce minimizează fereastra curentă. Revenirea la dimensiunea iniţială a ferestrei se face prin acceiaşi comandă. Acelaşi efect se obţine dacă se acţionează butonul din stînga al mouse-ului cînd cursorul este poziţionat pe simbolul [↓] aflat în partea din dreapta sus al ferestrei.

- *Close (Alt-F3)* - comandă de închidere a ferestrei curente.

- *Undo Close (Alt-F6)* - comandă de redeschidere a ultimei ferestre închise.

- *User screen (Alt-F5)* - comandă care prezintă ecranul sub forma în care ar apărea dacă s-ar executa programul utilizatorului. După vizualizarea ecranului, apăsarea oricărei taste determină revenirea la ecranul *Turbo Debugger*.

A.3 Meniuri generale

A.3.1 Meniul general

Funcţiile meniului general sînt disponibile tot timpul şi pot fi apelate prin tastele funcţionale. Descrierea funcţiilor meniului este prezentată pe bara de jos a ferestrei principale, aşa ca în figura A.1.

- F1 *Help* - obţinerea informaţiilor despre fereastra în care se află poziţionat cursorul.

- F2 *Bkpt* - plasarea unui punct de oprire la linia cursorului. Comandă similară cu meniul *Breakpoints/Toggle*.

- F3 *Mod* - deschiderea fereastrei corespunzătoare modulelor din cadrul programului. În fereastră se va afla codul sursă a modulului curent. Comandă similară cu meniul *View/ Module*.

- F4 *Here* - executarea programului pînă la linia pe care se află cursorul. Comandă similară cu meniul *Run/Go to cursor*.

- F5 *Zoom* - maximizarea ferestrei în care se află cursorul. Comandă similară cu meniul *Window/Zoom*.

- F6 *Next* - activarea următoarei ferestre deschise. Comandă similară cu meniul *Window/ Next*.

- F7 *Trace* - execuţia unei singure instrucţiuni din program. Comandă similară cu meniul *Run/Trace into*.
- F8 *Step* - execuţia unei singure instrucţiuni. Dacă instrucţiunea este apel de procedură, se execută toată procedura. Comandă similară cu meniul *Run/Step over*.
- F9 *Run* - execuţia întregului program. Comandă similară cu meniul *Run/Run*.
- F10 *Menu* - mutarea cursorului pe bara de sus, a meniului principal.

A.3.2 Meniul general alternativ

Funcţiile meniului general alternativ sînt disponibile tot timpul şi pot fi apelate prin apăsarea tastei ALT şi a unei taste funcţionale.

- Alt-F2 *Bkpt at* - stabilirea unui punct de oprire la adresa specificată. Comandă similară cu meniul *Breakpoints/At*.
- Alt-F3 *Close* - închiderea ferestrei curente. Comandă similară cu meniul *Windows/Close*.
- Alt-F4 *Back* - anularea rezultatului ultimei instrucţiuni executate. Comandă similară cu meniul *Run/Back trace*.
- Alt-F5 *User* - vizualizarea rezultatelor execuţiei programului în fereastra utilizator. Comandă similară cu meniul *Window/User screen*.
- Alt-F6 *Undo* - anularea efectului comenzii anterioare. Comandă similară cu meniul *Window/Undo*.
- Alt-F7 *Instr* - execuţia unei singure instrucţiuni. Comandă similară cu meniul *Run/ Instruction Trace*.
- Alt-F8 *Rtn* - execuţia programului pînă cînd funcţia curentă se reîntoarce în programul care a apelat-o. Comandă similară cu meniul *Run/Until return*.
- Alt-F9 *To* - execuţia programului pînă la adresa specificată. Comandă similară cu meniul *Run/Execute to*.
- Alt-F10 *Local* - deschiderea meniului local, asociat ferestrei active. Aceeaşi acţiune o determină şi apăsarea butonului din dreapta al mouse-ului.

A.4 Meniuri locale

Meniurile locale conţin comenzi specifice ferestrelor selectate. Apelarea meniurilor locale se poate face prin:

- comanda ALT-F10 din meniul general alternativ;
- apăsarea tastei CTRL;
- apăsarea butonului din dreapta al mouse-lui, în suprafaţa ferestrei selectate.

Informaţii suplimentare despre meniurile locale ale ferestrei curente se pot obţine prin apăsarea tastei F1.

A.4.1 Meniul ferestrei CPU

- G - *Goto* - poziţionarea cursorului la adresa indicată astfel încît cursorul se va afla pe prima linie a ferestrei.
- O - *Origin* - poziţionarea liniei pe care se află cursorul pe prima linie a ferestrei.
- F - *Follow* - poziţionarea cursorului pe linia de destinaţie a unei instrucţiuni de salt sau apel de procedură. Instrucţiunea curentă trebuie să fie una de salt sau apel de procedură, altfel instrucţiunea nu are nici un efect.
- C - *Caller* - poziţionarea cursorului pe instrucţiunea care a apelat întreruperea sau procedura curentă.
- P - *Previous* - poziţionarea cursorului pe instrucţiunea pe care a fost anterior poziţionării acestuia cu comanda *Follow* sau *Caller*.
- S - *Search* - căutarea unui byte sau a unei instrucţiuni în cod.
- V - *View source* - deschiderea unei ferestre *Module* care prezintă codul sursă asociat celui dezasamblat.
- A - *Assamble* - asamblarea unei instrucţiuni la adresa curentă a cursorului.
- N - *New* - determină ca următoarea instrucţiune executată să fie cea de la cursor. Adresa instrucţiunii pe care se află cursorul este încărcată în registrele CS şi IP. În acest mod, se poate sări peste o porţiune de cod.

A.4.2 Meniul ferestrei Dump

- G - *Goto* - salt la o adresă specificată de utilizator sub forma *segment:offset*.
- S - *Search* - căutarea unei secvenţe de biţi în fereastra curentă.
- N - *Next* - căutarea următoarei apariţii a secvenţei specificate de comanda *Search*.
- C - *Change* - modificarea conţinutului uneia sau mai multor locaţii de memorie de la poziţia curentă a cursorului.
- F - *Follow* - poziţionarea datelor sau codului la o nouă adresă pe baza datelor din memorie aflate la adresa curentă a cursorului.
- P - *Previous* - poziţionarea cursorului în locul anterior execuţiei comenzilor *Follow* sau *Caller*.
- D - *Display* - schimbarea modului de reprezentare a datelor în fereastra de memorie (Byte, Word, Long, Comp, Float, Real, Double, Extended).
- B - *Block* - manipularea unui bloc de memorie . Submeniul conţine opţiunile:

- *clear* - iniţializare cu zero a unui bloc de memorie;
- *move* - copierea unui bloc de memorie de la o adresă la alta;
- *set* - iniţializarea unui bloc de memorie cu o valoare specificată;
- *read* - citirea dintr-un fişier a datelor şi plasarea acestora într-un bloc de memorie;
- *write* - scrierea datelor dintr-un bloc de memorie într-un fişier.

A.4.3 Meniul ferestrei Register

- I - *Increment* - incremetarea conţinutului registrului selectat;
- D - *Decrement* - decremetarea conţinutului registrului selectat;
- Z - *Zero* - resetarea registrului selectat;
- C - *Change* - modificarea conţinutului registrului selectat;
- R - *Registers* - comutarea între afişarea registrelor pe 16 sau 32 de biţi.

A.4.4 Meniul ferestrei Flag

- T - *Toggle* - comutarea valorii indicatorului selectat. Acelaşi lucru se obţine prin apăsarea tastei ENTER după plasarea cusorului pe indicatorul respectiv.

A.4.5 Meniul ferestrei Stack

- G - *Goto* - mutarea cursorului la adresa specificată;
- O - *Origin* - mutarea cursorului la vîrful stivei, specificat de SS:SP;
- F - *Follow* - mutarea cursorului la adresa indicată în locaţia de date a poziţiei curente din stivă;
- P - *Previous* - mutarea cursorului la adresa anterioară unei poziţionări cu comenziile *Follow* sau *Caller*;
- C - *Change* - modificarea conţinutului locaţiei de memorie selectate.

Anexa B

Schemele machetei MPF1-B microprofessor

Această anexă prezintă schemele machetei MPF1-B Microprofessor echipată cu microprocesor Z80.

Figura B.1 prezintă schema bloc a machetei în care apar denumirea şi plasamentul componentelor şi al tastelor.

Figurile B.2, B.3, B.4, B.5 şi B.6 prezintă schemele electrice complete ale machetei.

Figura B.2 prezintă circuitul generator de semnal de ceas şi de generare a semnalelor de selecţie.

Figura B.3 prezintă conectarea circuitelor Z80-CPU, Z80-PIO şi Z80-CTC.

Figura B.4 prezintă conectarea circuitului 8255. În aceeaşi figură apar schemele circuitelor de intrare/ieşire ale machetei, difuzorului şi stabilizatorului de tensiune.

Figura B.5 prezintă circuitele de memorie şi tastatura machetei.

Figura B.6 prezintă blocul de afişaj al machetei realizat cu şase afişoare 7 segmente.

Figura B.1: Schema bloc a machetei MPF1-B microprofessor.

Figura B.2: Circuitul generator de semnal de ceas şi de generare a semnalelor de selecţie.

Figura B.3: Conectarea circuitelor Z80-CPU, Z80-PIO şi Z80-CTC.

Figura B.4: Conectarea circuitului 8255. Schemele circuitelor de intrare/ieşire ale machetei, difuzorului şi stabilizatorului de tensiune.

Figura B.5: Circuitele de memorie şi tastatura machetei.

Figura B.6: Blocul de afişaj al machetei.

1.1. Descrierea generala a procesoarelor x86 pe 16 biti

În figurile 1.2 şi 1.3 sunt prezentate schemele bloc ale celor două unităţi centrale. Aşa cum am spus ambele încorporează două blocuri de prelucrare separate, UE şi UIM. UE este identică la ambele procesoare, UIM diferind prin aceea că la 8086 ea lucrează pe o magistrală de date de 16 biţi cu coadă de instrucţiuni de 6 octeţi; la 8088 magistrala este de 8 biţi iar coada de 4 octeţi.

UE are funcţia de a executa toate instrucţiunile, comunicând prin date şi adrese cu UIM şi manipulând registrele generale, indicatorii de condiţii şi unitatea aritmetică/logică. Cu excepţia câtorva conexiuni externe de comandă, UE este complet izolată de exteriorul procesorului. UIM are funcţia de a executa toate ciclurile de acces la magistrala externă, fiind alcătuită din registrele de segment, registrele de comunicaţie internă, *pointer*-ul de instrucţiuni, stiva pentru codul obiect al instrucţiunii şi un sumator specializat. Operaţiile pe care le face UIM sunt: obţinerea adresei prin adunarea valorii segmentului cu valoarea deplasamentului, transferarea datelor spre UE pe magistrala de date internă, de 16 biţi, a unităţii aritmetice/logice, şi încărcarea în avans, preextragerea, codurilor obiect ale instrucţiunilor din memoria sistemului în coada de aşteptare, de unde vor fi preluate de UE.

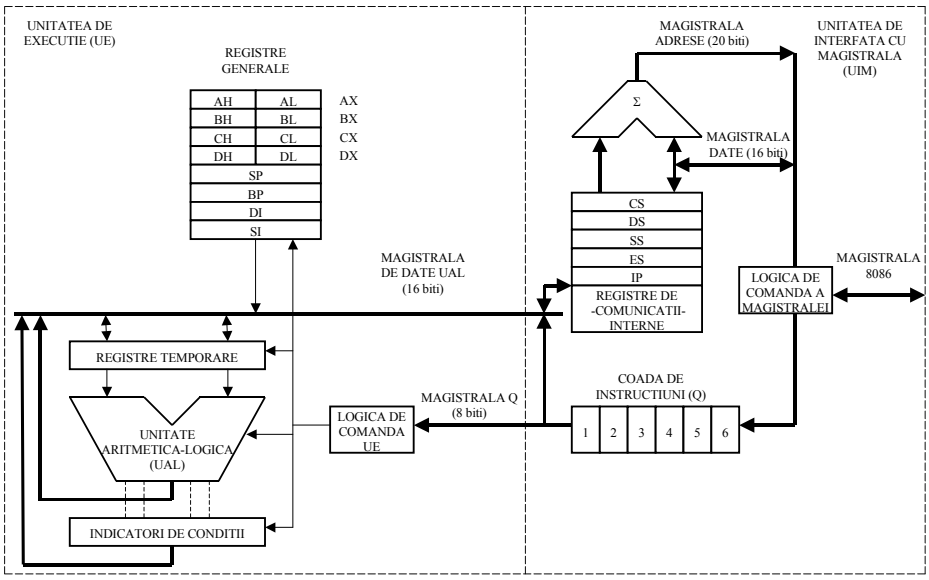


Figura 1.2. Schema bloc a microprocesorului 8086

UE extrage, atunci când este gata să execute o instrucţiune, codul obiect al acesteia din coada de aşteptare, trecând apoi la execuţia ei. În cazul în care coada de aşteptare nu are nici-un octet, UE va aştepta până ce UIM va extrage codul din memoria sistemului. Dacă în cursul execuţiei unei instrucţiuni este necesar accesul la o locaţie de memorie sau la un *port* de I/O, UE va face, de asemenea, o cerere către UIM pentru ca aceasta să execute ciclul de magistrală dorit.

Cele două blocuri, UE şi UIM, funcţionează independent. La 8086, dacă cel puţin doi octeţi din coadă sunt liberi şi UE nu are nici-o cerere către

UIM, aceasta din urmă va executa ciclii de extragere în scopul ocupării întregii cozi. La 8088, UIM extrage cod obiect atunci când coada are cel puţin un octet liber. Dacă UE are nevoie de un acces la magistrală şi UIM se află în timpul unui ciclu de extragere, UE va aştepta terminarea acestui ciclu înainte de a fi luată în considerare cererea.

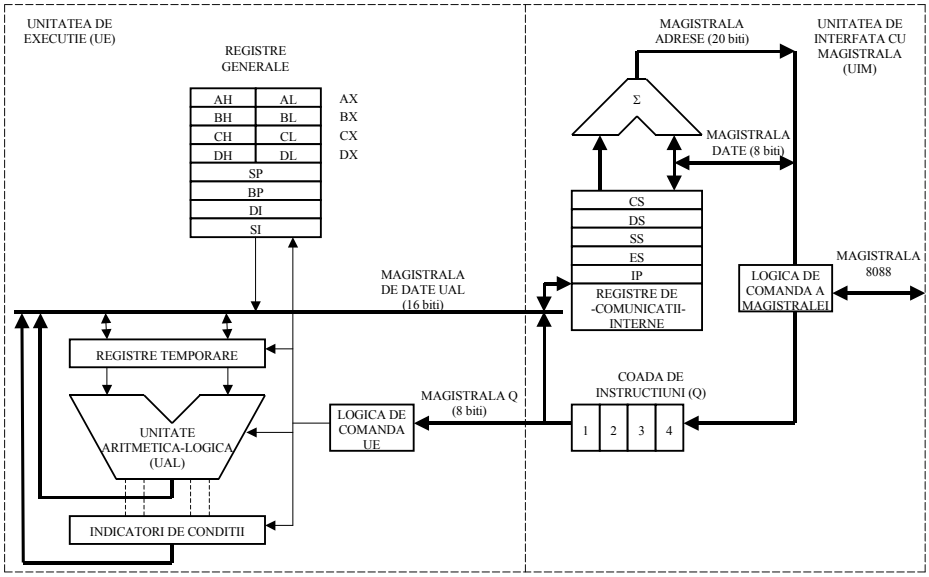


Figura 1.3. Schema bloc a microprocesorului 8088

Registrele programabile ale microprocesorului 8086 sunt înfăţişate în figura 1.4. Prezentăm în continuare registrele în legătură cu modurile de adresare ale lui 8086 deoarece multe din acestea sunt utilizate de logica de adresare a memoriei.

AX este acumulatorul principal utilizat, de exemplu, de instrucţiunile de I/O care transferă datele de obicei prin intermediul acestui registru. BX poate fi folosit atât ca acumulator cât şi ca registru bază pentru calcularea adreselor de memorie. CX poate fi acumulator sau se utilizează ca numărător în execuţia instrucţiunilor iterative. DX poate fi folosit ca acumulator sau ca registru numărător pentru adresarea zonelor de memorie în transferuri cu *port*-urile de intrare/ieşire.

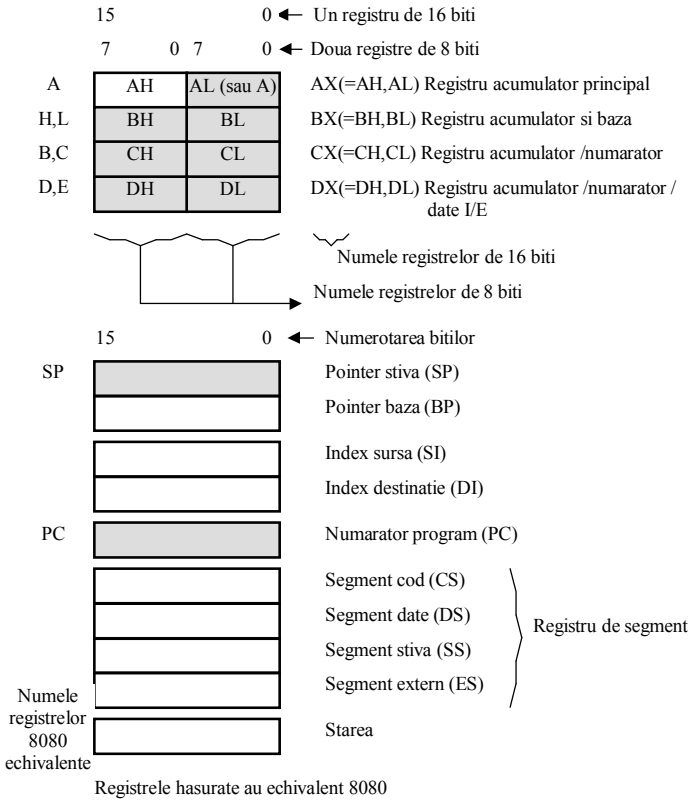


Figura 1.4. Registrele programabile ale microprocesorului 8086

Adresele de memorie pe 20 de biţi, generate de 8086, se calculează prin însumarea conţinutului unui *registru de segment* cu o *adresă efectivă*. Adresa efectivă se calculează printr-o varietate de moduri de adresare ca şi la alte microprocesoare. Registrul de segment selectat este întâi deplasat la stânga cu patru poziţii şi apoi adunat cu adresa efectivă, generându-se o adresă de 20 de biţi:

```
Conţinutul registrului de segment:   SSSS SSSS SSSS SSSS 0000 +
Adresa efectivă de memorie:          0000 EEEE EEEE EEEE EEEE

Adresa de memorie pe 20 de biţi:     SSSM MMMM MMMM MMMM MMMM
```

Registrele de segment sunt folosite deci ca registre de bază care pot puncta orice locaţie de memorie aflată la o adresă multiplu de 16. Fiecare registru de segment identifică începutul unei zone de memorie, al unui *segment*, de 64

kocteţi. Deoarece 8086 are patru registre de segment în orice moment, din întreaga memorie vor fi selectate numai patru segmente de câte 64 kocteţi. Nu se impune nici-o restricţie asupra conţinutului registrelor de segment, ceea înseamnă că aceste registre identifică numai *originea* segmentelor care pot fi plasate oriunde în spaţiul de adresare de 1 Moctet al microprocesorului. Memoria de 1 Moctet nu este deci împărţită în pagini de 64 kocteţi, ci adresată pe segmente care pot fi *separate* sau *suprapuse*.

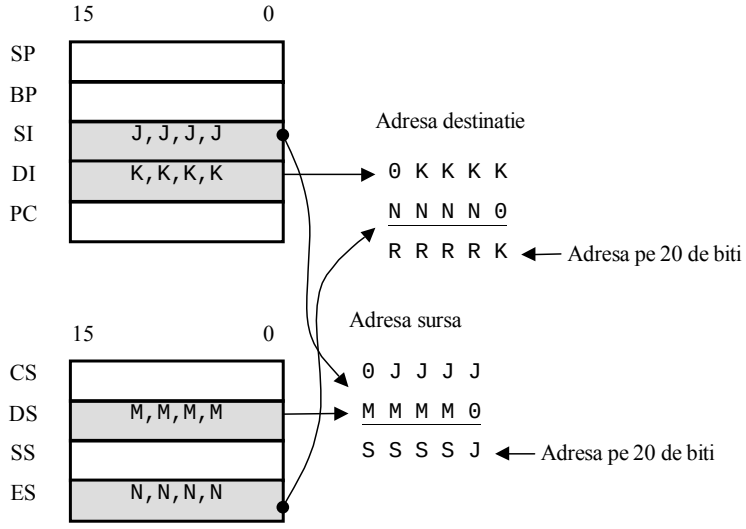


Figura 1.5. Calculul adresei de memorie pentru instrucţiuni pe şiruri de date

Cele patru registre de segment îndeplinesc funcţii de adresare diferite: în funcţie de tipul accesului la memorie,– cod, date, stivă şi extern –, calculul adresei foloseşte registrul de segment asociat. Astfel la o extragere de instrucţiune, numărătorul de program, PC, este adunat cu conţinutul registrului de segment de cod, CS. La orice instrucţiune de lucru cu stiva, de tip Push, Pop, Call sau Return, adresa de memorie se calculează adunând conţinutul *pointer*-ului de stivă, SP, cu conţinutul registrului de segment de stivă, SS. La instrucţiunile pe şiruri de date pentru calculul adreselor sursă şi destinaţie sunt utilizate registrele index SI şi DI împreună cu registrele de segment DS şi ES, figura 1.5. Observăm că pentru aceste instrucţiuni este necesar ca şirurile sursă şi destinaţie să fie plasate fiecare în cadrul unui singur spaţiu de adresare de 64 kocteţi, nu neapărat acelaşi. Instrucţiunile care accesează date calculează adresa prin adunarea adresei efective cu conţinutul registrelor de segment DS sau SS.

Dăm în tabelul 1.1 o prezentare sintetică a modurilor de calculare a adresei de memorie.

Tabelul 1.1. Calculul adreselor de memorie

Referinţa de memorie	Registrul de segment	Registrul bază	Registrul index	Deplasament pe 16 biţi, fără semn	Deplasament pe 8 biţi, semn extern	Fără deplasament
Date	DS (CS, SS, ES)[1]	–	SI	*[2]	*	*
			DI	*	*	*
		BX	SI	*	*	*
			DI	*	*	*
			–	*	*	*
	DS	–	–	*		
	SS (CS, DS, ES)[1]	BP	SI	*	*	*
			DI	*	*	*
			–	*	*	
Stivă	SS	SP	–			
Şiruri	DS	–	SI			
	ES	–	DI			
Extragere instrucţiune	CS	PC	–			
Salt	CS	PC	–		*	
Date I/O	DS	DX	–			

Note: 1. Depăşirea segmentului permite înlocuirea registrelor de segment DS sau SS cu unul din celelalte registre.
2. * semnifică deplasamentele care pot fi utilizate la calculul adresei

Vom prezenta întâi opţiunile de adresare a datelor. Cea mai simplă adresare este cea *directă*. În acest caz registrul DS este adunat cu un *offset* sau deplasament pe 16 biţi dat de doi octeţi din codul obiect al instrucţiunii. Dacă se folosesc registrele index SI sau DI, prin adunarea conţinutului unuia din acestea cu conţinutul registrului DS se obţine o adresare *implicită*. Prin adăugarea deplasamentului, de 16 sau 8 biţi, obţinem o adresare *directă indexată*. Dacă deplasamentul e pe 8 biţi atunci cel mai semnificativ bit, semnul, se extinde pe încă 8 biţi, mai semnificativi, pentru a forma un deplasament pe 16 biţi, de exemplu:

```
Deplasament:      …        1010 1101
Semn extins:   1111 1111   1010 1101   sau
Deplasament:      …        0101 1010
Semn extins:   0000 0000   0101 1010
```

Adresa mai poate fi calculată şi prin utilizarea unui registru-bază obţinându-se o adresare *relativă*. Pentru adresarea relativă a datelor se pot utiliza două registre de bază, BX şi BP, adresele găsindu-se în segmentul DS, pentru date, şi, respectiv, în segmentul SS, pentru stivă. În primul fel de adresare relativă cu bază a datelor, conţinutul registrului BX este adunat la adresa efectivă calculată în unul din modurile exemplificate mai sus. Putem avea deci trei feluri de adresări relative: directe, implicite sau directe indexate. În al doilea fel de adresare, calculul adresei se face adunând la adresa efectivă conţinutul registrului de segment SS.

În legătură cu utilizarea registrelor generale ca registre de bază atragem aici atenţia asupra unui mod de adresare a dispozitivelor de I/O prin intermediul registrului DX. În acest mod de adresare conţinutul registrului DX este plasat pe magistrala de adrese fără a fi adunat cu vreun registru de segment. Acesta este singurul mod de adresare în care conţinutul unui registru este scos direct pe magistrală ca adresă fără a fi în prealabil prelucrat în logica de segmentare.

Toate instrucţiunile de salt condiţionat utilizează adresarea relativă la programe, cu registru de bază PC, permiţându-se astfel generarea unui cod relocabil. Deplasamentul pe 8 biţi reprezintă un număr binar cu semn cuprins în gama (−128, +127).

Pentru instrucţiunile de salt necondiţionat şi de apel sunt folosite trei moduri de adresare: adresare relativă la program, registru de bază PC, cu deplasament pe 8 sau 16 biţi; adresare directă; adresare *indirectă*. În adresarea indirectă se utilizează una din formele de adresare descrise mai sus pentru a citi din memorie o nouă adresă de memorie. Există două opţiuni de adresare indirectă: una în care cuvântul de 16 biţi citit din memorie e încărcat în PC şi instrucţiunea de salt sau apel adresează o locaţie de memorie din cadrul segmentului CS curent; cealaltă în care se citesc două cuvinte de 16 biţi pentru a fi încărcate în PC, respectiv CS şi care permite adresarea oricărei locaţii din spaţiul de memorie al microprocesorului.

În figura 1.6 dăm structura registrului de stare al microprocesorului 8086 conţinând indicatorii de condiţii.

Indicatorul *Transport*, C sau CF, este inversat indicând de fapt un împrumut: după o scădere, realizată de procesor în complement faţă de 2, acest bit va fi pus pe "1" dacă nu a existat transport la bitul cel mai semnificativ şi pus pe "0" dacă a existat un transport.

Indicatorul *Paritate*, P sau PF, este pus pe "1" dacă rezultatul conţine un număr par de "1" şi pe "0" dacă rezultatul operaţiei are un număr impar de "1"-uri.

Indicatorul *Zero*, Z sau ZF, este standard fiind pus pe "1" dacă rezultatul operaţiei e zero şi pe "0" dacă rezultatul nu este zero.

Indicatorul *Semn*, S sau SF, este pus pe "1" dacă rezultatul unei operaţii este negativ având bitul cel mai semnificativ "1" şi pus pe "0" dacă rezultatul este pozitiv.

Indicatorul *Direcţie*, D sau DF, determină dacă operaţiile pe şiruri vor incrementa sau decrementa registrele index. Dacă *Direcţie* este "1" atunci conţinuturile registrelor SI şi DI vor fi decrementate, şirurile fiind accesate începând cu adresele cele mai mari. Dacă indicatorul este "0" conţinutul registrelor SI şi DI va fi incrementat, şirurile fiind accesate începând cu adresele cele mai mici.

Indicatorul *Validare/Invalidare Întrerupere*, I sau IF, trebuie să fie "1" pentru a valida întreruperile şi "0" pentru a le invalida (cu excepţia NMI).

Indicatorul *Derută*, T sau TF, face ca microprocesorul 8086 să intre în modul de lucru "pas-cu-pas" şi este strâns legat de sistemul de întreruperi (vezi §1.4.5.1 şi §2.3.6).

Indicatorul *Depăşire*, O sau OF este, de asemenea, legat de sistemul de întreruperi şi de execuţia instrucţiunii INTO.

Indicatorul *Transport Auxiliar*, A sau AF indică, după o operaţie, un transport din bitul 3 în 4.

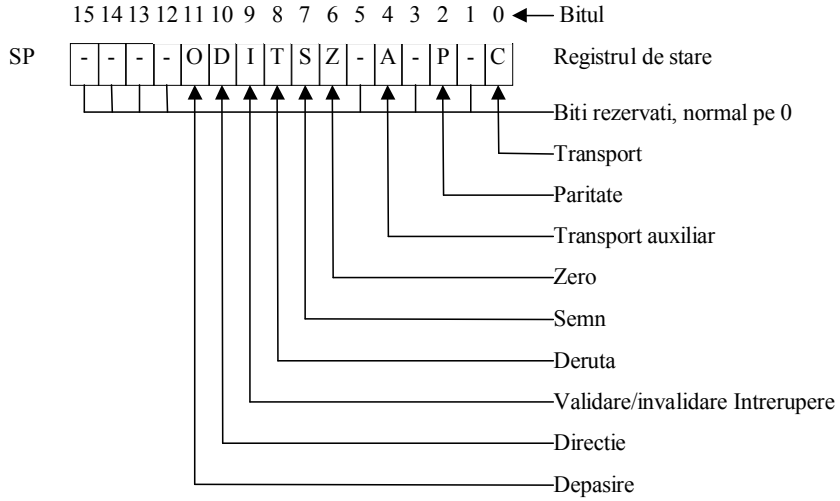


Figura 1.6. Registrul de stare

1.2. CONEXIUNILE EXTERNE

Microprocesoarele 8086 şi 8088 sunt fabricate în capsule DIL cu 40 de conexiuni externe. Aceste conexiuni sunt indicate în figurile 1.7 şi 1.8. Vom descrie în continuare semnificaţiile lor.

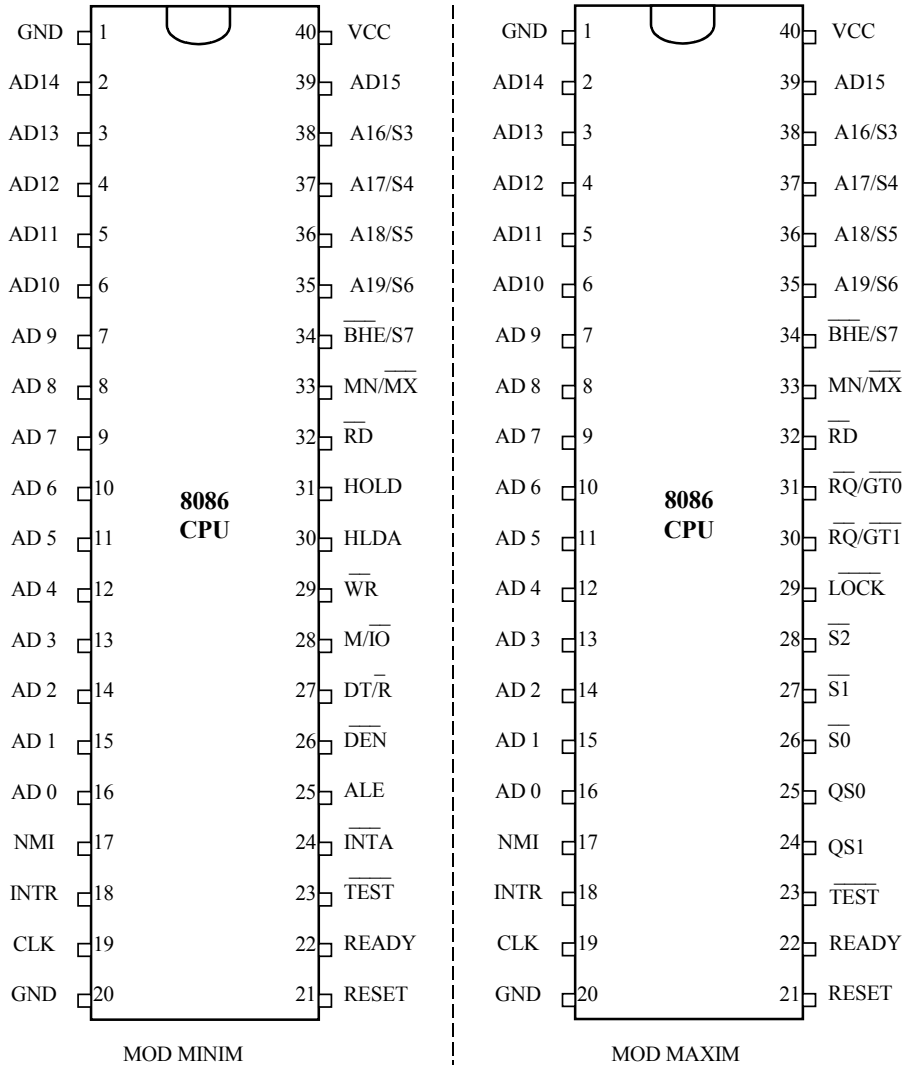


Figura 1.7. Conexiunile externe ale microprocesorului 8086

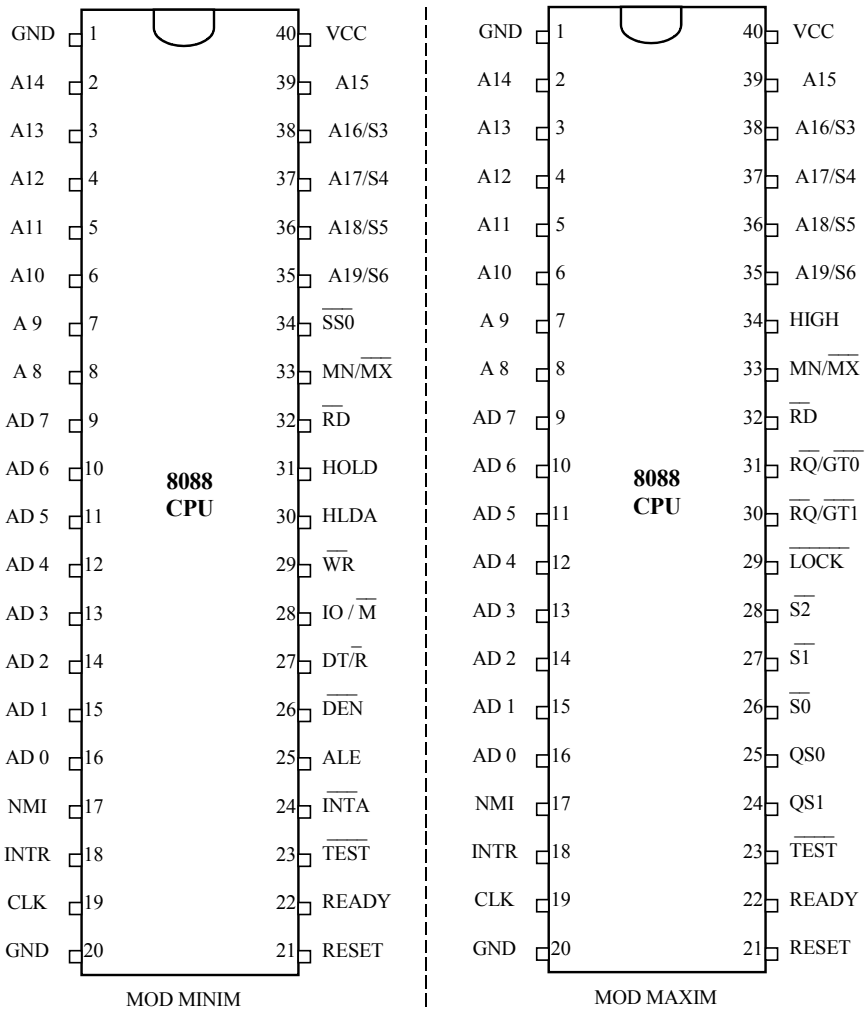


Figura 1.8. Conexiunile externe ale microprocesorului 8088

1.2.1. CONEXIUNILE EXTERNE ALE MICROPROCESORULUI 8086

1.2.1.1. **Semnalele comune modurilor de lucru minim şi maxim**

AD15÷AD0, *Address/Data Bus*, magistrala de adrese/date, intrări/ieşiri 3-stări active pe "1". Pe aceste linii sunt multiplexate în timp adresele de memorie (cei mai puţin semnificativi 16 biţi) sau de I/O, în starea T1, şi

magistrala de date, în stările T2, T3, TW, T4[1]. Multiplexarea magistralelor a fost impusă de necesitatea de a împacheta microprocesorul într-o capsulă de 40 de conexiuni. AD0, ca adresă A0, are pentru selecţia octetului de date mai puţin semnificativ, D7÷D0, aceeaşi funcţie ca şi semnalul $\overline{BHE}$, descris mai jos; pentru octetul mai semnificativ, D15÷D8, A0 este "0" în timpul stării T1, la transferarea unui octet de date pe porţiunea mai puţin semnificativă a magistralei, în operaţii cu memoria sau cu dispozitivele de I/O. AD15÷AD0 sunt trecute în starea a treia, de impedanţă înaltă, în timpul operaţiilor de achitare a întreruperii şi a cererii de preluare a magistralei.

A19/S6÷A16/S3, *Address/Status*, adrese/stări, ieşiri 3-stări. În timpul stării T1 reprezintă cei mai semnificativi biţi de adresă în operaţiile cu memoria; în operaţiile de I/O aceste semnale sunt "0". Pentru ambele tipuri de operaţii, pe timpul stărilor T2, T3, TW şi T4, la aceste ieşiri se găseşte o parte din starea microprocesorului: S6=0, indicând că 8086 accesează magistrala, S5=I, *Validare/Invalidare Întrerupere*, actualizat la începutul fiecărei perioade a ceasului CLK, iar S4 şi S3 precizând ce registru de relocare, DS, CS, SS sau ES, va fi utilizat în acel moment pentru accesarea datelor. Semnificaţia biţilor de stare S4 şi S3 este următoarea:

A17/S4	A16/S3	Semnificaţie
0	0	Date externe (relativ la segmentul din ES)
0	1	Stivă (relativ la segmentul din SS)
1	0	Cod sau nimic (relativ la segmentul din CS sau o valoare în lipsă "0")
1	1	Date (relativ la segmentul din DS)

Informaţia de stare este necesară în acţiunile de diagnosticare a structurilor realizate în jurul procesorului 8086. S4 şi S3 mai pot fi utilizaţi pentru selecţia bancurilor de memorie asociate fiecare câte unui registru de segment. Această tehnică permite partajarea memoriei în funcţie de segment în vederea expandării ei peste spaţiul de adresare directă de 1 Moctet. De asemenea, în acest fel, se asigură o posibilitate de protecţie în cazul operaţiilor eronate de scriere prin suprapunere în două segmente şi distrugerea informaţiei în unul dintre segmente. Conexiunile A19/S6÷A16/S3 sunt trecute în starea a treia în timpul operaţiei de achitare a unei cereri de preluare a magistralei microprocesorului.

$\overline{BHE}$/S7, *Bus High Enable/Status*, validare octet mai semnificativ/stare. Ieşire 3-stări. În timpul stării T1 ieşirea $\overline{BHE}$ are funcţia de a selecta transferul datelor pe porţiunea mai semnificativă a magistralei de date, biţii D15÷D8, fiind "0" în T1 timpul ciclilor de citire, scriere şi întrerupere. $\overline{BHE}$ poate fi utilizat la selecţia dispozitivelor de I/O pe 8 biţi conectate pe porţiunea mai semnificativă

[1] Descrierea stărilor T este dată în §1.4.1

a magistralei de date. Bitul de stare S7 este validat în timpul stărilor T2, T3 şi T4. Conexiunea este trecută în starea a treia în timpul unei operaţii de preluare a magistralei.

MN / $\overline{MX}$, *Minimum/Maximum*, comanda modului de lucru minim/maxim. Intrare. Dacă MN / $\overline{MX}$ este conectat la "0" microprocesorul tratează conexiunile 24÷31 în modul maxim, pentru care un circuit specializat pentru comanda magistralei, 8288, interpretează biţii de stare $\overline{S2}$, $\overline{S1}$ şi $\overline{S0}$ pentru a genera semnale de comandă compatibile cu standardul de interfaţă Multibus. Pentru MN / $\overline{MX}$ conectat la +5V, modul minim, 8086 generează singur, la conexiunile 24÷31, semnalele de comandă. În §1.4.2 şi §1.4.3 se dau două exemple de utilizare a procesorului 8086 în modurile minim şi maxim.

$\overline{RD}$, *Read*, citire. Ieşire 3-stări activă pe "0". Comandă de citire indicând că procesorul efectuează un ciclu de citire memorie sau I/O, funcţie de starea conexiunii $\overline{S2}$ (M / $\overline{IO}$). $\overline{RD}$ este activat pe "0" în timpul stărilor T2, T3 şi TW ale oricărui ciclu de citire, după trecerea în starea a treia a magistralei locale a microprocesorului. Conexiunea este trecută în starea de impedanţă înaltă în timpul unei operaţii de preluare a magistralei.

READY, gata. Intrare activă pe "1". Reprezintă, în timpul unei operaţii de scriere sau citire, un semnal de achitare din partea memoriei sau dispozitivului de I/O adresat, certificând validitatea transferului de date cu microprocesorul. Semnalul de achitare emis de memorie sau de I/O trebuie să fie sincronizat pentru a putea fi utilizat în mod corect de procesor prin satisfacerea timpilor de *set-up* şi *hold*. Această sincronizare se face cu ajutorul unui alt circuit specializat necesar în sistemele cu 8086: generatorul de ceas 8284.

INTR, *Interrupt Request*, cerere întrerupere. Intrare activă pe nivel "1". Procesorul eşantionează această intrare sincronizată intern în ultimul ciclu de ceas al fiecărei instrucţiuni, pentru a intra, dacă INTR=1, într-o operaţie de achitare a întreruperii. La achitarea întreruperii se va apela o subrutină pe baza unei tabele de vectori localizate în memoria sistemului. Întreruperea poate fi mascată intern prin program punând pe "0" bitul de validare a întreruperii.

$\overline{TEST}$, intrare activă pe "0". Această intrare, sincronizată intern cu frontul pozitiv al fiecărui impuls de ceas CLK, este examinată de instrucţiunea WAIT: dacă intrarea este "0", microprocesorul îşi continuă execuţia, dacă este "1", va aştepta intrând în aşa-numita stare inactivă, TI, executată în general de UIM atunci când nu poate să execute un ciclu de magistrală (vezi §1.4.1).

NMI, *Non-Maskable Interrupt*, întrerupere nemascabilă. Intrare activă pe front pozitiv producând la sfârşitul instrucţiunii în curs o întrerupere de tipul 2 (vezi §1.4.5.1). Achitarea acestui tip de întrerupere conduce la apelarea unei subrutine pornind de la o tabelă de vectori aflată în memorie. Intrarea este sincronizată intern şi nu poate fi mascată prin program.

RESET, iniţializare. Intrare activă pe "1". Semnalul aplicat la această conexiune este sincronizat intern şi, dacă este activ cel puţin patru perioade de ceas, conduce la terminarea activităţii curente a microprocesorului şi iniţializarea execuţiei după revenirea lui pe "0".

CLK, *Clock*, ceas. Intrare ce asigură funcţionarea sincronă a microprocesorului şi a controlorului de magistrală. Factorul de umplere al ceasului este 33%.

1.2.1.2. **Semnalele specifice modului de lucru minim**

Aşa cum am menţionat în §1.1, una dintre caracteristicile cele mai interesante ale microprocesoarelor 8086/8088 este şi posibilitatea de a selecta configuraţia de bază a maşinii, modul de lucru cel mai potrivit aplicaţiei, prin conectarea la VCC sau GND a intrării MN / $\overline{MX}$.

În modul minim, 8086 permite realizarea de unităţi centrale mai restrânse ca volum, satisfăcând zona aplicaţiilor pe 16 biţi mici şi medii. În acest mod de lucru procesorul îşi menţine capacitatea de adresare a memoriei de 1M, a I/O de 64k, precum şi magistrala de 16 biţi, generând conexiunile 24÷31 direct, fără ajutorul unui circuit de comandă specializat. Sunt toate semnalele necesare manipulării magistralelor, DT / $\overline{R}$, $\overline{DEN}$, ALE, M / $\overline{IO}$, semnalele de comandă pentru operaţiile de citire, scriere, achitarea întreruperii, $\overline{RD}$, $\overline{WR}$, $\overline{INTA}$, precum şi semnalele HOLD, HLDA necesare operaţiilor simple de transfer cu acces direct implementate cu ajutorul controloarelor DMA obişnuite, de exemplu 8257 (vezi şi [9]).

În continuare, vom descrie, pe scurt, semnificaţia conexiunilor 24÷31 ale circuitului 8086 în modul de lucru minim.

M / $\overline{IO}$, *Memory/Input-Output*, memorie/intrare-ieşire. Ieşire 3-stări utilizată pentru a distinge accesele la memorie, M / $\overline{IO}$ =1, de cele la dispozitivele de I/O, M / $\overline{IO}$ =0. Ieşirea este validată în starea T4 precedentă unui ciclu de magistrală rămânând activă până la sfârşitul stării T4 a ciclului curent. Ieşirea, echivalentă cu $\overline{S2}$ în modul maxim, este trecută în starea a treia în ciclii de achitare a cererii de preluare a magistralei microprocesorului.

$\overline{WR}$, *Write*, scriere. Ieşire 3-stări activă pe "0" ce indică faptul că procesorul execută un ciclu de scriere a memoriei sau a I/O. Ieşirea este activă în stările T2, T3 şi TW, fiind trecută în starea de impedanţă înaltă în ciclii de achitare a cererii de preluare a magistralei microprocesorului.

$\overline{INTA}$, *Interrupt Aknowledge*, achitare întrerupere. Ieşire activă pe "0" utilizată pentru generarea unui semnal de eşantionare a citirii în ciclii de achitare a întreruperii. Ieşirea $\overline{INTA}$ este activă în timpul stărilor T2, T3 şi TW ale fiecărui ciclu de întrerupere.

ALE, *Address Latch Enable*, validarea *latch*-urilor de adrese. Ieşire activă pe "1" utilizată pentru memorarea adreselor generate de microprocesor pe magistrala multiplexată locală în *latch*-urile de adrese ale sistemului.

DT / $\overline{R}$, *Data Transmit/Receive*, trasmisie/recepţie date. Ieşire utilizată pentru comanda direcţiei *transceiver*-elor de date: transmisie pentru DT / $\overline{R}$ =1, recepţie pentru DT / $\overline{R}$ =0, sensul fiind raportat la microprocesor. Ieşirea este trecută în starea a treia în timpul ciclilor de achitare a cererii de preluare a magistralei.

$\overline{DEN}$, *Data Enable*, validare date. Ieşire 3-stări activă pe "0" folosită pentru validarea *transceiver*-elor de date. Este "0" în timpul operaţiilor cu memoria şi cu dispozitivele de I/O precum şi în timpul ciclilor de achitare întrerupere ($\overline{INTA}$), pentru citiri (şi $\overline{INTA}$) de la mijlocul lui T2 la mijlocul lui T4, iar pentru scrieri de la începutul lui T2 la mijlocul lui T4. Ieşirea este trecută în starea de impedanţă mare în timpul ciclilor de achitare a unei cereri de preluare a magistralei.

HOLD, cerere de preluare a magistralei microprocesorului. Intrare activă pe "1".

HLDA, achitare a cererii de preluare a magistralei. Ieşire activă pe "1".

După recepţionarea unei cereri HOLD microprocesorul va activa, la mijlocul stării T1, semnalul de achitare HLDA, trecând în acelaşi timp în starea a treia magistrala locală (AD15÷AD0, A19/S6÷A16/S3) şi ieşirile de comandă ($\overline{RD}$, $\overline{BHE}$ /S7, M / $\overline{IO}$, DT / $\overline{R}$, $\overline{WR}$, $\overline{DEN}$). La terminarea operaţiei de transfer direct dispozitivul care a activat semnalul HOLD îl va dezactiva trecându-l pe "0", după care procesorul va dezactiva la rândul lui semnalul de achitare HLDA. Magistrala locală şi ieşirile de comandă vor fi activate când microprocesorul va avea nevoie să execute un ciclu de magistrală. Menţionăm că intrarea HOLD nu este asincronă şi că, pentru asigurarea timpului de *set-up*, este necesară sincronizarea externă.

1.2.1.3. **Semnalele specifice modului de lucru maxim**

În acest mod de lucru, obţinut prin legarea conexiunii MN / $\overline{MX}$ la GND, 8086 permite implementarea configuraţiilor multiprocesor şi/sau cuplarea procesoarelor pentru extinderea setului de instrucţiuni, a *coprocesoarelor*. În modul maxim este necesară folosirea controlorului de magistrală 8288 pentru ca, prin redefinirea conexiunilor 24÷31, să se obţină această creştere a performanţelor.

$\overline{RQ}$ / $\overline{GT1}$, $\overline{RQ}$ / $\overline{GT0}$, *Request/Grant Bus Access Control*, comenzi de cerere/achitare a accesului la magistrala microprocesorului. Intrări/ieşiri active pe "0" folosite de alte dispozitive de tip *master*, aflate pe magistrala locală, pentru a forţa microprocesorul să elibereze această magistrală la sfârşitul ciclului în curs. Aceste conexiuni sunt prevăzute cu rezistenţe interne legate la VCC, ceea ce permite să fie lăsate în gol. $\overline{RQ}$ / $\overline{GT0}$ este prioritară faţă de $\overline{RQ}$ / $\overline{GT1}$.

$\overline{LOCK}$, blocare. Ieşire 3-stări activă pe "0". Indică celorlalte dispozitive de tip *master* aflate pe magistrala locală a microprocesorului, că nu pot prelua această magistrală atâta timp cât $\overline{LOCK}$=0. Ieşirea este activată de o instrucţiune cu prefix `LOCK` rămânând activă încă o perioadă de ceas după execuţia acestei instrucţiuni cu prefix. Ieşirea este trecută în starea a treia pe timpul ciclurilor de achitare a unei cereri de preluare a magistralei.

$\overline{S2} \div \overline{S0}$, *Status*, stare. Ieşiri 3-stări active în timpul stărilor T4, T1 şi T2 şi inactive, în starea pasivă (1,1,1) în timpul stărilor T3 şi TW dacă semnalul de la conexiunea READY este "1". Starea $\overline{S2} \div \overline{S0}$ este folosită de controlorul 8288 pentru a genera semnalele de comandă necesare în operaţiile de acces la memorie sau la I/O. Orice schimbare a acestor semnale în timpul stării T4 indică începutul unui ciclu de magistrală, revenirea în starea pasivă în T3 sau TW indicând sfârşitul ciclului de magistrală. Semnificaţia ieşirilor $\overline{S2} \div \overline{S0}$ este următoarea:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Semnificaţie
0	0	0	Achitare întrerupere
0	0	1	Citire *port* I/O
0	1	0	Scriere *port* I/O
0	1	1	Oprire (*Halt*)
1	0	0	Citire instrucţiune
1	0	1	Citire memorie
1	1	0	Scriere memorie
1	1	1	Stare pasivă (nici-un ciclu de magistrală)

Conexiunile $\overline{S2} \div \overline{S0}$ sunt trecute în starea a treia în timpul ciclurilor de achitare a unei cereri de preluare a magistralei.

QS1, QS0, *Queue Status*, starea cozii de instrucţiuni. Ieşiri validate în ciclul de ceas următor unei operaţii cu stiva. Aceste ieşiri permit urmărirea din exterior a execuţiei instrucţiunilor având, atunci când sunt validate, următoarea semnificaţie:

QS1	QS0	Semnificaţie
0	0	Nici o operaţie
0	1	Extragerea primului octet al codului-operaţie
1	0	Coada goală (nici un octet în coadă)
1	1	Extragerea octetului următor al codului-operaţie

1.2.2. CONEXIUNILE EXTERNE ALE MICROPROCESORULUI 8088

Precizăm în acest subcapitol numai conexiunile a căror semnificaţie diferă de cea a microprocesorului 8086, prezentată în §1.3.1.

1.2.2.1. **Semnalele comune modurilor de lucru minim şi maxim**

AD7÷AD0, *Address/Data Bus*, magistrală de adrese/date. Diferă de magistrala multiplexată a procesorului 8086 numai prin mărimea de 8 biţi faţă de 16.

A15÷A8, *Address Bus*, magistrală de adrese. Ieşiri 3-stări reprezentând biţii 8÷15 ai adresei. Spre deosebire de AD15÷AD8, aceşti biţi nu mai sunt multiplexaţi nefiind necesară memorarea lor în *latch*-uri cu ajutorul semnalului ALE. A15÷A8 rămân valizi pe durata întregului ciclu de magistrală, stările T1÷T4.

Conexiunile MN / $\overline{\text{MX}}$, $\overline{\text{RD}}$, READY, INTR, $\overline{\text{TEST}}$, NMI, RESET, CLK au aceleaşi semnificaţii cu cele ale microprocesorului 8086.

1.2.2.2. **Semnalele specifice modului de lucru minim**

IO / $\overline{\text{M}}$, *Input-Output/Memory*, intrare-ieşire/memorie. Ieşire 3-stări semnificând pentru IO / $\overline{\text{M}}$ =1 un ciclu de I/O iar pentru IO / $\overline{\text{M}}$ =0 un ciclu de memorie. Activarea acestei ieşiri respectă aceleaşi relaţii de timp ca şi ieşirea M / $\overline{\text{IO}}$ a lui 8086.

$\overline{\text{SS0}}$, *S0 Status*, starea S0. Ieşire 3-stări echivalentă cu bitul de stare S0 din modul maxim. Împreună cu IO / $\overline{\text{M}}$ şi DT / $\overline{\text{R}}$, ultima conexiune având aceeaşi semnificaţie ca şi la 8086, codifică starea ciclului de magistrală în curs:

IO / $\overline{M}$	DT / $\overline{R}$	$\overline{SS0}$	Semnificaţie
1	0	0	Achitare întrerupere
1	0	1	Citire *port* I/O
1	1	0	Scriere *port* I/O
1	1	1	Oprire (*Halt*)
0	0	0	Citire instrucţiune
0	0	1	Citire memorie
0	1	0	Scriere memorie
0	1	1	Stare pasivă

Celelalte conexiuni specifice modului minim de lucru al microprocesorului 8088, $\overline{WR}$, $\overline{INTA}$, ALE, $\overline{DEN}$, HOLD şi HLDA, rămân cu aceleaşi funcţii ca la 8086.

1.2.2.3. **Semnalele specifice modului de lucru maxim**

Semnalele specifice modului de lucru maxim al microprocesorului 8088, $\overline{S2}$, $\overline{S1}$, $\overline{S0}$, $\overline{RQ}$ / $\overline{GT1}$, $\overline{RQ}$ / $\overline{GT0}$, $\overline{LOCK}$, QS1 şi QS0 sunt aceleaşi cu ale microprocesorului 8086 în modul maxim. Conexiunea $\overline{SS0}$ este "1" în acest mod de lucru.

1.3. FUNCŢIONAREA MICROPROCESORULUI 8086

Vom descrie în cele ce urmează funcţionarea microprocesorului 8086 din punct de vedere al utilizării lui hardware.

1.3.1. CICLII DE MAGISTRALĂ AI MICROPROCESORULUI 8086. PREZENTARE GENERALĂ

8086 comunică cu exteriorul prin intermediul unei magistrale de comenzi şi a unei magistrale multiplexate în timp de adrese, stări şi date. Aşa cum am menţionat, tehnica multiplexării în timp a permis, atunci când a fost dezvoltat circuitul, utilizarea cea mai eficientă a celor 40 de conexiuni exterioare ale capsulei în care s-a împachetat microprocesorul. Această magistrală multiplexată, denumită şi magistrală locală, poate fi amplificată direct şi condusă în sistem, *latch*-area, memorarea adresei făcându-se distribuit în modulele de memorie sau de I/O. O altă abordare este cea a demultiplexării unice a adreselor şi datelor lângă procesor, cu ajutorul unui singur grup de

latch-uri pentru adrese şi a *transceiver*[2]-elor pentru date. Microprocesorul extrage cod sau transferă date executând aşa numiţii *cicli de magistrală*. Un ciclu de magistrală poate fi definit ca un eveniment asincron în care întâi se validează o adresă, a unei locaţii de memorie sau a unui dispozitiv periferic, apoi se generează fie un semnal de comandă a citirii, pentru capturarea datelor de la dispozitivul adresat, fie un semnal de comandă a scrierii asociat cu emiterea datelor ce trebuie transmise dispozitivului adresat. Observăm întrebuinţarea termenului ciclu de magistrală în locul mai vechiului termen *ciclu maşină* folosit în descrierea funcţionării unor procesoare pe 8 biţi ca 8080 sau Z80. Ciclii de magistrală trebuie înţeleşi ca grupuri de perioade de ceas reprezentând activitatea externă a microprocesorului fără a avea o legătură explicită şi ordonată cu execuţia curentă a unei instrucţiuni aşa cum aveau ciclii maşină (vezi de exemplu ciclii M_1, M_2, M_3 ai unui ciclu-instrucţiune Z80 [9]). În figura 1.9 este înfăţişată diagrama de bază reprezentând ciclii de citire şi scriere pe care îi execută microprocesorul pe magistrala sa de comunicaţie cu exteriorul. Această diagramă va fi detaliată în capitolele următoare pentru modurile de lucru minim şi maxim.

Orice ciclu de magistrală, TCY, are cel puţin patru perioade de ceas, numite *stări T*. În timpul primei stări T, T1, procesorul validează adresa A19÷A0 pe magistrala multiplexată. Tot acum se generează semnalul ALE, de către 8086 sau de controlorul de magistrală 8288, în funcţie de modul de lucru. Acest semnal serveşte memorării adresei în circuite de tip *latch*, de exemplu 74LS373 sau 8282, pe frontul negativ al acestui impuls garantându-se validitatea adresei. A doua stare, T2, este destinată schimbării direcţiei magistralei, microprocesorul invalidând adresa şi, pentru un ciclu de citire, trecând magistrala, cei mai puţin semnificativi 16 biţi, în starea a treia, iar pentru un ciclu de scriere validând datele. *Transceiver*-ele de date sunt validate, cu ajutorul semnalului DEN, în T1 sau T2, în funcţie de configuraţia sistemului şi direcţia transferului. Tot în T2 sunt generate şi comenzile de citire, $\overline{\text{RD}}$, scriere, $\overline{\text{WR}}$ sau achitare întrerupere, $\overline{\text{INTA}}$, precum şi starea S7÷S3 în locul biţilor de adresă A19÷A16 şi a semnalului $\overline{\text{BHE}}$.

În T3, pentru un ciclu de scriere, microprocesorul menţine biţii de stare şi datele ce trebuie transmise, iar în cazul unui ciclu de citire 8086 eşantionează datele păstrând $\overline{\text{RD}}$=0. Dacă memoria sau dispozitivul de I/O selectat nu sunt capabile să asigure transferul datelor până în T4 ele trebuie să avertizeze de acest lucru procesorul poziţionând, înainte de T3, semnalul READY pe "0". În acest fel ciclul de magistrală se prelungeşte prin introducerea de către microprocesor, între T3 şi T4, a unor stări de aşteptare TW al căror număr depinde de timpul necesar dispozitivului adresat pentru a asigura transferul. Activitatea procesorului pe magistrală în timpul unei stări TW este aceeaşi ca în

[2] Circuite amplificatoare bidirecţionale cum sunt, de exemplu, 8286 sau 74LS245

T3. Ieşirea din stările TW se face la iniţiativa dispozitivului, după trecerea timpului necesar asigurării corectitudinii transferului de date, prin trecerea semnalului READY pe "1".

Un ciclu de magistrală se termină cu starea T4 prin dezactivarea comenzilor şi eliberarea magistralei locale.

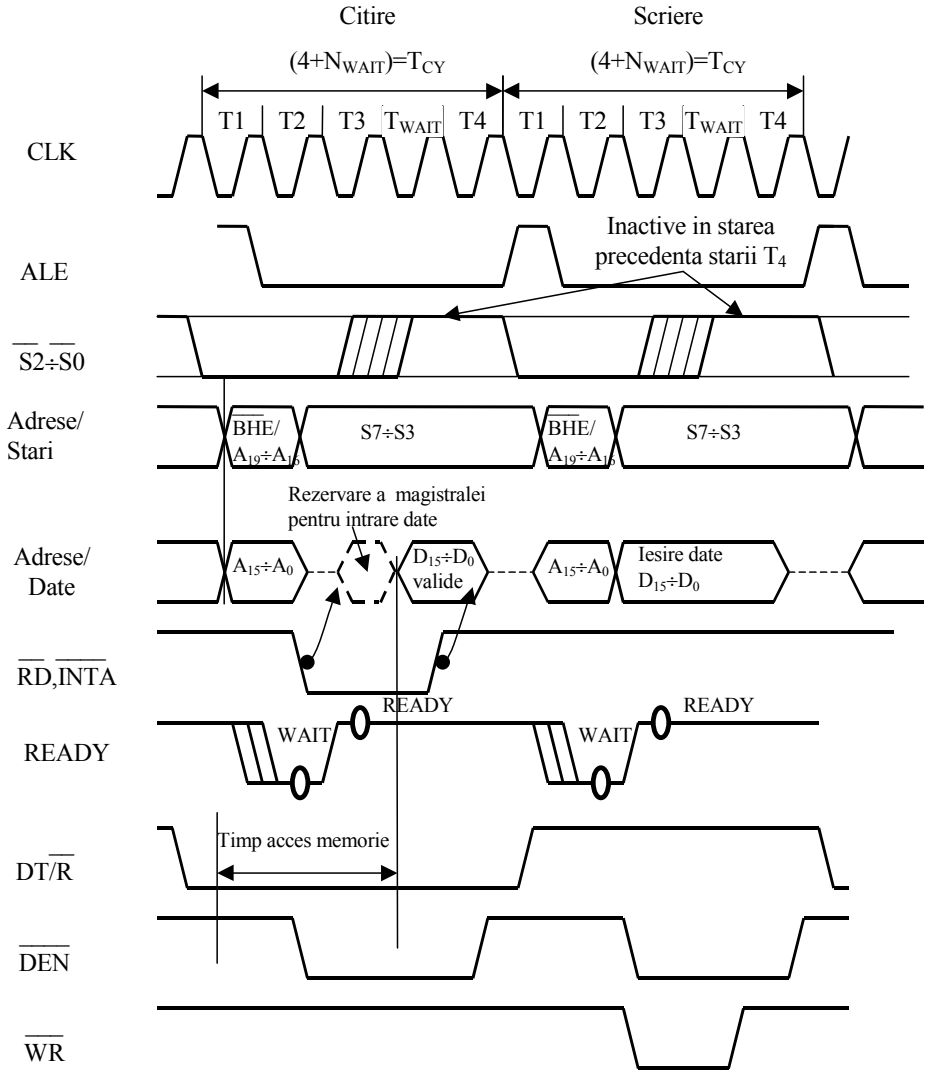


Figura 1.9. Ciclii de magistrală ai microprocesorului 8086. Diagrama de bază

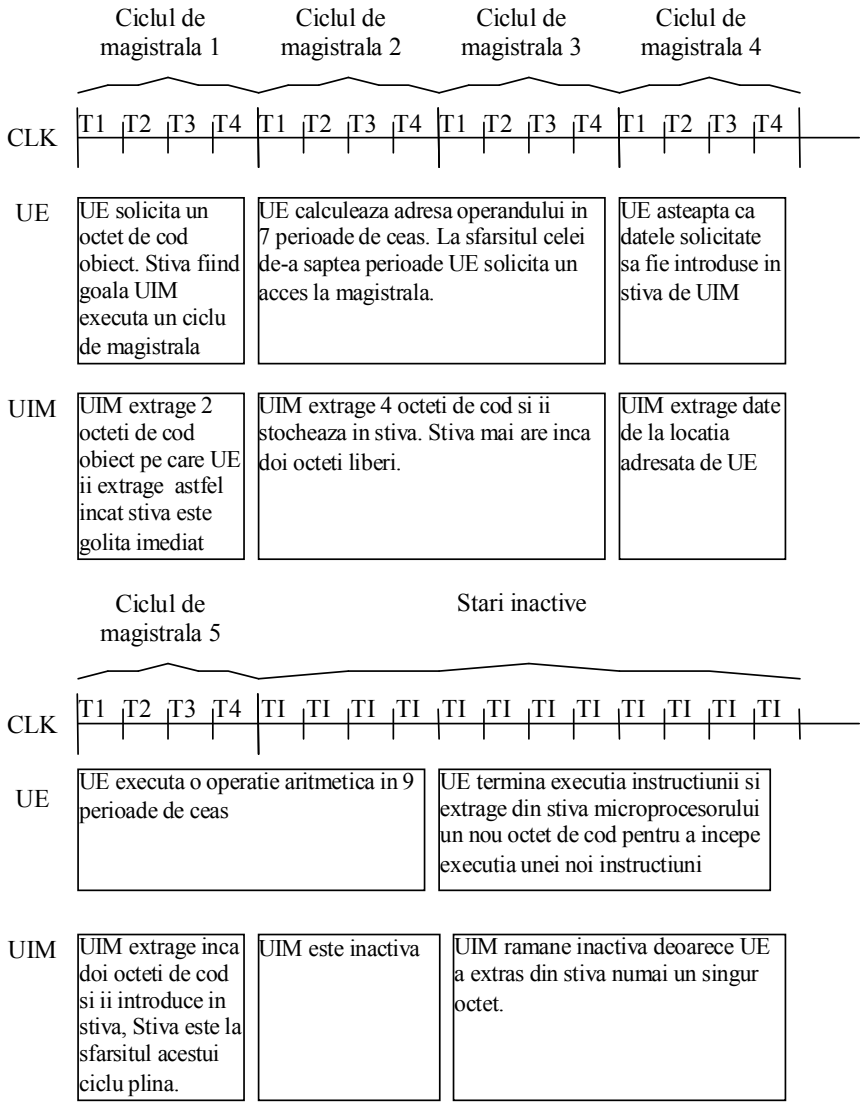


Figura 1.10. Un exemplu de funcţionare suprapusă a celor două blocuri funcţionale ale microprocesorului 8086

Între ciclii de magistrală, executaţi de unitatea centrală numai atunci când trebuie transferate instrucţiuni sau operanzi la memorie sau dispozitivele de I/O, pot apărea aşa numitele stări inactive, TI, în timpul cărora procesorul

execută operaţii interne. Dacă ciclul precedent a fost o scriere, 8086 va menţine în timpul stării TI informaţia de stare S7÷S3 din ciclul de magistrală precedent şi datele anterior emise pe magistrala multiplexată. Atunci când operaţia precedentă a fost o citire, în afara menţinerii stării S7÷S3, microprocesorul va lăsa în TI magistrala de date în starea de impedanţă înaltă.

Datorită modului de lucru suprapus al celor două blocuri funcţionale, UE şi UIM, activitatea exterioară a microprocesorului nu va apărea, ca în cazul maşinilor din generaţiile anterioare, de exemplu Z80, ca o succesiune de operaţii de extragere a codului, de *fetch*-uri, şi de transferuri corespunzătoare de date cu memoria sau dispozitivele de I/O. La 8086 extragerea codului şi transferurile de operanzi asociate unei instrucţiuni pot fi separate prin ciclii de magistrală executaţi de UIM în scopul umplerii stivei interne de 6 octeţi. De asemenea, pot apărea desincronizări, întârzieri, între citirea şi începutul execuţiei unei instrucţiuni. În figura 1.10, [4], se exemplifică modul de lucru suprapus al celor două blocuri funcţionale ale microprocesorului. Observăm că UE este activă în timpul execuţiei instrucţiunilor şi inactivă atunci când aşteaptă codul obiect sau date pe care nu le poate obţine decât prin intermediul unor cicli de magistrală executaţi de UIM. Activitatea UE se desfăşoară în secvenţe cu un număr variabil de perioade de ceas, negrupate, aşa cum am spus mai sus, în cicli maşină de lungime fixă. Pe de altă parte, UIM grupează perioadele de ceas în cicli de magistrală, dar numai când microprocesorul execută un transfer de cod sau date cu exteriorul. În restul timpului UIM rămâne inactivă.

1.3.2. FUNCŢIONAREA MICROPROCESORULUI ÎN MODUL MINIM

Conectarea intrării MN / $\overline{MX}$ a microprocesorului 8086 la +5V permite selectarea semnificaţiei funcţionale a grupului de conexiuni externe 24÷31 ale circuitului în vederea utilizării lui în modul minim, pentru realizarea de configuraţii simple sau medii, cu o singură unitate centrală. În figura 1.11 este prezentată o astfel de configuraţie organizată în jurul lui 8086 funcţionând în modul minim. Cu linie continuă sunt desenate circuitele şi semnalele necesare pentru alcătuirea unei structuri foarte simple. Cu linie punctată sunt indicate circuitele şi semnalele suplimentare necesare pentru implementarea unor structuri de complexitate medie.

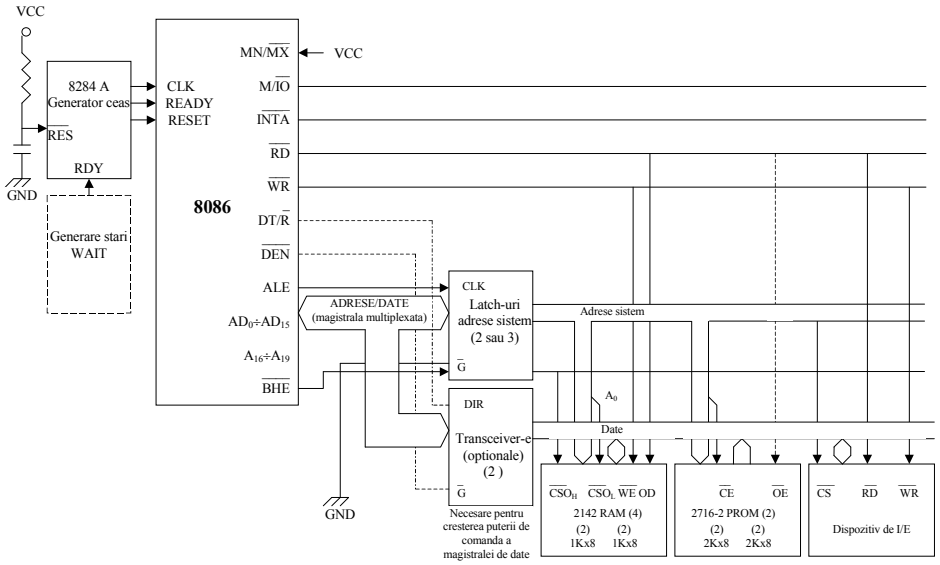


Figura 1.11. Microprocesorul 8086 în modul minim

În acest subcapitol vom interpreta diagramele de timp corespunzătoare operaţiilor externe, de magistrală, ale microprocesorului 8086 lucrând în modul minim. În analizele de timp care urmează ne vom referi la figura 1.12 unde sunt date diagramele detaliate de catalog, pentru modul de lucru minim şi la tabelele 1.2 şi 1.3 cu valorile parametrilor asociaţi acestor diagrame [6]. În funcţionarea pe magistrală a lui 8086 se pot pune în evidenţă, în modul minim ca şi în modul maxim, următoarele secvenţe de timp principale: adresarea, ciclul de citire, ciclul de scriere, achitarea întreruperii, introducerea stărilor de aşteptare, preluarea magistralei.

1.3.2.1. **Adresarea**

Adresarea memoriei sau a dispozitivelor de I/O se face prin demultiplexarea în timp a informaţiilor de pe magistrala locală a microprocesorului, cu ajutorul semnalului ALE destinat memorării adreselor în *latch*-urile de adrese ale sistemului. În operaţia de adresare vom analiza întâi raportul între adrese şi ALE apoi întârzierea adreselor până la apariţia lor pe magistrala demultiplexată a sistemului.

Tabelul 1.2. Parametrii 8086 pentru modul minim. Cerinţe de timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă	Condiţii de test şi observaţii
TCLCL	Perioada ceasului CLK	200ns	500ns	
TCLCH	Timpul cât CLK=0	118ns		
TCHCL	Timpul cât CLK=1	69ns		
TCH1CH2	Frontul crescător al ceasului CLK		10ns	Măsurat între valorile 1,0V şi 3,5V ale semnalului CLK
TCL2CL1	Frontul descrescător al ceasului CLK		10ns	Măsurat între valorile 3,5V şi 1,0V ale semnalului CLK
TDVCL	Timpul de stabilizare, de *set-up*, al datelor la citire	30ns		
TCLDX	Timpul de menţinere, de *hold*, al datelor la citire	10ns		
TR1VCL	Timpul de *set-up* al semnalului RDY la intrarea în circuitul 8284A măsurat faţă de frontul descrescător al lui CLK	35ns		Acest timp este necesar pentru semnalul asincron RDY pentru a garanta recunoaşterea lui în următoarea perioada a lui CLK
TCLR1X	Timpul de *hold* al semnalului RDY la intrarea în 8284A	0ns		
TRYHCH	Timpul de *set-up* al semnalului READY la intrarea în 8086	118ns		
TCHRYH	Timpul de *hold* al semnalului READY la intrarea în 8086	30ns		
TRYLCL	Timpul necesar ca intrarea READYsă fie considerată inactivă faţă de începutul stării T3	–8ns		READY mai poate deveni inactiv maximum 8ns în T3 transformând această stare într-o stare de aşteptare TW
THVCH	Timpul de *set-up* al semnalului HOLD	35ns		
TINVCH	Timpul de *set-up* al semnalelor INTR, NMI, $\overline{\text{TEST}}$	30ns		Timp necesar pentru ca aceste semnale asincrone să fie recunoscute în următoarea perioadă a ceasului CLK
TILIH	Frontul de creştere, pozitiv, al semnalelor de intrare (cu excepţia lui CLK)		20ns	De la 0,8V la 2,0V
TIHIL	Frontul descrescător, negativ, al semnalelor de intrare (exceptând CLK)		12ns	De la 2,0V la 0,8V

Tabelul 1.3. Parametrii 8086 pentru modul minim. Răspunsuri în timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă	Condiţii de test şi observaţii
TCLAV	Întârzierea validării adresei faţă de CLK	10ns	110ns	Măsurătorile au fost făcute adăugându- se la ieşirile corespun-zătoare ale microprocesorului C_L=20…100pF
TCLAX	Timpul de menţinere al adresei	10ns		
TCLAZ	Întârzierea până la trecerea adreselor în starea a treia	TCLAX	80ns	
TLHLL	Durata impulsului ALE	TCLCH–20ns		
TCLLH	Întârzierea activării semnalului ALE		80ns	
TCHLL	Întârzierea dezactiv. semnalului ALE		85ns	
TLLAX	Timpul de *hold* al adresei vs. ALE	TCHCL–10ns		
TCLDV	Întârzierea validării datelor vs. CLK	10ns	110ns	
TCHDX	Timpul de *hold* al datelor vs. CLK	10ns		
TWHDX	Timpul de *hold* al datelor după $\overline{WR}$	TCLCH–30ns		
TCVCTV	Întârzierea de activare a comenzilor $\overline{DEN}$, $\overline{WR}$, $\overline{INTA}$	10ns	110ns	
TCHCTV	Întârzierea de activare/dezactivare a comenzii $DT/\overline{R}$	10ns	110ns	
TCVCTX	Întârzierea la inactivarea comenzilor $\overline{DEN}$, $\overline{WR}$, $\overline{INTA}$	10ns	110ns	
TAZRL	Întârzierea între momentul trecerii adreselor în starea a treia şi activarea comenzii de citire $\overline{RD}$	0ns		
TCLRL	Întârzierea activării comenzii $\overline{RD}$	10ns	165ns	
TCLRH	Întârzierea la inactivarea comenzii $\overline{RD}$	10ns	150ns	
TRHAV	Întârzierea între inactivarea $\overline{RD}$ şi activarea următoarei adrese	TCLCL–45ns		
TCLHAV	Întârzierea validării semnalului HLDA	10ns	160ns	
TRLRH	Durata comenzii $\overline{RD}$	2·TCLCL –75ns		
TWLWH	Durata comenzii $\overline{WR}$	2·TCLCL –60ns		
TAVAL	Întârzierea între validarea adresei şi terminarea impulsului ALE	TCLCH–60ns		
TOLOH	Frontul crescător al semnalelor de ieşire		20ns	De la 0,8V la 2,0V
TOHOL	Frontul descrescător al semnalelor de ieşire		12ns	De la 2,0V la 0,8V

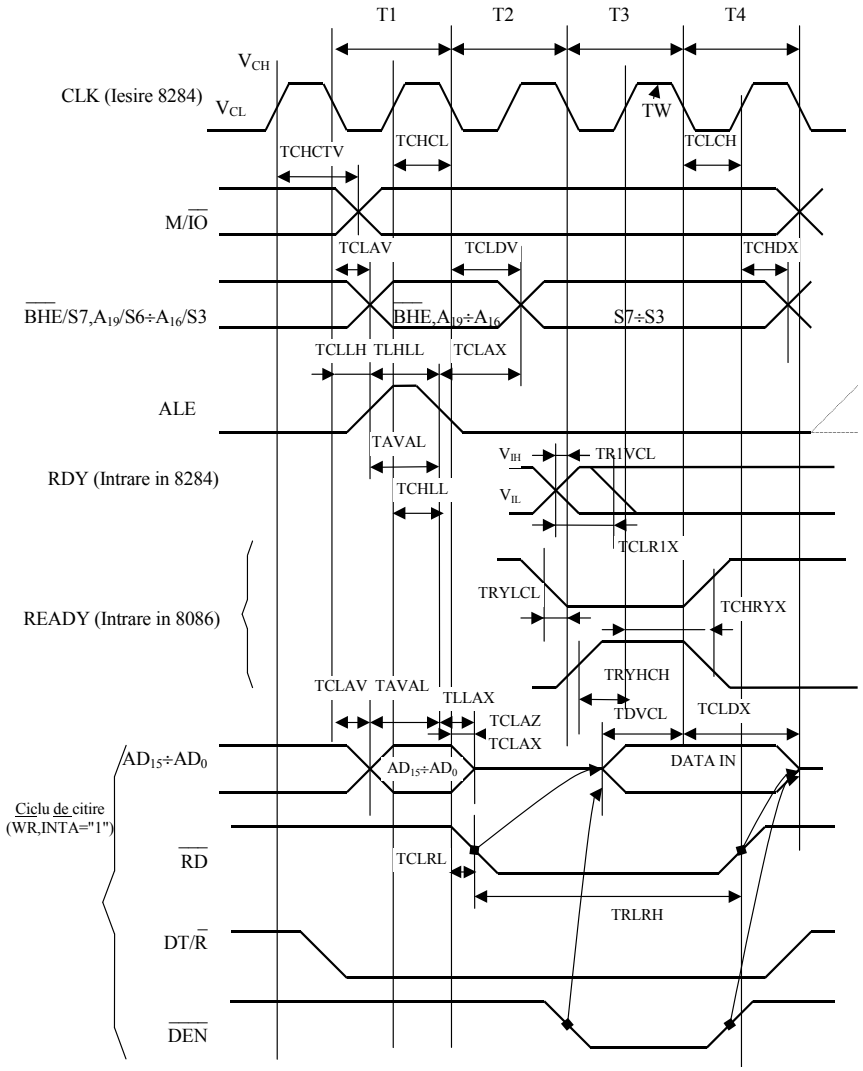


Figura 1.12. Diagrame de timp pentru modul minim

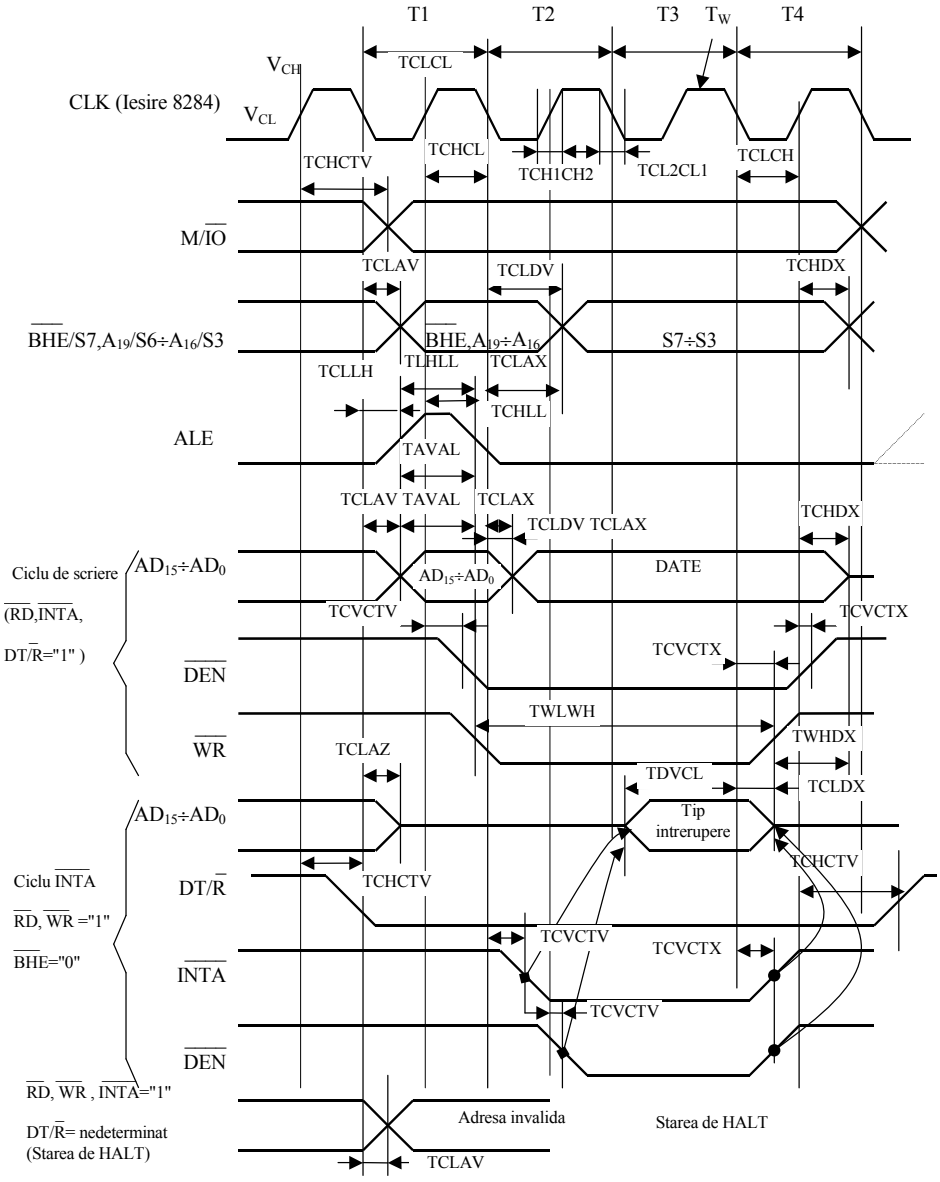


Figura 1.12. *Continuare*

Relaţia de timp între adresă şi ALE este garantată de fabricantul microprocesorului astfel încât capturarea unei adrese valide să se facă simplu prin folosirea unor *latch*-uri ca 8282 sau 74LS373. Este bine totuşi să discutăm

detaliat relaţia de timp între adresă şi ALE ea fiind importantă în aprecierea corectitudinii memorării unei adrese valide de pe magistrala multiplexată a lui 8086. Se cunoaşte că *latch*-urile folosite în mod obişnuit, ca 8282 sau 74LS373, sunt transparente atunci când ALE=1 ele memorând, capturând, datele de la intrare, în cazul nostru o adresă, pe frontul negativ al semnalului ALE. În acest caz ne interesează starea liniilor de adresă faţă de momentul terminării impulsului ALE. O situaţie defavorabilă va surveni atunci când întârzierea de apariţie a adreselor, TCLAV, este maximă şi ALE se termină cel mai repede, TCHLL fiind minim, zero. Rezultă de aici că adresa ar fi garantată validă, înainte de terminarea lui ALE, cu $TCLCH_{min}-TCLAV_{max}$= 118ns−110ns= 8ns ceea ce, deşi ar asigura un timp de stabilizare, de *set-up*, suficient pentru *latch*-urile menţionate mai sus (0ns pentru 8282 şi 5ns pentru 74LS373), nu constituie o garanţie de proiectare sigură. Într-adevăr, analiza de mai sus ignoră un alt parametru, cu adevărat important în relaţia de timp adresă/ALE şi anume TAVAL, timpul de garantare al unei adrese valide înaintea terminării semnalului ALE. Acest timp este dat, conform tabelului 1.2, de ecuaţia TAVAL= TCLCH−60ns şi este prioritar faţă de rezultatul obţinut mai sus prin luarea în considerare numai a relaţiilor de timp independente ale semnalelor faţă de ceasul microprocesorului. Înlocuind, obţinem în situaţia cea mai defavorabilă, pentru 8086 lucrând la 5MHz, TAVAL= 118ns−60ns= 58ns.

Timpul de menţinere, de *hold*, al adresei după frontul negativ al semnalului ALE este dat de TLLAX= TCHCL−10ns prioritar, de asemenea, faţă de valoarea obţinută prin luarea în considerare a întârzierilor TCHLL şi TCLAX faţă de CLK. În situaţia cea mai defavorabilă fabricantul garantează un timp de menţinere minim $TLLAX_{min}$= $TCHCL_{min}$−10ns= 69ns−10ns= 59ns faţă de o valoare minimă negativă obţinută arificial prin $TCHCL_{min}-TCHLL_{max}+TCLAX_{min}$= 69ns−85ns+10ns= −6ns. Semnificaţia lui $TCLAX_{min}$ se referă la performanţa microprocesorului de a-şi inactiva magistrala în cazul în care nu este condiţionată de un semnal ALE prea lent datorat, de exemplu, încărcării capacitive mai mari a ieşirii. Timpii TLLAX şi TCLAX definesc momentul comutării, după terminarea impulsului ALE, a întregii magistrale multiplexate, atât pentru ciclii de citire cât şi pentru cei de scriere. În timpul operaţiilor de citire, după ALE, conexiunile AD15÷0 vor fi trecute în starea de impedanţă înaltă iar, pentru operaţiile de scriere, comutate pe datele ce urmează a fi scrise. AD19÷16 comută din adrese în stări pentru ambele tipuri de cicli. Durata impulsului ALE este garantată de relaţia TLHLL= TCLCH−20ns, minimum 98ns, şi nu de TCLCH−TCLLH+TCHLL care, în cazul cel mai defavorabil, ar fi condus la o valoare minimă pentru ALE de 118ns−80ns+0ns= 38ns.

O altă relaţie de timp care ne interesează în faza de adresare este întârzierea adreselor demultiplexate, după *latch*-uri, faţă de adresele emise de

microprocesor pe magistrala locală multiplexată. Utilizând *latch*-uri ca 8282 sau 74LS373 adresa nu se va transmite decât după ce ALE=1. Comparând întârzierea maximă a emiterii adreselor, $TCLAV_{max}$= 110ns, cu întârzierea maximă a apariţiei impulsului ALE, $TCLLH_{max}$=80ns, deducem că întârzierea maximă a apariţiei adreselor pe magistrala demultiplexată de adrese a sistemului este dată de $TCLAV_{max}+t_{latch}$ unde t_{latch} este întârzierea maximă de propagare prin *latch*-uri (30ns pentru 8282, 18ns pentru 74LS373)

1.3.2.2. **Ciclul de citire**

Operaţia de citire propriu-zisă constă din: (1) trecerea magistralei locale AD15÷AD0 în starea a treia pentru a permite preluarea de către microprocesor a datelor, (2) activarea comenzii de citire $\overline{RD}$, (3) validarea *transceiver*-elor de date, dacă sunt prevăzute, cu ajutorul comenzii $\overline{DEN}$ şi (4) stabilirea direcţiei acestora prin intermediul semnalului $DT/\overline{R}$. Precizarea direcţiei de circulaţie a datelor în *transceiver* se face prin poziţionarea $DT/\overline{R}$=0, recepţie-date, la începutul ciclului de magistrală astfel încât acest semnal nu ridică probleme de timp. Problema cea mai importantă, ca relaţie de timp, la citirea datelor de către microprocesor este ca acestea să fie validate, plasate, pe magistrala locală respectându-se timpii de stabilizare, *set-up*, şi menţinere, *hold*, impuşi de 8086, TDVCL respectiv TCLDX. Corectitudinea datelor depinde, pe de-o parte, de selecţia şi întârzierea în răspuns ale dispozitivului de memorie sau de I/O adresat, iar pe de altă parte de validarea şi întârzierea *transceiver*-elor folosite în sistem.

Activarea comenzii $\overline{DEN}$ asigură un timp suficient pentru ca *transceiver*-ele să fie deschise circulaţiei datelor spre magistrala locală a microprocesorului asigurându-se timpul minim de stabilizare a datelor. Valoarea minimă a acestui timp pentru frecvenţa de 5MHz a ceasului e dată de relaţia $TCLCL + TCHCL_{min} - TCVCTV_{max} - TDVCL_{min}$= 200ns + 69ns −110ns − 30ns = 129ns. Invalidarea datelor prin invalidarea *tansceiver*-ului asigură timpul minim de menţinere al datelor, $TCLDX_{min}$=10ns, deoarece întârzierea minimă la dezactivarea comenzii $\overline{DEN}$, faţă de aceeaşi referinţă ca şi TCLDX, este $TCVCTX_{min}$=10ns şi întârzierea trecerii în starea a treia a *transceiver*-elor este cel puţin 0ns. De asemenea, importantă aici este şi durata între momentul invalidării *transceiver*-elor, implicând eliberarea magistralei locale a microprocesorului, şi momentul emiterii de către 8086 pe această magistrală a unei noi adrese. Această durată este pentru frecvenţa de 5MHz a ceasului de minimum $TCLCL - TCVCTX_{max} + TCLAV_{min}$= 200ns−110ns+10ns= 100ns.

Întârzierea plasării datelor pe magistrală de către dispozitivul citit se datorează întâi selecţiei circuitului, apoi comenzii efective de citire. De obicei,

selecţia se face pe baza adreselor, ceea ce nu pune probleme de timp, adresarea făcându-se, aşa cum am văzut, la începutul ciclului de magistrală. Rămâne întârzierea datorată comenzii efective de citire $\overline{RD}$. Timpul minim "oferit" de microprocesor dispozitivului citit pentru a plasa datele pe magistrala locală, cu ajutorul comenzii $\overline{RD}$, fără introducerea unor stări de aşteptare TW, la frecvenţa ceasului de 5MHz şi asigurându-se timpul minim de stabilizare este dat de formula $2 \cdot TCLCL - TCLRL_{max} - TDVCL_{min} = 2 \cdot 200ns - 165ns - 30ns = 205ns$. Acesta este timpul pe care îl au la dispoziţie datele pentru a fi "extrase", "accesate", cu ajutorul comenzii $\overline{RD}$ din dispozitivul adresat şi a parcurge magistralele şi circuitele interpuse până la microprocesor. Introducerea stărilor de aşteptare adaugă la acest timp câte o perioadă TCLCL pentru fiecare stare TW introdusă. Durata minimă a impulsului de citire, parametru important pentru dispozitivele de memorie sau I/O ce urmează a fi utilizate în sistem, este garantată de relaţia $TRLRH = 2 \cdot TCLCL - 75ns = 325ns$ prioritară faţă de formula $2 \cdot TCLCL - TCLRL + TCLRH$ care ar conduce în cazul cel mai defavorabil la $2 \cdot 200ns - 165ns + 10ns = 245ns$. Timpul de menţinere minim al datelor, $TCLDX_{min} = 10ns$, este asigurat şi pe această cale datorită faptului că întârzierea minimă a dezactivării comenzii $\overline{RD}$, faţă de aceeaşi referinţă ca şi TCLDX, este $TCLRH_{min} = 10ns$.

În sisteme configurate minimal, fără *transceiver*-e, dispozitivele de memorie şi/sau I/O sunt plasate direct pe magistrala locală multiplexată a microprocesorului. În această situaţie nu mai este necesară folosirea semnalelor $\overline{DEN}$ şi $DT/\overline{R}$, adresele şi comanda efectivă de citire, $\overline{RD}$, fiind suficiente. Utilizarea corectă în timp a magistralei locale este asigurată de garantarea activării semnalului $\overline{RD}$ cel puţin odată cu trecerea magistralei locale în starea a treia, $TAZRL_{min} = 0ns$, şi de asigurarea unui timp minim între dezactivarea lui $\overline{RD}$, şi validarea unei adrese noi de către 8086, $TRHAV_{min} = TCLCL - 45ns = 155ns$.

1.3.2.3. **Ciclul de scriere**

Scrierea efectivă presupune generarea datelor, activarea comenzii de scriere şi comanda *transceiver*-elor. După cum se vede în figura 1.12, pe timpul operaţiei de scriere $DT/\overline{R}$ este menţinut pe "1" ceea ce se asigură prin trecerea pe "1" a acestui semnal la sfârşitul oricărui ciclu de magistrală care implică citire de date, aşa cum am arătat deja în §1.4.2.2. Deci comanda direcţiei *transceiver*-elor, $DT/\overline{R}$, vine în ciclul de scriere poziţionată pe "1" din ciclul sau ciclii precedenţi, ea rămânând nemodificată, pe transmisie, în timpul operaţiei de scriere. Astfel microprocesorul poate să activeze comanda de

validare a *transceiver*-elor, $\overline{\text{DEN}}$, încă din starea T1, pe timpul operaţiei de adresare, fără să perturbe adresa emisă pe magistrala locală. Această validare avansată este necesară pentru minimizarea întârzierii datelor, momentul emiterii efective a lor găsind *transceiver*-ele deschise. Comanda de scriere, $\overline{\text{WR}}$, şi datele sunt activate cu acelaşi front negativ al ceasului CLK de la începutul stării T2. La începutul scrierii apare o zonă de incertitudine datorată întârzierilor diferite TCVCTV=10÷110ns pentru $\overline{\text{WR}}$ şi TCLDV=10÷110ns pentru date, ea fiind de maximum 100ns între cele două evenimente, de exemplu dacă TCVCTV=10ns şi TCLDV=110ns. Această incertitudine a relaţiei între date şi comanda $\overline{\text{WR}}$ la începutul comenzii de scriere impune proiectantului să utilizeze în sistem dispozitive de memorie sau I/O care să captureze datele pe frontul pozitiv al semnalului $\overline{\text{WR}}$ sau să imagineze diverse adaptări, circuite, care să asigure preluarea de către dispozitive a datelor de scriere după trecerea perioadei de incertitudine. Microprocesorul 8086 garantează validitatea datelor faţă de frontul pozitiv al comenzii de scriere $\overline{\text{WR}}$, fără introducerea unor stări de aşteptare, o durată dată de formula 2·TCLCL−TCLDV+TCVCTX. Introducerea stărilor TW adaugă un timp TCLCL pentru fiecare stare de aşteptare. În situaţia cea mai defavorabilă, pentru frecvenţa de 5MHz a ceasului, obţinem un timp garantat de minimum 2·200ns−110ns+10ns=300ns. Timpul de menţinere al datelor după dezactivarea comenzii de scriere este garantat de TWHDX=TCLCH−30ns fiind de minimum 118ns−30ns=80ns. TWHDX este prioritar faţă de timpul obţinut prin referiri la ceasul microprocesorului cu formula TCLCH+TCHDX−TCVCTX care ar conduce la o valoare minimă de 118ns+10ns−110ns=18ns.

La sfârşitul scrierii microprocesorul comută datele în adrese, dacă urmează imediat un nou ciclu de magistrală, T1 după T4, sau se trece în starea a treia, în cazul în care magistrala va fi cedată datorită achitării unei cereri HOLD sau $\overline{\text{RQ}}$. Aşa cum am spus şi în §1.4.1, dacă nu urmează imediat un nou ciclu de magistrală procesorul va menţine datele emise anterior în timpul operaţiei de scriere. Asigurarea timpului de menţinere a datelor la sfârşitul scrierii, după *transceiver*-e, la dispozitivul receptor, se face şi prin invalidarea comenzii $\overline{\text{DEN}}$ cu o întârziere de minimum $TCLCH_{min}$+ $TCVCTX_{min}$− $TCVCTX_{max}$= 118ns + 10ns −110ns = 18ns. Menţionăm acum că acest rezultat este abstract el presupunând în formula de mai sus că o întârziere de acelaşi tip, aici TCVCTX, poate fi simultan minimă şi maximă. În realitate acest lucru nu este posibil, o componentă electronică nefiind capabilă să demonstreze în acelaşi timp, pentru acelaşi parametru, întârziere maximă şi minimă. Argumentul conduce la concluzia că rezultatele obţinute în analize de timp cum este şi cea de mai sus reprezintă într-adevăr cazurile cele mai defavorabile, nemai fiind necesare amendări suplimentare pentru a da garanţii proiectantului. În situaţia de faţă

întârzierea reală între momentul inactivării comenzilor $\overline{WR}$ şi $\overline{DEN}$ este de aproximativ 60ns [10].

1.3.2.4. **Ciclul de achitare a întreruperii**

Vom descrie aici numai ciclul de magistrală specific achitării întreruperii mascabile, urmând ca într-un alt capitol să prezentăm pe larg structura mecanismului de întreruperi al microprocesorului 8086 şi întreaga lui activitate de pe magistrală la achitarea unei întreruperi.

Întreruperile mascabile generate de sistem sunt sesizate de procesor la intrarea INTR şi sunt mascate de bitul I, – indicatorul de condiţii pentru Validare/Invalidare Întreruperi –, din registrul de stare. În timpul ultimului ciclu de ceas al fiecărei instrucţiuni, cu unele excepţii ce vor fi precizate ulterior, microprocesorul eşantionează linia INTR. Dacă INTR este găsit pe "1" şi I poziţionat pe Validare-întreruperi 8086 va executa o secvenţă de achitare a întreruperii. Pentru a garanta achitarea de către microprocesor a întreruperii INTR va trebui menţinut pe "1" până când procesorul intră în ciclul de achitare şi activează semnalul de răspuns $\overline{INTA}$. Semnalul de cerere de întrerupere este de tip activ pe nivel el fiind sincronizat intern de către procesor cu frontul pozitiv al ceasului CLK şi testat apoi, cum am mai spus, în ultimul ciclu de ceas al instrucţiunii în curs. Dacă detectarea întreruperii, la sfârşitul execuţiei instrucţiunii în curs, se produce în timp ce UIM execută un ciclu de magistrală, extrăgând o nouă instrucţiune în vederea umplerii cozii de aşteptare, INTR trebuie să satisfacă şi un timp de stabilizare de cel puţin două perioade de ceas înainte de ultima stare, T4, a ciclului executat de UIM, pentru ca întreruperea să fie achitată imediat după terminarea acestuia. În cazul în care timpul de *set-up* faţă de sfârşitul acestui eventual ciclu de magistrală executat de UIM nu este asigurat, achitarea întreruperii se mai poate amâna până după derularea a încă unui ciclu de magistrală, dacă există vreunul în aşteptare.

Secvenţa hardware, specifică, de achitare a întreruperii mascabile, INTR, de către microprocesorul 8086 este compusă din cicli $\overline{INTA}$ separaţi prin două stări inactive TI, figura 1.13. Cei doi cicli de magistrală $\overline{INTA}$ sunt, aşa cum se poate vedea şi în figura 1.12, asemănători logic cu ciclii de citire, comanda $\overline{RD}$ înlocuindu-se cu semnalul de achitare $\overline{INTA}$. Diferă doar relaţiile de timp referitoare la $\overline{INTA}$ şi magistrala AD15÷AD0.

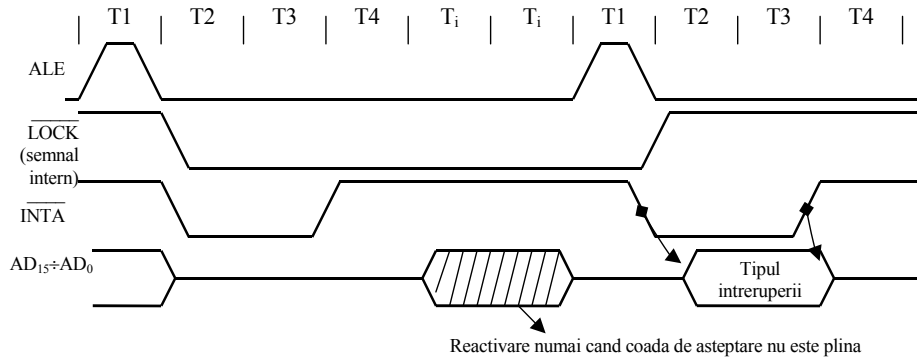


Figura 1.13. Secvenţa de achitare a unei întreruperi în modul minim

În timpul ciclilor $\overline{\text{INTA}}$ faza de adresare nu este efectivă, în sensul că ALE poate încărca în *latch*-uri o adresă nedeterminată, magistrala multiplexată fiind trecută în starea a 3-a în T1. Această situaţie impune ca dispozitivele de memorie sau I/O să nu fie selectate şi/sau validate pe magistrala sistemului în operaţiile de citire cu semnale care să reprezinte numai decodificări ale adreselor specifice ci cu semnale care să înglobeze şi comanda efectivă de citire, $\overline{\text{RD}}$, pentru modul de lucru minim. Semnalul ALE este generat în timpul ciclilor de achitare cu scopul de a fi utilizat, cum se va arăta mai târziu, în sisteme complexe cu mai multe controloare de întreruperi, 8086 lucrând în modul maxim.

Primul ciclu $\overline{\text{INTA}}$ este destinat semnalizării începutului achitării întreruperii atenţionând sistemul să-şi pregătească răspunsul pentru al doilea ciclu $\overline{\text{INTA}}$. În timpul acestui prim ciclu microprocesorul nu va citi magistrala de date, el efectuând anumite operaţii interne necesare achitării întreruperii. Răspunsul dispozitivului care întrerupe este un octet reprezentând aşa-numitul *tip al întreruperii* ce urmează a fi citit şi prelucrat apoi de procesor pentru a se obţine adresa efectivă a subrutinei de tratare a întreruperii. Tipul întreruperii este citit de microprocesor, în al doilea ciclu $\overline{\text{INTA}}$, pe jumătatea mai puţin semnificativă AD7÷AD0 a magistralei locale. Aceasta impune ca dispozitivele specifice, controloarele de întreruperi, care, în cadrul secvenţelor de achitare, răspund prin generarea tipului de întrerupere să fie plasate pe jumătatea mai puţin semnificativă a magistralei de date, la adrese pare.

Pe durata secvenţei de achitare, magistrala multiplexată de adrese/date este trecută în starea a 3-a, de impedanţă mare, în T1, la începutul fiecărui ciclu $\overline{\text{INTA}}$, cu o întârziere dată de parametrul TCLAZ faţă de frontul negativ de la începutul stării T1. Liniile superioare de adrese/stări, A19/S6÷A16/S3 nu sunt folosite în timpul ciclilor $\overline{\text{INTA}}$. Starea însă, S7÷S3 comutată în T2 este validă şi are valoarea: S7=S6=0, S5=IF, S4=S3=0. Magistrala AD15÷AD0 rămâne în starea a 3-a dar, adresa, A19÷A16, are o valoare nedeterminată ca şi primii 16 biţi A15÷A0. Starea însă, S7÷S3 comutată în T2 este validă şi are valoarea: S7=S6=0, S5=IF, S4=S3=0. Magistrala AD15÷AD0 rămâne în starea a 3-a dar, adresa, A19÷A16, are o valoare nedeterminată ca şi

3-a până în ciclul de ceas următor stării T4 a fiecăruia din cei doi cicli de magistrală $\overline{INTA}$. În cazul în care, pe timpul achitării întreruperii, coada de aşteptare a microprocesorului nu este plină, pe magistrala AD pot fi plasate, în timpul stărilor inactive TI, date în condiţiile discutate anterior în §1.4.1. Datele reprezentând tipul întreruperii trebuie să satisfacă aceleaşi cerinţe pentru timpii de stabilizare şi menţinere, TCLDV şi TCLDX, faţă de frontul negativ al ceasului de la începutul stării T4 din al doilea ciclu $\overline{INTA}$, ca şi datele dintr-un ciclu de citire.

Ieşirea $M/\overline{IO}$ va fi poziţionată pe "0" indicând sistemului, în timpul ciclilor $\overline{INTA}$, o operaţie de intrare/ieşire. Semnalul $\overline{LOCK}$, intern în cazul funcţionării microprocesorului 8086 în modul minim, va fi şi el activat între stările T2 ale celor doi cicli de achitare pentru a preveni achitarea de către UIM a unei cereri de magistrală între cei doi cicli $\overline{INTA}$. Asupra priorităţii între cererea de întrerupere şi cererea de magistrală vom reveni în §1.4.5.4. Comenzile de validare şi sens pentru *transceiver*-e, $\overline{DEN}$ şi $DT/\overline{R}$, sunt activate în fiecare ciclu $\overline{INTA}$ având aceleaşi relaţii de timp ca şi în cazul ciclilor de citire. Între cei doi cicli $\overline{INTA}$, $\overline{DEN}$ şi $DT/\overline{R}$ sunt dezactivate.

Relaţiile de timp pentru semnalul de achitare $\overline{INTA}$ sunt identice cu cele ale comenzii de scriere $\overline{WR}$. Astfel, pe baza acestor relaţii obţinem un timp de acces, de la validarea comenzii $\overline{INTA}$ până la stabilizarea pe magistrala microprocesorului a datelor reprezentând timpul întreruperii, dat de formula 2·TCLCL – TCVCTV– TDVCL care, pentru cazul cel mai defavorabil conduce la valoarea $2\cdot TCLCL - TCVCTV_{max} - TDVCL_{min} = 2\cdot 200ns - 110ns - 30ns = 260ns$. Mărirea timpului de acces se poate face prin introducerea de stări TW care, pentru fiecare stare introdusă, adaugă la acest timp o perioadă TCLCL. Garantarea timpului de menţinere TCLDX necesar procesorului pentru capturarea datelor pe frontul negativ de la începutul stării T4 se asigură prin condiţionarea menţinerii datelor pe magistrală cu $\overline{INTA}=0$ şi $\overline{DEN}=0$ şi invalidarea acestor comenzi cu cel puţin $TCVCTX_{min}=10ns$ după frontul CLK menţionat. Durata minimă a impulsului $\overline{INTA}$ rezultă din relaţia 2·TCLCL–TCVCTV+TCVCTX, conducând, pentru un ceas de 5MHz, la o valoare teoretică minimă de 2·200ns–110ns+10ns= 300ns. Având în vedere că în realitate microprocesorul nu poate să manifeste simultan, pentru acelaşi semnal, atât întârzieri maxime cât şi minime, rezultă o valoare practică minimă de 340ns [10]. Pentru eliberarea magistralei la sfârşitul celor doi cicli $\overline{INTA}$ trebuie ţinut cont, pe de-o parte, de întârzierea maximă a comenzii $\overline{INTA}$, $TCVCTX_{max}=110ns$, iar pe de altă parte de momentul când microprocesorul activează magistrala în starea T1 imediat următoare stării T4 din al doilea ciclu $\overline{INTA}$. Rezultă un timp minim de $TCLCL–TCVCTX_{max}+TCLAV_{min}= 200ns–110ns+10ns= 100ns$ în care dispozitivul care a plasat pe magistrală tipul întreruperii trebuie să-şi invalideze ieşirile. Acelaşi timp rezultă şi luând în

considerare invalidarea datelor datorită comenzii $\overline{DEN}$, pentru sisteme în care se folosesc *transceiver*-e.

1.3.2.5. **Introducerea stărilor de aşteptare**

Stările de aşteptare, TW, pot fi inserate, în timpul operaţiilor de citire, scriere sau achitare a întreruperii pentru a se facilita transferul de date cu dispozitive de memorie sau de I/O mai lente. Stările TW se inserează între stările T3 şi T4. Pentru a introduce o stare de aşteptare într-un ciclu de magistrală semnalul READY trebuie inactivat, pus pe "0", la sfârşitul stării T2 iar pentru a nu intra într-o stare de aşteptare sau pentru a ieşi din ea, intrarea READY a microprocesorului trebuie activata, trecută pe "1", înainte de frontul pozitiv al ceasului din T3, respectiv din TW.

Introducerea stărilor de aşteptare se poate face în două moduri dictate, în general, de complexitatea configuraţiilor: primul este acela al sistemelor considerate în mod normal *not ready* – nepregătite, al doilea al sistemelor aflate în mod normal în starea *ready* – pregătite pentru transfer.

În cazul primei abordări, considerate clasică, dacă dispozitivul selectat care primeşte o comandă, $\overline{RD}$, $\overline{WR}$ sau $\overline{INTA}$, are timp să o execute în ritmul impus de microprocesor el va trebui să activeze în timp util intrarea READY pentru a permite procesorului să-şi încheie ciclul de magistrală prevenindu-se astfel introducerea stărilor TW, ca în figura 1.14. Dacă dispozitivul selectat nu poate executa comanda în ritmul microprocesorului el va introduce una sau mai multe stări de aşteptare menţinând pe "0" un timp corespunzător intrarea READY a lui 8086. Această soluţie se pretează, de obicei, sistemelor complexe cu mai multe procesoare sau celor realizate pe magistrale standard cum ar fi Multibus, unde întârzierile de propagare şi de acces la magistrală sunt relativ mari şi multe din operaţiile externe microprocesorului nu se pot executa la viteza lui maximă. Dispozitivele cele mai rapide, puţine în cazul adaptării prezentei soluţii, vor trebui să activeze semnalul READY ca în figura 1.14 pentru a permite funcţionarea sistemului la viteza maximă. În cazul în care nu se respectă relaţiile de timp necesare poate fi introdusă o stare TW în plus.

A doua abordare presupune că majoritatea operaţiilor externe microprocesorului se pot face în ritmul impus de el şi deci că semnalul READY este în mod normal activ, pe "1". Dispozitivele mai lente, mai puţine sau mai puţin solicitate în configuraţiile pentru care alegem această implementare, vor trebui să dezactiveze semnalul READY pentru a introduce stări TW ca, de exemplu, în figura 1.15. Metoda sistemelor în stare normală *ready* conduce în general la scheme mai simple şi este potrivită cu configuraţiile mici, monoprocesor, unde nu se pun probleme de viteză legate, cum am menţionat, de accesul la o magistrală complexă sau la resurse localizate pe module diferite

cum e cazul configuraţiilor complexe, multiprocesor. Dezavantajul acestei metode apare în posibilitatea terminării anormale a unui ciclu de magistrală datorită dezactivării incorecte, fără respectarea relaţiilor de timp impuse, a semnalului READY. De aceea, pentru a obţine siguranţă în funcţionarea sistemului, metoda impune analizarea atentă a relaţiilor de timp legate de accesul microprocesorului la toate dispozitivele de memorie şi/sau I/O.

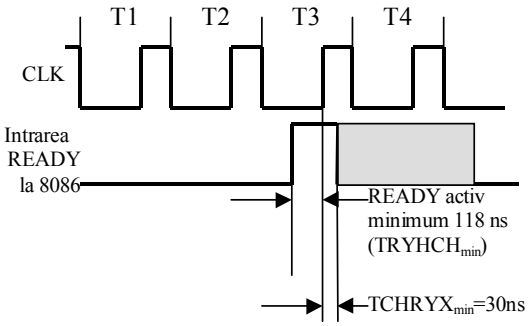


Figura 1.14. Prevenirea introducerii unor stări de aşteptare în sisteme normale "*not-ready"*

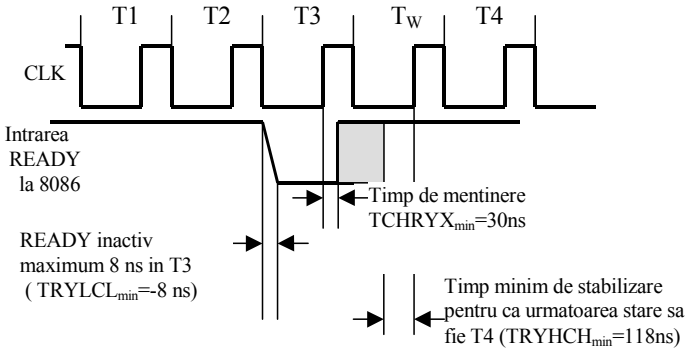


Figura 1.15. Introducerea unei stări de aşteptare în sisteme normal "*ready*"

După cum vedem în figurile detaliate 1.14 şi 1.15 dar şi în diagramele de catalog din figura 1.12 utilizarea intrării READY ascultă de cerinţe diferite în funcţie de metoda folosită. În cazul sistemelor în starea normală *ready*, pentru a introduce o stare TW, intrarea READY a microprocesorului trebuie pusă pe "0" în cel mult 8ns (TRYLCL$_{min}$=-8ns) după sfârşitul stării T2, începutul stării T3. După depăşirea acestei limite permise, pentru garantarea unei funcţionări corecte a microprocesorului, fabricantul interzice modificarea intrării READY din "1" în "0" pe timpul cât CLK mai este "0" în starea T3. Menţionăm că aceste restricţii sunt rezolvate în cazul utilizării circuitului de ceas 8284 care asigură şi formarea semnalului READY aşa cum vom arăta în §1.4.4.4. În cazul sistemelor

în starea normală *not ready* pentru a preveni introducerea stărilor de aşteptare intrarea READY trebuie trecută pe "1" cu cel puţin 118ns, $TRYHCH_{min}$, înainte de frontul pozitiv al ceasului din T3. În cele două situaţii prezentate mai sus, ilustrând introducerea respectiv prevenirea introducerii unei stări TW, pentru ca procesorul să ia o decizie corectă este necesar ca semnalul READY, "0" sau "1", să fie menţinut un timp minim dat de parametrul $TCHRYX_{min}$= 30ns faţă de frontul pozitiv din T3.

Este necesar să precizăm că în sistemele în stare normală *not ready* programatorii pot să ia o măsură de precauţie specifică nescriind cod executabil în ultimii şase octeţi ai memoriei existente. Acest lucru elimină situaţia în care, datorită preextragerilor de instrucţiuni şi existenţei codului executabil în ultimele locaţii fizice, 8086 ar putea, depăşind spaţiul de memorie fizică, să se blocheze în TW deoarece, în acest caz, fără măsuri speciale de proiectare, memoria nu mai răspunde prin activarea semnalului READY. Neasignând cod executabil ultimilor şase octeţi rezervăm această zonă de memorie extragerilor în avans ale UIM, executate, aşa cum ştim, de microprocesor în scopul umplerii cozii de aşteptare, şi pentru care memoria poate răspunde corect, fără complicaţii hardware, prin trecerea pe "1" a semnalului READY.

Semnalul READY este generat de obicei fie numai pe baza decodificării de adresă fie pe baza decodificării de adresă şi condiţionării cu ajutorul comenzii, $\overline{RD}$, $\overline{WR}$ sau $\overline{INTA}$. Pentru sisteme normale *not ready* timpul în care trebuie activat semnalul READY pornind de la adresă pentru a preântâmpina introducerea unei stări TW, cazul dispozitivelor rapide ce pot funcţiona în ritmul impus de microprocesor, este dat de 2·TCLCL+TCLCH−TCLAV−TRYHCH care, în situaţia cea mai defavorabilă, conduce la 2·200ns+118ns−110ns−118ns= 290ns. Acesta este timpul pus la dispoziţie de microprocesorul 8086 unui sistem normal *not ready* pentru ca el să decodifice adresa şi să activeze semnalul READY luând astfel decizia preîntâmpinării introducerii unei stări TW. În cazul în care este necesară introducerea unor stări de aşteptare sistemul nu are altceva de făcut decât să întârzie activarea liniei READY cu numărul de perioade de ceas corespunzător.

Pentru sistemele normale *ready* timpul în care sistemul poate de data aceasta inactiva semnalul READY, pentru a introduce stări TW pe baza decodificării de adresă, este dat de 2·TCLCL−TCLAV−TRYLCL. În situaţia cea defavorabilă obţinem 2·200ns−110ns−(−8ns)=298ns. Pentru dispozitivele rapide sistemul nu va acţiona asupra liniei READY lăsând-o în "1".

Revenirea în starea normală a semnalului READY, în cazul modificării lui, se face la începutul ciclului de magistrală următor imediat după schimbarea adresei prin deschiderea *latch*-urilor datorită semnalului ALE.

În cazul condiţionării semnalului READY de comenzile $\overline{RD}$, $\overline{WR}$ sau $\overline{INTA}$ timpii puşi la dispoziţie de microprocesor logicii externe sunt mult mai mici. Astfel pentru citiri şi sisteme normale *not ready* acest timp, dat de relaţia TCLCL+TCLCH−TCLRL−TRYHCH, este în situaţia cea mai defavorabilă de 200ns+118ns−165ns−118ns= 35ns. Pentru sisteme normale *ready* rezultă din relaţia TCLCL−TCLRL−TRYLCL un timp de 200ns−165ns−(−8ns)= 43ns. Pentru scrieri şi cicli $\overline{INTA}$ formulele se modifică prin înlocuirea parametrului TCLRL cu TCVCTV rezultând timpii de 90ns şi respectiv 98ns. Atragem atenţia că timpii obţinuţi mai sus se referă la semnalul READY de la intrarea microprocesorului 8086. În §1.4.4.4, referitor la generarea semnalului READY cu ajutorul circuitului 8284, vom reveni precizând şi restricţiile impuse semnalului RDY de la intrarea acestui circuit. Analiza relaţiilor de timp ale semnalului READY de la intrarea microprocesorului 8086 este utilă mai ales la realizarea unor structuri care nu folosesc circuitul specializat 8284.

Din compararea rezultatelor obţinute mai sus concluzionăm că restricţiile de timp impuse sistemelor la generarea semnalului READY sunt mai largi pentru situaţiile în care semnalul se formează prin decodificarea adresei faţă de situaţiile în care acesta este condiţionat de comenzi. De aici recomandările ca pentru sistemele în stare normală *not ready* formarea semnalului READY să se facă, pentru dispozitivele rapide care nu impun introducerea de stări TW, numai prin decodificarea de adresă, iar pentru dispozitivele lente şi prin folosirea, dacă este cazul, a comenzilor. Pentru sistemele în stare normală *ready* în cazul dispozitivelor rapide READY nu trebuie acţionat iar în situaţia dispozitivelor lente semnalul READY se recomandă a fi dezactivat din timp, numai pe baza decodificării de adresă, şi reactivat ţinând cont eventual şi de comandă. Există, de asemenea, posibilitatea utilizării semnalului $M/\overline{IO}$ la formarea lui READY în sistemele pentru care memoria este rapidă şi nu are nevoie de prelungirea ciclilor de magistrală, fiind în schimb necesară introducerea unui număr fix de stări TW pentru toate dispozitivele de I/O. $M/\overline{IO}$ permite declanşarea unei scheme care să introducă un astfel de număr prefixat de stări TW la orice ciclu $\overline{RD}$, $\overline{WR}$ sau $\overline{INTA}$ indiferent de adresa circuitului accesat.

1.3.2.6. **Preluarea controlului magistralei locale (accesul direct)**

Preluarea controlului magistralei locale a microprocesorului 8086 funcţionând în modul minim se face printru-un procedeu de cerere-achitare

analog cu cel folosit de microprocesoarele 8080 şi Z80 [5, 9, 27, 28]. Semnalele utilizate sunt HOLD pentru cererea de magistrală, analog cu semnalele HOLD şi $\overline{\text{BUSRQ}}$ de la 8080 şi respectiv Z80, şi HLDA pentru achitarea cererii, echivalent cu HLDA şi $\overline{\text{BUSAK}}$. Pentru a prelua controlul magistralei locale un dispozitiv, de exemplu un coprocesor sau un circuit pentru comanda accesului direct la memorie, trebuie să activeze intrarea HOLD a procesorului şi să aştepte activarea semnalului de achitare pentru a putea efectua transferuri de date pe magistrala locală. Intrarea HOLD este eşantionată de 8086 cu frontul negativ al ceasului, figura 1.16. Decizia de a răspunde la cererea HOLD este luată în unitatea de interfaţă cu magistrala fiind influenţată de următorii factori: activitatea curentă a magistralei, starea semnalului intern $\overline{\text{LOCK}}$, activat de prefixul software LOCK, şi starea întreruperilor. După eşantionare microprocesorul poate achita cererea HOLD numai dacă următoarea stare T este T4 sau TI. Dacă nu, el aşteaptă sfârşitul ciclului de magistrală în curs pentru a trece pe "1" ieşirea HLDA. Dispozitivul care a emis HOLD trebuie să menţină acest semnal activ pe toată durata cât are nevoie de magistrală. Aşa cum am spus mai sus, dialogul pentru preluarea magistralei are loc între dispozitivul solicitant şi UIM. UE îşi poate continua activitatea până la golirea stivei sau până la apariţia necesităţii transferării unui operand. La terminarea operaţiei, HOLD va fi trecut pe "0" microprocesorul răspunzând la rândul lui prin HLDA=0 şi, dacă este cazul, prin reactivarea magistralei locale. Precizăm, din nou, că datorită funcţionarii de tip *pipeline* a celor două blocuri constructive, UE şi UIM, operaţia HOLD/HLDA se poate face în paralel cu execuţia unor operaţii interne de către UE. În această situaţie este posibil ca la încheierea ciclului de cerere/achitare de magistrală microprocesorul să nu aibă nevoie de un acces în exterior şi deci, nefiind cazul, să nu activeze imediat magistrala locală. Va apărea atunci o perioadă de timp în care magistrala locală nu va fi controlată de nici-un dispozitiv rămânând în starea a treia. De aceea se recomandă, pentru sistemele care folosesc facilitatea HOLD/HLDA, să se conecteze ieşirile de comandă la +5V prin rezistenţe de 22kΩ, asigurându-se astfel menţinerea, în perioada de incertitudine, a nivelului inactiv "1".

Cererea de preluare a controlului, HOLD, nefiind sincronizată de procesor, trebuie să respecte un timp minim de stabilizare faţă de frontul pozitiv al ceasului. Acest timp, $THVCH_{min}$= 35ns, poate fi asigurat dacă se face o sincronizare externă folosindu-se ceasul microprocesorului, oricare din cele două fronturi fiind utilizabil. Trecerea magistralei locale în starea a treia se poate face teoretic după validarea semnalului de achitare HLDA. Întârzierea între activarea semnalului HLDA şi eliberarea magistralei este dată de diferenţa TCLAZ–TCLHAV fiind de maximum 70ns. Această situaţie poate să deranjeze în sistemele în care activarea lui HLDA conduce imediat la activarea de către dispozitivul solicitant a liniilor de comandă. De aceea pentru aceste cazuri se

recomandă întârzierea lui HLDA astfel încât să se anuleze zona de suprapunere de maximum 70ns. Întârzierile minime cu care microprocesorul conduce din nou magistrala locală, după inactivarea cererii HOLD, sunt date, pentru comenzi, de $THVCH_{min}+3\cdot TCLCL= 635ns$ şi pentru adrese/date/stări de $THVCH_{min}+3\cdot TCLCL+ TCHCL_{min}= 704ns$, figura 1.17 (s-a considerat că între fronturile ceasului ultimei perioade şi activarea semnalelor nu există întârziere). Dacă aceste întârzieri nu sunt suficiente pentru ca dispozitivul solicitant să elibereze magistrala, el trebuie să întârzie la microprocesor momentul trecerii lui HOLD pe "0". În sfârşit, întârzierea între momentul trecerii lui HLDA pe "0" şi preluarea comenzilor este de minimum $TCLCL+TCLCH_{min}-TCLHAV_{max}= 200ns+118ns-160ns= 158ns$ iar până la preluarea adreselor/datelor de $2\cdot TCLCL-TCLHAV_{max}= 240ns$. Întârzierile legate de semnalele HOLD/HLDA şi magistrala locală sunt importante pentru proiectant care, ţinând cont de ele, va trebui să asigure controlul disjunct, fără suprapuneri, al magistralei locale din cele două direcţii: dinspre microprocesor şi dinspre dispozitivele solicitante.

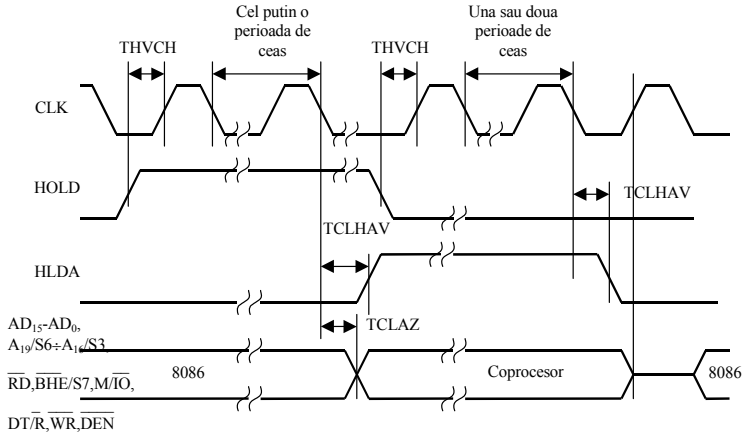


Figura 1.16. Secvenţa de preluare a controlului magistralei locale în modul minim

Răspunsul microprocesorului la cererea de preluare a magistralei, HOLD, depinde aşa cum am spus mai sus de activitatea curentă de pe magistrală, starea semnalului $\overline{LOCK}$ şi a întreruperilor. Dacă cererea HOLD apare când $\overline{LOCK}$ nu este activ, microprocesorul nu execută un ciclu de achitare a întreruperii şi UIM este la sfârşitul unui ciclu de magistrală sau într-o stare TI achitarea cu HLDA va avea întârzierea minimă dată de $THVCH_{min}+ TCHCL_{min}+TCLCL+TCLHAV_{min}= 35ns+65ns+200ns+10ns= 310ns$. În aceleaşi condiţii întârzierea maximă este dată de $(THVCH_{min}-1ns)+ TCLCL+ TCHCL_{max}+TCLCL+ TCLHAV_{max}= 34ns+200ns+82ns+200ns+160ns= 676ns$ şi apare atunci când se pierde o perioadă de ceas la eşantionarea lui HOLD

datorită nerespectării cu puţin, în relaţia de mai sus cu 1ns, a timpului de stabilizare minim. Dacă cererea HOLD apare când UIM trebuie să înceapă un ciclu de magistrală întârzierea maximă a lui HLDA, incluzând pierderea unei perioade la eşantionare, ca mai înainte, şi o perioadă de stabilizare internă pentru HLDA, este dată de ($THVCH_{min}$−1ns) + TCLCL+ $TCHCL_{max}$+ TCLCL+ 4·TCLCL+ TCLCL + $TCLHAV_{max}$= 34ns+7·200ns+82ns+160ns=1,676μs. La această întârziere se mai pot adăuga stări TW. De asemenea, dacă transferul pe care îl are de executat UIM este pe cuvânt la o adresă impară, vor trebui efectuaţi doi cicli consecutivi de magistrală înainte ca procesorul să achite cererea HOLD ceea ce conduce la o întârziere suplimentară de încă patru stări T. Se obţine în această situaţie, neţinând cont de eventuale stări de aşteptare o întârziere de 2,476μs.

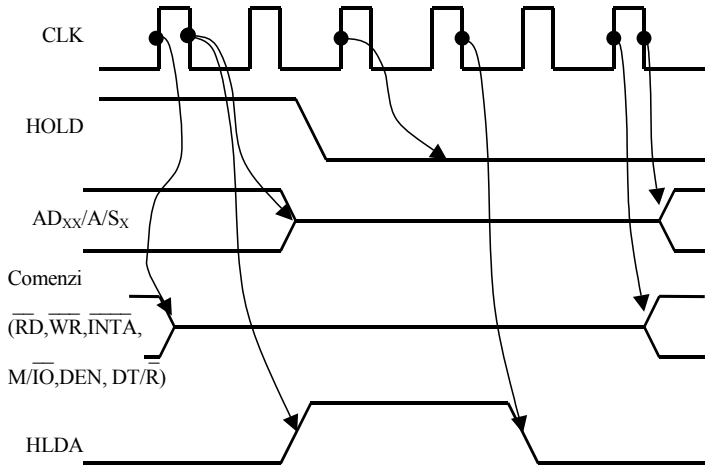


Figura 1.17. Eliberarea şi preluarea magistralei locale în modul minim

Cu toate că în modul minim 8086 nu utilizează ieşirea $\overline{LOCK}$, prefixul LOCK întrebuinţat la programare înaintea unei instrucţiuni nu va permite achitarea unei eventuale cereri HOLD decât după terminarea instrucţiunii respective. În acest mod se poate asigura execuţia fără întrerupere a oricăror instrucţiuni care necesită mai mulţi cicli de magistrală. Ţinând cont de faptul că semnalul intern LOCK este activ încă o stare T după terminarea instrucţiunii "protejate" întârzierea maximă a activării lui HLDA va fi dată de ($THVCH_{min}$−1ns)+TCLCL+$TCHCL_{max}$+(M+1)·TCLCL+TCLCL+$TCLHAV_{max}$, unde M este numărul de stări T necesar execuţiei instrucţiunii care urmează

prefixului LOCK iar adăugarea ultimei perioade TCLCL este impusă de necesitatea respectării unui timp de stabilizare intern pentru HLDA. Rezultă, la o frecvenţă a ceasului de 5MHz, o întârziere de M·200ns +876ns.

Pentru situaţiile în care cererea de preluare a magistralei se face la începutul unei secvenţe de achitare a întreruperii întârzierea maximă până la activarea semnalului de achitare HLDA va fi dată de ($THVCH_{min}$−1ns) + TCLCL+ $TCHCL_{max}$+ TCLCL+ 10·TCLCL+ TCLCL+ $TCLHAV_{max}$ = 2,876μs. Ultima perioadă T reprezintă, ca şi în cazul prefixului LOCK, un timp de stabilizare intern.

1.3.3. FUNCŢIONAREA MICROPROCESORULUI ÎN MODUL MAXIM

Prin legarea intrării MN / $\overline{MX}$ la GND se obţine funcţionarea microprocesorului în modul maxim. În acest mod de lucru, pentru controlul sistemului, se utilizează circuitul specializat de comandă a magistralei 8288, obţinându-se structuri tipice cum este şi cea din figura 1.18.

Modul de lucru maxim a fost conceput pentru a asigura microprocesorului posibilităţi de funcţionare în sisteme multi-microprocesor sau cu procesoare de extensie a setului de instrucţiuni, aşa numitele *coprocesoare*. Prin adăugarea controlorului 8288 conexiunile externe, destinate în modul minim comenzilor şi controlului magistralei, sunt redefinite, §1.3.2.3, oferindu-se următoarele facilităţi: două canale, $\overline{RQ}$ / $\overline{GT0}$ şi $\overline{RQ}$ / $\overline{GT1}$, pentru preluarea magistralei locale de către procesoare suplimentare aflate pe această magistrală şi care pot utiliza circuitele de interfaţă ale lui 8086 pentru a avea acces la magistrala sistemului; semnalele QS0 şi QS1 indicând starea cozii de aşteptare utilizabile de către module specializate sau de coprocesoare pentru a "urmări" execuţia instrucţiunilor; mecanism de blocare hardware a magistralei cu ajutorul semnalului $\overline{LOCK}$ permiţând controlul accesului la resursele divizate în sisteme multiprocesor prin prevenirea pierderii voluntare sau forţate a magistralei de către unitatea centrală, activă la un moment dat, şi garantarea execuţiei mai multor cicli de magistrală fără intervenţia unei alte unităţi centrale; alte posibilităţi de extindere a comenzilor şi configuraţiilor de sistem cu ajutorul unor circuite suplimentare, cum este circuitul 8289 pentru arbitrarea magistralei de sistem. Analizele de timp descrise în continuare vor ţine cont atât de semnalele generate de 8086 cât şi de cele generate de controlorul de magistrală. 8288 generează şi semnalele suplimentare de control şi comandă care permit alcătuirea unor sisteme complexe.

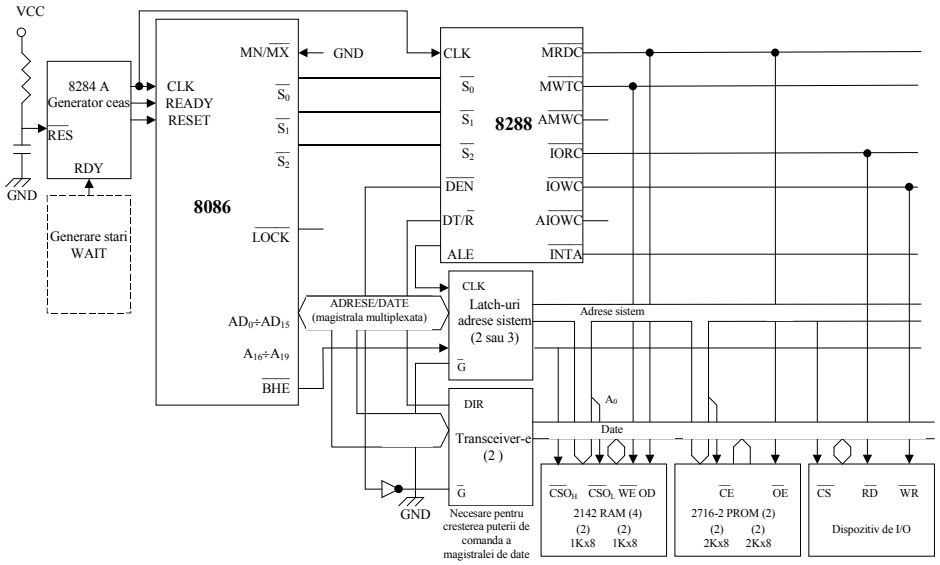


Figura 1.18. Microprocesorul 8086 în modul maxim

În cele ce urmează descriem întâi funcţionarea circuitului 8288 după care vom interpreta diagramele de timp corespunzătoare operaţiilor externe: adresarea, ciclul de citire, ciclul de scriere, achitarea întreruperii, introducerea stărilor de aşteptare, preluarea magistralei. La sfârşitul capitolului §1.4.3 vom atrage atenţia asupra unor particularităţi ale funcţionării în modul maxim.

1.3.3.1. **Controlorul de magistrală 8288**

Schema-bloc a circuitului şi conexiunile sale externe sunt prezentate în figurile 1.19 şi 1.20. În continuare vom descrie conexiunile externe şi, succint, particularităţile de funcţionare ale circuitului.

$\overline{S2}, \overline{S1}, \overline{S0}$, *Status*, biţi de stare. Intrări corespunzătoare biţilor de stare ai microprocesoarelor 8086 sau 8088. 8288 decodifică, aşa cum se poate vedea şi în figura 1.19, aceste intrări pentru a genera semnalele de comandă. Atunci când nu sunt utilizate, microprocesorul aflându-se în starea pasivă, aceste intrări sunt în "1" datorită unor rezistenţe interne legate la +5V.

CLK, *Clock*, ceas. Semnal de ceas de la generatorul de ceas 8284 necesar pentru stabilirea momentelor în care se generează comenzile.

ALE, *Address Latch Enable*, validare *latch*-are adrese. Ieşire activă pe "1" destinată memorării adreselor în *latch*-urile de adrese ale sistemului.

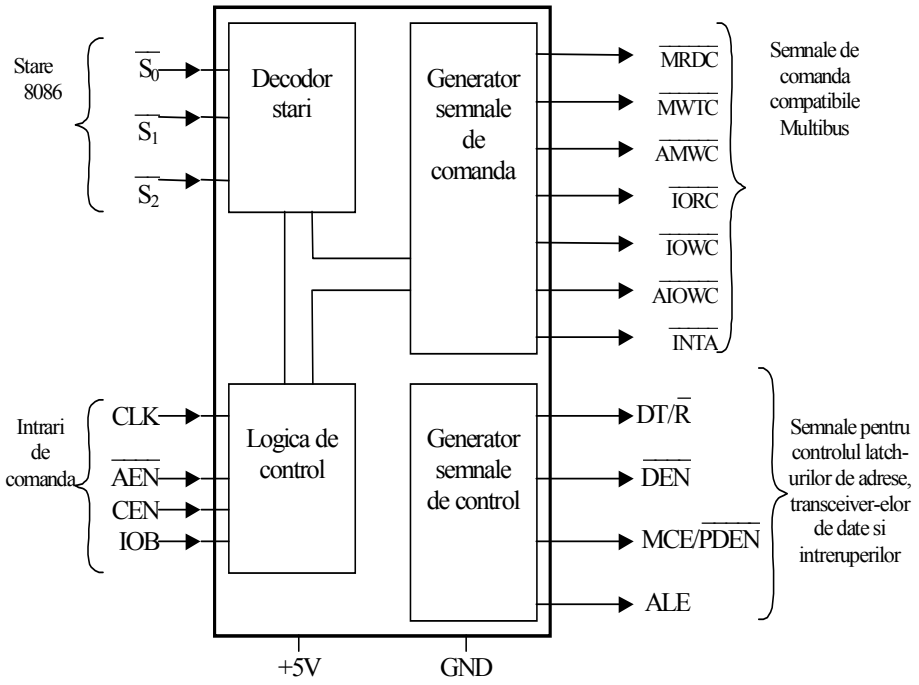


Figura 1.19. Schema bloc a controlorului de magistrală 8288

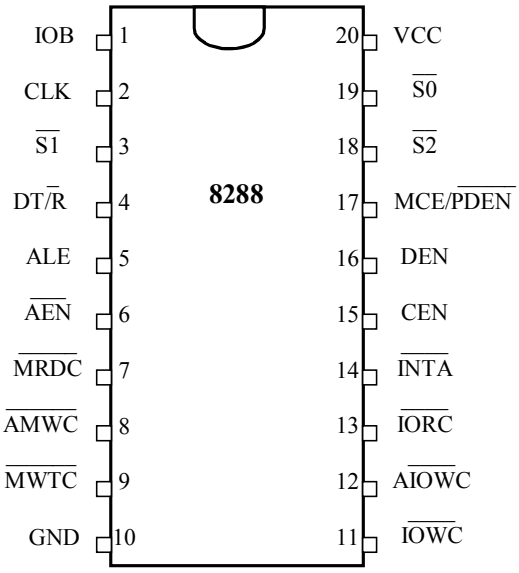


Figura 1.20. Conexiunile externe ale controlorului de magistrală 8288

DEN, *Data Enable*, validare date. Ieşire activă pe "1" utilizată pentru validarea *transceiver*-elor de date pe magistrala locală sau pe magistrala sistemului.

DT/$\overline{R}$, *Data Transmit/Receive*, transmisie/recepţie date. Ieşire folosită pentru precizarea direcţiei *transceiver*-elor de date, "1" pentru transmisie, scriere în dispozitivele de I/O sau memorie, "0" pentru recepţie, citire din dispozitivele de I/O sau memorie.

$\overline{AEN}$, *Address Enable*, validare adrese. Intrare activă pe "0" utilizată pentru validarea semnalelor de comandă ale controlorului de magistrală. $\overline{AEN}$ nu va afecta liniile de comandă ale circuitului dacă intrarea IOB a lui este conectată la "1", 8288 lucrând în această situaţie în modul *magistrală de I/O*.

CEN, *Command Enable*, validare comenzi. Intrare activă pe "1" utilizată pentru validarea semnalelor de comandă ale controlorului precum şi a ieşirilor DEN şi $\overline{PDEN}$. Când CEN=0 semnalele de comandă şi DEN şi $\overline{PDEN}$ sunt forţate în starea inactivă. Intrarea poate fi folosită pentru a implementa partiţionări de memorie şi a elimina conflicte de adresare între dispozitivele plasate pe magistrala de sistem şi cele de pe magistralele locale.

IOB, *Input/Output Bus Mode*, modul *magistrală de I/O*. Intrare cu ajutorul căreia este selectat modul de lucru al controlorului: "0" pentru modul *magistrală sistem*, "1" pentru modul *magistrală de I/O*.

$\overline{AIOWC}$, *Advanced Input/Output Write Command*, comandă de scriere I/O avansată. Ieşire activă pe "0" reprezentând o comandă de scriere avansată, anticipată, pentru dispozitivele de I/O care au nevoie mai devreme de un astfel de semnal pentru decodificări de adrese sau validări. Utilizarea acestui semnal avansat elimină necesitatea introducerii unei stări de aşteptare pentru unele dispozitive de I/O mai lente. Semnalul are aceeşi cronogramă cu a semnalelor de citire.

$\overline{IOWC}$, *Input/Output Write Command*, comandă de scriere I/O. Ieşire activă pe "0" reprezentând comanda normală de scriere pentru dispozitivele de I/O.

$\overline{IORC}$, *Input/Output Read Command*, comandă de citire I/O. Ieşire activă pe "0" reprezentând comanda de citire pentru dispozitivele de I/O.

$\overline{AMWC}$, *Advanced Memory Write Command*, comandă de scriere memorie avansată. Ieşire activă pe "0" utilizată pentru generarea unei comenzi de scriere anticipate solicitate de unele memorii mai lente pentru decodificări sau validări. Ca şi $\overline{AIOWC}$ are aceeaşi cronogramă cu cea a semnalelor de citire.

$\overline{MWTC}$, *Memory Write Command*, comandă de scriere memorie. Ieşire activă pe "0" reprezentând comanda normală de scriere la memorie.

$\overline{MRDC}$, *Memory Read Command*, comandă de citire memorie. Ieşire activă pe "0" reprezentând comanda de citire din memorie.

$\overline{\text{INTA}}$, *Interrupt Aknowledge*, achitare întreruperi. Ieşire activă pe "0" utilizată pentru a comunica dispozitivului care întrerupe achitarea întreruperii şi pentru a comanda plasarea de către acesta din urmă, pe magistrala de date, a vectorului de întrerupere.

MCE / $\overline{\text{PDEN}}$, *Master Cascade Enable/Peripheral Data Enable*. Este o ieşire având două funcţii, MCE şi $\overline{\text{PDEN}}$, în funcţie de tensiunea aplicată intrării de comandă IOB, "0" respectiv "1". MCE este o ieşire activă pe "1" care se utilizează în timpul unei secvenţe de achitare a întreruperii pentru citirea dintr-un controlor de întreruperi *master* a adresei de cascadă corespunzătoare controlorului de întreruperi *slave*, conectat în cascadă, care a solicitat efectiv întreruperea. $\overline{\text{PDEN}}$ este o ieşire activă pe "0" realizând, analog cu ieşirea DEN, funcţia de validare a *transceiver*-elor de date pentru dispozitivele de I/O.

În figurile 1.21, 1.22 şi 1.23 se prezintă diagramele care ilustrează relaţiile de timp între semnalele utilizate şi generate de circuitul 8288 iar în tabelele 1.4 şi 1.5 se dau valorile minime şi maxime ale parametrilor ce caracterizează aceste relaţii [11].

Aşa după cum se poate înţelege din figura 1.19 liniile de stare activate de microprocesor sunt decodificate de controlorul de magistrală 8288 pentru a emite comanda corespunzătoare conform tabelului 1.6. Comenzile se emit în două feluri, în funcţie de modul de lucru al circuitului selectat cu ajutorul conexiunii IOB.

Modul *magistrală de I/O*. În acest mod de lucru, IOB=1, toate comenzile de I/O, $\overline{\text{IORC}}$, $\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$, $\overline{\text{INTA}}$, sunt permanent validate, indiferent de starea semnalului $\overline{\text{AEN}}$. Atunci când microprocesorul iniţiază un ciclu de magistrală pentru o operaţie de I/O, 8288 va activa imediat liniile de comandă corespunzătoare împreună cu semnalele $\overline{\text{PDEN}}$ şi DT / $\overline{\text{R}}$ pentru controlul *transceiver*-elor de magistrală. În acest mod de lucru comenzile de I/O nu pot fi utilizate pe o magistrală unică de sistem, aceeaşi pentru memorie şi dispozitivele de I/O, datorită lipsei unui arbitraj. Acest mod de lucru se foloseşte atunci când în cadrul sistemului există două magistrale separate pentru memorie şi I/O. În aceste situaţii microprocesorul nu va aştepta nimic când accesează magistrala de I/O. Comenzile pentru memorie, $\overline{\text{MRDC}}$, $\overline{\text{MWTC}}$, $\overline{\text{AMWC}}$, rămân dependente de intrarea $\overline{\text{AEN}}$ fiind validate numai după ce această intrare devine "0" datorită unui semnal de tip *magistrală gata*. Modul de lucru *magistrală de I/O* este avantajos în sistemele multi-microprocesor pentru microprocesoarele care au toate dispozitivele de I/O dedicate.

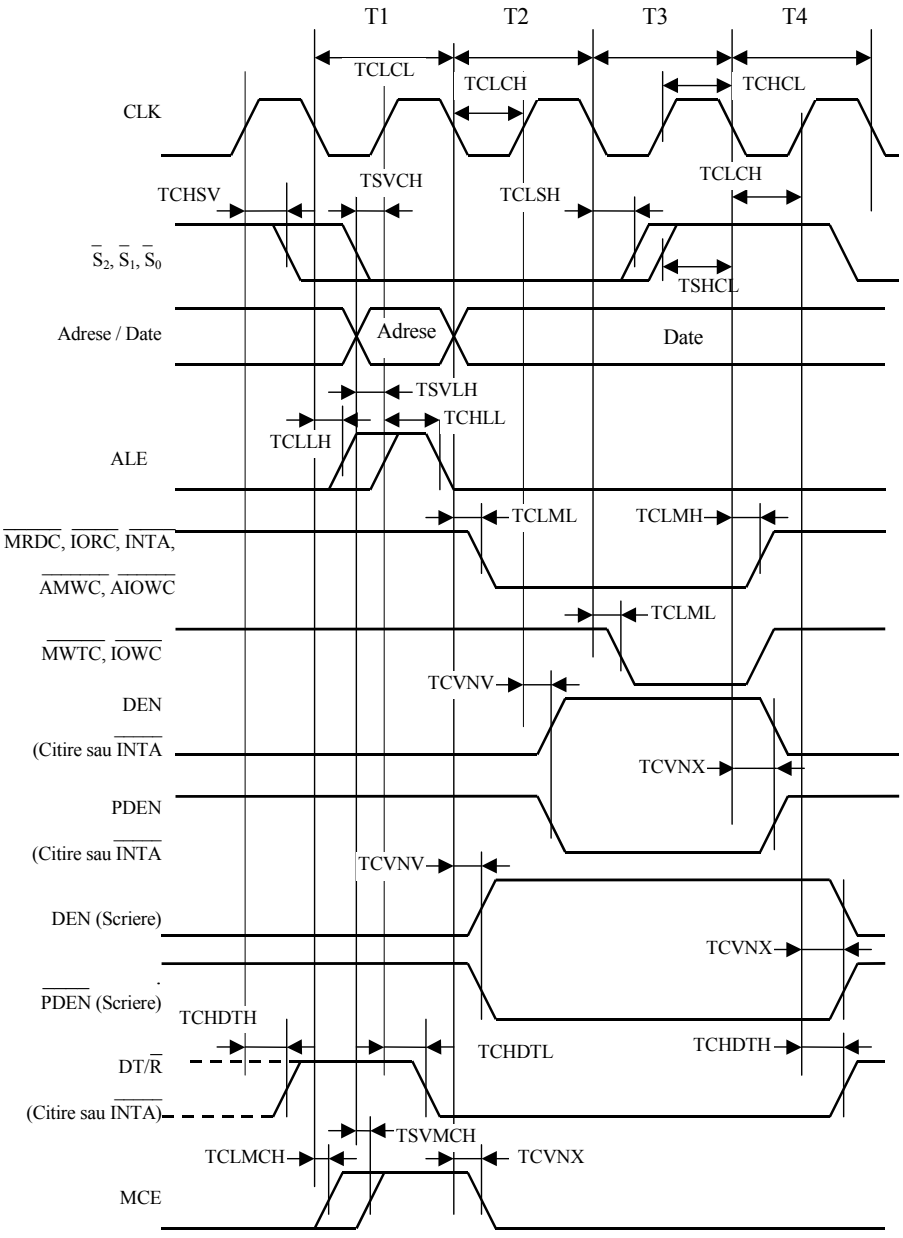


Figura 1.21. Diagramele de timp pentru circuitul 8288

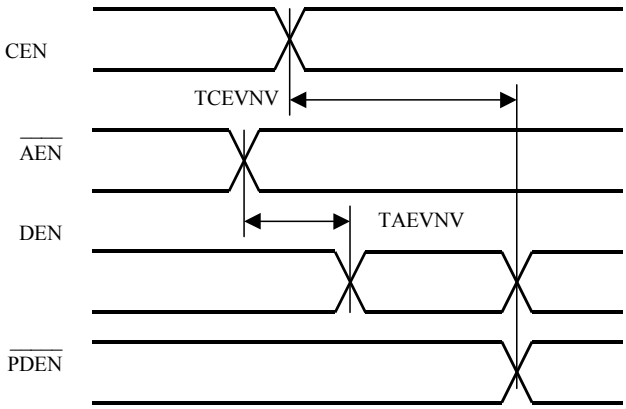


Figura 1.22. Condiţionarea semnalelor DEN şi $\overline{\text{PDEN}}$

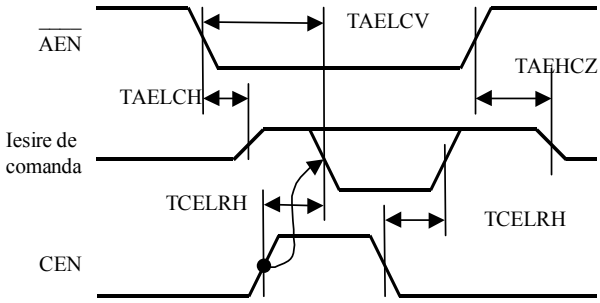


Figura 1.23. Întârzierile la validarea/invalidarea comenzilor

Tabelul 1.4. Parametrii 8288. Cerinţe de timp

Parametru	Semnificaţie	Valoare minimă
TCLCL	Perioada ceasului CLK	125ns
TCLCH	Timpul cât CLK=0	66ns
TCHCL	Timpul cât CLK=1	40ns
TSVCH	Timpul de stabilizare, de *set-up*, al semnalelor de stare active	65ns
TCHSV	Timpul de menţinere, de *hold*, al semnalelor de stare inactive	10ns
TSHCL	Timpul de stabilizare al semnalelor de stare inactive	55ns
TCLSH	Timpul de menţinere al semnalelor de stare active	10ns

Tabelul 1.5. Parametrii 8288. Răspunsuri în timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă
TCVNV	Întârzierea la activarea semnalelor de control	5ns	45ns
TCVNX	Întârzierea la inactivarea semnalelor de control	10ns	45ns
TCLLH	Întârzierea la activarea semnalului ALE în raport cu CLK		15ns
TCLMCH	Întârzierea la activarea semnalului MCE în raport cu CLK		15ns
TMHNL	Întârzierea între activarea semnalului DEN şi activarea semnalelor de comandă	TCLCH−5ns	
TSVLH	Întârzierea între activarea stărilor şi activarea semnalului ALE		15ns
TSVMCH	Întârzierea între activarea stărilor şi activarea semnalului MCE		15ns
TCHLL	Întârzierea la inactivarea semnalului ALE		15ns
TCLML	Întârzierea la activarea comenzilor	10ns	35ns
TCLMH	Întârzierea la inactivarea comenzilor	10ns	35ns
TCHDTL	Întârzierea la activarea semnalului DT / $\overline{R}$		50ns
TCHDTH	Întârzierea la inactivarea semnalului DT / $\overline{R}$		30ns
TAELCH	Întârzierea la validarea comenzilor		40ns
TAEHCZ	Întârzierea la invalidarea comenzilor		40ns
TAELCV	Întârzierea la activarea comenzilor	115ns	200ns
TAEVNV	Întârzierea între $\overline{AEN}$ şi DEN		20ns
TCEVNV	Întârzierea între CEN şi DEN sau $\overline{PDEN}$		20ns
TCELRH	Întârzierea între CEN şi comenzi		TCLML

Tabelul 1.6. Decodificarea stărilor şi comenzile emise de 8288

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Starea microprocesorului	Comanda emisă de 8288
0	0	0	Achitare întrerupere	$\overline{INTA}$
0	0	1	Citire *port* de I/O	$\overline{IORC}$
0	1	0	Scriere *port* de I/O	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Oprire (*Halt*)	–
1	0	0	Acces cod	$\overline{MRDC}$
1	0	1	Citire memorie	$\overline{MRDC}$
1	1	0	Scriere memorie	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Stare pasivă	–

Modul *magistrală de sistem*. În acest mod de lucru, IOB=0, toate comenzile sunt validate cu $\overline{AEN} = 0$. Modul de funcţionare se utilizează în sistemele realizate pe o singură magistrală, memoria cât şi dispozitivele de I/O fiind accesibile numai prin intermediul acestei magistrale. Folosirea circuitului 8288 în acest mod de lucru impune realizarea în cadrul sistemului a unei logici

de arbitraj care va informa controlorul, prin intermediul liniei $\overline{\text{AEN}}$, dacă magistrala este liberă pentru a fi accesată de microprocesor.

O facilitate suplimentară pentru utilizarea controlorului în sisteme multi-microprocesor este şi ieşirea MCE activă numai în modul *magistrală de sistem*. Aşa cum s-a arătat şi în §1.4.2.4 pe timpul unei secvenţe de achitare a întreruperii apar doi cicli de întrerupere succesivi. Pe timpul primului ciclu, necesar pentru efectuarea unor operaţii interne în microprocesor, nu are loc nici-un transfer de adrese sau date. De aceea, logica asociata controloarelor de întreruperi va trebui să mascheze semnalul MCE pe durata acestui prim ciclu $\overline{\text{INTA}}$. Imediat însă înaintea celui de-al doilea ciclu, semnalul MCE activat va putea, pentru situaţii în care se folosesc mai multe controloare de întreruperi legate în cascadă, să valideze o adresă de cascadă emisă de controlorul *master* pe magistrala locală a procesorului pentru a fi memorată cu ajutorul semnalului ALE în *latch*-uri de adrese. În acest fel, la sfârşitul celui de-al doilea ciclu $\overline{\text{INTA}}$, controlorul *slave* adresat cu adresa de cascadă va plasa pe magistrala de date a sistemului vectorul propriu de întrerupere. Vom reveni cu detalii referitoare la organizarea sistemului de întreruperi în §1.4.4.

1.3.3.2. **Adresarea**

În cele ce urmează ne vom referi atât la diagramele şi relaţiile de timp pentru microprocesorul 8086, figura 1.24 şi tabelele 1.7 şi 1.8, cât şi la diagramele şi relaţiile de timp ale controlorului de magistrală 8288, date în figurile 1.21, 1.22, 1.23 şi în tabelele 1.4 şi 1.5.

În modul de lucru maxim adresele sunt generate de microprocesor, ca şi în modul minim, în timp ce semnalul de strobare ALE este emis de 8288. Pentru aprecierea situaţiei celei mai defavorabile observăm că activarea semnalului ALE are două căi de întârziere: faţă de stări şi faţă de ceas. În raport cu stările $\overline{\text{S2}}$, $\overline{\text{S1}}$, $\overline{\text{S0}}$ apare o întârziere, măsurată de la frontul pozitiv anterior stării T1, dată de suma TCHSV+TSVLH. A doua cale dă întârzierea TCLLH măsurată de la frontul negativ al ceasului CLK care marchează începutul stării T1. De aici rezultă că, în situaţia când microprocesorul este mai lent pe calea de activare a stărilor şi alternanţa pozitivă a ceasului este minimă, ALE va fi generat cu o întârziere TSVLH după schimbarea stărilor. Pe de altă parte, dacă stările $\overline{\text{S2}}$, $\overline{\text{S1}}$, $\overline{\text{S0}}$ se schimbă înainte de începerea stării T1 8288 garantează activarea semnalului ALE cel mult cu o întârziere TCLLH faţă de începutul stării T1. Pentru prima situaţie, cea mai defavorabilă comparativ cu TCLLH_{max}=15ns, se obţine, faţă de începutul stării T1, o întârziere maximă de TCHSV_{max} + TSVLH_{max}–TCHCL_{min}=110ns+15ns–69ns=56ns.

Tabelul 1.7. Parametrii 8086 pentru modul maxim. Cerinţe de timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă	Observaţii
TCLCL	Perioada ceasului CLK	200ns	500ns	
TCLCH	Timpul cât CLK=0	118ns		
TCHCL	Timpul cât CLK=1	69ns		
TCH1CH2	Frontul crescător al ceasului CLK		10ns	Măsurat între valorile 1,0V şi 3,5V
TCL2CL1	Frontul descrescător al ceasului		10ns	Măsurat între valorile 3,5V şi 1,0V
TDVCL	Timpul de stabilizare, de *set-up*, al datelor la citire	30ns		
TCLDX	Timpul de menţinere, de *hold*, al datelor la citire	10ns		
TR1VCL	Timpul de stabilizare al semnalului RDY la intrarea în 8284A măsurat faţă de frontul descrescător al ceasului CLK	35ns		Acest timp este necesar semnalului asincron RDY pentru a garanta recunoaşterea lui în următoarea perioadă a ceasului CLK
TCLR1X	Timpul de menţinere al semnalului RDY la intrarea în 8284A	0ns		
TRYHCH	Timpul de stabilizare al semnalului READY la intrarea în 8086	118ns		
TCHRYX	Timpul de menţinere al semnalului READY la intrarea în 8086	30ns		
TRYLCL	Timpul necesar ca intrarea READY să fie considerată inactivă faţă de începutul stării T3	–8ns		READY mai poate deveni inactiv maximum 8ns în T3 transformând această stare într-o stare de aşteptare TW
TINVCH	Timpul de stabilizare pentru recunoaşterea semnalelor INTR, NMI, $\overline{\text{TEST}}$	30ns		
TGVCH	Timpul de stabilizare pentru semnalele $\overline{\text{RQ}}$ / $\overline{\text{GT}}$	30ns		
TCHGX	Timpul de menţinere al semnalului RQ la intrarea în 8086	40ns		
TILIH	Frontul crescător al semnalelor la intrările în 8086		20ns	Măsurat între 0,8V şi 2V
TIHIL	Frontul descrescător al semnalelor la intrările în 8086		12ns	Măsurat între 2V şi 0,8V

Tabelul 1.8. Parametrii 8086 pentru modul maxim. Răspunsuri în timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă	Observaţii
TCLML	Întârzierea la activarea comenzilor	10ns	35ns	Comenzile generate de 8288
TCLMH	Întârzierea la inactivarea comenzilor	10ns	35ns	
TRYHSH	Întârzierea între activarea lui READY şi trecerea stărilor $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ în starea pasivă		110ns	
TCHSV	Întârzierea la activarea stărilor $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ faţă de CLK	10ns	110ns	
TCLSH	Întârzierea la inactivarea stărilor $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ faţă de CLK	10ns	130ns	
TCLAV	Întârzierea la validarea adresei vs. CLK	10ns	110ns	
TCLAX	Timpul de menţinere al adresei	10ns		
TCLAZ	Întârzierea până la trecerea adreselor în starea a 3-a	TCLAX	80ns	
TSVLH	Întârzierea între validarea stării şi punerea pe "1" a lui ALE		15ns	ALE şi MCE generate de 8288
TSVMCH	Întârzierea între validarea stării şi punerea pe "1" a lui MCE		15ns	
TCLLH	Întârzierea între frontul descrescător al ceasului CLK şi validarea lui ALE		15ns	
TCLMCH	Întârzierea între frontul descrescător al ceasului CLK şi validarea lui MCE		15ns	
TCHLL	Întârzierea la inactivarea lui ALE		15ns	
TCLMCL	Întârzierea la inactivarea lui MCE		15ns	
TCLDV	Întârzierea validării datelor faţă de CLK	10ns	110ns	
TCHDX	Timpul de menţinere al datelor vs. CLK	10ns		
TCVNV	Întârzierea faţă de CLK la activarea semnalului de control DEN	5ns	45ns	DEN generat de 8288
TCVNX	Întârzierea la inactivarea semnalelor de control	10ns	45ns	
TAZRL	Întârzierea între trecerea în starea a 3-a a magistralei şi activarea lui $\overline{RD}$	0ns		
TCLRL	Întârzierea faţă de CLK a activării lui $\overline{RD}$	10ns	165ns	
TCLRH	Întârzierea faţă de CLK a inactivării lui $\overline{RD}$	10ns	150ns	
TRHAV	Întârzierea între inactivarea lui $\overline{RD}$ şi activarea adresei următoare	TCLCL –45ns		
TCHDTL	Întârzierea faţă de CLK la activarea semnalului $DT/\overline{R}$		50ns	$DT/\overline{R}$ generat de 8288
TCHDTH	Întârzierea la inactivarea semnalului $DT/\overline{R}$		30ns	
TCLGL	Întârzierea la activarea lui GT	0ns	85ns	
TCLGH	Întârzierea la inactivarea lui GT	0ns	85ns	
TRLRH	Lăţimea semnalului $\overline{RD}$	2TCLCL–75ns		
T0L0H	Frontul crescător al semnalelor generate		20ns	0,8V ↑ 2V
T0H0L	Frontul descrescător al semnalelor generate		12ns	2V ↓ 0,8V

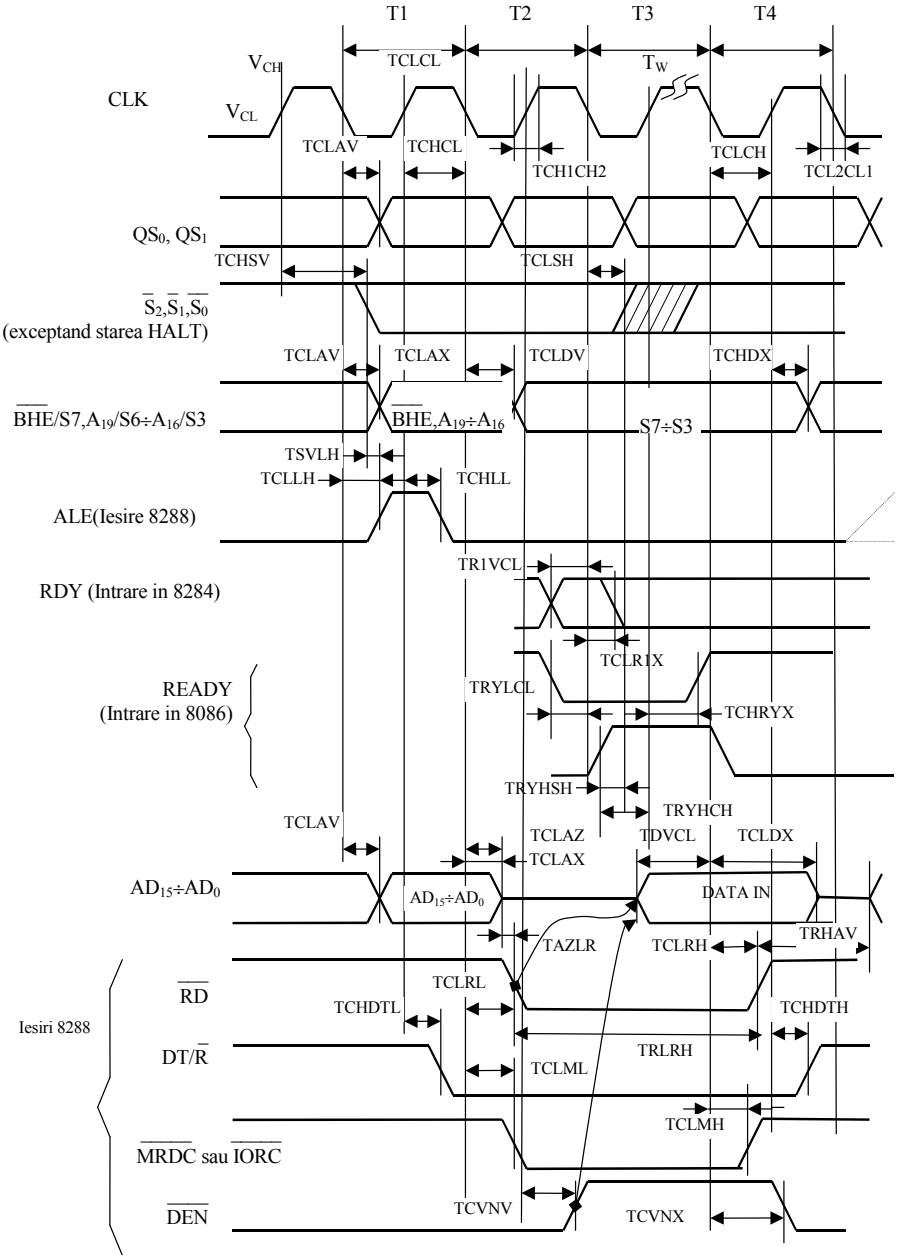


Figura 1.24. Diagrame de timp pentru modul maxim

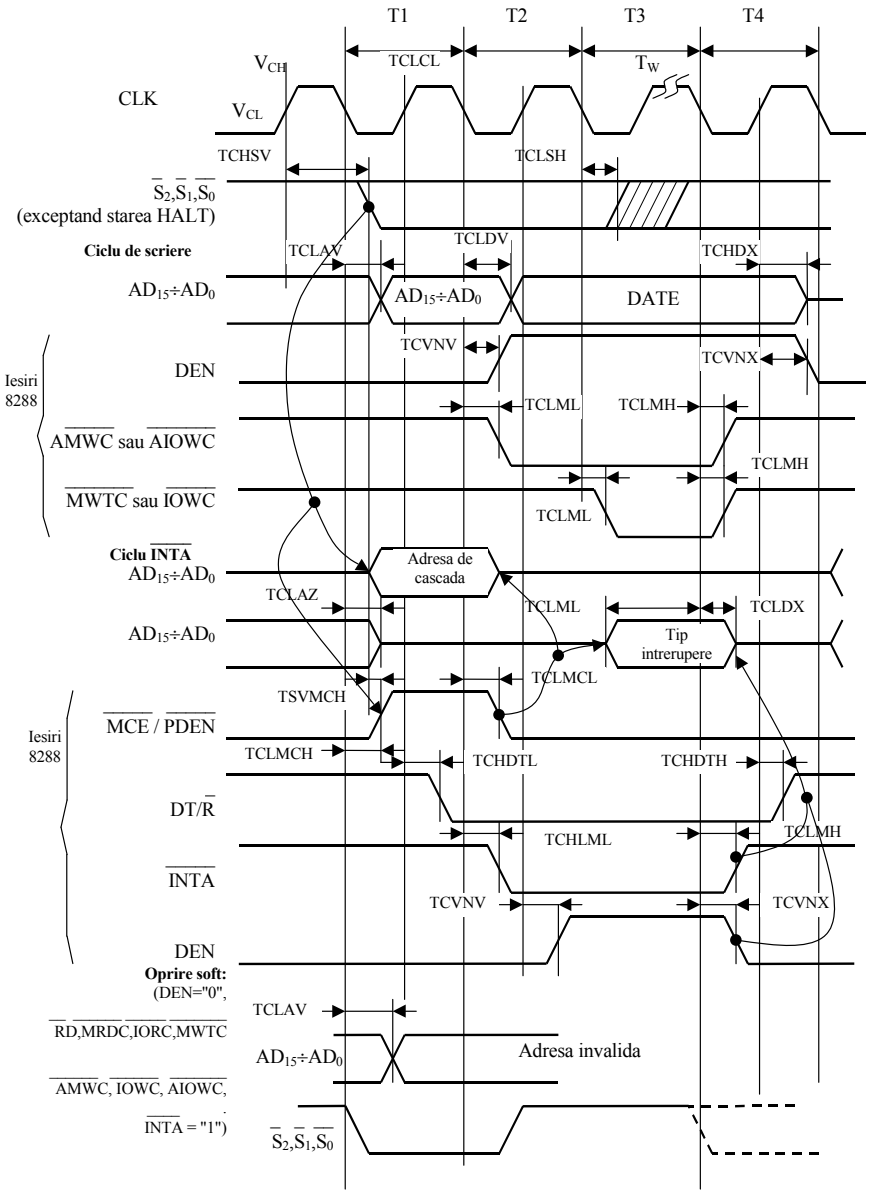


Figura 1.24. *Continuare*

Dezactivarea semnalului ALE se face cu frontul pozitiv din T1 indiferent de întârzierea la activare. De aici rezultă, la o frecvenţă a ceasului de 5MHz, o durată minimă pentru ALE de $TCLCH_{max}$+ $TCHLL_{min}$− 56ns= 131ns+0ns−56ns= 75ns.

Timpul de stabilizare minim al adreselor faţă de frontul negativ al lui ALE, necesar pentru asigurarea capturării adreselor în *latch*-uri, este dat de $TCLCH_{min}-TCLAV_{max}+TCHLL_{min}$. Se obţine la 5MHz, presupunând din nou aproape teoretic $TCHLL_{min}$= 0ns, un timp de *set-up* garantat de minimum 118ns−110ns+0ns=8ns ceea ce permite utilizarea unor *latch*-uri ca 8282, 8283 sau 74LS373. Timpul de menţinere al adreselor faţă de frontul căzător al semnalului ALE se poate obţine din relaţia TCHCL−TCHLL care, în cazul cel mai defavorabil şi pentru o frecvenţă a ceasului de 5MHz, conduce la valoarea minimă de $TCHCL_{min}-TCHLL_{max}$=69ns−15ns=54ns.

Timpii de comutare ai magistralei multiplexate din adrese în stări şi în starea a treia sau date funcţie de tipul ciclului, citire respectiv scriere, rămân aceeaşi ca în modul minim.

1.3.3.3. **Ciclul de citire**

În modul maxim se pot folosi comenzile de citire generate de 8288, dar şi comanda $\overline{RD}$ activată direct de microprocesor, ale cărei relaţii de timp sunt aceleaşi ca în modul minim. Deoarece parametrii dinamici ai semnalelor emise de controlorul de magistrală sunt mai buni decât cei ai semnalelor emise de microprocesor, cum se poate vedea din comparaţia tabelelor 1.3 şi 1.5, se recomandă folosirea acestor semnale pentru accesul la dispozitivele de memorie sau I/O plasate pe magistrala demultiplexată, de sistem, magistrală care datorită unor *buffer*-e suplimentare şi a unei încărcări capacitive mai mari introduce, de obicei, întârzieri suplimentare. Semnalul $\overline{RD}$ poate fi utilizat pentru citirea unor dispozitive apropiate, conectate direct pe magistrala locală multiplexată.

Comenzile de citire a dispozitivelor de memorie sau de I/O emise de 8288, $\overline{MRDC}$, $\overline{IORC}$ şi $\overline{INTA}$, au aceleaşi caracteristici dinamice: sunt activate cu o întârziere TCLML faţă de începutul stării T2 şi sunt dezactivate cu o întârziere TCLMH faţă de începutul stării T4. Durata minimă a unei astfel de comenzi va deci dată de $2\cdot TCLCL-TCLML_{max}+TCLMH_{min}$ şi va fi la 5MHz de 400ns−35ns+10ns=375ns. Timpul minim oferit pentru accesarea datelor de către microprocesor va fi $2\cdot TCLCL- TCLML_{max}- TDVCL_{max}$= 400ns− 35ns− 30ns = 335ns simţitor mai mare decât cel de 205ns pe care îl avem la dispoziţie în modul minim.

După cum se vede în figurile 1.21 şi 1.24 direcţia *transceiver*-elor se stabileşte în T1 iar validarea lor pe magistrala locală a microprocesorului se face în T2 asigurându-se o întârziere minimă de $TCLCH_{min}+ TCVNV_{min}$=118ns+5ns=123ns faţă de momentul comandării trecerii în starea a treia a magistralei locale. Timpul minim rămas la dispoziţie pentru propagarea

datelor spre procesor cu respectarea intervalului de stabilizare, de *set-up*, este dat de relaţia $TCHCL_{min}$ + TCLCL − $TCVNV_{max}$ − $TDVCL_{min}$= 69ns+ 200ns − 45ns−30ns= 194ns, mai mare decât timpul de 129ns asigurat în modul minim. Timpul de menţinere al datelor, $TCLDX_{min}$=10ns, este garantat din punctul de vedere al controlului *transceiver*-elor deoarece atât DEN cât şi $DT/\overline{R}$ sunt menţinute în T4 minimum $TCVNX_{min}$=10ns, respectiv $TCLCH_{min}$+ $TCHDTH_{min}$=118ns+0ns=118ns. Pe de altă parte, *transceiver*-ele vor fi invalidate înainte de următorul ciclu al microprocesorului cu minimum TCLCL−$TCVNX_{max}$=200ns−45ns=155ns.

1.3.3.4. **Ciclul de scriere**

Spre deosebire de modul de lucru minim, în modul maxim 8288 generează două tipuri de comenzi de scriere pentru memorie şi I/O: avansate, $\overline{AMWC}$ şi $\overline{AIOWC}$, şi normale, $\overline{MWTC}$, $\overline{IOWC}$. Comenzile avansate de scriere, generate cu o perioadă de ceas înaintea celor normale, au aceeaşi desfăşurare în timp ca şi comenzile de citire, durata lor fiind de 2·TCLCL−TCLML+TCLMH. La 5MHz această durată este de minimum 2·200ns−35ns+10ns=375ns. Pentru comenzile normale lăţimea impulsului de scriere, dată de TCLCL−TCLML+TCLMH, va fi, la 5MHz, de minimum 175ns. Întârzierea datelor, calculată faţă de începutul stării T2, depinde atât de microprocesor, parametrul TCLDV, cât şi de comanda DEN pentru validarea *transceiver*-elor, TCVNV. Se observă din tabelele 1.3 şi 1.5 că întârzierea cea mai mare se datorează microprocesorului: $TCLDV_{max}$=110ns faţă de $TCVNV_{max}$=45ns. Pentru comenzile avansate va fi deci posibil ca datele să fie valide la ieşirile microprocesorului după activarea comenzii de scriere de către 8288 cu o întârziere dată de diferenţa TCLDV−TCLML, maximum 100ns. Pentru comenzile normale datele vor fi întotdeauna valide înaintea activării acestor comenzi cu cel puţin TCLCL−$TCLDV_{max}$+$TCLML_{min}$= 200ns−110ns+ 10ns =100ns. Faţă de frontul pozitiv al ambelor tipuri de comenzi de scriere datele sunt stabile, la o frecvenţă a ceasului de 5MHz, cu minimum 2·TCLCL−$TCLDV_{max}$+$TCLMH_{min}$=2·200ns−110ns+10ns=300ns. Timpul de menţinere, de *hold*, al datelor după dezactivarea comenzilor de scriere este, ca şi timpul de *set-up*, dependent atât de microprocesor cât şi de comanda de invalidare a *transceiver*-elor emisă de 8288. Procesorul garantează stabilitatea datelor după frontul crescător al comenzilor de scriere minimum $TCLCH_{min}$−$TCLMH_{max}$+ $TCHDX_{min}$= 118ns− 35ns+ 10ns= 93ns. Invalidarea *transceiver*-elor cu DEN se face cu cel puţin $TCLCH_{min}$− $TCLMH_{max}$+ $TCVNX_{min}$=118ns−35ns +10ns=93ns. Astfel, din punct de vedere al

procesorului, 8086+8288, datele vor fi garantat stabile cel puţin 93ns după comanda de scriere.

1.3.3.5. **Achitarea întreruperii**

Secvenţa de achitare a întreruperii, generarea celor doi cicli $\overline{\text{INTA}}$, este logic identică cu cea din modul minim, diferenţele funcţionale fiind legate de posibilităţile oferite de 8288, prin generarea semnalului MCE, de a realiza sisteme cu mai multe controloare de întreruperi. Relaţiile de timp ale semnalelor DEN, DT / $\overline{\text{R}}$ şi $\overline{\text{INTA}}$ sunt aceleaşi ca în ciclul de citire, $\overline{\text{INTA}}$ fiind echivalent cu o comandă de citire, $\overline{\text{MRDC}}$ sau $\overline{\text{IORC}}$. Ca şi în modul minim, pe timpul celor doi cicli $\overline{\text{INTA}}$ magistrala locală multiplexată a microprocesorului va fi trecută în starea a treia la începutul stării T1 a fiecărui ciclu. Timpii de stabilizare şi menţinere pentru datele ce reprezintă tipul de întrerupere, plasate pe magistrala locală a procesorului, sunt aceiaşi ca pentru orice citire: TDVCL şi TCLDX.

Diferenţele de funcţionare în modul maxim sunt datorate posibilităţilor oferite de controlorul de magistrală în vederea implementării de sisteme cu mai multe circuite de control al întreruperilor folosind semnalele MCE şi ALE. După cum s-a menţionat în §1.4.3.1., în modul de lucru *magistrală de sistem* 8288 generează, pe timpul stării T1, în acelaşi timp cu semnalul de strobare ALE, semnalul MCE. MCE poate fi utilizat în vederea validării unei adrese de cascadă, de exemplu CAS2÷0 pentru 8259A, generate de controlorul de întreruperi principal, *master*, pe liniile cele mai semnificative, AD15÷AD13, ale magistralei locale. Această adresă poate fi capturată cu ALE în *latch*-urile de adresă ale sistemului pentru a selecta controlorul de întreruperi secundar, *slave*, care a generat efectiv întreruperea. Un exemplu de schemă care utilizează această facilitate este dat în figura 1.24A. MCE este generat în fiecare ciclu $\overline{\text{INTA}}$ dar el trebuie folosit numai în al doilea ciclu $\overline{\text{INTA}}$ deoarece la începutul primului ciclu microprocesorul nu garantează starea de impedanţă înaltă a magistralei locale: TCLAZ_{max}=80ns, în timp ce MCE poate fi activat cu o întârziere maximă de TSVMCH_{max}=15ns. De asemenea, 8259A, controlorul de întreruperi folosit în mod uzual în sistemele cu 8086 sau 8088, generează adresa de cascadă în al doilea ciclu $\overline{\text{INTA}}$. Selecţia celui de-al doilea semnal MCE se face cu ajutorul semnalului $\overline{\text{LOCK}}$. Reamintim că în modul minim acest semnal neutilizabil este generat intern, la ieşirea corespunzătoare a procesorului generându-se comanda de scriere $\overline{\text{WR}}$. În modul maxim $\overline{\text{LOCK}}$ va fi activat în T2 din primul ciclu $\overline{\text{INTA}}$ şi dezactivat în starea T2 a celui de-al doilea ciclu, figura 1.25. Întârzierea la schimbarea stării semnalului $\overline{\text{LOCK}}$, TCLAV, figura 1.26, este aceeaşi cu întârzierea de validare a adreselor,

cuprinsă între 10ns şi 110ns. Timpul de stabilizare minim al adresei de cascadă, faţă de frontul negativ al semnalului de *latch*-are ALE, este condiţionat atât de întârzierea apariţiei adresei la ieşirile controlorului de întreruperi cât şi de raportul între semnalele MCE şi ALE. Suprapunerea minimă între MCE şi ALE garantează, din al doilea punct de vedere, un timp minim de stabilizare de $TCLCH_{min}$+ $TCHLL_{min}$– $TCLMCH_{max}$= 118ns + 0ns – 15ns = 103ns. Timpul de menţinere al adresei de cascadă este asigurat, de asemenea, din punct de vedere al semnalului MCE, mai durând după ALE cel puţin $TCHCL_{min}$+ $TCVNX_{min}$– $TCHLL_{max}$= 69ns+10ns–15ns= 64ns.

În modul de lucru *magistrală de I/O*, 8288 nu mai generează semnalul MCE şi toate operaţiile de intrare/ieşire, inclusiv ciclii $\overline{\text{INTA}}$, se presupune că se referă la dispozitive aflate pe magistrala locală şi nu pe magistrala de sistem. În acest caz nu mai este necesară validarea unei adrese de cascadă pe magistrala de sistem, vezi şi figura 1.26A, prin capturare în *latch*-urile de adresă. În această situaţie toate întreruperile mascabile vin din zona locală a sistemului. În acest mod de lucru, în timpul ciclilor $\overline{\text{INTA}}$ se va genera semnalul de validare $\overline{\text{PDEN}}$ în locul semnalului DEN utilizabil în cazul când şi pe magistrala locală se folosesc *transceiver*-e.

1.3.3.6. **Introducerea stărilor de aşteptare**

Introducerea stărilor de aşteptare, TW, se face la fel ca în modul minim între stările T3 şi T4, prin implementarea uneia din cele două metode: sisteme normale *ready* sau sisteme normale *not ready*. Timpul minim oferit dispozitivelor de memorie sau de I/O pentru a dezactiva, trece pe "0", intrarea READY a microprocesorului, timp măsurat între momentul activării unei comenzi şi sfârşitul stării T2 (este vorba despre comenzile de citire inclusiv $\overline{\text{INTA}}$ şi cele avansate), este dat de TCLCL– $TCLML_{max}$– $TRYLCL_{min}$= 200ns– 35ns– (–8ns)= 173ns. După cum se vede în figura 1.24 comenzile normale sunt validate în T3 ele neasigurând generarea corectă a semnalului READY. Având în vedere însă că 8288 generează în acelaşi timp comenzi normale şi comenzi avansate, la generarea semnalului READY se pot utiliza întotdeauna numai comenzile avansate beneficiind de timpul calculat mai sus. În acest timp minim trebuie inclusă şi întârzierea datorată circuitului 8284 de sincronizare a semnalului READY, aspect asupra căruia vom reveni în §1.4.4.4. Timpul minim oferit dispozitivelor periferice pentru activarea, trecerea pe "1", a intrării READY a procesorului, măsurat între momentul activării comenzii şi frontul pozitiv al ceasului din T3, este TCLCL– $TCLML_{max}$+ $TCLCH_{min}$– $TRYHCH_{min}$= 200ns– 35ns+ 118ns–118ns= 165ns.

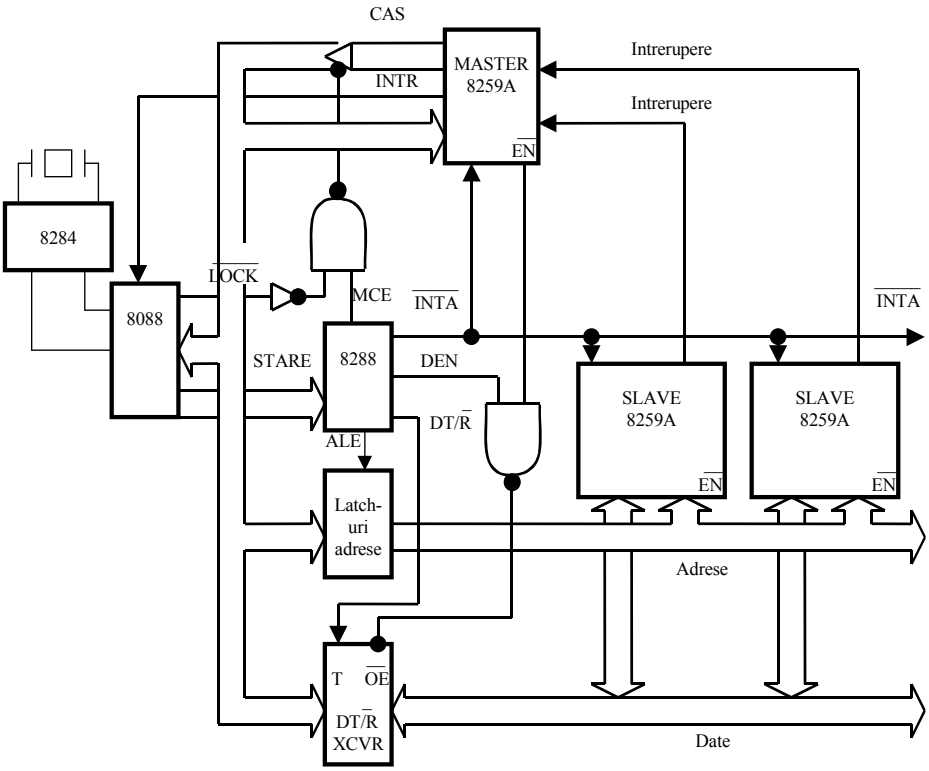


Figura 1.24A. 8086 în modul maxim cu un controlor de întreruperi *master* pe magistrala locală şi mai multe controloare de întreruperi *slave* pe magistrala de sistem

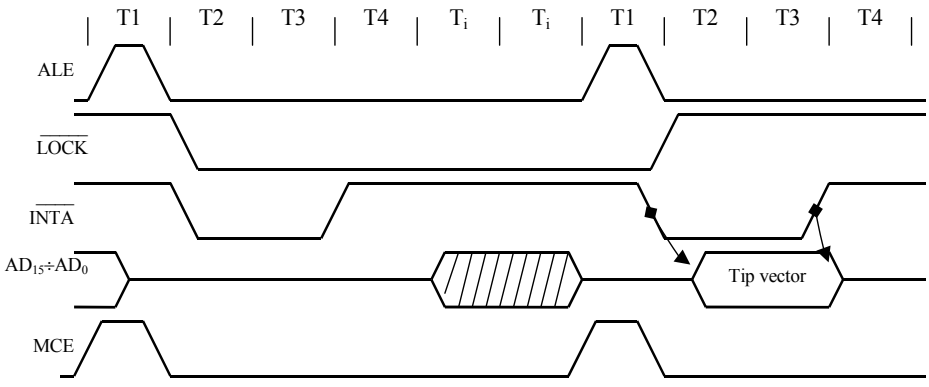


Figura 1.25. Secvenţa de achitare a întreruperii mascabile în modul maxim

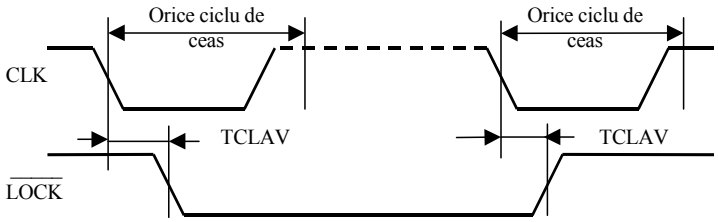


Figura 1.26. Întârzierile semnalului $\overline{LOCK}$

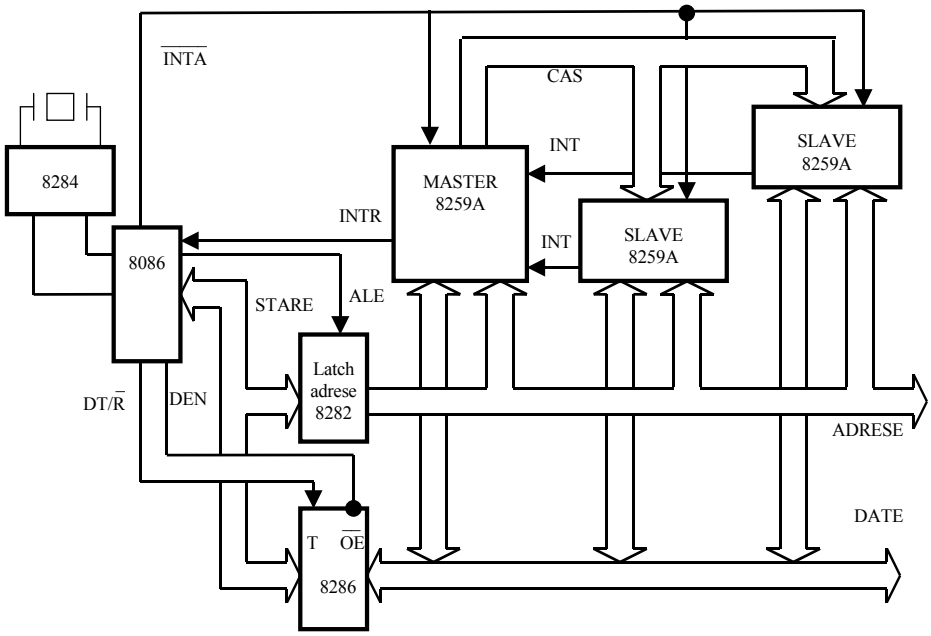


Figura 1.26A. 8086 în modul maxim cu un controlor de întreruperi *master* pe magistrala locală şi mai multe controloare de întreruperi *slave* pe magistrala de sistem

1.3.3.7. **Preluarea controlului magistralei locale (accesul direct)**

În modul de lucru maxim al microprocesorului preluarea controlului magistralei locale, spre deosebire de modul minim unde aceasta se realiza printr-un procedeu de cerere/achitare cu ajutorul semnalelor HOLD şi HLDA, se face prin intermediul unei secvenţe mai sofisticate implementate pe două niveluri de prioritate cu ajutorul unor conexiuni bidirecţionale $\overline{RQ}/\overline{GT0}$ şi $\overline{RQ}/\overline{GT1}$, figura 1.27.

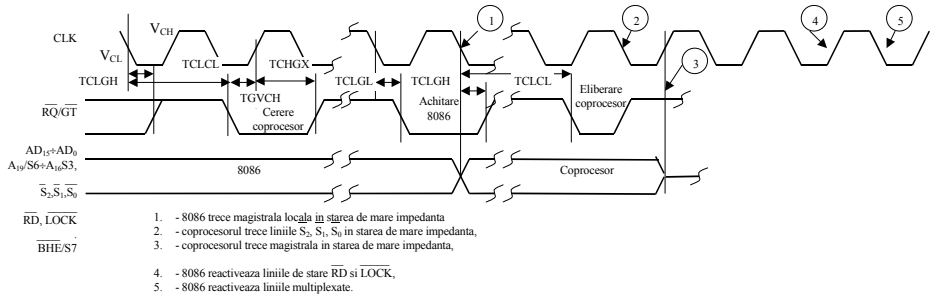


Figura 1.27. O secvenţă $\overline{RQ}$ / $\overline{GT}$

Ca şi secvenţa HOLD/HLDA, secvenţele $\overline{RQ}$ / $\overline{GT}$ sunt prevăzute pentru preluarea controlului magistralei *locale* a microprocesorului de către dispozitive de tip *master*, de exemplu coprocesoare sau circuite specializate pentru comanda accesului direct, care se găsesc pe această magistrală şi care folosesc în întregime circuitele de interfaţă ale microprocesorului cu magistrala de sistem. Secvenţele $\overline{RQ}$ / $\overline{GT}$ nu sunt destinate implementării funcţiei de arbitraj pe magistrala sistemului, în sisteme complexe, multi-*master*, cum ar fi, de exemplu, cele realizate pe interfaţa MULTIBUS. Pentru astfel de sisteme sunt folosite circuite specializate de arbitraj ca 8289.

Protocolul $\overline{RQ}$ / $\overline{GT}$ permite plasarea directă pe magistrala locală a cel mult două procesoare de extensie a setului de instrucţiuni, aşa numitele coprocesoare, sau a altor procesoare specializate, cum este şi procesorul de intrare/ieşire 8089. O secvenţă de preluare a magistralei implementate pe una din liniile bidirecţionale $\overline{RQ}$ / $\overline{GT}$ constă, figura 1.27, într-o *cerere* de la unul din procesoarele suplimentare, o *achitare* din partea procesorului 8086 indicând eliberarea magistralei locale şi o *eliberare* a magistralei la terminarea intervenţiei solicitate de procesorul suplimentar. Cele trei evenimente sunt materializate prin trecerea liniilor $\overline{RQ}$ / $\overline{GT}$ pe "0". Între cele două linii prioritară este $\overline{RQ}$ / $\overline{GT0}$. Aplicarea acestei priorităţi se face numai în cazurile în care apar cereri pe ambele linii înainte de emiterea de către procesor a unei achitări. Dacă procesorul a emis deja o achitare pentru canalul $\overline{RQ}$ / $\overline{GT1}$ o cerere prioritară apărută ulterior acestei achitări pe $\overline{RQ}$ / $\overline{GT0}$ va aştepta eliberarea magistralei de către coprocesorul care a emis cererea pe canalul $\overline{RQ}$ / $\overline{GT1}$. Pe timpul preluării magistralei 8086 se comportă la fel ca în modul minim: pe de-o parte unitatea de execuţie funcţionează intern până în momentul apariţiei necesităţii accesului la magistrală pentru date sau cod, iar pe de altă parte în situaţiile când achitarea apare înainte ca UE să aibă nevoie de magistrală microprocesorul va întârzia cu activarea magistralei până când va avea nevoie efectiv de ea.

Aşa cum se poate vedea în figura 1.27 după primirea şi achitarea unei cereri, 8086 va trece în starea a treia magistrala multiplexată, AD15÷AD0, $\overline{BHE}$ / S7, A19/S6÷A16/S3, liniile de stare $\overline{S2} \div \overline{S0}$, ieşirile $\overline{LOCK}$ şi $\overline{RD}$. Această acţiune nu va invalida ieşirile controlorului 8288 ci, datorită existenţei în 8288 a unor rezistenţe interne pentru semnalele $\overline{S2} \div \overline{S0}$, se va confunda cu apariţia unor stări pasive pe timpul cărora, aşa cum s-a arătat, 8288 nu va emite nici-o comandă sau semnal de validare a *transceiver*-elor. În această situaţie, 8288 poate fi controlat de coprocesorul căruia i s-a acordat magistrala. Dacă coprocesorul nu foloseşte controlorul 8288 el trebuie să-i invalideze ieşirile de comandă, trecându-le în starea a treia prin poziţionarea pe "1" a intrării $\overline{AEN}$ a lui 8288.

O cerere de preluare a magistralei locale pe una din liniile $\overline{RQ} / \overline{GT}$ se materializează printr-un impuls pe zero, figura 1.27, a cărui durată nu va fi mai mare decât perioada ceasului CLK dar care trebuie sincronizat cu ceasul microprocesorului pentru a fi asiguraţi timpii de stabilizare şi menţinere faţă de frontul pozitiv al ceasului cu care se eşantionează liniile $\overline{RQ} / \overline{GT}$. După generarea impulsului de cerere solicitantul, coprocesor sau procesor specializat, va trebui să urmărească starea liniei $\overline{RQ} / \overline{GT}$ pe care a emis cererea. Achitarea de către microprocesor a unei cereri se face de asemenea printr-un impuls pe zero emis de microprocesor pe linia bidirecţională pe care s-a făcut cererea. Precizăm că, deoarece microprocesorul poate emite achitarea cu frontul negativ al ceasului CLK imediat următor frontului pozitiv cu care a strobat cererea, linia $\overline{RQ} / \overline{GT}$ pe care se desfăşoară dialogul poate să rămână pe "0" datorită unei scurte suprapuneri a începutului achitării cu cererea care nu s-a terminat. Această menţinere pe "0" a liniei, semnificând totuşi un dialog, impune, în vederea capturării unei achitări, folosirea unei logici sincrone acţionate pe ceasul CLK. O logică bazată pe acţionări pe front nu va fi potrivită datorită tocmai posibilităţii apariţiei unor eventuale suprapuneri, aşa cum s-a explicat mai sus.

După recepţionarea achitării, dispozitivul solicitant poate să preia magistrala locală. Datorită faptului că microprocesorul va trece magistrala în starea a treia după o perioadă de ceas de la emiterea achitării se impune dispozitivului solicitant să respecte la activarea magistralei întârzierea TCLAZ faţă de frontul negativ al ceasului CLK, figura 1.27. Această precauţie este recomandată deoarece detectarea unei achitări de către dispozitivul solicitant se poate face chiar cu frontul pozitiv al ceasului cuprins în perioada de ceas în care microprocesorul emite achitarea, deci înainte de dezactivarea magistralei de către 8086. Aceasta ar putea conduce la apariţia prin suprapunerea a doi emiţători a unei zone de incertitudine a magistralei la preluarea ei de către solicitant.

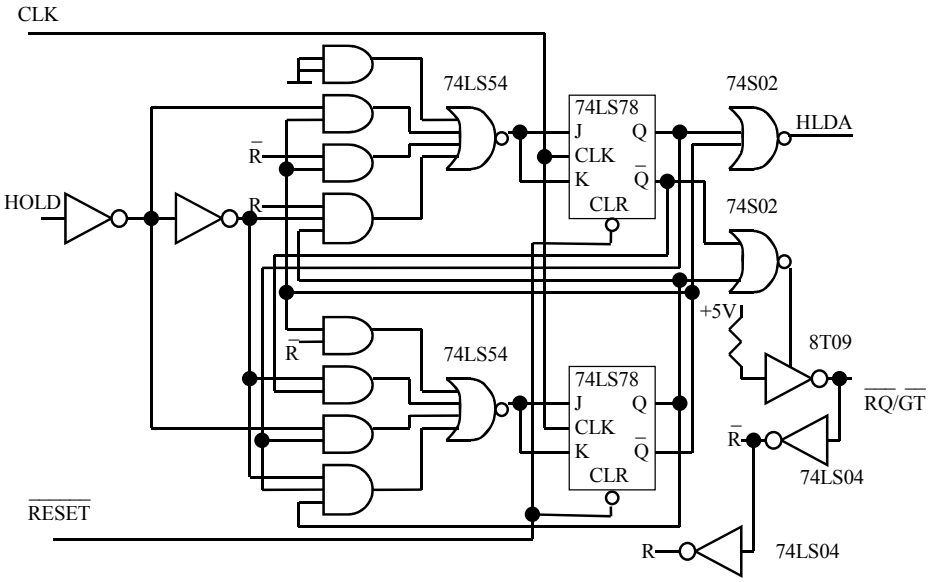


Figura 1.28. O schemă de conversie HOLD/HLDA în $\overline{RQ}$ / $\overline{GT}$

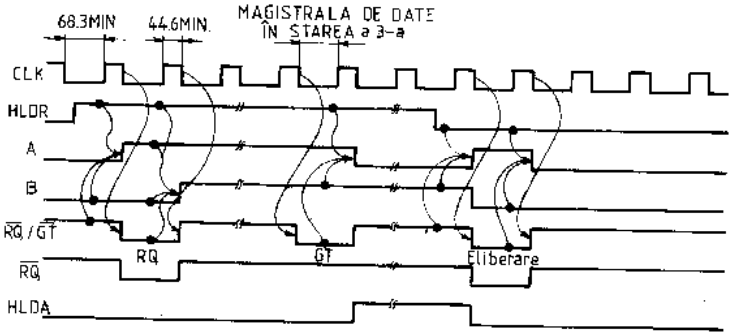


Figura 1.29. Diagramele de timp pentru schema din figura 1.28

Pentru a reda controlul magistralei procesorului dispozitivul solicitant va trebui să elibereze magistrala şi să emită pe linia $\overline{RQ}$ / $\overline{GT}$ un impuls de eliberare a magistralei. 8086 va activa liniile de stare $\overline{S2} \div \overline{S0}$ şi $\overline{LOCK}$ trei perioade de ceas după detectarea impulsului de eliberare. Liniile de adrese/date vor fi activate după cel puţin un interval $TCHCL_{min}$ faţă de liniile de stare. Între timp *latch*-urile de adresă şi 8288-ul asociate lui 8086 vor trebui şi ele activate pentru ca microprocesorul să preia efectiv controlul magistralei locale.

Pentru ilustrarea mai bună a protocolului de preluare a magistralei locale în modul maxim dăm în figurile 1.28 şi 1.29 o schemă de conversie a semnalelor HOLD/HLDA în $\overline{RQ}/\overline{GT}$ şi diagrama de timp corespunzătoare [10]. Această schemă poate fi utilizată pentru adaptarea de exemplu a unui circuit de control al accesului direct în sisteme cu 8086 lucrând în modul maxim.

Răspunsul microprocesorului la o cerere pe una din liniile $\overline{RQ}/\overline{GT}$ depinde, ca şi în modul minim, de activitatea curentă pe magistrală, de starea semnalului $\overline{LOCK}$ şi de întreruperi. În [10] se dă pentru calculul întârzierii între detecţia unei cereri de către microprocesor şi detecţia unei achitări de către solicitant următoarea formulă: (Întârzierea de la HOLD la HLDA)–(THVCH+TCHCL+TCLHAV). Aplicarea acestei formule conduce pentru situaţiile în care unitatea de interfaţă cu magistrala UIM se află într-o stare T4 sau TI la o întârziere de o perioadă de ceas. Spre deosebire de modul minim unde dialogul cu procesorul pentru preluarea magistralei se făcea numai pe un canal HOLD/HLDA, în modul maxim pot apărea întârzieri datorită dialogului pe cele două canale $\overline{RQ}/\overline{GT}$. Astfel, dacă 8086 a generat o achitare pe una din liniile $\overline{RQ}/\overline{GT}$, o posibilă cerere apărută pe cealaltă linie nu va primi achitarea decât după ce primul solicitant va elibera magistrala. Întârzierea între eliberarea unei linii $\overline{RQ}/\overline{GT}$ şi achitarea unei cereri în aşteptare emise pe cealaltă linie este în general de o perioadă de ceas, figura 1.30. Uneori, situaţiile în care pe timpul preluării magistralei pe $\overline{RQ}/\overline{GT1}$ apare o cerere pe canalul $\overline{RQ}/\overline{GT0}$ întârzierea între eliberarea canalului 1 şi achitarea canalului 0 poate fi de două perioade de ceas, în funcţie de existenţa unei cereri de transfer în aşteptare emise de unitatea de execuţie. Această cerere de transfer emisă de UE va fi amânată în favoarea rezolvării cererii de pe $\overline{RQ}/\overline{GT0}$, dar cu "preţul" decalării achitării pe canalul 0 cu încă o perioadă de ceas. Observăm deci că microprocesorul, fiind cel mai puţin prioritar ca solicitant de magistrală, sistemul poate fi blocat, *agăţat*, de unul dintre solicitanţii exteriori. Ieşirea dintr-o astfel de blocare se poate face cu ajutorul unei tehnici de tip *ceas de gardă, watchdog*.

1.3.3.8. **Alte particularităţi ale funcţionării în modul maxim**

Starea cozii de aşteptare dată de ieşirile QS1, QS0, indică ce tip de informaţii este extras din coada de aşteptare internă şi când aceasta din urmă este iniţializată datorită unui transfer al controlului. Monitorizând liniile de stare $\overline{S2}$, $\overline{S1}$ şi $\overline{S0}$ pentru instrucţiunile extrase de microprocesor, – $\overline{S2}, \overline{S1}, \overline{S0}$ =1,0,0 indică acces-cod –, A0 şi $\overline{BHE}$ care precizează dacă accesul se face pe octet sau pe cuvânt, şi liniile QS1 şi QS0 pentru instrucţiunile care

părăsesc coada de aşteptare, se poate "urmări" din exterior execuţia instrucţiunilor. Această tehnică este utilizată atât de coprocesoare pentru detectarea execuţiei unei instrucţiuni ESCAPE prin care li se comunică sarcini specifice cât şi, de exemplu, pentru depanare şi punere la punct, în cadrul unor module de logică specializate, de tip *analizor logic*, care pot să "prindă" o execuţie la o anumită adresă de memorie. Starea cozii de aşteptare este validă în ciclul de ceas următor efectuării operaţiei.

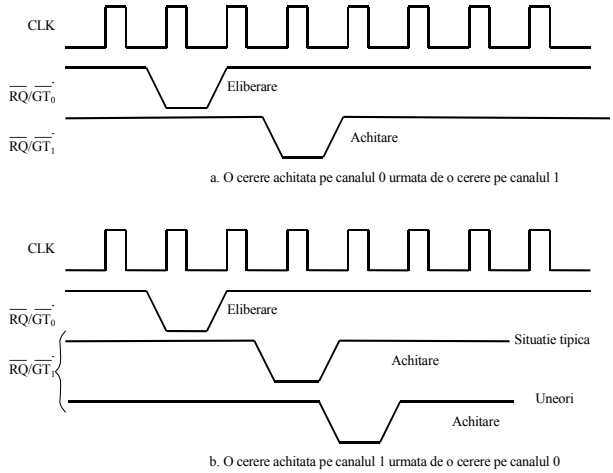


Figura 1.30. Întârzieri la achitarea cererii de preluare a magistralei pe liniile $\overline{RQ}$ / $\overline{GT}$

Controlul accesului la resursele divizate în sisteme complexe, multi-microprocesor, se poate face în modul de lucru maxim, aşa cum am mai spus, cu ajutorul ieşirii $\overline{LOCK}$. Această ieşire este activată la execuţia unei instrucţiuni precedate de prefixul LOCK, mai exact, în primul ciclu de ceas următor "execuţiei" prefixului. Ieşirea va rămâne activă încă un ciclu de ceas după executarea instrucţiunii precedate de prefixul LOCK, figura 1.31. În această figură se poate vedea, de exemplu, comportarea ieşirii $\overline{LOCK}$ la execuţia instrucţiunii XCHG cu prefix LOCK utilizate pentru implementarea unui mecanism uzual de tip *testare şi poziţionare semafor*:

```
lock xchg reg, memorie      ; reg este orice registru
                            ; MEMORIE este adresa
                            ; semaforului
```

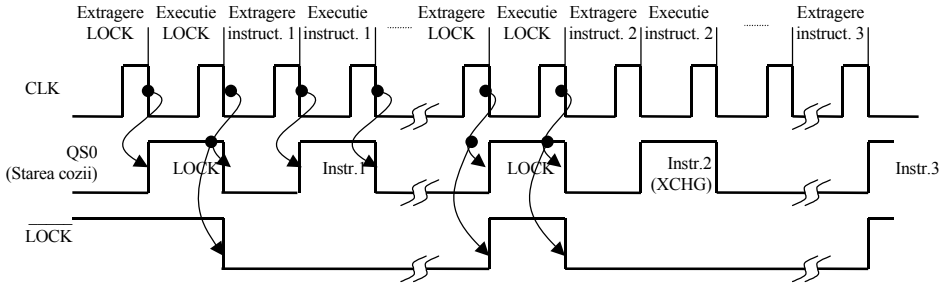


Figura 1.31. Activarea semnalului $\overline{\text{LOCK}}$

Aici prefixul LOCK previne accesul unei alte unităţi centrale la semafor, locaţia MEMORIE, pe timpul operaţiei de testare şi poziţionare a semaforului. O altă întrebuinţare eficientă a prefixului LOCK este în cazul transferurilor pe blocuri folosite pentru schimbarea rapidă de mesaje între unităţile centrale ale sistemelor multiprocesor. În figura 1.31 starea cozii, validată în timpul ciclului de ceas următor operaţiei efectuate asupra cozii, este reprezentată de QS0, poziţionat pe "1" la extragerea din coadă a primului octet al codului-operaţie. În figură este prezentată execuţia succesivă a trei instrucţiuni. Observăm întâi că sunt necesare două perioade de ceas pentru extragere şi decodare – execuţia prefixului LOCK –, echivalentă cu întârzierea până la activarea ieşirii $\overline{\text{LOCK}}$. Ieşirea $\overline{\text{LOCK}}$ va deveni aşadar activă odată cu începutul instrucţiunii pe care o precede prefixul. Ea va rămâne activă încă o perioadă după terminarea instrucţiunii, trecând pe "1" între execuţiile a două instrucţiuni succesive, chiar dacă ambele sunt precedate de prefix, cazul din figură. Dacă în coada de aşteptare se află numai prefixul, instrucţiunea pe care o precede nefiind încă extrasă din memorie, ieşirea $\overline{\text{LOCK}}$ va fi activată imediat după execuţia prefixului şi va rămâne aşa până după aducerea instrucţiunii în coadă şi executarea ei. Mai trebuie făcută şi observaţia că pe timpul execuţiei unei instrucţiuni "blocate", precedate de prefixul LOCK, UIM mai poate să încarce coada de aşteptare a microprocesorului, magistrala fiind de fapt numai a lui pe durata execuţiei instrucţiunii datorită prefixului LOCK.

Pe timpul execuţiei unei instrucţiuni "blocate" orice cerere $\overline{\text{RQ}}$ / $\overline{\text{GT}}$ va fi înregistrată urmând a fi achitată după terminarea execuţiei instrucţiunii. De asemenea, întreruperile care apar în timpul execuţiei unui prefix nu vor fi achitate, dacă sunt evident validate, decât după execuţia instrucţiunii precedate de prefix. Excepţie fac instrucţiunile care permit servirea întreruperilor pe timpul execuţiei lor: HALT, WAIT sau instrucţiunile pe şiruri.

1.3.4. GENERAREA CEASULUI ŞI A SEMNALELOR DE INIŢIALIZARE ŞI READY CU AJUTORUL CIRCUITULUI 8284

1.3.4.1. **Circuitul 8284**

Circuitul 8284, realizat în tehnologie bipolară, asigură generarea semnalelor de ceas în sistemele realizate în jurul microprocesoarelor 8086 sau 8088, sincronizarea semnalului READY şi generarea semnalului de iniţializare RESET. Schema-bloc a circuitului este dată în figura 1.32 iar conexiunile externe în figura 1.33. Prezentăm în continuare succint semnificaţia conexiunile externe.

$\overline{AEN1}$, $\overline{AEN2}$, *Address Enable*, validare adresă. Intrări active pe "0" servind pentru validarea semnalelor READY corespunzătoare, RDY1 respectiv RDY2. Cele două perechi de semnale $\overline{AEN1,2}$/RDY1,2 sunt utilizabile în configuraţii de sisteme în care procesorul deservit de circuitul 8284, poate avea acces la două magistrale, pe fiecare putându-se găsi mai multe unităţi centrale, de exemplu două magistrale de tip MULTIBUS. În configuraţiile simple, cu o singură unitate centrală intrările $\overline{AEN}$ se pot conecta la "0" validând tot timpul sincronizarea semnalelor READY.

RDY1, RDY2, *Bus Ready (Transfer Complete)*, magistrală liberă (transfer terminat). Intrare activă pe "1" indicând procesorului că transferul cu unul din dispozitivele aflate pe magistrala de date a sistemului s-a încheiat: octetul a fost recepţionat sau emis.

READY, Gata. Ieşire activă pe "1" reprezentând semnalul cu care se poate comanda intrarea READY a microprocesoarelor 8086 sau 8088 pentru a le trece în starea de aşteptare. Semnalul se obţine prin sincronizarea cu ceasul CLK a semnalelor asincrone apărute la intrările RDY1 sau RDY2. READY va fi şters după un timp ce satisface timpul de menţinere impus de microprocesor.

X1, X2. Intrări la care se conectează cuarţul a cărui frecvenţă trebuie să fie de trei mai mare decât frecvenţa dorită a ceasului microprocesorului.

$F/\overline{C}$, *Frequency/Cristal Select*, selecţie frecvenţă/cuarţ. Intrare prin care se poate selecta sursa de obţinere a semnalelor de ceas pentru microprocesor: frecvenţă externă la intrarea EFI sau oscilator cu cuarţ. $F/\overline{C}=1$ selectează generarea ceasului cu ajutorul intrării EFI.

EFI, *External Frequency Input*, intrare de frecvenţă externă. Semnalul prezentat la această intrare este de tip TTL cu factor de umplere 1/2 şi frecvenţă de trei ori mai mare decât frecvenţa dorită a ceasului microprocesorului.

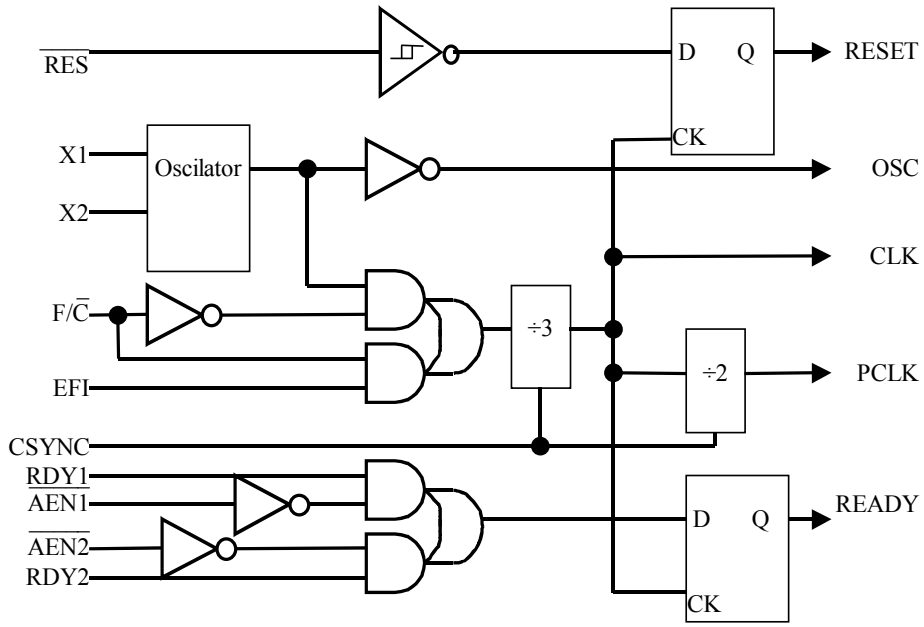


Figura 1.32. Schema bloc a circuitului 8284

CLK, *Processor Clock*, ceas pentru procesor. Ieşire de ceas utilizată de microprocesor şi dispozitivele conectate pe magistrala lui locală. Frecvenţa ceasului CLK este de trei mai mică decât frecvenţa cuarţului sau a semnalului de la intrarea EFI şi are un factor de umplere 1/3. Nivelul "1" al acestei ieşiri, garantat de cel puţin 4V la o tensiune de alimentare VCC=5V±10% şi un consum de 1mA, permite utilizarea ceasului CLK atât pentru dispozitive bipolare cât şi pentru cele MOS.

PCLK, *Peripheral Clock*, ceas pentru periferice. Ieşire de ceas utilizabilă pentru dispozitivele periferice mai lente. Frecvenţa este de două ori mai mică decât a lui CLK şi factorul de umplere 1/2. PCLK este de tip TTL.

OSC, *Oscillator Output*, ieşirea oscilatorului. Ieşirea TTL, negată, a oscilatorului având frecvenţa cuarţului. Ieşirea nu este afectată de opţiunea $F/\overline{C}$ fiind comandată direct de oscilatorul cu cuarţ. Dacă nu se foloseşte cuarţul ci intrarea externă EFI ieşirea OSC este nedeterminată. Raporturile de frecvenţă între OSC, CLK şi PCLK se pot vedea în figura 1.34.

$\overline{RES}$, *Reset In*, intrare de iniţializare. Intrare de tip *trigger Schmitt* activă pe "0" la care se poate conecta o reţea RC pentru generarea semnalului de iniţializare la punerea sub tensiune sau cu ajutorul unui comutator.

RESET, iniţializare. Ieşire activă pe "1" destinată iniţializării microprocesorului şi circuitelor aferente.

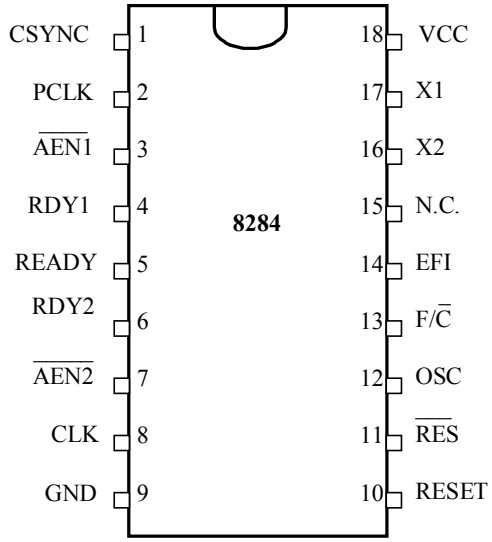


Figura 1.33. Conexiunile externe ale circuitului 8284

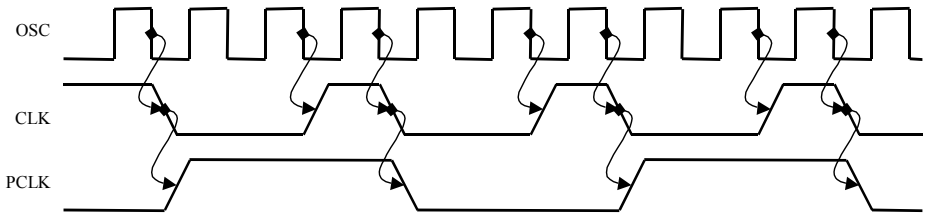


Figura 1.34. Relaţiile între OSC, CLK şi PCLK

CSYNC, *Clock Synchrozation*, sincronizare ceas. Intrare de sincronizare activă pe "1" cu ajutorul căreia se pot controla mai multe circuite 8284 pentru a genera ceas în fază. Numărătoarele interne ale circuitului sunt iniţializate cu CSYNC=1 şi validate cu CSYNC=0. Generarea semnalului de sincronizare aplicat la intrarea CSYNC trebuie făcută sincron cu EFI. CSYNC este utilizat deci împreună cu EFI, în cazul folosirii ca sursă a ceasului a oscilatorului intern intrarea CSYNC legându-se la masă.

Diagramele de timp care ilustrează funcţionarea circuitului se dau în figura 1.35 iar parametrii de timp în tabelele 1.9 şi 1.10.

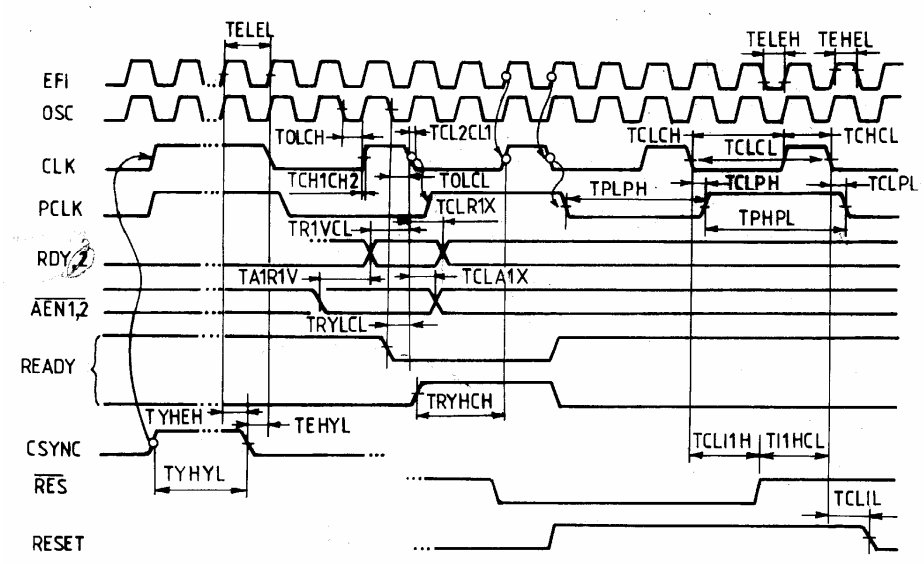


Figura 1.35. Diagramele de timp pentru circuitul 8284

Tabelul 1.9. Parametrii 8284. Cerinţe de timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă
TEHEL	Timpul cât EFI este "1"	13ns	
TELEH	Timpul cât EFI este "0"	13ns	
TELEL	Perioada EFI	TEHEL+TELEH+δ[1]	
	Frecvenţa cuarţului	12MHz	25MHz
TR1VCL	Timpul de stabilizare al semnalelor RDY1,2 faţă de CLK	35ns	
TCLR1X	Timpul de menţinere al semnalelor RDY1,2 faţă de CLK	0ns	
TA1VR1V	Timpul de stabilizare al semnalelor $\overline{\text{AEN1,2}}$ faţă de RDY1,2	15ns	
TCLA1X	Timpul de menţinere al semnalelor $\overline{\text{AEN1,2}}$ faţă de CLK	0ns	
TYHEH	Timpul de stabilizare al semnalului CSYNC faţă de EFI	20ns	
TEHYL	Timpul de menţinere al semnalului CSYNC faţă de EFI	20ns	
TYHYL	Durata lui CSYNC	2·TELEL	
TI1HCL	Timpul de stabilizare al semnalului $\overline{\text{RES}}$ faţă de CLK	65ns	
TCLI1H	Timpul de menţinere al semnalului $\overline{\text{RES}}$ faţă de CLK	20ns	

Nota 1: δ=durata frontului crescător + durata frontului descrescător al semnalului EFI(maximum 5ns+5ns=10ns)

Tabelul 1.10. Parametrii 8284. Răspunsuri în timp

Parametru	Semnificaţie	Valoare minimă	Valoare maximă
TCLCL	Perioada ceasului CLK	125ns	
TCHCL	Timpul cât CLK este "1"	(1/3)·TCLCL+2ns	
TCLCH	Timpul cât CLK este "0"	(2/3)·TCLCL−15ns	
TCH1CH2 TCL2CL1	Durata fronturilor crescător şi descrescător ale ceasului CLK		10ns
TPHPL	Timpul cât PCLK este "1"	TCLCL−20ns	
TPLPH	Timpul cât PCLK este "0"	TCLCL−20ns	
TRYLCL	Timpul între dezactivarea semnalului READY şi CLK	−8ns	
TRYHCH	Timpul între activarea semnalului READY şi CLK	(2/3)·TCLCL−15ns	
TCLIL	Întârzierea între CLK şi RESET	40ns	
TCLPH	Întârzierea între CLK şi frontul crescător al PCLK		22ns
TCLPL	Întârzierea între CLK şi frontul descrescător al PCLK		22ns
TOLCH	Întârzierea între OSC şi frontul crescător al CLK	−5ns	12ns
TOLCL	Întârzierea între OSC şi frontul descrescător al CLK	2ns	20ns

1.3.4.2. **Generarea ceasului**

În figura 1.36 se dau caracteristicile pe care trebuie să le îndeplinească semnalul de ceas CLK al microprocesorului 8086. După cum se vede, 8086 are nevoie de un ceas cu fronturi rapide, maximum 10ns, şi niveluri de tensiune "0" între −0,5V şi +0,6V şi "1" între 3,9V şi VCC+1V. Frecvenţa maximă a ceasului pentru 8086, varianta standard, este de 5MHz. Frecvenţa minimă a ceasului, având în vedere existenţa în interiorul microprocesorului a unor celule de memorie dinamică, este de 2MHz. Din cauza acestei ultime restricţii funcţionarea pas-cu-pas a lui 8086, la nivel de instrucţiune sau de ciclu-maşină, nu poate fi implementată hardware prin blocarea ceasului. Ea este posibilă numai prin poziţionarea software a indicatorului de condiţii T, aşa cum vom vedea în §1.4.5.1 şi în §2.3.6. Pentru a satisface cerinţele minime de timp care să asigure o funcţionare internă optimă, ceasul microprocesorului se recomandă, după cum am menţionat, să aibă un factor de umplere 1/3. Această recomandare apare mai stringentă cu cât ne apropiem de frecvenţa maximă de lucru. Circuitul 8284, prezentat în capitolul anterior, asigură generarea unui ceas cu factor de umplere 1/3, nivelurile de tensiune şi fronturile lui respectând cerinţele impuse de microprocesor.

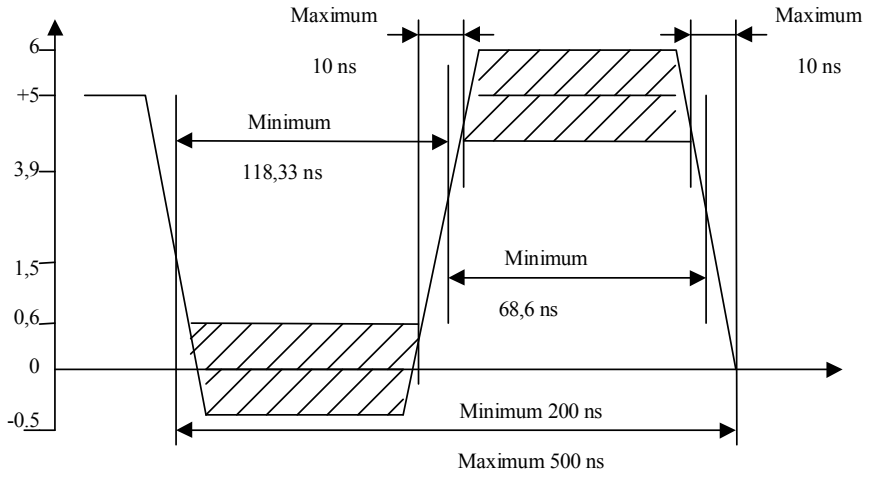


Figura 1.36. Caracteristicile ceasului pentru microprocesorul 8086

După cum se poate deduce din prezentarea conexiunilor externe şi a schemei bloc ale circuitului 8284, ca sursă pentru generarea ceasului se poate folosi fie un semnal extern la intrarea EFI, fie oscilatorul din interior proiectat să oscileze cu un cuarţ extern conectat între X1 şi X2 pe frecvenţa de rezonanţă serie a acestuia. Sursa trebuie să oscileze pe o frecvenţă de trei ori mai mare decât frecvenţa dorită a ceasului CLK. Pentru o generare cât mai stabilă şi precisă a ceasului fabricanţii circuitului recomandă utilizarea cuarţurilor ce oscilează pe frecvenţa fundamentală şi au o rezistenţă serie cât mai mică. De asemenea, deoarece oscilatorul nu apare pentru cuarţ ca o sursă ideală de semnal având o componentă inductivă, pentru a anula efectul acestei componente – oscilaţia pe o frecvenţă mai mică decât frecvenţa de rezonanţă serie pură –, trebuie adăugat în serie cu cristalul, la intrarea X2, un condensator, CL în figura 1.37. Acest condensator serveşte, de asemenea, şi la separarea în curent continuu a cuarţului ceea ce asigură o protecţie a acestuia din urmă împotriva unor polarizări în tensiune continuă care ar putea să deranjeze şi chiar să distrugă structura cristalină a cuarţului.

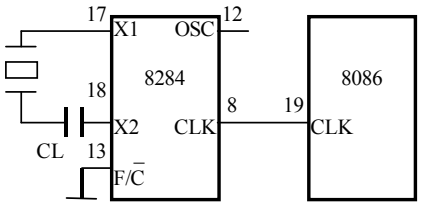


Figura 1.37. Conectarea unui cuarţ la 8284

Impedanţa condensatorului, *XCL*, depinde de frecvenţa de oscilaţie, fiind data de:

$$XCL = \frac{1}{2 \cdot \pi \cdot F \cdot CL}.$$

Se recomandă ca rezistenţa-serie a cristalului plus *XCL* să fie mai mică decât 1kΩ pentru a nu opri funcţionarea oscilatorului prin reducerea amplificării în buclă sub 1.

Valoarea condensatorului descreşte deci cu frecvenţa cristalului fiind, de exemplu, aproximativ 24pF la 12MHz şi 8pF la 22MHz. Dacă sistemul nu necesită alegerea precisă a unei frecvenţe, ceea ce ar fi impus corelarea exactă a valorii condensatorului cu frecvenţa de rezonanţă serie a cuarţului, ci doar stabilitatea ei, fabricanţii recomandă utilizarea unui condensator de 12÷15pF pentru un cuarţ de 15MHz în vederea generării ceasului CLK la 5MHz. Circuitul imprimat, în general elementele tehnologice folosite, fire, lipituri, socluri, pot modifica valoarea componentei inductive şi deci conduce la schimbarea valorii condensatorului CL.

Intrarea EFI se utilizează atunci când este necesară o sursă de semnal a cărei frecvenţă să fie foarte precisă sau să fie variabilă şi pentru situaţiile când mai multe circuite 8284 comandând unităţi centrale care trebuie sincronizate, au nevoie de o sursă comună, unică, de oscilaţie, figura 1.38. Semnalul extern aplicat la intrarea EFI a circuitului trebuie să fie compatibil TTL, să aibă un factor de umplere 1/2 şi o frecvenţă de trei ori mai mare decât frecvenţa dorită a ceasului CLK. Frecvenţa maximă a acestui semnal este puţin peste 24MHz, timpul minim pe "0" sau pe "1" fiind de 13ns. Frecvenţa minimă este impusă de microprocesor. La folosirea unei surse comune pentru generarea ceasului de către mai multe circuite 8284 distribuite în cadrul sistemului fiecare 8284 va trebui comandat direct de la sursă printr-o legătură proprie. Se recomandă, pentru minimizarea zgomotului, ca aceste legături să fie realizate cu fir bifilar, torsadat, semnalul fiind emis şi recepţionat cu porţi ca 74LS04, având fronturi mai lente, masa firului torsadat conectând mesele sursei şi receptorului, figura 1.37. De asemenea, pentru micşorarea alunecării ceasului aceste legături trebuie să fie de aceeaşi lungime. Cu toate acestea variaţia întârzierii între EFI şi ceasurile CLK generate de mai multe circuite 8284, ca în figura 1.38, poate ajunge la 35÷40ns. Această variaţie se reduce la 15÷25ns dacă circuitele sunt împachetate în acelaşi tip de capsulă, au acelaşi tensiune de alimentare şi lucrează la aceeaşi temperatură.

Pentru sincronizarea ceasurilor cu evenimente externe proiectanţii au prevăzut intrarea CSYNC: CSYNC=1 forţează ieşirile de ceas CLK şi PCLK pe "1" iar CSYNC=0 validează generarea ceasurilor, prin pornirea numărătoarelor,

cu primul front pozitiv al semnalului emis de sursa de frecvenţă, oscilatorul cu cuarţ sau EFI. CSYNC trebuie să fie activ pe "1" cel puţin două perioade ale semnalului emis de sursa de frecvenţă. În figurile 1.39a şi b se prezintă două moduri de obţinere a unui semnal de comandă CSYNC sincronizat, pornind de la o condiţie de sincronizare externă şi funcţie de sursa de generare a ceasurilor, EFI respectiv OSC. Cele două bistabile din figura 1.39a sunt acţionate cu $\overline{EFI}$ pentru a se asigura timpii de stabilizare şi menţinere impuşi comenzii CSYNC, figura 1.39c. Această negare a semnalului sursă nu mai este necesară la utilizarea oscilatorului local deoarece OSC reprezintă ieşirea negată a semnalului emis de acesta. Încă o observaţie: deoarece activarea lui CSYNC poate conduce, prin trecerea imediată pe "1" a ceasului CLK, la violarea timpului TCLCH de minimum 118ns, se recomandă ca această activare să se facă fie pe timpul iniţializării, fie pe timpul cât CLK=1. Dacă se face pe timpul iniţializării, invalidarea lui CSYNC trebuie realizată cu minimum patru perioade ale ceasului CLK înainte de terminarea RESET-ului pentru a se garanta iniţializarea microprocesorului.

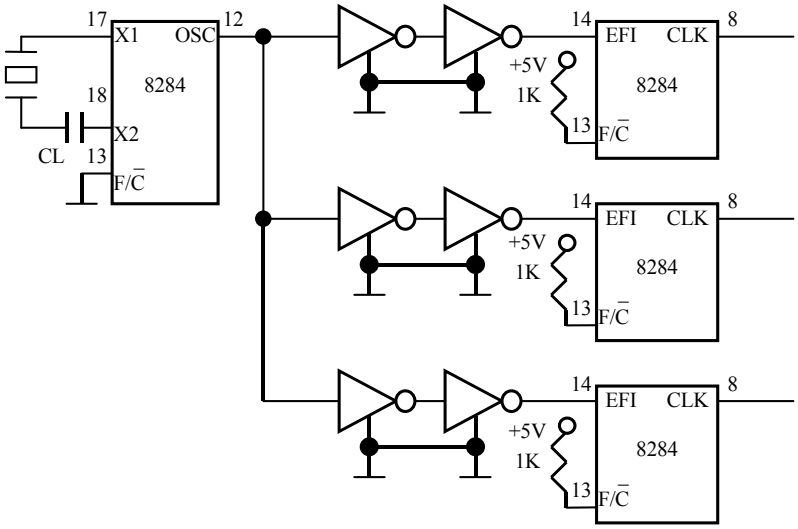


Figura 1.38. Mai multe circuite 8284 având o sursă comună de oscilaţie

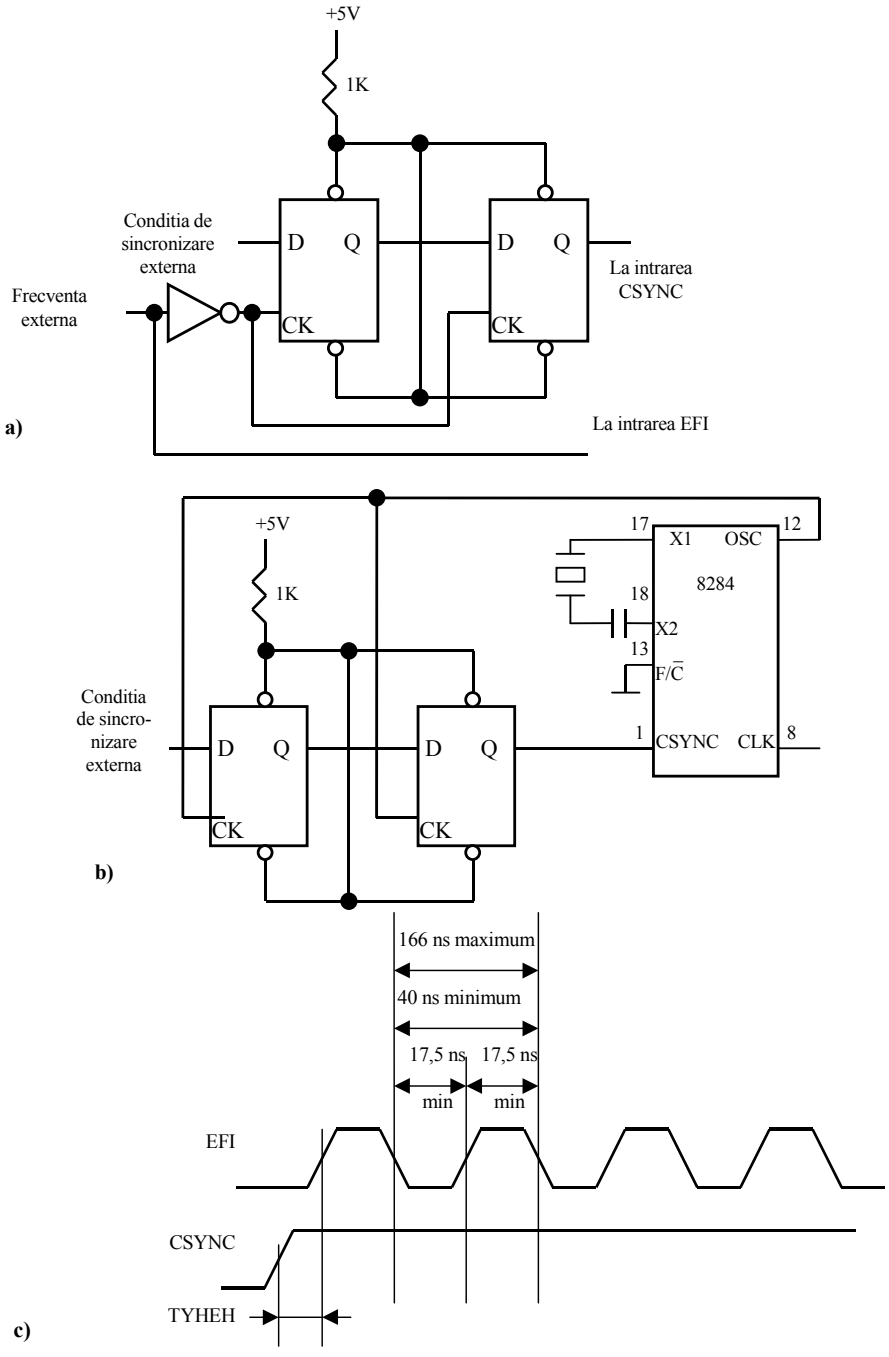


Figura 1.39. Sincronizarea comenzii CSYNC : cu o sursă externă de frevenţă (a); cu oscilatorul cu cuarţ (b); timpul de stabilizare al comezii CSYNC faţă de EFI (c)

1.3.4.3. **Iniţializarea**

Iniţializarea microprocesorului 8086 se realizează prin activarea, trecerea pe "1", a conexiunii RESET. Durata minimă de activare a intrării RESET este de patru perioade ale ceasului CLK, cu excepţia iniţializării la punerea sub tensiune când este necesar un impuls de 50μs. La apariţia semnalului RESET 8086 va termina operaţia în curs şi va rămâne inactiv până la terminarea impulsului de iniţializare. Semnalul RESET, sincronizat intern, va declanşa după revenirea sa pe "0" o secvenţă internă de iniţializare de aproximativ zece cicli CLK. După acest interval, microprocesorul îşi va relua funcţionarea normală începând cu instrucţiunea de la adresa FFFF0H.

La iniţializare, microprocesorul 8086 va elibera magistrala de date, adrese şi comenzi conform tabelului 1.11. Aşa cum se vede magistrala multiplexată va fi trecută odată cu activarea intrării RESET în starea a treia. O altă parte a semnalelor va fi întâi poziţionată pe "1", pe durata unei semiperioade a ceasului, când CLK este "0" – figura 1.40, după care va fi trecută în starea a treia. În modul minim ALE şi HLDA vor fi inactivate, trecute pe "0". În modul maxim liniile $\overline{RQ}/\overline{GT}$ sunt, de asemenea, inactivate, trecute pe "1", iar liniile de stare a cozii QS1,0 devin 0,0 – *nici o operaţie*. Deoarece în această din urmă situaţie la iniţializare indicatorii stivei, după cum se vede, nu sunt resetaţi, QS1,0=1,0, se impune, în sistemele unde se foloseşte o logică externă de urmărire a stării cozii, iniţializarea ei cu semnalul general RESET.

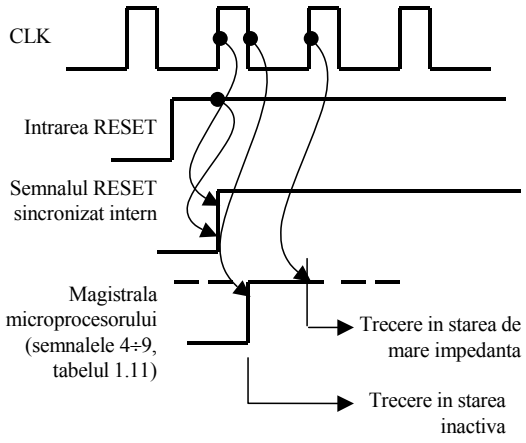


Figura 1.40. Inactivarea magistralei microprocesorului 8086 la iniţializare

Tabelul 1.11. Starea magistralei microprocesorului 8086 la iniţializare

Denumirea semnalului	Starea semnalului la iniţializare
AD15÷AD0	Starea a 3-a
A19/S6÷A16/S3	Starea a 3-a
$\overline{BHE}$ / S7	Starea a 3-a
$\overline{S2}$(M / $\overline{IO}$)	Trecut pe "1", apoi în starea a 3-a
$\overline{S1}$(DT / $\overline{R}$)	Trecut pe "1", apoi în starea a 3-a
$\overline{S0}$($\overline{DEN}$)	Trecut pe "1", apoi în starea a 3-a
$\overline{LOCK}$ / $\overline{WR}$	Trecut pe "1", apoi în starea a 3-a
$\overline{RD}$	Trecut pe "1", apoi în starea a 3-a
$\overline{INTA}$	Trecut pe "1", apoi în starea a 3-a
ALE	"0"
HLDA	"0"
$\overline{RQ}$ / $\overline{GT0}$	"1"
$\overline{RQ}$ / $\overline{GT1}$	"1"
QS0	"0"
QS1	"0"

Pentru a se asigura starea inactivă a liniilor de comandă şi control ale microprocesorului în special în sistemele unde curenţii de scurgere sau capacitatea parazită pot conduce la niveluri de tensiune "1" sub nivelul admis de dispozitivele utilizate, se recomandă legarea la VCC a acestor linii cu rezistenţe de 22kΩ. În sistemele care lucrează în modul maxim această cerinţă este asigurată de 8288 care conţine rezistenţe interne legate la VCC. Datorită acestor rezistenţe 8288 interpretează la iniţializare liniile de stare S2÷S0, pe "1", ca reprezentând starea pasivă a microprocesorului şi îşi poziţionează ieşirile conform tabelului 1.12. Dacă iniţializarea apare în timpul unui ciclu de magistrală trecerea liniilor S2÷S0 în starea pasivă va conduce la terminarea ciclului şi revenirea liniilor de comandă în starea inactivă. Reţinem deci că 8288 nu îşi trece ieşirile de comandă în starea a treia atunci când S2÷S0 semnifică starea pasivă a microprocesorului. Dacă în sistem, la iniţializare, este necesară această trecere a comenzilor în starea de impedanţă înaltă, semnalul de iniţilizare va trebui conectat la intrarea $\overline{AEN}$ a lui 8288 precum şi, pentru invalidarea întregii unităţi centrale, la intrările de validare-ieşire, $\overline{OE}$, ale *latch*-urilor de adrese. *Transceiver*-ele pentru date sunt dezactivate cu ajutorul ieşirii DEN, figura 1.41. Un nivel "1" corect al ieşirilor de comandă ale circuitului 8288 la trecerea lor în starea a treia se asigură prin legarea lor la VCC cu ajutorul unor rezistenţe de 2,2kΩ.

Tabelul 1.12. Ieşirile circuitului 8288 la iniţializare

Denumirea semnalului	Starea semnalului la iniţializare
ALE	0
DEN	0
DT / $\overline{R}$	1
MCE / $\overline{PDEN}$	0/1
Comenzi	1

Semnalul de iniţializare a microprocesorului, RESET, poate fi generat, aşa cum am spus în §1.4.4.1, cu ajutorul circuitului 8284. Comanda acestui semnal se face la intrarea de tip *trigger Schmitt*, $\overline{RES}$, figura 1.42a. După cum se ştie, astfel de intrări asigură, prin utilizarea unei reacţii interne pozitive care accelerează tranziţiile lente şi fixează praguri diferite pentru fronturile pozitive respectiv negative, transformarea semnalelor de intrare cu variaţie lentă în semnale de ieşire clar definite, fără oscilaţii, stabile. Diferenţa între cele două praguri, *histerezis*-ul la intrarea $\overline{RES}$, este de minimum 0,25V. Aceasta înseamnă că un semnal "0" la intrarea $\overline{RES}$, a cărui valoare maximă este V_{ILmax}=0,8V, va rămâne activ în tranziţia pozitivă spre valoarea "1" până la 1,05V. În sensul invers un semnal "1" la intrarea $\overline{RES}$, a cărui valoare minimă este V_{IHmin}=2,6V, va fi considerat în continuare "1", în tranziţia spre "0", până va scădea sub 2,35V.

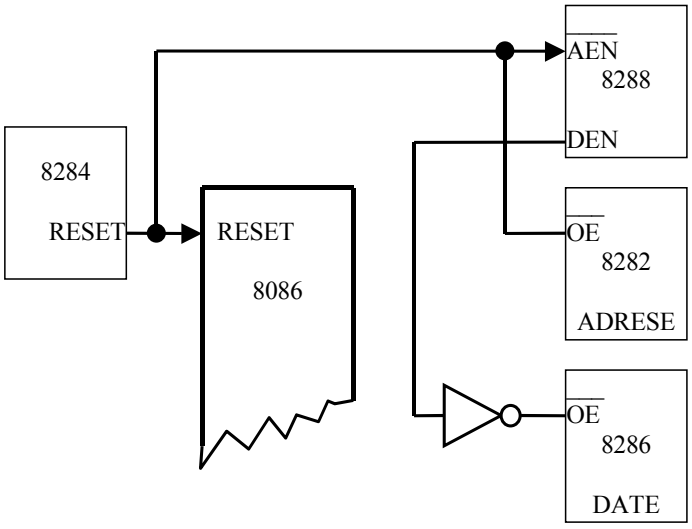


Figura 1.41. Dezactivarea la iniţializare a controlorului 8288, a *latch*-urilor de adrese şi a *transceiver*-elor de date

La punerea sub tensiune, intrarea $\overline{\text{RES}}$ trebuie să rămână sub 1,05V cel puţin 50μs după ce tensiunea de alimentare VCC a atins 4,5V, figura 1.42b. *Histerezis*-ul de la intrarea $\overline{\text{RES}}$ permite asigurarea acestei cerinţe prin conectarea unui circuit RC simplu, figura 1.42a. Constanta RC va rezulta din formula obişnuită:

$$V(t) = V\left(1 - \mathrm{e}^{-\frac{t}{RC}}\right),$$

unde V=4,5V. Valoarea constantei RC rezultată din această formulă nu ţine cont de timpul în care tensiunea de alimentare atinge 4,5V şi de sarcina acumulată în condensator în acest timp. Pentru t=50μs şi $V(t)$=1,05V, în formula de mai sus, se obţine întâi constanta de timp RC=$188 \cdot 10^{-6}$ şi apoi durata iniţializării de 162μs după ce tensiunea de alimentare a atins 4,5V. Această durată reprezintă timpul de creştere a tensiunii la intrarea $\overline{\text{RES}}$ până la valoarea de comutare $V(t)$=2,6V. Dacă se doreşte o precizie mai bună pentru durata iniţializării încărcarea condensatorului de la intrarea $\overline{\text{RES}}$ poate fi făcută cu un generator de curent constant care va asigura o creştere liniară şi nu invers exponenţială a tensiunii la intrarea $\overline{\text{RES}}$.

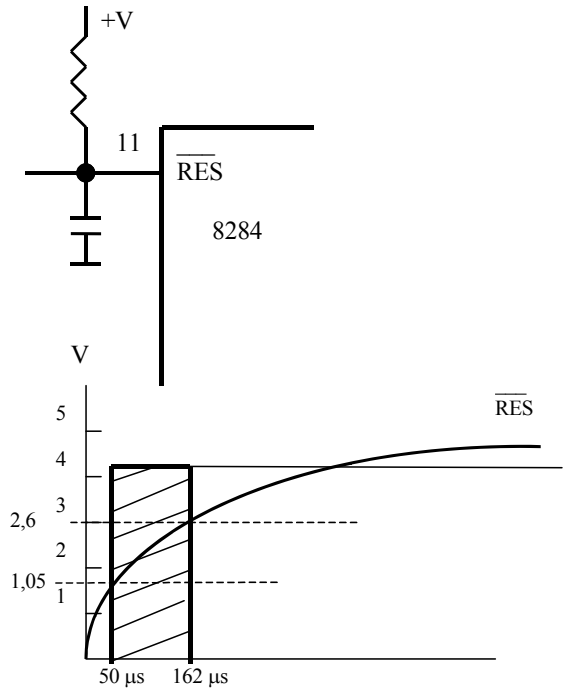


Figura 1.42. Comanda iniţializării cu ajutorul circuitului 8284 : circuitul RC de la intrarea $\overline{\text{RES}}$ (a); semnalul aplicat la intrarea $\overline{\text{RES}}$ (b)

1.3.4.4. **Generarea semnalului READY**

Pentru formarea către microprocesor a semnalului READY, necesar pentru introducerea stărilor de aşteptare, care să satisfacă cerinţele de timp prezentate în §1.4.2.5 şi §1.4.3.6, se poate utiliza, aşa cum am mai spus, circuitul 8284 care are prevăzute în acest scop intrările separate RDY1 şi RDY2 validate de $\overline{\text{AEN1}}$, respectiv $\overline{\text{AEN2}}$. La aceste intrări se va conecta logica propriu-zisă de generare a stărilor de aşteptare realizată în funcţie de configuraţia sistemului şi dispozitivele de memorie şi/sau I/O folosite. Cele două intrări RDY1 şi RDY2, validate, sunt trecute printr-o poartă SAU LOGIC formând un semnal intern eşantionat la începutul fiecărui ciclu de ceas, figura 1.43. Semnalul eşantionat este generat la ieşirea READY cu o întârziere de maximum 8ns, ceea ce satisface timpii de stabilizare la intrarea READY a microprocesorului, figura 1.15. Semnalul eşantionat, fiind memorat, nu se poate modifica decât cel mai devreme cu următorul front negativ al ceasului CLK, ceea ce asigură şi timpii de menţinere impuşi de 8086. Semnalele generate la intrările RDY1 şi RDY2 ale circuitului 8284 trebuie să satisfacă timpul de stabilizare minim $TR1VCL_{min}$=35ns faţă de frontul negativ al ceasului iar cele generate la intrările AEN, minimum $TR1VCL_{min}$+ $TA1VR1V_{min}$= 35ns+15ns =50ns.

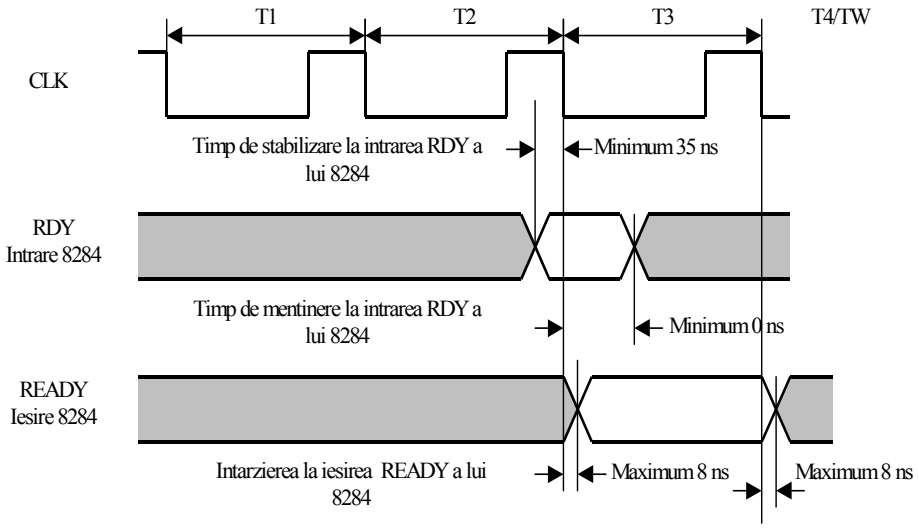


Figura 1.43. Relaţii de timp la generarea semnalului READY cu ajutorul circuitului 8284

Pentru sistemele care au nevoie de o singură intrare RDY intrarea $\overline{AEN}$ corespunzătoare se poate lega la GND, cealaltă intrare de validare conectându-se la VCC printr-o rezistenţă de aproximativ 1kΩ, figura 1.44a. Dacă semnalul generat de logica de introducere a stărilor de aşteptare este activ pe "0" se pot utiliza intrările $\overline{AEN}$, figura 1.44b. Un exemplu de logică pentru inserarea unei singure stări TW, suficientă pentru marea majoritate a dispozitivelor de memorie sau I/O care nu pot să lucreze la viteza maximă a microprocesorului, este dat figura 1.45. Selecţia unuia dintre dispozitivele lente din sistem, la noi $\overline{CS1}$ sau $\overline{CS2}$, conduce la bascularea bistabilului şi trecerea semnalului RDY pe "0", figura 1.46. Trecerea pe "0" a bistabilului se face cu frontul pozitiv din T2, ceea ce asigură formarea corectă a semnalului READY către microprocesor şi introducerea unei stări TW după T3. Revenirea pe "1" a bistabilului cu frontul pozitiv din T3 indică procesorului ieşirea din TW şi intrarea în T4 pentru a încheia ciclul. Următoarea trecere pe "0" a bistabilului în TW nu mai influenţează desfăşurarea operaţiei. Schema se iniţializează cu ALE la începutul fiecărui ciclu-maşină.

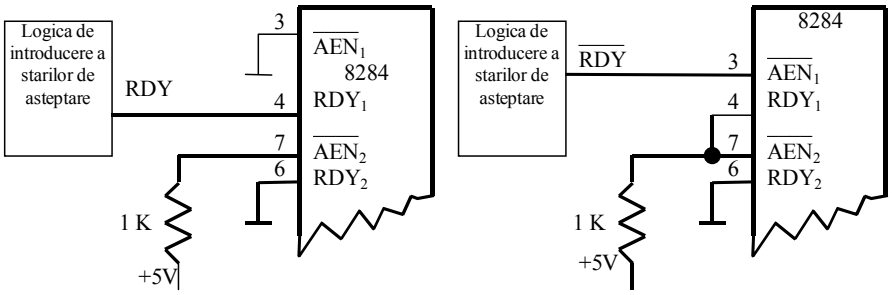


Figura 1.44. Generarea semnalului READY cu ajutorul circuitului 8284 : utilizarea intrărilor RDY1 sau RDY2 (a); utilizarea intrărilor $\overline{AEN1}$, $\overline{AEN2}$ (b)

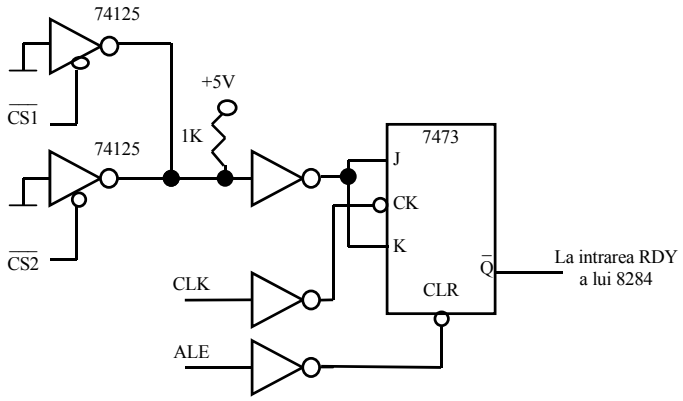


Figura 1.45. Un circuit pentru inserarea unei singure stări TW

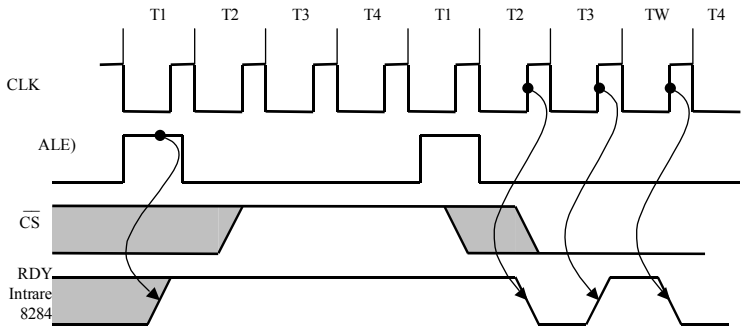


Figura 1.46. Diagrama de funcţionare a circuitului din figura 1.45

1.3.5. SISTEMUL DE ÎNTRERUPERI

La 8086 întreruperile pot fi cauzate atât de evenimente hardware, cât şi de evenimente software. Întreruperile hardware, generate de circuite exterioare microprocesorului, pot fi mascabile sau nemascabile. Întreruperile software, nemascabile, grupate mai demult în categoria *derute* [12], sunt generate de logica programelor executate pe 8086.

Sistemul de întreruperi al procesorului 8086 are la bază o tabelă a vectorilor de întrerupere plasată în memoria sistemului între adresele 00000H şi 003FFH, figura 1.47. Fiecare vector constă din doi octeţi reprezentând numărătorul de program, *pointer*-ul de instrucţiune, IP, şi doi octeţi ce specifică valoarea asociată a segmentului de cod, registrul CS. Cele două valori, IP şi CS, formează adresa rutinei de serviciu (vezi capitolul 1.2). Tabela conţine maximum 256 de vectori de întrerupere precizând deci adresele de început ale rutinelor de serviciu ale întreruperilor. Aceste rutine pot fi plasate oriunde în spaţiul de adresare de 1Moctet al microprocesorului. Fiecărui vector din tabelă îi este asociat, aşa cum se poate vedea în figura 1.47, un număr ce reprezintă tipul întreruperii. Numărul, preluat din logica externă sau din program, multiplicat cu 4, dă deplasamentul faţă de începutul tabelei la care se găseşte vectorul de întrerupere asociat. Acest sistem de întreruperi "vectorizat" este deosebit de flexibil permiţând utilizatorului să specifice cum crede de cuviinţă adresele de memorie ale tuturor rutinelor de serviciu întrebuinţate. Dacă nu sunt definite toate cele 256 de întreruperi, utilizatorul va preciza numai adresele rutinelor folosite, recomandându-se, totuşi, cel puţin, în perioada de punere la punct, asignarea tipurilor de întrerupere neutilizate cu adresa unei rutine separate pentru detectarea uşoară a întreruperilor parazite.

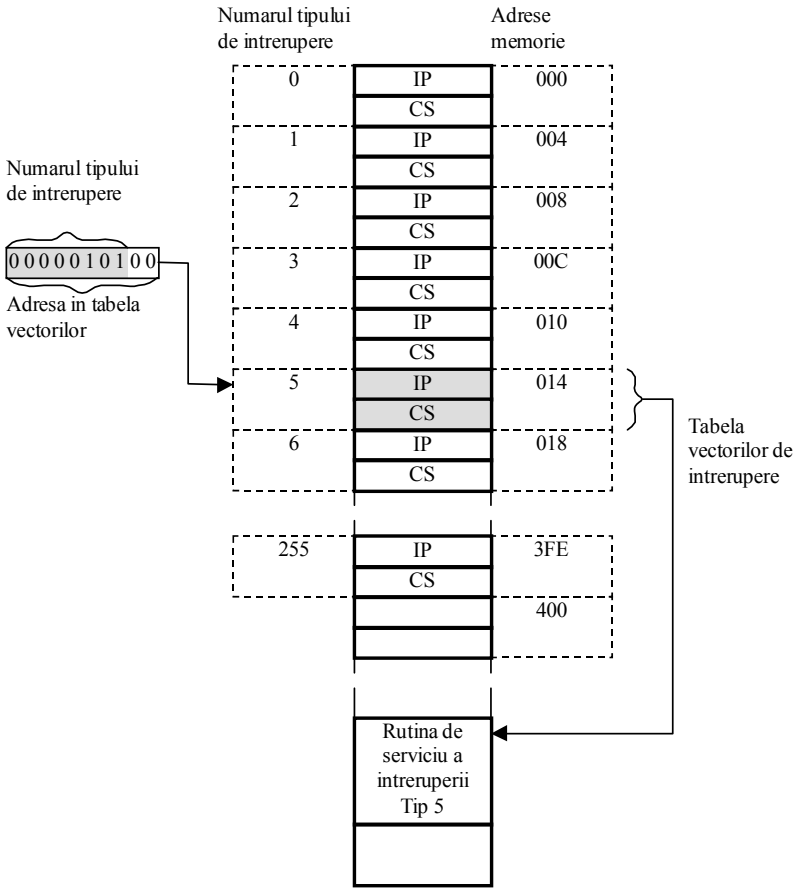


Figura 1.47. Tabela vectorilor de întrerupere la 8086

Întreruperile microprocesorului 8086 se mai pot clasifica în trei grupe: întreruperi predefinite generate de funcţii speciale, întreruperi hardware definite de utilizator şi întreruperi software definite de utilizator. Vom descrie pe scurt aceste tipuri de întreruperi, vezi şi figura 1.47.

1.3.5.1. **Întreruperi predefinite**

La invocarea din hardware sau din software a unei întreruperi predefinite microprocesorul va transfera controlul rutinei a cărei adresă este specificată de vectorul asociat tipului de întrerupere. Utilizatorul are sarcina să scrie rutinele de serviciu şi să iniţializeze tabela vectorilor cu adresele corespunzătoare.

Tipul 0 – împărţire la 0. Această întrerupere este invocată la orice încercare de împărţire pentru care câtul depăşeşte valoarea maximă, cum este, de exemplu, cazul împărţirii la zero. Întreruperea nu este mascabilă şi poate fi considerată ca o secvenţă aparţinând operaţiei de împărţire.

Tipul 1 – pas-cu-pas. Întreruperea apare după o instrucţiune de la poziţionarea indicatorului de condiţie *Derută*, T, în registrul de stare al microprocesorului. Se permite astfel introducerea software a funcţionării pas-cu-pas în cadrul unei secvenţe de program. Secvenţa de întrerupere începe cu salvarea registrului de stare şi a numărătorului de program şi ştergerea indicatorului T permiţându-se astfel execuţia normală, nu pas-cu-pas, a rutinei de serviciu specifice. La revenire în programul principal, modul de execuţie pas-cu-pas, este asigurat prin restaurarea IP, CS şi a indicatorilor de condiţii, deci şi a lui T. Nici această întrerupere nu poate fi mascată.

Tipul 2 – întrerupere nemascabilă, NMI. Întreruperea este de tip hardware şi are prioritatea cea mai înaltă. Intrarea corespunzătoare a microprocesorului, conexiunea NMI, este comutată pe front şi apoi sincronizată în interiorul procesorului cu ceasul CLK. Pentru ca o întrerupere NMI să fie recunoscută, semnalul intern sincronizat trebuie să fie activ cel puţin două perioade ale ceasului. De asemenea, dacă intrarea NMI rămâne pe "1" mai multă vreme, revenirea pe "0", timpul cât stă pe "0", înainte ca o nouă întrerupere să fie declanşată este de minimum două perioade CLK. Semnalul de la intrarea lui 8086 poate fi dezactivat înainte de intrarea în rutina de serviciu. O atenţie deosebită trebuie acordată eliminarii *spike*-urilor, impulsurilor parazite, care pot genera întreruperi. Întreruperea NMI este rezervată, de obicei, evenimentelor catastrofale cum sunt căderile de tensiune sau semnalizările de la un sistem de supraveghere de tip *ceas de gardă*, *watchdog*.

Tipul 3 – întrerupere pe un octet. Este o întrerupere software nemascabilă invocată de o instrucţiune reprezentată pe un octet, INT 3, destinată în primul rând ca întrerupere de tip *breakpoint* pe parcursul punerii la punct a programelor.

Tipul 4 – întrerupere la depăşire. Este o întrerupere software nemascabilă care apare la execuţia instrucţiunii INTO dacă indicatorul de condiţie *Depăşire*, O, este poziţionat. Instrucţiunea permite derutarea programului la o rutină de serviciu în cazul apariţiei unei erori de depăşire.

Întreruperile de tip 0 sau 2 pot apărea fără ca programatorul să acţioneze în vreun fel specific, cu excepţia unei împărţiri la 0 care ar putea fi o greşeală de programare. Întreruperile de tip 1, 3 şi 4 necesită pentru a fi generate acte conştiente din partea programatorului. Toate tipurile de întreruperi prezentate mai sus, cu excepţia NMI, sunt invocate software şi sunt asociate direct cu câte o instrucţiune specifică.

1.3.5.2. **Întreruperi software definite de utilizator**

Întreruperile software sunt generate de instrucţiunea `INT` nn unde nn reprezintă numărul tipului de întrerupere definit de utilizator. Instrucţiunile `INT`, deci întreruperile software, nu sunt mascabile cu ajutorul indicatorului de condiţii *Validare/Invalidare Întrerupere*, I. Revenirea din rutina de serviciu apelată printr-o întrerupere software se face cu instrucţiunea `IRET`.

Transferul controlului la o întrerupere software se face la sfârşitul instrucţiunii `INT` nn fără ca microprocesorul să iniţieze pe magistrală un ciclu de achitare INTA. De asemenea, întreruperile software vor invalida întreruperile mascabile prin punerea pe "0" a indicatorilor I şi T.

1.3.5.3. **Întreruperi hardware definite de utilizator**

Aşa cum am mai spus, întreruperile hardware definite de utilizator sunt iniţiate de circuite speciale prin activarea, trecerea pe "1", a semnalului de la intrarea INTR a procesorului. Aceste întreruperi sunt mascabile cu ajutorul bitului I din registrul de stare.

Starea intrării INTR este testată în timpul ultimului ciclu de ceas al fiecărei instrucţiuni. Excepţie de la această regulă fac instrucţiunile `MOV` şi `POP` cu un registru de segment, instrucţiunea `WAIT`, instrucţiunile pe şiruri precedate de prefixul de repetare `REP`.

În cazul instrucţiunilor `MOV` şi `POP` cu un registru de segment microprocesorul va testa starea conexiunii INTR după executarea instrucţiunii următoare instrucţiunilor `MOV` şi `POP` menţionate. Astfel, se permite încărcarea unui *pointer* de stivă de 32 de biţi în registrele SS şi SP fără pericolul apariţiei unei întreruperi între cele două încărcări. O secvenţă cum este şi cea de mai jos nu va fi deci interuptibilă:

```
mov ss,  segment_stiva_nou
mov sp,  pointer_stiva_nou
```

În timpul instrucţiunii `WAIT` care aşteaptă trecerea pe "0" a intrării $\overline{\text{TEST}}$ a microprocesorului se testează şi starea semnalului la conexiunea INTR pentru a se permite executarea rutinelor de întrerupere în timpul aşteptării. Particularitea în această situaţie constă în faptul că atunci când se dectectează o întrerupere, 8086 mai extrage încă o dată instrucţiunea `WAIT` înainte de a

transfera controlul rutinei de serviciu. Aceasta pentru a garanta revenirea din rutină tot la instrucţiunea de aşteptare WAIT.

O altă situaţie specială apare şi în cazul instrucţiunilor cu prefix. Deoarece prefixele sunt considerate ca parte a instrucţiunii pe care o preced, procesorul va eşantiona semnalul INTR la sfârşitul execuţiei instrucţiunii incluzând în aceasta şi prefixele. Excepţia survine la instrucţiunile pe şiruri precedate de prefixul REP. Pentru aceste instrucţiuni testarea întreruperii se face la sfârşitul fiecărei repetări. Atunci când instrucţiunile repetitive pe şiruri mai sunt precedate şi de alte prefixe, de exemplu LOCK, şi apare o întrerupere microprocesorul va restaura la revenirea din rutina de serviciu numai prefixul imediat precedent instrucţiunii. De aceea, pentru a nu perturba printr-o întrerupere execuţia completă a instrucţiunii repetitive trebuie utilizată o secvenţă de program de tipul:

```
transfer_repetitiv: lock rep movs dest,cs:sursa
                    and cx,cx
                    jnz transfer_repetitiv
```

Codul obiect pe octeţi generat de asamblorul 8086 pentru instrucţiunea MOVS va fi în ordinea crescătoare a adreselor: prefixul LOCK, prefixul REP, prefixul *Depăşire Segment (:)*, codul propriu-zis al instrucţiunii MOVS. La revenirea dintr-o rutină de serviciu a întreruperii va fi refăcut numai prefixul *Depăşire Segment* garantându-se astfel execuţia corectă a încă unui transfer. Următoarea instrucţiune verifică terminarea transferului şirului prin testarea valorii contorului asociat instrucţiunii MOVS, registrul CX. Dacă această valoare e diferită de zero se sare la începutul secvenţei pentru a se termina execuţia instrucţiunii repetitive cu prefixe.

1.3.5.4. **Prioritatea întreruperilor şi a cererii DMA**

Ordinea de prioritate a întreruperilor respectă următoarele reguli:

– INTR, întreruperea hardware, este singura întrerupere mascabilă şi dacă ea este detectată simultan cu alte întreruperi, ştergerea indicatorului I din registrul de stare o va masca. Aceasta face ca INTR să fie de fapt întreruperea cu prioritatea cea mai scăzută atâta timp cât rutinele de serviciu ale celorlalte întreruperi nu revalidează INTR prin poziţionarea lui I;

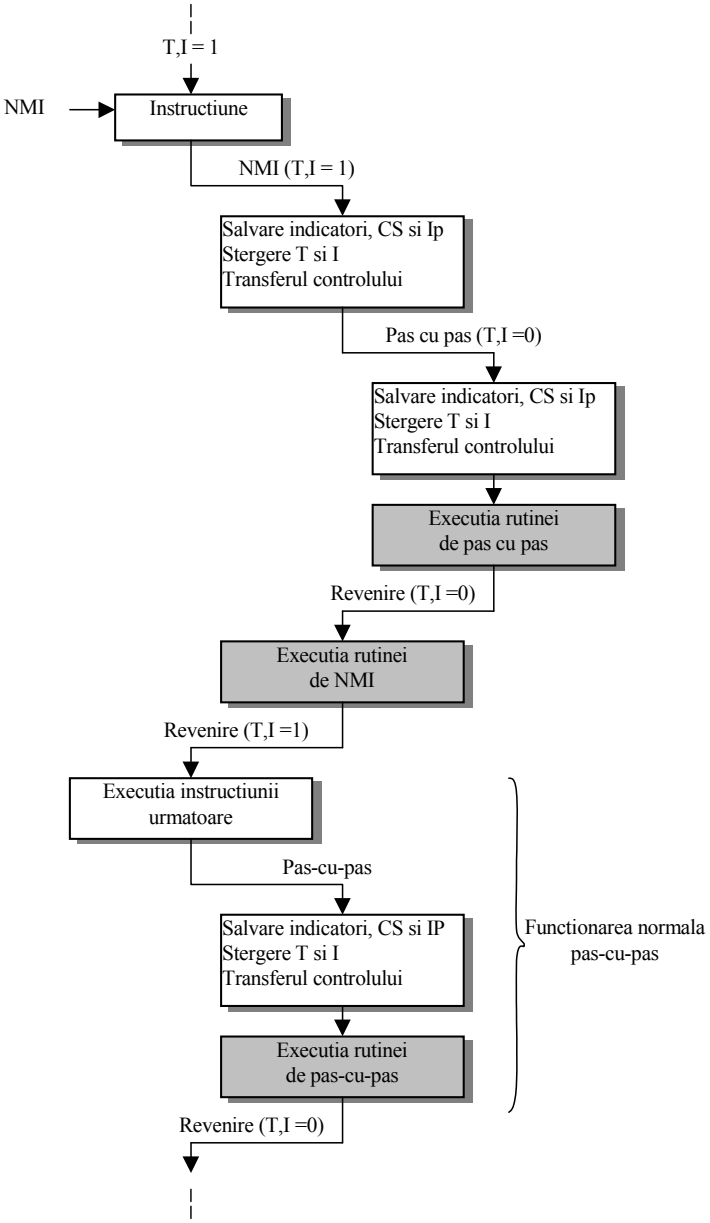


Figura 1.48. O întrerupere NMI pe timpul funcţionării pas-cu-pas şi funcţionarea normală pas-cu-pas

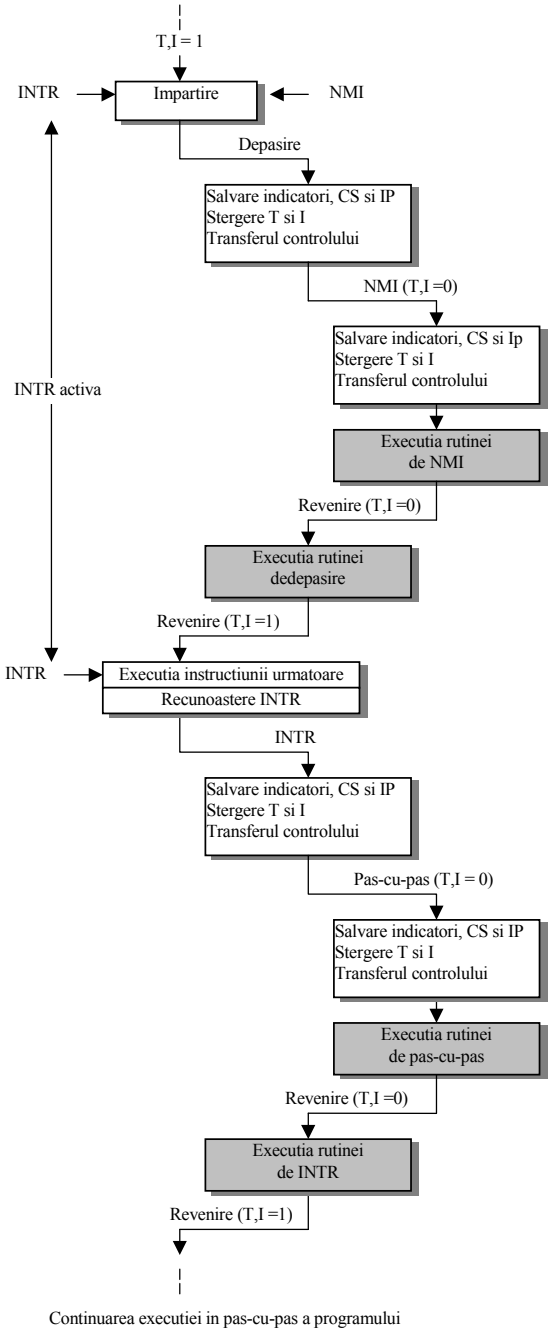


Figura 1.49. Apariţia simultană a întreruperilor NMI, INTR, pas-cu-pas şi întreruperii de depăşire

– în cadrul întreruperilor nemascabile, NMI, pas-cu-pas şi întreruperi software, întreruperea pas-cu-pas are prioritatea cea mai înaltă urmată de NMI şi întreruperile software. Această ordine de prioritate este valabilă numai în relaţiile dintre două tipuri de întreruperi nemascabile, ca în exemplul din figura 1.48. Excepţia apare în cazul în care toate cele trei întreruperi nemascabile sunt în aşteptare. În această situaţie ordinea de execuţie a acestor rutine este următoarea: NMI, întreruperea software, întreruperea pas-cu-pas. Aşadar funcţionarea pas-cu-pas se reia la execuţia instrucţiunii următoare instrucţiunii care a provocat întreruperea software, figura 1.49.

În legătură cu cererea DMA precizăm că apariţia simultană a unei cereri DMA şi a întreruperii INTR, de exemplu în timpul execuţiei unei instrucţiuni precedate de prefixul LOCK, se rezolvă prin servirea întâi a cererii DMA şi apoi a întreruperii, HOLD fiind deci prioritar faţă de INTR.

1.3.6. STAREA HALT, OPERAŢII CU $\overline{\text{LOCK}}$, SINCRONIZAREA EXTERNĂ CU AJUTORUL INTRĂRII $\overline{\text{TEST}}$

1.3.6.1. **Starea HALT a microprocesorului**

La execuţia unei instrucţiuni HALT, microprocesorul 8086 va indica intrarea în starea HALT, oprire, în funcţie de modul de lucru. În modul de lucru minim, 8086 va genera semnalul de eşantionare ALE fără nici-o comandă ($\overline{\text{RD}}$, $\overline{\text{WR}}$ sau $\overline{\text{INTA}}$), figura 1.12. În modul maxim, procesorul generează starea HALT la ieşirile $\overline{\text{S2}}$, $\overline{\text{S1}}$ şi $\overline{\text{S0}}$, §1.3.1.3 şi figura 1.24, ALE fiind emis de controlorul de magistrală 8288. Starea HALT nu este părăsită decât la apariţia unei întreruperi sau la iniţializare. Microprocesorul nu va ieşi din HALT la apariţia unei cereri HOLD de preluare a magistralei. În ultima situaţie, 8086 va regenera starea HALT.

1.3.6.2. **Operaţii de citire/modificare/scriere cu $\overline{\text{LOCK}}$**

Activarea semnalului $\overline{\text{LOCK}}$ se poate face cu ajutorul prefixului LOCK atunci când este necesară execuţia consecutivă a două cicluri de magistrală. Această necesitate poate apărea la execuţia unei instrucţiuni de interschimbare, XCHG, memorie/registru când procesorul trebuie să realizeze cu memoria o operaţie de citire/modificare/scriere, de exemplu a unui semafor, fără a fi perturbat. Semnalul $\overline{\text{LOCK}}$ este pus pe "0" în ciclul de ceas următor celui în

care UE decodifică codul-obiect al prefixului LOCK şi este dezactivat, pus pe "1", la sfârşitul ultimului ciclu de magistrală aparţinând instrucţiunii precedate de prefixul LOCK. Pe timpul cât $\overline{LOCK}$ este activ, în modul maxim, cererile de preluare a magistralei de la intrările RQ/GT sunt memorate pentru a fi achitate după dezactivarea lui $\overline{LOCK}$.

1.3.6.3. Sincronizarea externă cu ajutorul intrării $\overline{TEST}$

Pentru legătura cu exteriorul, microprocesorul 8086 mai poate testa prin program starea intrării $\overline{TEST}$ cu ajutorul instrucţiunii WAIT. Dacă instrucţiunea WAIT găseşte intrarea $\overline{TEST}$ inactivă, pe "1", procesorul intră în aşteptare executând în mod repetitiv instrucţiunea WAIT. Pe durata execuţiei acestei instrucţiuni, microprocesorul e în starea pasivă şi nu execută cicli de magistrală. Ieşirea din starea de aşteptare se face prin activarea, cel puţin 5 perioade de ceas, a intrării $\overline{TEST}$. Dacă în timpul aşteptării, execuţiei instrucţiunii WAIT, apare o cerere HOLD toate ieşirile microprocesorului trec în starea a treia. De asemenea, apariţia unei cereri de întrerupere conduce la suspendarea aşteptării, procesarea întreruperii şi revenirea la execuţia instrucţiunii de aşteptare WAIT. Pentru a reveni la WAIT procesorul va mai extrage o dată după apariţia întreruperii codul instrucţiunii WAIT pentru a putea memora în stivă adresa de revenire din rutina de serviciu a întreruperii.

1.4. FUNCŢIONAREA MICROPROCESORULUI 8088

Unitatea centrală 8088 are aceeşi structură internă ca 8086 majoritatea funcţiilor fiind identice, figurile 1.2 şi 1.3. 8088 manevrează magistralele de adrese, date şi comenzi în acelaşi mod cu 8086 cu deosebirea că schimbul de date se face pe 8 biţi. Operanzii de 16 biţi vor fi deci accesaţi în două cicluri de magistrală consecutivi. Cele două unităţi centrale sunt identice din punct de vedere al programatorului, diferenţele apărând numai la timpii de execuţie. De asemenea, structura registrelor este aceeaşi.

Prezentăm sintetic diferenţele dintre microprocesoarele 8088 şi 8086:

– lungimea cozii la 8088 este de 4 octeţi faţă de 6 cât este la 8086. Scurtarea s-a făcut cu scopul de a preveni ocuparea excesivă a magistralei de către UIM din 8088 datorită timpului suplimentar impus de accesul pe 8 biţi;

– în acelaşi timp cu micşorarea cozii, pentru optimizare, proiectanţii au modificat şi algoritmul de pre-extragere. UIM din 8088 va extrage o nouă

instrucţiune dacă în coadă este disponibil cel puţin un octet spre deosebire de 8086 unde trebuie să fie neocupaţi cel puţin doi octeţi;

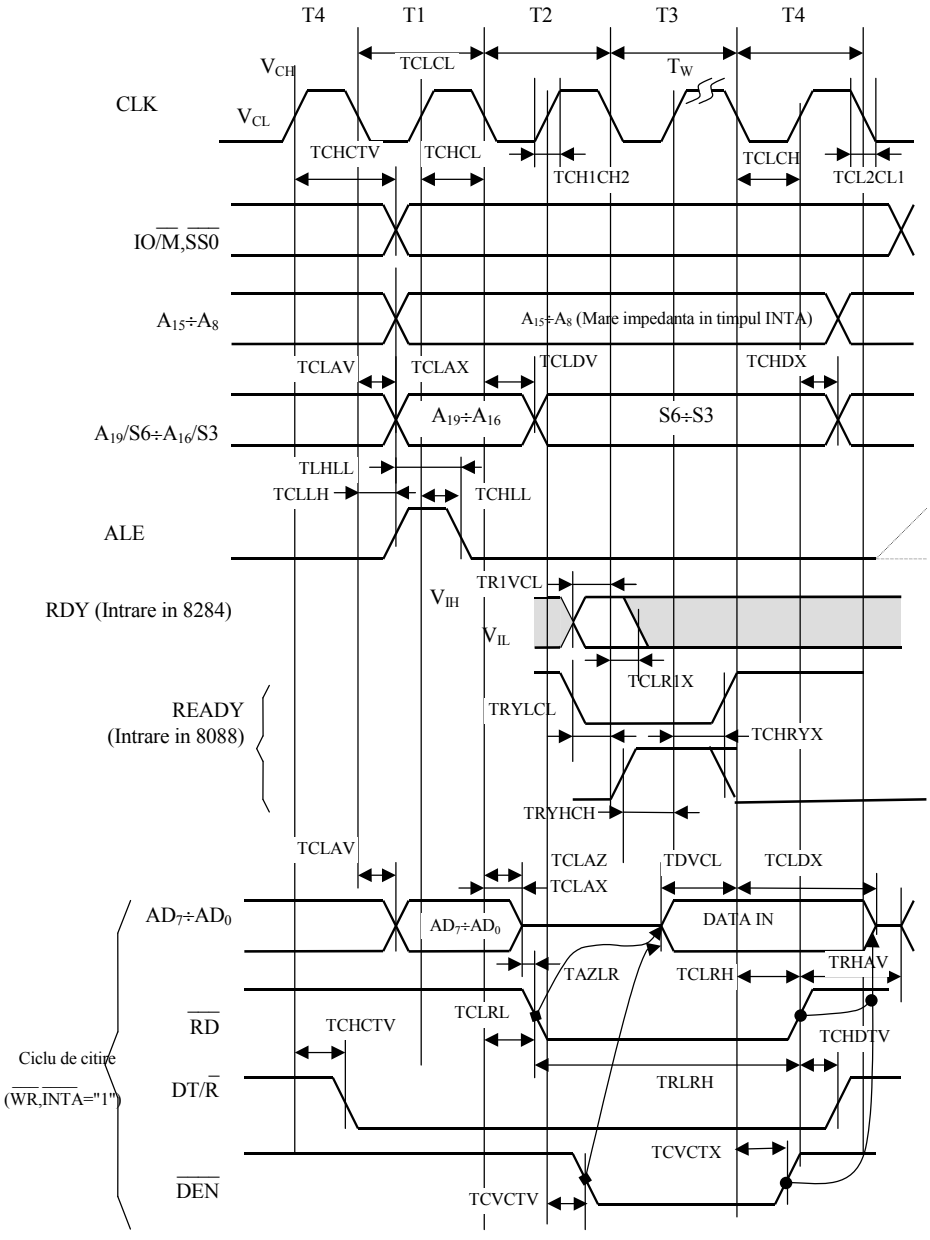


Figura 1.50. Diagrame de timp pentru 8088 în modul minim

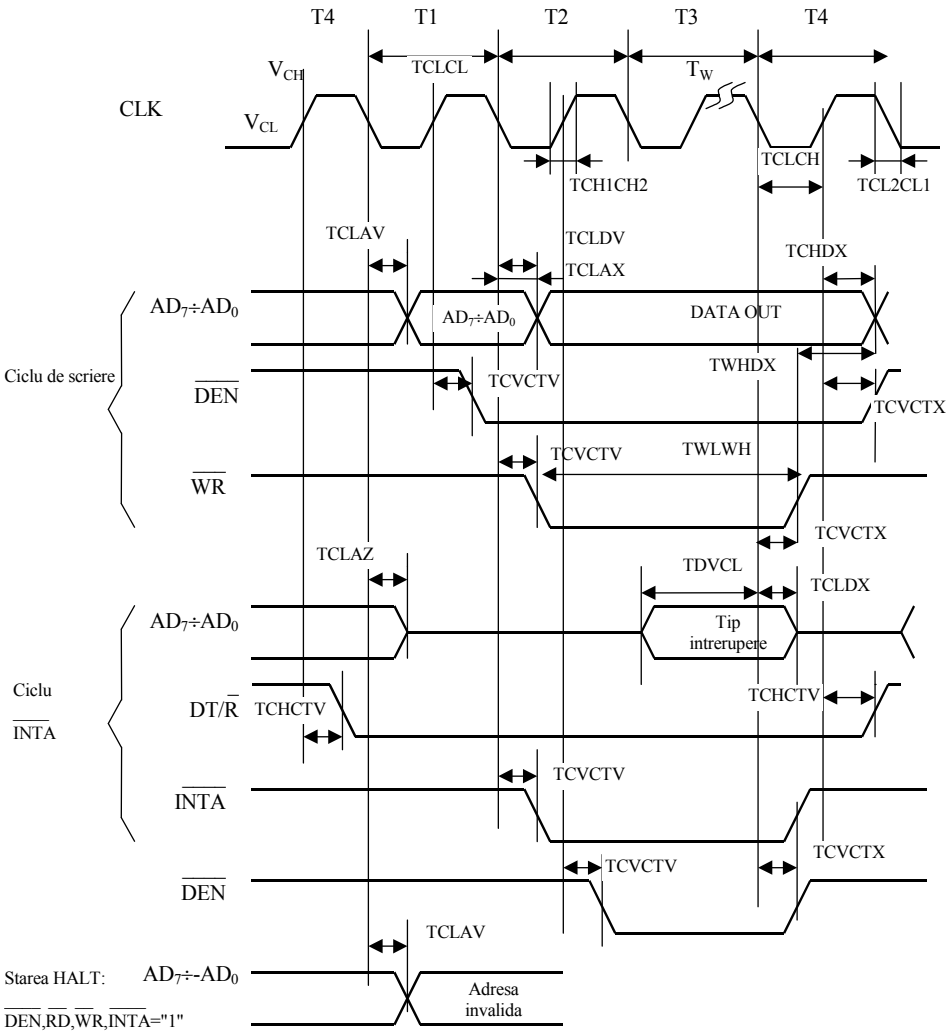


Figura 1.50. *Continuare*

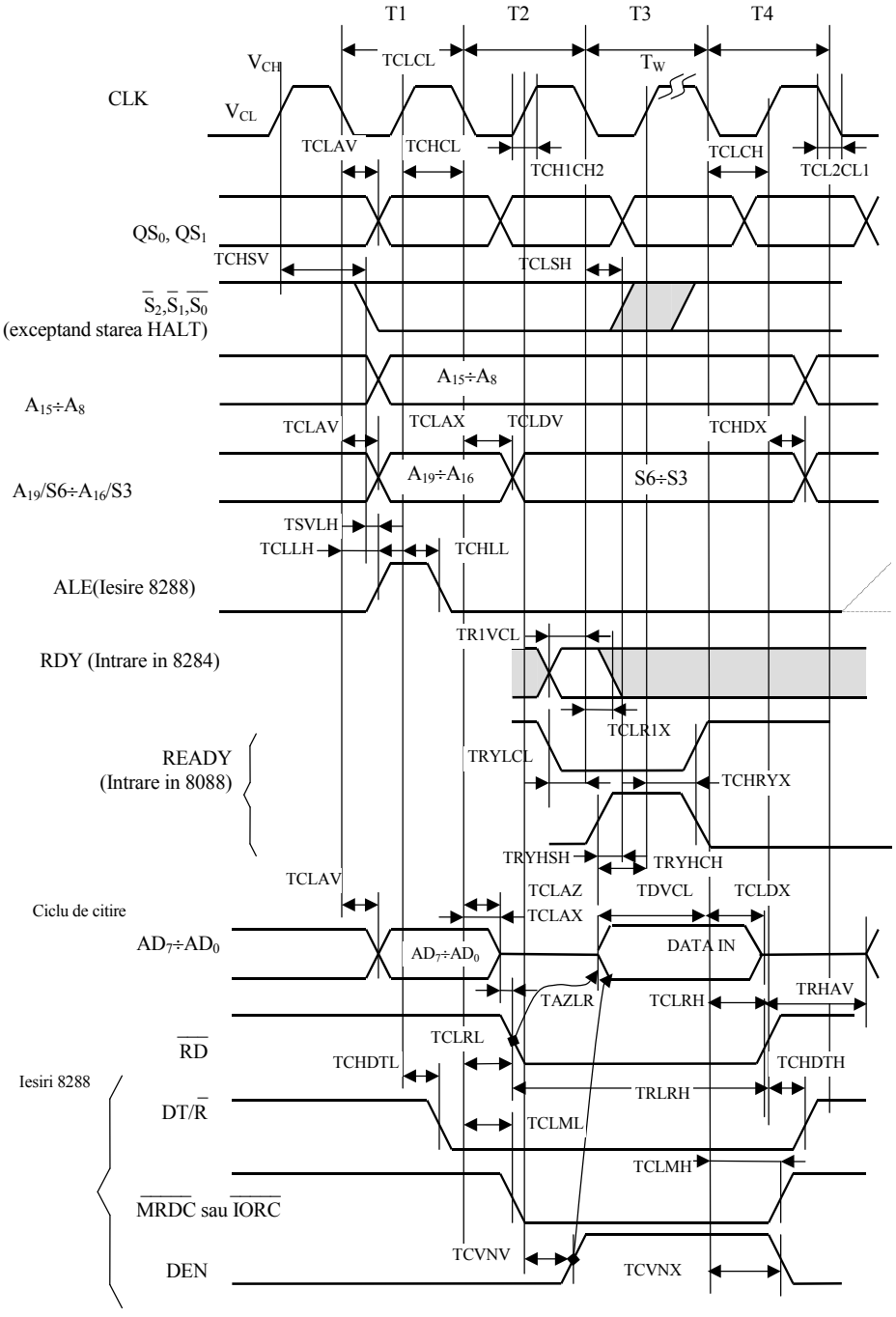


Figura 1.51. Diagrame de timp pentru 8088 în modul maxim

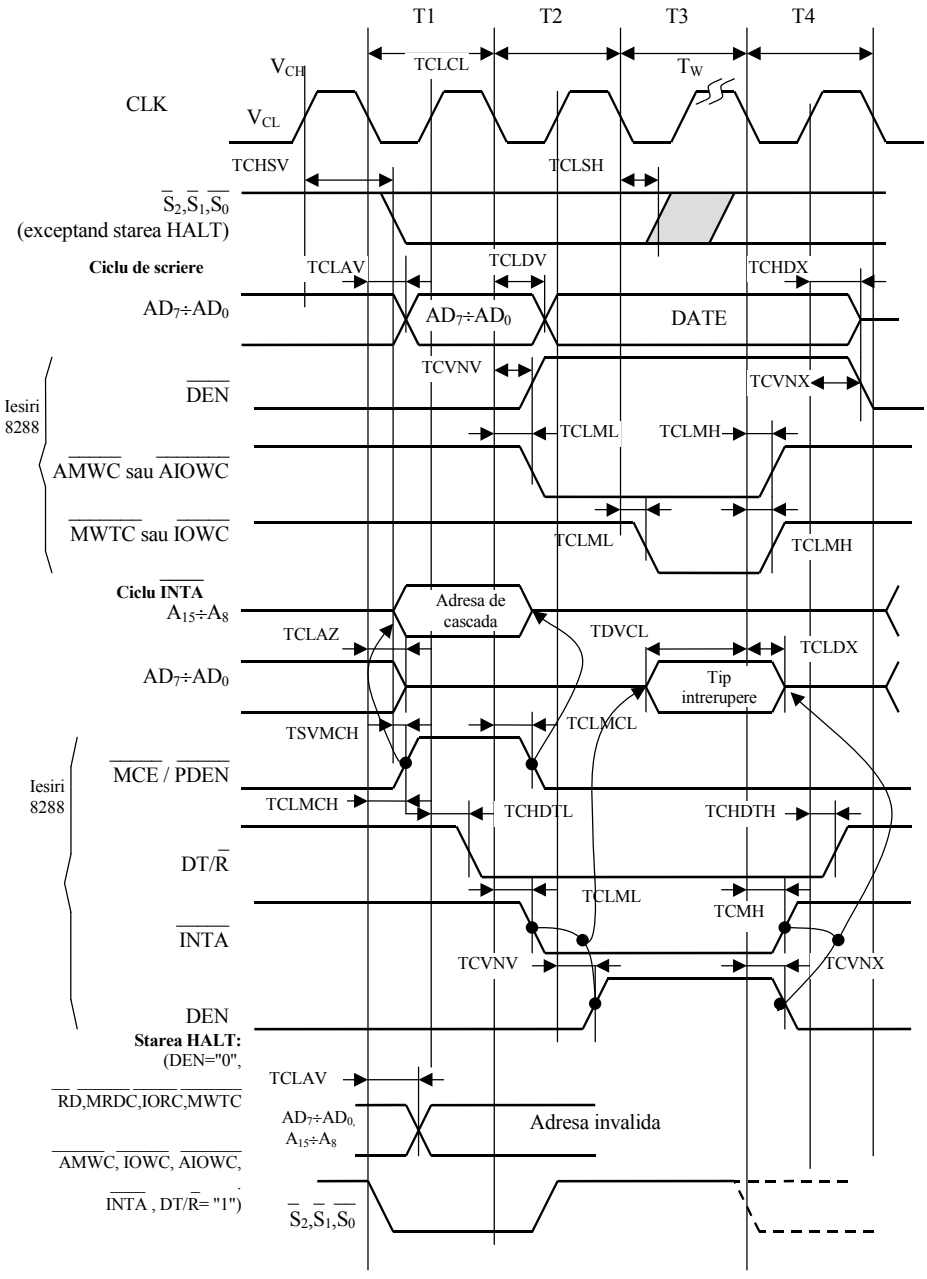


Figura 1.51. *Continuare*

– timpii de execuţie ai instrucţiunilor sunt afectaţi de accesul pe 8 biţi la resursele externe, memoria şi dispozitivele de I/O. Toate scrierile şi citirile operanzilor de 16 biţi se prelungesc cu încă un ciclu de magistrală. Viteza de execuţie este, de asemenea, micşorată şi de întârzierile apărute la extragerea codului. Această a două micşorare a vitezei de execuţie apare numai în cazul succesiunilor de operaţii simple. În situaţiile utilizării unor instrucţiuni mai sofisticate UIM din 8088 are timp să umple stiva şi execuţia nu va mai fi limitată decât de viteza UE din microprocesor.

Cum am mai spus 8088 şi 8086 sunt complet compatibile software deoarece au unităţile de execuţie identice. De aceea programele care nu sunt dependente de organizarea sistemului vor funcţiona corect pe amândouă microprocesoarele. Pe de altă parte, programele care depind de organizarea sistemului vor trebui revăzute pentru a putea fi transferate între 8088 şi 8086.

Conexiunile externe ale microprocesorului 8088 sunt aproape identice cu cele ale lui 8086, figura 1.7 şi 1.8, apărând la 8088 următoarele modificări funcţionale:

– biţii de adresă A8÷A15 sunt la 8088 numai ieşiri fiind *latch*-ate intern şi rămânând deci valizi pe durata întregului ciclu de magistrală, asemănător cu 8085;

– semnalul $\overline{\text{BHE}}$ a fost eliminat, nemaifiind necesar la 8088;

– ieşirea $\overline{\text{SS0}}$ asigură în modul minim informaţii de stare având semnificaţiile bitului de stare $\overline{\text{S0}}$, vezi §1.3.2.2. Această ieşire este activă numai în modul minim fiind "1" în modul maxim. Împreună cu $\text{IO} / \overline{\text{M}}$ şi $\text{DT} / \overline{\text{R}}$ codifică în modul de lucru minim starea ciclului de magistrală în curs;

– ieşirea $\text{IO} / \overline{\text{M}}$ a fost inversată pentru a se obţine o compatibilitate la nivel de magistrală cu microprocesorul 8085;

– la o instrucţiune HALT în modul minim, semnalul ALE este întârziat cu o perioadă de ceas pentru a se putea *latch*-a cu el starea microprocesorului.

Diagramele de timp pentru procesorul 8088 lucrând în modul minim sunt date în figura 1.50 iar pentru modul maxim în figura 1.51. Parametrii lui 8088 atât în modul minim cât şi în cel maxim, cerinţe şi răspunsuri în timp, sunt aceeşi ca ai lui 8086, tabelele 1.2, 1.3 şi 1.7, 1.8.

2

SETUL DE INSTRUCŢIUNI AL MICROPROCESOARELOR 8086/8088

2.1. GENERALITĂŢI. ARHITECTURA UNUI SET DE INSTRUCŢIUNI

Arhitectura unui de instrucţiuni este, aşa cum se cunoaşte, *partea vizibilă* a maşinii, accesibilă programatorului. Seturile de instrucţiuni se pot clasifica pe baza a cinci direcţii de structurare [15]:

– memorarea operanzilor în UC;
– numărul de operanzi explicit specificaţi într-o instrucţiune;
– localizarea operanzilor;
– tipurile de operaţii;
– tipurile şi dimensiunile operanzilor.

Memorarea operanzilor în UC diferenţiază arhitecturile în trei mari clase: arhitecturi de tip *stivă*, de tip *acumulator* şi de tip *registre generale* (*GPR – General Purpose Registers*). Maşinile mai vechi reprezintă arhitecturi de tip stivă sau acumulator în timp ce maşinile actuale au la bază arhitecturi cu registre generale. Această evoluţie are două motivaţii principale: întâi că registrele, ca şi alte structuri de memorare din cadrul UC, sunt mai rapide decât memoria, şi apoi, ele sunt mai uşor şi mai eficient de utilizat la compilare în comparaţie cu celelalte forme de memorare internă.

Arhitecturile GPR se pot diferenţia la rândul lor cu ajutorul a două caracteristici mari. Prima se referă la *numărul de operanzi ai unei instrucţiuni UAL*: doi sau trei. În formatul cu trei operanzi instrucţiunea conţine doi operanzi sursă şi un operand rezultat. În formatul cu doi operanzi unul dintre aceştia este atât sursă cât şi destinaţie. A doua diferenţiere se referă la *numărul de operanzi dintr-o instrucţiune UAL ce pot fi adrese de memorie*. Acest număr poate varia pentru o instrucţiune UAL tipică între 0 şi 3. Combinaţiile celor două caracteristici sunt exemplificate în tabelul 2.1.

Tabelul 2. 1. Arhitecturi de tip "registru general" – *GPR*

Număr de adrese de memorie în instrucţiuni UAL	Număr maxim de operanzi în instrucţiuni UAL	Exemple de maşini
0	2	IBM RT-PC
	3	SPARC, MIPS, HP Precision
1	2	PDP-10, 68000, IBM 360, 8086
	3	IBM 360 (instrucţiunile RS)
2	2	PDP-11, 32X32, IBM 360 (instr. SS)
	3	–
3	3	VAX (are şi formate de doi operanzi)

Deşi, aşa cum se poate observa, sunt şapte combinaţii posibile, numai trei sunt considerate tipice şi permit clasificarea majorităţii maşinilor de calcul actuale. Acestea sunt denumite *registru-registru* (sau *încărcare/memorare – load/store*), *registru-memorie* şi *memorie-memorie*. Fiecare din cele trei tipuri de arhitecturi are avantajele şi dezavantajele sale. Astfel, arhitecturile registru-registru sunt în general simple, au instrucţiuni de lungime fixă care se execută de obicei în acelaşi număr de ceasuri. Dezavantajul lor constă în aceea că programele vor avea un număr de instrucţiuni mai mare. Arhitecturile registru-memorie prezintă avantajul accesării directe a datelor din registre, fără încărcări prealabile din memorie. Dezavantajul lor principal apare datorită neechivalenţei operanzilor, cel care este sursă şi destinaţie fiind distrus în urma unei operaţii binare. Maşinile memorie-memorie sunt cele mai compacte şi pot să nu mai folosească registre pentru memorări temporare. Pe de altă parte aceste arhitecturi au instrucţiuni de lungimi şi complexităţi variabile, care se pot executa în perioade de timp foarte diferite. Eficienţa acestor ultime tipuri de maşini poate fi sever limitată de accesul la memorie, *gâtuitura von Neumann*. Avantajele şi dezavantajele arhitecturale de mai sus sunt calitative, nu au un caracter absolut, şi, în ultimă instanţă, impactul lor real depinde atât de calitatea compilatoarelor cât şi de realizarea hardware a maşinii.

8086, este, aşa cum am spus deja în §1.1, o arhitectură hibridă acumulator/registre generale, cu instrucţiuni UAL tipice cu doi operanzi dintre care unul poate fi adresă de memorie. Setul de instrucţiuni prevede operaţii atât pe 8 biţi, octeţi, cât şi pe 16 biţi, cuvinte. Această distincţie referitoare la tipul operandului apare atât în operaţiile cu registre cât şi în accesele la memorie.

Aşa cum am văzut în primul capitol, spaţiul de adrese al microprocesorului 8086, deşi este de 20 de biţi, este "spart" în segmente de 64kB adresabile cu ajutorul unor deplasamente de 16 biţi. Adresa de 20 de biţi este astfel formată dintr-o adresă efectivă de 16 biţi – *offset*-ul sau

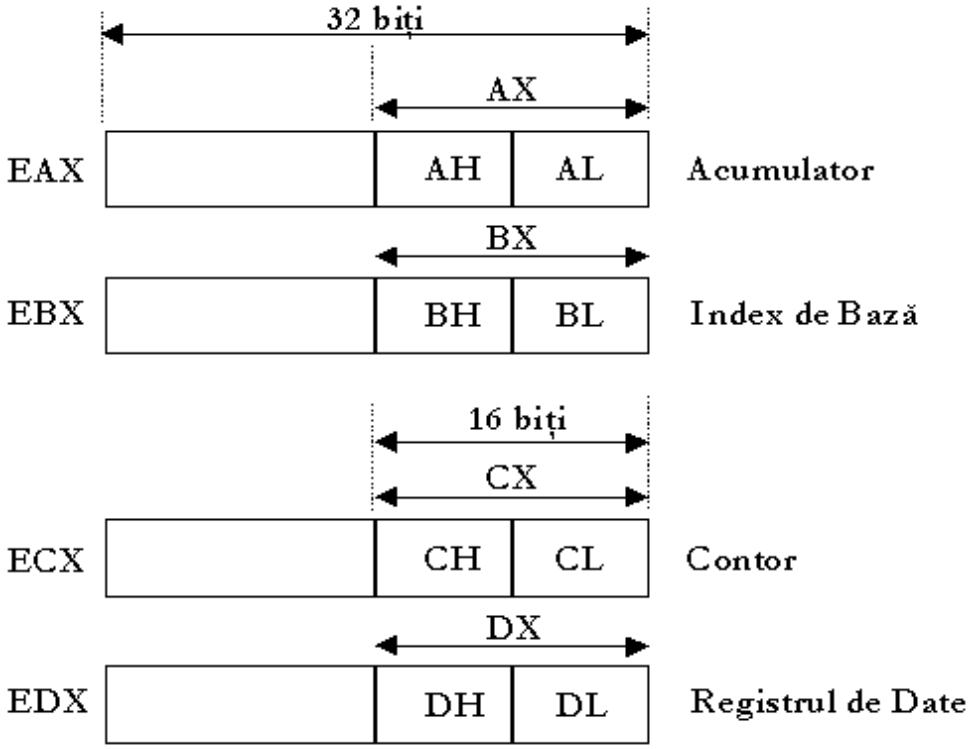
deplasamentul în cadrul unui segment –, care se adaugă la o altă adresă pe 16 biţi – baza segmentului. Adresa bazei segmentului se obţine prin deplasarea stânga cu 4 poziţii a conţinutului unui registru de segment de 16 biţi.

Cele 14 registre ale microprocesorului 8086, prezentate pe larg în prima parte a lucrării, figura 1.4, pot fi împărţite din punctul de vedere al arhitecturii setului de instrucţiuni în patru clase: registre de date (AX, BX, CX şi DX), registre de adresare (SP, BP, SI, DI), registre de segment (CS, SS, DS, ES) şi registre de control (IP, Starea). Registrele de date sunt utilizate pentru memorarea datelor şi operaţii asupra lor, registrele de adresare sunt folosite pentru formarea adreselor de memorie efective de 16 biţi (în cadrul segmentelor), registrele de segment pentru formarea unei adrese de memorie reale pe 20 de biţi iar registrele de control pentru păstrarea stării maşinii şi controlul programului.

Introducere în limbajul de asamblare

Ne vom referi în cele ce urmează la familia de microprocesoare intitulată iAPx86 ce stau la baza calculatoarelor IBM PC, începând de la procesoarele 8088 şi 8086, continuând cu 80286, 80386, 80486, Pentium, ş.a.m.d. Procesorul 8086 reprezintă, de fapt, baza familiei ce este cunoscută pe scurt sub denumirea de familia microprocesoarelor x86. De aceea se vor face referiri în continuare la această arhitectură (8086).

Elementele arhitecturale de bază ale microprocesorului



Notă:
Regiştrii pe 32 de biţi nu apar la 8086, 8088, 80286

Figura 1. Regiştrii de uz general – acumulator, index de bază, contor şi de date

Regiştrii microprocesorului

Regiştrii (sau registrele) microprocesorului reprezintă locaţii de memorie speciale aflate direct pe cip; din această cauză reprezintă cel mai rapid tip de memorie. Alt lucru deosebit legat de regiştri este faptul că fiecare dintre aceştia au un scop bine precizat, oferind anumite funcţionalităţi speciale, unice. Există patru mari

categorii de regiştri: regiştrii de uz general, registrul indicatorilor de stare (*flags*), regiştrii de segment şi registrul pointer de instrucţiune.

Regiştrii de uz general

Regiştrii de uz general (vezi figura 1 şi figura 2) sunt implicaţi în operarea majorităţii instrucţiunilor, drept operanzi sursă sau destinaţie pentru calcule, copieri de date, pointeri la locaţii de memorie sau cu rol de contorizare. Fiecare dintre cei 8 regiştri de uz general AX, BX, CX, DX, SP, BP, DI,SI sunt regiştri pe 16 biţi pentru microprocesorul 8086, iar de la procesorul 80386 încoace au devenit regiştri pe 32 de biţi, denumiţi, respectiv: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI (litera E provine de la *Extended – extins* în engleză). Mai mult, cei mai puţin semnificativi 8 biţi ai regiştrilor AX, BX, CX, DX formează respectiv regiştrii AL, BL, CL, DL (litera L provine de la *Low – jos* în engleză), iar cei mai semnificativi 8 biţi ai aceloraşi regiştri formează regiştrii AH, BH, CH, DH (litera H provine de la *High – înalt* în engleză) (figura 1).

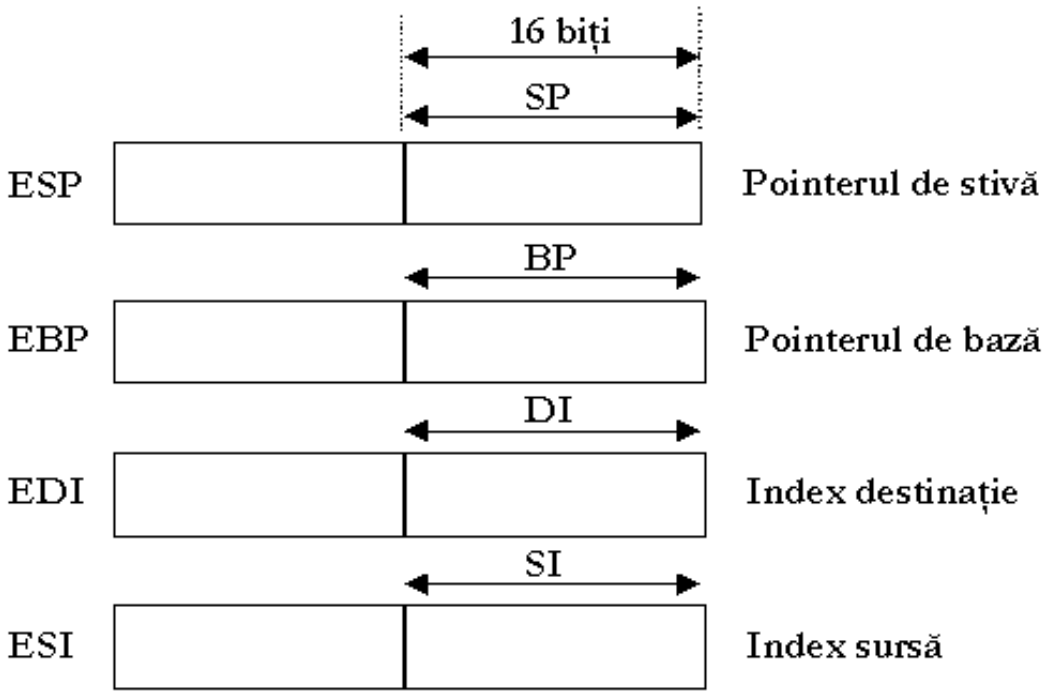


Figura 2. Regiştrii de uz general index şi pointer

Ne vom concentra în continuare atenţia asupra regiştrilor generali pe 16 biţi; fiecare dintre aceştia poate stoca o valoare pe 16 biţi, poate fi folosit pentru stocarea unei valori din memorie sau poate fi utilizat pentru operaţii aritmetice şi logice. Spre exemplu, următoarele instrucţiuni:

```
…
MOV  BX, 2
MOV  DX, 3
```

```
ADD  BX, DX
…
```

încarcă valoarea 2 în registrul BX, valoarea 3 în registrul DX, adună cele două valori iar rezultatul (5) este memorat în registrul BX. În exemplul anterior putem utiliza oricare dintre regiştrii de uz general în locul regiştrilor BX şi DX. În afara proprietăţii de a stoca valori şi de a folosi drept operanzi sursă sau destinaţie pentru instrucţiunile de manipulare a datelor, fiecare dintre cei 8 regiştri de uz general au propria "personalitate". Vom vedea în continuare care sunt caracteristicile specifice fiecăruia dintre regiştrii de uz general.

Registrul AX (EAX)

Registrul AX (EAX) este denumit şi registrul *acumulator*, fiind principalul registru de uz general utilizat pentru operaţii aritmetice, logice şi de deplasare de date. Totdeauna operaţiile de înmulţire şi împărţire presupun implicarea registrului AX. Unele dintre instrucţiuni sunt optimizate pentru a se executa mai rapid atunci când este folosit AX. În plus, registrul AX este folosit şi pentru toate transferurile de date de la/către porturile de Intrare/Ieşire. Poate fi accesat pe porţiuni de 8, 16 sau 32 de biţi, fiind referit drept AL (cei mai puţin semnificativi 8 biţi din AX), AH (cei mai semnificativi 8 biţi din AX), AX (16 biţi) sau EAX (32 de biţi). Prezentăm în continuare alte câteva exemple de instrucţiuni ce utilizează registrul AX. De remarcat este faptul că transferurile de date se fac pentru instrucţiunile (denumite şi *mnemonice*) Intel de la dreapta spre stânga, exact invers decât la Motorola (vom vedea şi alt exemplu asemănător la scrierea datelor în memorie sub format diferit la Motorola faţă de Intel), unde transferul se face de la stânga la dreapta.

Instrucţiunea: **MOV AX, 1234H** încarcă valoarea 1234H (4660 în zecimal) în registrul acumulator AX. După cum spuneam, cei mai puţini semnificativi 8 biţi ai registrului AX sunt identificaţi de AL (A-Low) iar cei mai semnificativi 8 biţi ai aceluiaşi registru sunt identificaţi ca fiind AH (A-High). Acest lucru este utilizat pentru a lucra cu date pe un octet, permiţând ca registrul AX să fie folosit pe postul a doi regiştri separaţi (AH şi AL). Aceeaşi regulă este valabilă şi pentru regiştrii de uz general BX, CX, DX. Următoarele trei instrucţiuni setează registrul AH cu valoarea 1, incrementează cu 1 această valoare şi apoi o copiază în registrul AL:

```
MOV  AH, 1
INC  AH
MOV AL, AH
```

Valoarea finală a registrului AX va fi 22 (AH = AL = 2).

Registrul BX (EBX)

Registrul BX (Base), sau registrul de bază poate stoca adrese pentru a face referire la diverse structuri de date, cum ar fi vectorii stocaţi în memorie. O valoare reprezentată pe 16 biţi stocată în registrul BX poate fi utilizată ca fiind o porţiune din adresa unei locaţii de memorie ce va fi accesată. Spre exemplu, următoarele instrucţiuni încarcă registrul AH cu valoarea din memorie de la adresa 21.

```
MOV  AX,  0
MOV  DS, AX
MOV  BX, 21
MOV  AH, [ BX ]
```

Se observă că am încărcat valoarea 0 în registrul DS înainte de a accesa locaţia de memorie referită de registrul BX. Acest lucru este datorat segmentării memoriei (segmentare discutată mai în detaliu în secţiunea consacrată regiştrilor de segment); implicit, atunci când este folosit ca pointer de memorie, BX face referire relativă la registrul de segment DS (adresa la care face referire este o adresă relativă la adresa de segment conţinută în registrul DS).

Registrul CX (ECX)

Specializarea registrului CX (Counter) este numărarea; de aceea, el se numeşte şi registrul contor. De asemenea, registrul CX joacă un rol special atunci când se foloseşte instrucţiunea LOOP. Rolul de contor al registrului CX se observă imediat din exemplul următor:

```
MOV CX, 5
start:
…
<instrucţiuni ce se vor executa de 5 ori>
…
SUB CX, 1
JNZ start
```

Deoarece valoarea iniţială a lui CX este 5, instrucţiunile cuprinse între eticheta start şi instrucţiunea JNZ se vor executa de 5 ori (până când registrul CX devine 0). Instrucţiunea SUB CX, 1 decrementează registrul CX cu valoarea 1 iar instrucţiunea JNZ start determină saltul înapoi la eticheta start dacă CX nu are valoarea 0. În limbajul microprocesorului există şi o instrucţiune specială legată de ciclare. Aceasta este instrucţiunea LOOP, care este folosită în combinaţie cu registrul CX. Liniile de cod următoare sunt echivalente cu cele anterioare, dar aici se utilizează instruxţiunea LOOP:

```
MOV CX, 5
start:
…
<instrucțiuni ce se vor executa de 5 ori>
…
LOOP start
```

Se observă că instrucțiunea LOOP este folosită în locul celor două instrucțiuni SUB şi JNZ anterioare; LOOP decrementează automat registrul CX cu 1 şi execută saltul la eticheta specificată (*start*) dacă CX este diferit de zero, totul într-o singură instrucțiune.

Registrul DX (EDX)

Registrul de uz general DX (Data register), denumit şi registrul de date, poate fi folosit în cazul transferurilor de date Intrare/Ieşire sau atunci când are loc o operație de înmulțire sau de împărțire. Instrucțiunea **IN AL, DX** copiază o valoare de tip Byte dintr-un port de intrare, a cărui adresă se află în registrul DX. Următoarele instrucțiuni determină scrierea valorii 101 în portul I/O 1002:

```
...
MOV  AL, 101
MOV  DX, 1002
OUT  DX, AL
…
```

Referitor la operațiile de înmulțire şi împărțire, atunci când împărțim un număr pe 32 de biți la un număr pe 16 biți, cei mai semnificativi 16 biți ai deîmpărțitului trebuie să fie în DX. După împărțire, restul împărțirii se va afla în DX. Cei mai puțin semnificativi 16 biți ai deîmpărțitului trebuie să fie în AX iar câtul împărțirii va fi în AX. La înmulțire, atunci când se înmulțesc două numere pe 16 biți, cei mai semnificativi 16 biți ai produsului vor fi stocați în DX iar cei mai puțin semnificativi 16 biți în registrul AX.

Registrul SI

Registrul SI (Source Index) poate fi folosit, ca şi BX, pentru a referi adrese de memorie. De exemplu, secvența de instrucțiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
MOV  AL, [ SI ]
```

Încarcă valoarea (pe 8 biţi) din memorie de la adresa 33 în registrul AL. Registrul SI este, de asemenea, foarte folositor atunci când este utilizat în legătură cu instrucţiunile dedicate tipului string (şir de caractere). Secvenţa următoare :

```
CLD
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
LODSB
```

nu numai că încarcă registrul AX cu valoarea de la adresa de memorie referită de registrul SI, dar adună, de asemenea, valoarea 1 la SI. Acest lucru este deosebit de eficient atunci când se accesează secvenţial o serie de locaţii de memorie, cum ar fi şirurile de caractere. Instrucţiunile de tip string se pot repeta de mai multe ori, astfel încât o singură instrucţiune poate avea ca efect sute sau mii de operaţii.

Registrul DI

Registrul DI (Destination Index) este utilizat în mod asemănător registrului SI. În secvenţa de instrucţiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  DI, 1000
ADD  BL, [ DI ]
```

se adună la registrul BL valoarea pe 8 biţi stocată la adresa 1000. Registrul DI este puţin diferit faţă de registrul SI în cazul instrucţiunilor de tip string; dacă SI este întotdeauna pe post de pointer sursă de memorie, registrul DI serveşte drept pointer destinaţie de memorie. Mai mult, în cazul instrucţiunilor de tip string, registrul SI adresează memoria relativ la registrul de segment DS, în timp ce DI conţine referiri la memorie relativ la registrul de segment ES. În cazul în care SI şi DI sunt utilizaţi cu alte instrucţiuni, ei fac referire la registrul de segment DS.

Registrul BP

Pentru a înţelege mai bine rolul regiştrilor BP şi SP, a sosit momentul să spunem câteva lucruri despre porţiunea de memorie denumită stivă (în engleză *stack*). Stiva (vezi figura 3) reprezintă o porţiune specială de locaţii adiacente din memorie. Aceasta este conţinută în cadrul unui segment de memorie şi identificată de un selector de segment memorat în registrul SS (cu excepţia cazului în care se foloseşte modelul nesegmentat de memorie în care stiva poate fi localizată oriunde în spaţiul de adrese liniare al programului). Stiva este o porţiune a memoriei unde valorile pot

fi stocate şi accesate pe principul LIFO (Last In – First Out), drept urmare ultima valoare stocată în stivă este prima ce va fi citită din stivă. De regulă, stiva este utilizată la apelul unei proceduri sau la întoarcerea dintr-un apel de procedură (principalele instrucţiuni folosite sunt CALL şi RET).

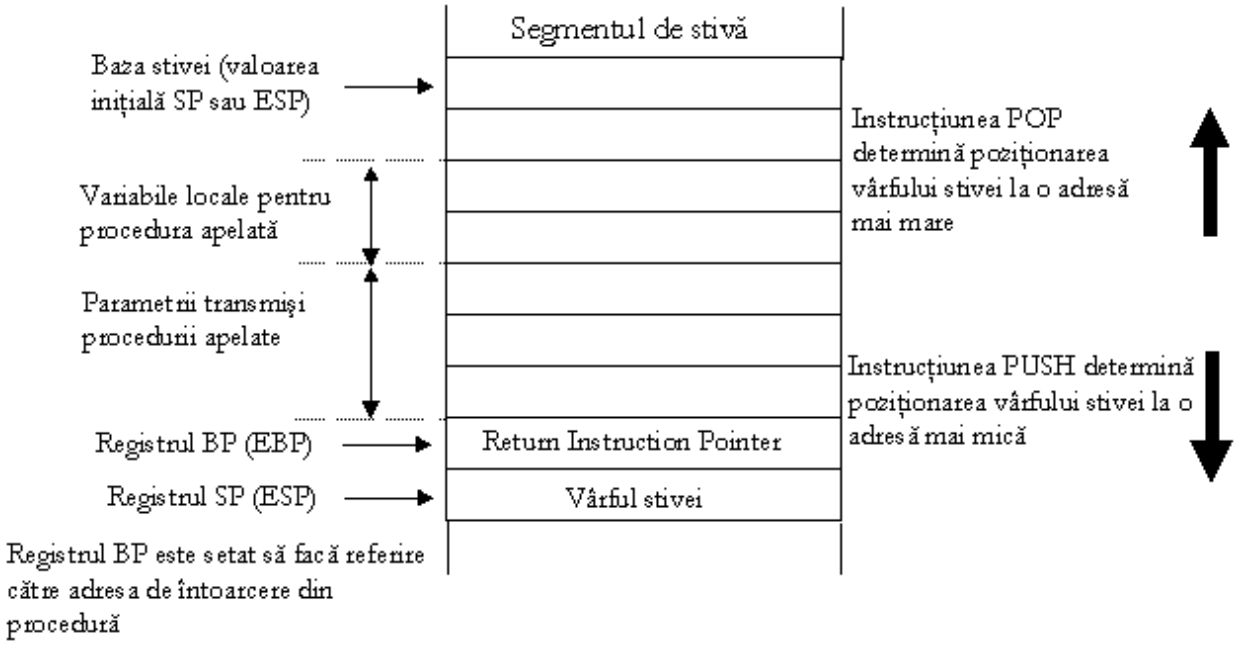


Figura 3. Structura stivei

Registrul pointer de bază, BP (Base Pointer) poate fi utilizat ca pointer de memorie precum regiştrii BX, SI şi DI. Diferența este aceea că, dacă BX, SI şi DI sunt utilizaţi în mod normal ca pointeri de memorie relativ la segmentul DS, registrul BP face referire relativ la segmentul de stivă SS. Principiul este următorul: o modalitate de a trece parametrii unei subrutine este aceea de a utiliza stiva (acest lucru se întâmplă în mod obişnuit în limbajele de nivel înalt, C sau Pascal, spre exemplu). Dacă stiva se află în porțiunea de memorie referită de registrul de segment SS (Stack Segment), datele se află în mod normal în segmentul de memorie referit de către DS, registrul segment de date. Deoarece BX, SI şi DI se referă la segmentul de date, nu există o modalitate eficientă de a folosi regiştrii BX, SI, DI pentru a face referire la parametrii salvaţi în stivă din cauză că stiva este localizată într-un alt segment de memorie. Registrul BP oferă rezolvarea acestei probleme asigurând adresarea în segmentul de stivă. Spre exemplu, instrucţiunile:

```
PUSH  BP
MOV  BP, SP
MOV  AX, [ BP+4 ]
```

fac să se acceseze segmentul de stivă pentru a încărca registrul AX cu primul parametru trimis de un apel C unei rutine scrise în limbaj de asamblare. În concluzie, registrul BP este conceput astfel încât să ofere suport pentru accesul la parametri, variabile locale şi alte necesităţi legate de accesul la porţiunea de stivă din memorie.

Registrul SP

Registrul SP (Stack Pointer), sau pointerul de stivă, reţine de regulă adresa de deplasament a următorului element disponibil în cadrul segmentului de stivă. Acest registru este, probabil, cel mai puţin „general" dintre regiştrii de uz general, deoarece este dedicat mai tot timpul administrării stivei.

Registrul BP face în fiecare clipă referire la vârful stivei – acest vârf al stivei reprezintă adresa locaţiei de memorie în care va fi introdus următorul element în stivă. Acţiunea de a introduce un nou element în stivă se numeşte „împingere" (în engleză *push*); de aceea, instrucţiunea respectivă poartă numele de PUSH. În mod asemănător, operaţia de scoatere a unui element din vârful stivei poartă, în engleză, numele de *pop*, iar instrucţiunea echivalentă operaţiei se numeşte POP. În figurile 3 şi 4 sunt ilustrate modificările survenite în conţinutul stivei şi al regiştrilor SP, BX şi CX ca urmare a execuţiei instrucţiunilor următoare (se presupune că registrul SP are iniţial valoarea 1000):

```
MOV  BX, 9
PUSH  BX
MOV  CX, 10
PUSH  CX
POP  BX
POP  CX
```

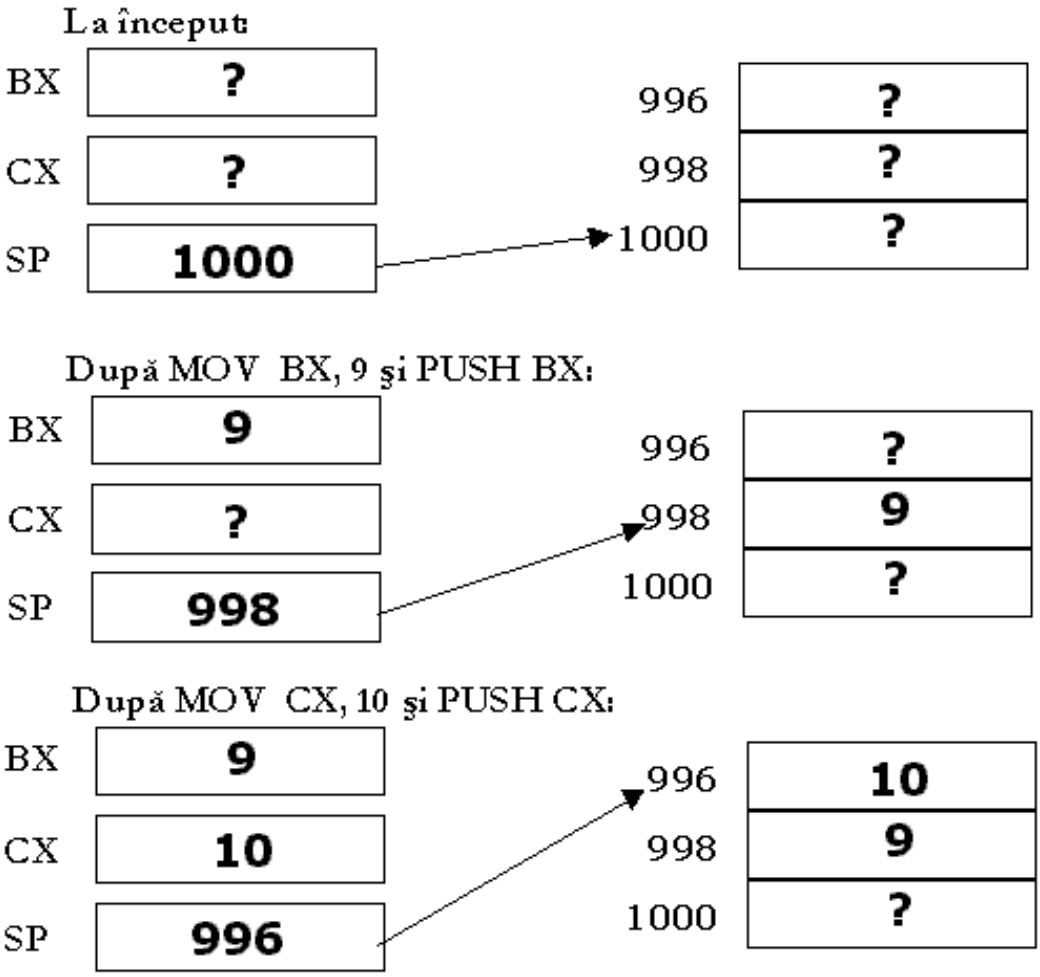


Figura 3. Modalitatea de funcţionare a stivei după execuţia primelor 4 instrucţiuni

Este permisă stocarea valorilor în registrul SP precum şi modificarea valorii sale prin adunare sau scădere la fel ca şi în cazul celorlalţi regiştri de uz general; totuşi, acest lucru nu este recomandat dacă nu suntem foarte siguri de ceea ce facem. Prin modificarea registrului SP, vom modifica adresa de memorie a vârfului stivei, ceea ce poate avea efecte neprevăzute, aceasta pentru că instrucţiunile PUSH şi POP nu reprezintă unicele modalităţi de utilizare a stivei. Indiferent dacă apelăm o subrutină sau ne întoarcem dintr-un astfel de apel de subrutină, fie procedură sau funcţie, în acest caz este folosită stiva. Unele resurse de sistem, precum tastatura sau ceasul de sistem, pot folosi stiva în momentul trimiterii unei întreruperi la microprocesor. Acest lucru presupune că stiva este folosită continuu, deci dacă se modifică registrul SP (adică adresa stivei), datele din noile locaţii de memorie nu vor mai fi cele corecte. În concluzie, registrul SP nu trebuie modificat în mod direct; el este modificat automat în urma instrucţiunilor POP, PUSH, CALL, RET. Oricare dintre ceilalţi regiştri de uz general pot fi modificaţi în mod direct în orice moment.

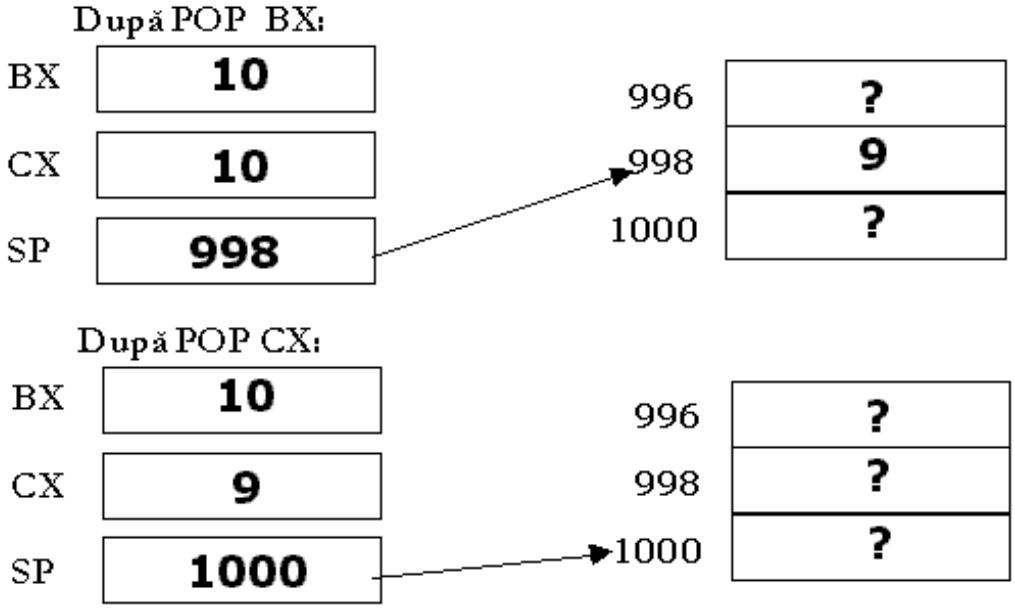


Figura 4. Funcţionarea stivei după ultimile două instrucţiuni POP

Registrul pointer de instrucţiuni (IP)

Registrul pointer de instrucţiuni (IP – Instruction Pointer, vezi figura 5) este folosit, întotdeauna, pentru a stoca adresa următoarei instrucţiuni ce va fi executată de către microprocesor. Pe măsură ce o instrucţiune este executată, pointerul de instrucţiune este incrementat şi se va referi la următoarea adresă de memorie (unde este stocată următoarea instrucţiune ce va fi executată). De regulă, instrucţiunea ce urmează a fi executată se află la adresa imediat următoare instrucţiunii ce a fost executată, dar există şi cazuri speciale (rezultate fie din apelul unei subrutine prin instrucţiunea CALL, fie prin întoarcerea dintr-o subrutină, prin instrucţiunea RET). Pointerul de instrucţiuni nu poate fi modificat sau citit în mod direct; doar instrucţiuni speciale pot încărca acest registru cu o nouă valoare. Registrul pointer de instrucţiune nu specifică pe de-a întregul adresa din memorie a următoarei instrucţiuni ce va fi executată, din aceeaşi cauză a segmentării memoriei. Pentru a aduce o instrucţiune din memorie, registrul CS oferă o adresă de bază iar registrul pointer de instrucţiune indică adresa de deplasament plecând de la această adresă de bază.

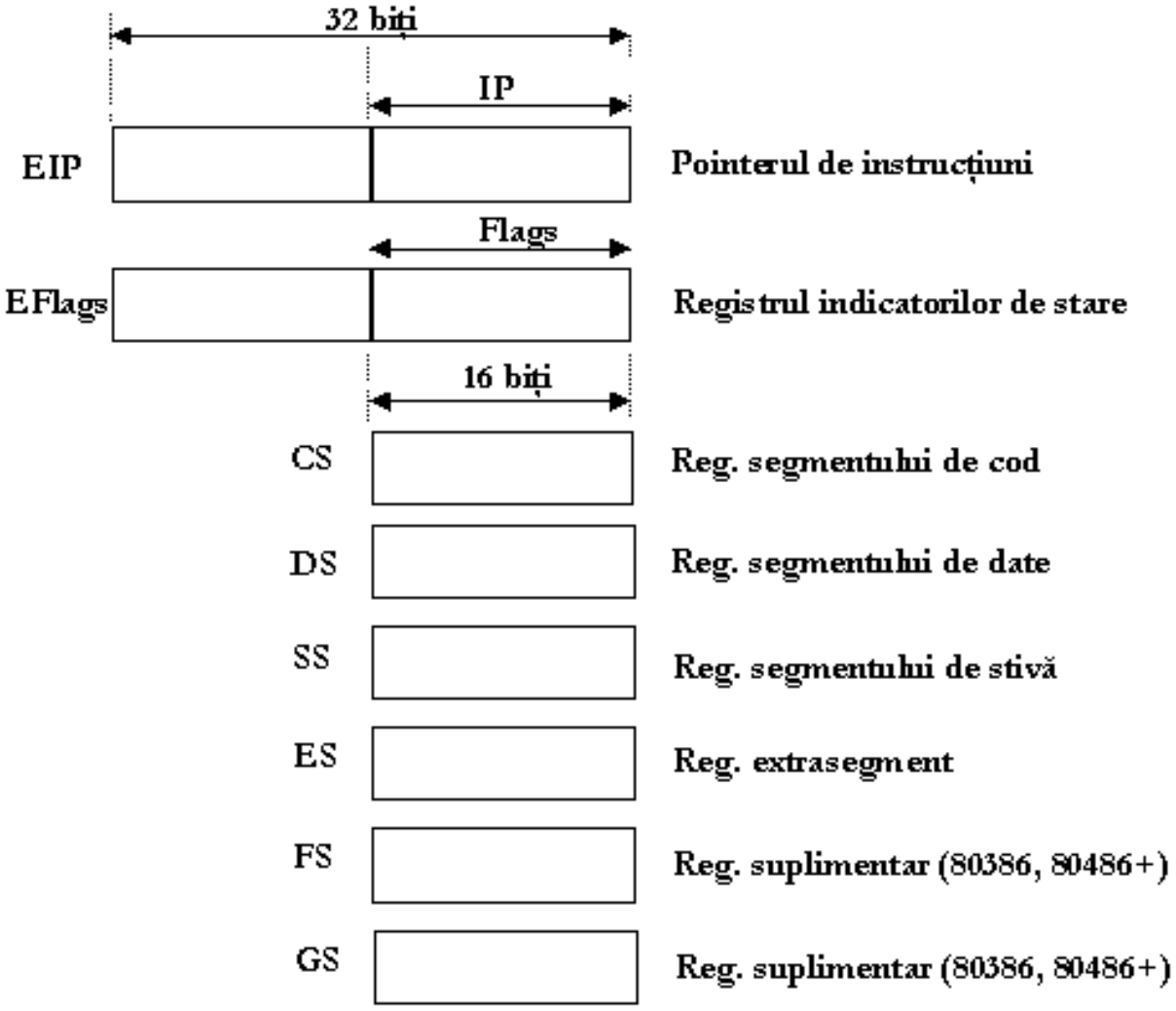


Figura 5. Regiştrii de segment, pointerul de instrucţiuni şi registrul indicatorilor de stare

Registrul indicatorilor de stare (FLAGS)

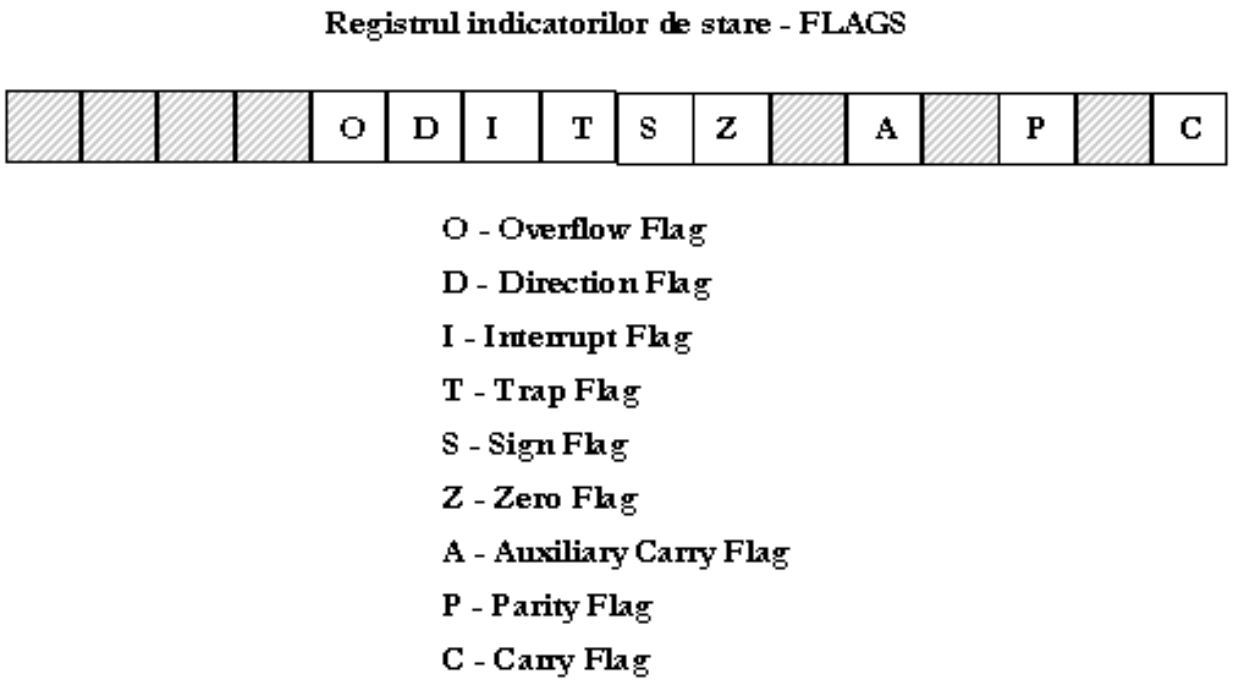


Figura 6. Registrul indicatorilor de stare - detaliu

Registrul indicatorilor de stare (FLAGS) pe 16 biţi conţine informaţii legate de starea microprocesorului precum şi de rezultatele ultimilor instrucţiuni executate. Un indicator de stare (*flag*) este în sine o locaţie de memorie de 1 bit ce indică starea curentă a microprocesorului şi modalitatea sa de operare. Un indicator se spune că "este setat" dacă are valoarea 1 şi "nu este setat" în caz contrar. Indicatorii de stare se modifică după execuţia unor instrucţiuni aritmetice sau logice. Exemple de indicatori de stare (vezi figura 6):

- C (Carry) indică apariţia unei cifre binare de transport în cazul unei adunări sau împrumut în cazul unei scăderi;
- O (Overflow) apare în urma unei operaţii aritmetice. Dacă este setat, înseamnă că rezultatul nu încape în operandul destinaţie;
- Z (Zero) indică faptul că rezultatul unei operaţii aritmetice sau logice ste zero;
- S (Sign) indică semnul rezultatului unei operaţii aritmetice;
- D (Direction) – când este zero, procesarea elementelor şirului se face de la adresa mai mică la cea mai mare, în caz contrar este invers;
- I (Interrupt) controlează posibilitatea microprocesorului de a răspunde la evenimente externe (apeluri de întrerupere);
- T (Trap) este folosit de programele de depanare (de tip debugger), activând sau nu posibilitatea execuţiei programului pas cu pas. Dacă este setat, UCP

întrerupe fiecare instrucțiune, lăsând programul depanator să execute programul respectiv pas cu pas;
- A (Auxiliary carry) suportă operații în codul BCD. Majoritatea programelor nu oferă suport pentru reprezentarea numerelor în acest format, de aceea se utilizează foarte rar;
- P (Parity) este stetat în conformitate cu paritatea biților cei mai puțin semnificativi ai unei operații cu date. Astfel, dacă rezultatul unei operații conține un număr par de biți 1, acest indicator este setat. Dacă numărul de biți 1 din rezultat este impar, atunci indicatorul PF este zero. Este folosit de regulă de programe de comunicații, dar Intel a introdus acest indicator nu pentru a îndeplini o anumită funcționalitate, ci pentru a asigura compatibilitatea cu vechile microprocesoare ale familiei x86.

Regiştrii de segment

Proprietățile regiştrilor de segment (vezi figura 5) sunt în strânsă legătură cu noțiunea de segmentare a memoriei. Premiza de la care se pleacă este următoarea: 8086 este capabil să adreseze 1MB de memorie, astfel că sunt necesare adrese pe 20 de biți pentru a cuprinde toate locațiile din spațiul de 1 MB de memorie. Totuşi, registrele utilizate sunt registre pe 16 biți, deci a trebuit să se găsească o soluție pentru această problemă. Soluția găsită se numeşte *segmentarea memoriei*; în acest caz memoria de 1MB este împărțită în 16 segmente de câte 64 KB (16*64 KB = 1024 KB = 1 MB).

Noțiunea de segmentare a memoriei presupune utilizarea unor adrese de memorie formate din două părți. Prima parte reprezintă adresa segmentului iar cea de-a doua porțiune reprezintă adresa de deplasament, sau offset-ul (figura 7).

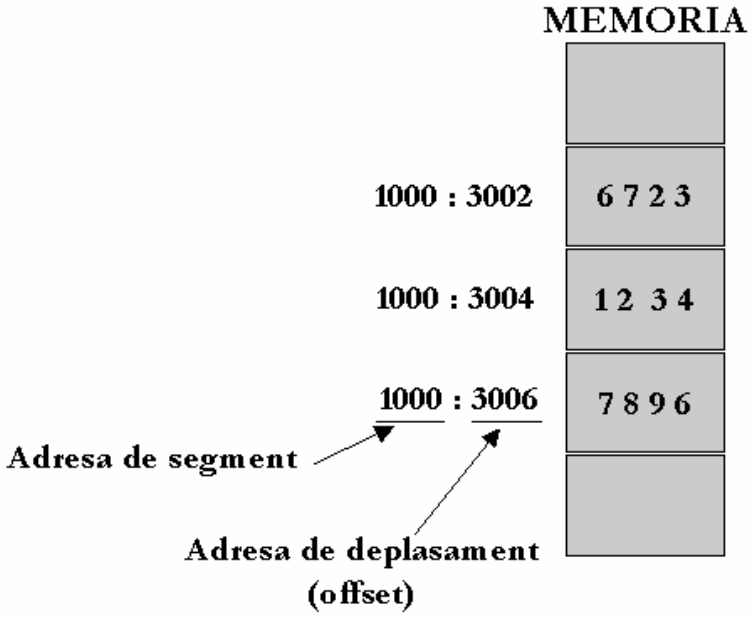


Figura 7. Cele două porțiuni ale unei adrese segmentate

Fiecare pointer de memorie pe 16 biți este combinat cu conținutul unui registru de segment pe 16 biți pentru a forma o adresă completă pe 20 de biți. Adresa de segment împreună cu adresa de deplasament sunt combinate în felul următor: valoarea de segment este deplasată la stânga cu 4 biți (înmulțită cu $16 = 2^4$) și apoi adunată cu valoarea adresei de deplasament. Adresa astfel construită se numește adresă efectivă; fiind o adresă pe 20 de biți poate accesa 2^{20} octeți de memorie, adică 1 MB de memorie. Construirea unei adrese efective este prezentată în figura 8.

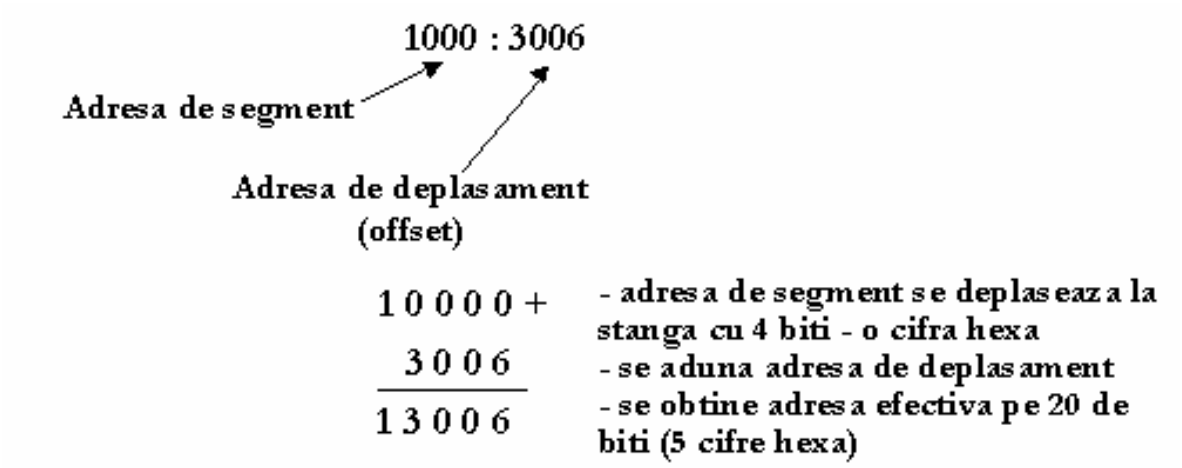


Figura 8. Exemplu de calcul al adresei efective

Registrul CS – acest registru face referire la începutul blocului de 64 KB de memorie în care se află codul programului (segmentul de cod). Microprocesorul 8086 nu poate aduce altă instrucțiune pentru execuție decât cea definită de CS. Registrul CS poate fi modificat de un număr de instrucțiuni, precum instrucțiuni de salt, apel sau de întoarcere. El nu poate fi încărcat în mod direct cu o valoare, ci doar prin intermediul unui alt registru general.

Registrul DS – face referire către începutul segmentului de date, unde se află mulțimea de date cu care lucrează programul aflat în execuție.

Registrul ES – face referire la începutul blocului de 64KB cunoscut sun denumirea de extra-segment. Acesta nu este dedicat nici unui scop anume, fiind disponibil pentru diverse acțiuni. Uneori acesta poate fi folosit pentru creearea unui bloc de memorie de 64 KB adițional pentru date. Acest extra-segment lucrează foarte bine în cazul instrucțiunilor de tip STRING. Toate instrucțiunile de tip STRING ce scriu în memorie folosesc adresarea ES : DI ca adresă de memorie.

Registrul SS – face referire la începutul segmentului de stivă, care este blocul de 64 KB unde se află stiva. Toate instrucțiunile ce folosesc implicit registrul SP (instrucțiunile POP, PUSH, CALL, RET) lucrează în segmentul de stivă deoarece registrul SP este capabil să adreseze memoria doar în segmentul de stivă.

Formatul general al unei instrucţiuni în limbaj de asamblare

O linie de cod scrisă în limbaj de asamblare are următorul format general:

<nume> <instructiune/directiva> <operanzi> <;comentariu>

unde:

- **<nume>** - reprezintă un nume simbolic opţional;
- **<instructiune/directiva>** - reprezintă mnemonica (numele) unei instrucţiuni sau a unei directive;
- **<operanzi>** - reprezintă o combinaţie de unul, doi sau mai mulţi operanzi (sau chiar nici unul), care pot fi constante, referinţe de memorie, referinţe de regiştri, şiruri de caractere, în funcţie de structura particulară a instrucţiunii;
- **<;comentariu>** - reprezintă un comentariu opţional ce poate fi plasat după caracterul „;” până la sfârşitul liniei respective de cod.

Nume de variabile şi etichete

Numele folosite într-un program scris în limbaj de asamblare pot identifica variabile numerice, variabile şir de caractere, locaţii de memorie sau etichete. Spre exemplu, următoarea secvenţă de cod, care calculează valoarea lui trei factorial (3!=1x2x3=6) cuprinde câteva nume de variabile şi etichete:

```
.MODEL  small
.STACK  200h
.DATA
Valoare_Factorial  DW  ?
Factorial  DW  ?
.CODE

Trei_Factorial  PROC
MOV  ax, @data
MOV ds, ax
MOV  [Valoare_Factorial], 1
MOV [Factorial], 2
MOV  cx, 2
Ciclare:
MOV ax, [Valoare_Factorial]
MUL [Factorial]
MOV  [Valoare_Factorial], ax
INC [Factorial]
LOOP Ciclare
```

```
RET
Trei_Factorial ENDP
END
```

Numele *Valoare_Factorial* şi *Factorial* sunt utilizate pentru definirea a două variabile de tip word (pe 16 biţi), *Trei_Factorial* identifică numele procedurii (subrutinei) ce conţine codul pentru calculul factorialului, permiţând apelul său din altă parte a programului. *Ciclare* reprezintă un nume de etichetă, identificând adresa instrucţiunii MOV ax, [Valoare_Factorial], astfel încât instrucţiunea LOOP folosită mai jos să poată face un salt înapoi la această instrucţiune. Numele de variabile pot conţine următoarele caractere: literele a-z şi A-Z, cifrele de la 0-9 precum şi caracterele speciale _ (*underscore* – liniuţă de subliniere), @ (*„at"* în engleză – citit şi „*a rond"* sau *„coadă de maimuţă"*), $ şi ?. Se poate folosi si caracterul punct (".") drept prim caracter al numelui unei etichete. Cifrele 0-9 nu pot fi utilizate pe prima poziţie a numelui; de asemenea, nu pot fi folosite nume care să conţină un singur caracter $ sau ?. Fiecare nume poate fi definit *o singură dată* (numele sunt *unice*) şi pot fi utilizate ca operanzi de oricâte ori se doreşte într-un program. Un nume poate să apară într-un program singur pe o linie (linia respectivă nu mai conţine altă instrucţiune sau directivă). În acest caz, valoarea numelui este dată de adresa instrucţiunii sau directivei de pe linia următoare din program. De exemplu, în secvenţa următoare:

```
...
JMP scadere
...
scadere:
SUB AX, CX
...
```

următoarea instrucţiune care va fi executată după instrucţiunea **JMP scadere** va fi instrucţiunea **SUB AX, CX**. Exemplul anterior este echivalent cu secvenţa:

```
...
JMP scadere
...
scadere: SUB AX, CX
...
```

Există unele avantaje atunci când scriem instrucţiunile pe linii separate. În primul rând, atunci când scriem un nume de etichetă pe o singură linie, este mai uşor să folosim nume lungi de etichete fără a strica „forma" programului scris în limbaj de asamblare. În al doilea rând, este mai uşor să adăugăm ulterior o nouă instrucţiune în dreptul etichetei dacă aceasta nu este scrisă pe aceeaşi linie cu instrucţiunea.

Numele variabilelor sau etichetelor folosite într-un program nu trebuie să se confunde cu numele *rezervate* de asamblor, cum ar fi numele de directive şi instrucţiuni, numele regiştrilor, etc. De exemplu, o declaraţie de genul:

```
...
ax  DW  0
BYTE:
...
```

nu poate fi acceptată, deoarece AX este numele registrului acumulator, AX, iar BYTE reprezintă un cuvânt cheie rezervat.

Orice nume de etichetă ce apare pe o linie fără instrucţiuni sau apare pe o linie cu instrucţiuni trebuie să aibă semnul „:" după numele ei. Totodată, se încearcă să se dea un nume sugestiv etichetelor din program. Fie următorul exemplu:

```
...
CMP AL, 'a'
JB  Nu_este_litera_mica
CMP AL, 'z'
JA  Nu_este_litera_mica
SUB AL, 20H         ; se transforma in litera mare
Nu_este_litera_mica:
…
```

comparativ cu:

```
...
CMP AL, 'a'
JB  x5
CMP AL, 'z'
JA  x5
SUB AL, 20H         ; se transforma in litera mare
x5:
…
```

Dacă în primul caz am folosit un nume sugestiv de etichetă (Nu_este_litera_mica), în cazul al doilea, identic din punct de vedere al funcţionalităţii cu primul, eticheta a fost denumită x5, absolut nesugestiv!

Observaţie:

Limbajul de asamblare nu este *case sensitive*. Aceasta semnifică faptul că, într-un program scris în limbaj de asamblare, numele de variabile, etichete, instrucţiuni, directive, mnemonice, etc., pot fi scrise atât cu litere mari cât şi cu litere mici, nefăcându-se diferenţa între ele (*Nu_este_litera_mica* este acelaşi lucru cu *nu_este_litera_mica* sau *Nu_Este_Litera_Mica*, etc.).

Directive de segment simplificate

Datorită faptului că regiştrii microprocesorului 8086 sunt regiştri pe 16 biţi, s-a impus folosirea unor segmente de memorie de câte 64Ko (maxim cât se poate adresa având la dispoziţie 16 biţi - 64Ko=2^16=65536). Într-un program scris în limbaj de asamblare (vom folosi în continuare prescurtarea ASM) există trei segmente: segmentul de cod, segmentul de date şi segmentul de stivă.

Directivele de segment (fie sub formă standard, fie sub formă simplificată) sunt necesare în orice program scris în limbaj de asamblare pentru a defini şi controla utilizarea segmentelor iar directiva END este folosită întotdeauna pentru a încheia codul programului.

Exemple de directive de segment simplificate sunt:

```
.STACK
.CODE
.DATA
.MODEL
DOSSEG
END
```

.STACK,.CODE,.DATA definesc, respectiv, segmentele de stivă, de cod şi de date.

De exemplu, **.STACK 200H** defineşte o stivă de 512 octeţi (în ASM valorile ce sunt încheiate cu litera H semnifică faptul că este vorba despre hexazecimal). O astfel de valoare pentru stivă este suficientă în mod normal; unele programe, însă (îndeosebi cele recursive) pot necesita dimensiuni mai mari ale stivei.

Directiva **.CODE** marchează începutul segmentului de cod.

Directiva **.DATA** marchează începutul segmentului de date, adică locul în care vom plasa variabilele de memorie. Reprezentativ aici este faptul că trebuie încărcat în mod explicit registrul de segment DS cu valoarea "@data" înaintea accesării locaţiilor de memorie în segmentul definit de .DATA. Având în vedere că un registru de segment poate fi încărcat fie dintr-un registru general fie dintr-o locaţie de memorie dar nu poate fi încărcat direct cu o constantă, registrul de segment DS este încărcat în general printr-o secvenţă de 2 instrucţiuni:

```
...
mov ax, @data
mov ds, ax
...
```

(se poate folosi şi alt registru general în locul lui AX).

Secvenţa anterioară semnifică faptul că DS se va referi către segmentul de date ce începe cu directiva .DATA.

Considerăm în continuare un exemplu de program ce afişează textul memorat în DataString pe ecran:

```
;Program p01.asm
.MODEL small          ;se specifică modelul de memorie SMALL
.STACK 200H           ;se defineşte o stivă de 512 octeţi
.DATA                 ;se specifică începutul segmentului de
                      ;date
DataString DB 'Hello!$'  ;se      defineşte      variabila
                      ;DataString, iniţializată cu valoarea
                      ;"Hello!"
.CODE                 ;începutul segmentului de cod al
                      ;programului
ProgramStart:         ;orice program are o etichetă de
                      ;început
mov bx,@data          ;secvenţa ce setează registrul DS să
                      ;facă referire la segmentul de date ce
                      ;începe cu .DATA
mov ds,bx
mov dx, OFFSET DataString ;se încarcă în DX adresa
;variabilei DataString
mov ah,09             ;codul funcţiei DOS de afişare a unui
                      ;string
int 21H               ;apelul DOS de afişare a string-ului
mov ah, 4cH           ;codul funcţiei DOS de terminare a
                      ;programului
int 21H               ;apelul DOS de terminare a programului
END ProgramStart      ;directiva de terminare a codului
                      ;programului
```

Explicaţii:

1. Se pot introduce comentarii într-un program ASM prin folosirea ";". Tot ce urmează după ";" şi până la sfârşitul liniei este considerat comentariu.

2. Nu are importanţă dacă programul este scris folosind litere mari sau mici (nu este "case sensitive").

3. Fără cele două instrucţiuni care setează registrul DS către segmentul definit de .DATA, funcţia de afişare a string-ului nu ar fi funcţionat cum trebuie. Variabila DataString se află în segmentul .DATA şi nu poate fi accesată dacă DS nu este poziţionat către acest segment. Acest lucru se explică în modul următor: atunci când facem apelul DOS de afişare a unui string, trebuie să parcurgem întreaga adresă de tipul segment:offset a string-ului în DS:DX. De aceea, de abia după ce am încărcat DS cu segmentul .DATA şi DX cu adresa (offset-ul) lui DataString avem o referinţă completă segment:offset către DataString.

Observaţii.

Nu trebuie să încărcăm în mod explicit registrul de segment CS deoarece DOS face acest lucru automat în momentul când rulăm un program. Astfel, dacă CS nu ar fi deja setat la momentul execuţiei primei instrucţiuni din program, procesorul nu ar şti unde să găsească instrucţiunea şi programul nu ar rula niciodată. În mod asemănător, registrul de segment SS este setat de DOS înainte de execuţia programului şi de regulă rămâne nemodificat pe perioada execuţiei programului.

Cu registrul de segment DS lucrurile stau altfel. În timp ce registrul CS se referă la intrucţiuni (cod), SS se referă ("pointează") la stivă, DS "pointează" la date. Programele nu manipulează direct instrucţiuni sau stive dar au de-a face în mod direct cu date. De asemenea, programele vor acces la date situate în segmente diferite în orice moment. Se poate dori încărcarea în DS a unui segment, accesarea datelor din acel segment şi apoi încărcarea lui DS cu un alt segment pentru a accesa un bloc diferit de date. În programe mici sau medii nu vom avea nevoie de mai mult de un segment de date dar programe mai complexe folosesc deseori segmente de date multiple.

Următorul program va afişa un caracter pe ecran, folosind încărcarea registrului ES în locul lui DS.

```
;Program p02.asm
.MODEL small
.STACK 200H
.DATA
OutputChar DB 'B'         ;definirea variabilei OutputChar
                          ;iniţializată cu valoarea "B"
.CODE
ProgramStart:
mov dx, @data
mov es, dx           ;spre deosebire de programul anterior,
                     se foloseşte ES pentru specificarea
                     segmentului de date
mov bx, offset OutputChar     ;se încarcă BX cu adresa
                              ;variabilei OutputChar
mov dl, es:[bx]           ;se încarcă AL cu valoarea de la
                          ;adresa explicită es:[bx]
                          ;(adresare indexată)
mov ah,02                 ;codul funcţiei DOS de afişare a
                          ;unui caracter
int 21H              ;apelul DOS de execuţie a afişării
mov ah, 4cH          ;codul funcţiei DOS de terminare a
                     ;programului
int 21H              ;apelul DOS de terminare a programului
END ProgramStart     ;directiva de terminare a codului
                     ;programului
```

DOSSEG este directiva ce face ca segmentele dintr-un program să fie grupate conform convenţiilor Microsoft de adresare a segmentelor.

Directiva **.MODEL**

Este directiva ce specifică modelul de memorie pentru un program ASM ce foloseşte directive de segment simplificate.

Definiţii: "near" înseamnă adresa (offset-ul) pe 16 biţi din cadrul aceluiaşi segment, în timp ce "far" înseamnă o adresă completă de tip segment:offset, din cadrul altui segment decât cel curent.

Modelele de memorie ce se pot specifica prin intermediul directivei .MODEL sunt:

- ***tiny*** - atât codul cât şi datele programului încap în acelaşi segment de 64Ko. Atât codul cât şi datele sunt de tip near.
- ***small*** - codul programului trebuie să fie într-un singur segment de 64Ko şi datele într-un bloc separat de 64Ko; codul şi datele sunt near
- ***medium*** - codul programului poate fi mai mare decât 64Ko dar datele trebuie să fie într-un singur segment de 64 Ko. Codul este far, datele sunt near.
- ***compact*** - codul programului poate fi într-un singur segment, datele pot fi mai mari de 64 Ko. Codul este near, datele sunt far.
- ***large*** - atât codul cât şi datele pot depăşi 64Ko, dar nici un masiv de date nu poate depăşi 64 Ko. Atât codul cât şi datele sunt far.
- ***huge*** - atât codul cât şi datele pot depăşi 64Ko şi masivele de date pot depăşi 64 Ko. Atât codul cât şi datele sunt far. Pointerii la elementele dintr-un masiv sunt far.

În continuare sunt prezentate câteva exemple legate de modalităţile de declarare a variabilelor şi de adresare a memoriei.

```
var1 DW 01234h      ;se defineste o variabila word cu
                    ;valoarea 1234h
var2 DW 01234       ;se defineste o variabila word cu
                    ;valoarea zecimala 1234 (4D2 in hexa)
var3 RESW 1         ;se rezerva spatiu pentru o variabila
                    ;word (de valoare 0)
var4 DW ABCDh       ;atribuire ilegala!

 mesajsco2 DB 'SCO 2 este cursul preferat!'

...start:

  mov ax,cs         ; setarea  segmentului de date
  mov ds,ax         ; DS=CS
```

```
     ; orice referinta de memorie se presupune ca este relativa
la segmentul DS

        mov ax,[var2]           ; AX <- var2
                                ; == mov ax,[2]

        mov si,var2       ;se  foloseste  SI  ca  pointer  catre
                          var2 (cod C echivalent SI=&var2)

        mov ax,[si]       ;se  citeste  din  memorie  valoarea  lui
                          ;var2 (*(&myvar2))
                          ;(referinta indirecta)

        mov bx,mesajsco2 ; BX este pointer la un string
                          ; (cod C echivalent:  BX=&mesajsco2)

        dec BYTE [bx+1]  ; transforma 'C' in 'B' !

        mov si, 1         ; Foloseste SI cu rol de index
        inc byte [mesajsco2+SI]  ; == inc byte [SI + 8]
                                 ; == inc byte [9]

     ; Memoria poate fi adresata folosindu-se 4 registri:
     ; SI  -> Implica DS
     ; DI  -> Implica DS
     ; BX  -> Implica DS
     ; BP  -> Implica SS !  (nu este foarte des utilizat)
     ;
     ;Exemple:

     mov ax,[bx]     ; ax <- word in memorie referit de BX
     mov al,[bx]     ; al <- byte in memorie referit de BX
     mov ax,[si]     ; ax <- word referit de SI
     mov ah,[si]     ; ah <- byte referit de SI
     mov cx,[di]     ; di <- word referit de DI

     mov ax,[bp]     ; AX <- [SS:BP] Operatie cu stiva!

     ; In plus, sunt permise BX+SI si BX+DI:

     mov ax,[bx+si]
     mov ch,[bx+di]

     ; Deplasamente pe 8 sau 16 biti:

     mov ax,[23h]         ; ax <- word in memorie DS:0023
     mov ah,[bx+5]        ; ah <- byte in memorie [DS:BX+5]
     mov ax,[bx+si+107]   ; ax <- word la adresa [DS:BX+SI+107]
     mov ax,[bx+di+47]    ; ax <- word la adresa [DS:BX+DI+47]
```

```
; ATENTIE: copierea din memorie in memorie este ilegala!
;Totdeauna trebuie sa se treaca valoarea copiata printr-un
;registru

mov [bx],[si]   ;Ilegal
mov [di],[si]   ;Ilegal

; Caz special: operatiile cu stiva!

pop word [var]          ; var <- [SS:SP]
```

Adrese de memorie şi valori

Un program scris în limbaj de asamblare se poate referi fie la o adresă de memorie (OFFSET = DEPLASAMENT), fie la o valoare stocată de variabilă în memorie. Din păcate, limbajul de asamblare nu este nici strict, nici intuitiv cu privire la modurile în care aceste două tipuri de referire sunt făcute şi, drept urmare, referirile la OFFSET sau la valoare sunt deseori confundate. În figura 9 sunt ilustrate conceptele de adresă de deplasament (offset) şi valoare stocată în memorie.

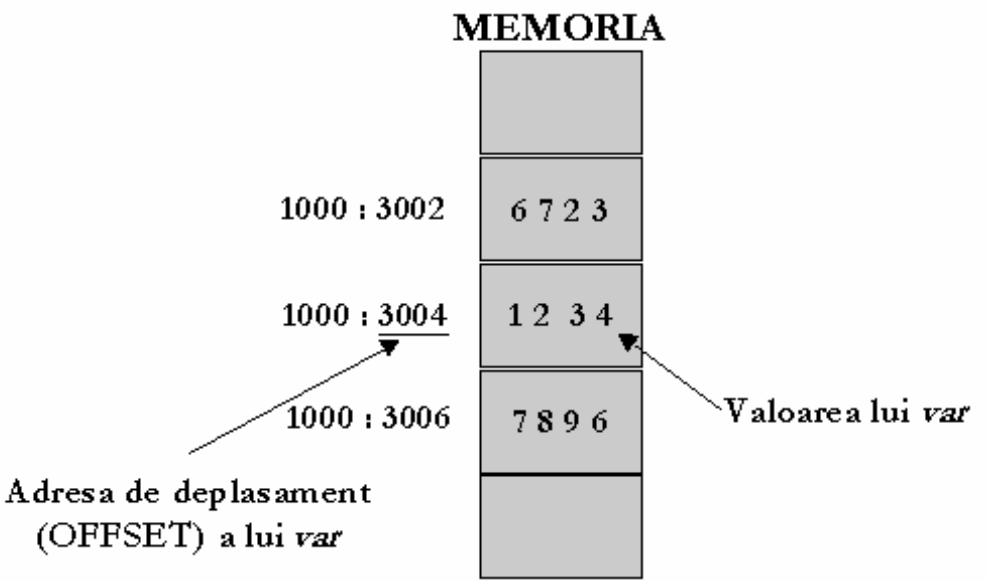


Figura 9. Ilustrarea noţiunilor de adresă de deplasament şi valoare stocată în memorie

Deplasamentul unei variabile de memorie *var* de dimensiune word este valoarea constantă 5004H, obţinută cu operatorul OFFSET. Spre exemplu, instrucţiunea:

MOV BX, OFFSET var

Încarcă valoarea 5004H în registrul BX. Valoarea 5004H nu se modifică; ea este construită în cadrul instrucţiunii. Valoarea lui *var* este 1234H, citită din memorie la adresa dată de offset-ul 5004H din segmentul de date. O modalitatea de citire a acestei valori este de a încărca registrele BX, SI, DI sau BP cu offset-ul lui *var* şi apoi folosirea registrului respectiv pentru adresarea memoriei. Instrucţiunile:

```
MOV BX, OFFSET var
MOV AX, [ BX ]
```

Au ca efect încărcarea valorii lui *var* (1234H) în registrul AX.
De asemenea, se poate încărca valoarea lui *var* direct în AX folosind:

```
MOV AX, var
Sau
MOV AX, [ var ]
```

În timp ce valoarea deplasamentului rămâne constantă, valoarea 1234H nu este permanent asociată cu *var*. De exemplu, instrucţiunile:

```
MOV [ var ], 5555H
MOV AX, [ var ]
```

Au ca efect încărcarea valorii 5555H în registrul AX.
Cu alte cuvinte, în timp ce deplasamentul lui *var* este o valoare constantă ce descrie o adresă fixă dintr-un segment de date, valoarea variabilei *var* este un număr ce poate fi modificat şi care se află memorat la adresa (de memorie) respectivă. Instrucţiunile:

```
MOV [ var ], 1
ADD [ var ], 2
```

Modifică valoarea lui var la 3, dar instrucţiunea:

ADD OFFSET var, 2 este echivalentă cu **ADD 5002H, 2**, ceea ce este un lucru fără sens, deoarece este imposibil să se însumeze o constantă cu alta.

O problemă ce poate apărea adesea în timpul programării este aceea a omiterii lui OFFSET; de exemplu, dacă scriem **MOV SI, var** atunci când, de fapt, dorim încărcarea în SI a deplasamentului lui *var*. Nu va fi semnalată nici o eroare în acest caz, având în vedere că *var* este o variabilă de tip word. Totuşi, în momentul execuţiei programului, registrul SI va fi încărcat cu valoarea lui *var* (1234H), în loc de OFFSET, ceea ce poate conduce la rezultate imprevizibile. În acest caz, referirile la constantele de adresă vor fi precedate de OFFSET iar referirile la valori din memorie să fie cuprinse între paranteze drepte („[” şi „]”), eliminând astfel ambiguitatea.

Instrucţiuni ale microprocesorului Intel

Microprocesoarele din familia Intel x86 dispun de o serie impresionantă de instrucţiuni, asemeni tuturor procesoarelor din clasa procesoarelor CISC (Complex Instruction Set Computer). Instrucţiunile se pot împărţi în: instrucţiuni logice, aritmetice, de transfer şi de control. Prezentăm în continuare câteva exemple din fiecare clasă de instrucţiuni.

Instrucţiuni logice

Instrucţiunile logice implementează funcţiile logice de bază, pe un octet sau pe cuvânt. Ele acţionează bit cu bit, deci se aplică funcţia logică respectivă tuturor biţilor sau perechilor de biţi corespunzători operanzilor. Instrucţiunile logice sunt următoarele:

- **NOT**: A =~A
- **AND**: A &= B
- **OR**: A |= B
- **XOR**: A ^= B
- **TEST**: A & B

De regulă, instrucţiunile logice au efect asupra indicatorilor de stare, cu excepţia instrucţiunii NOT, care nu are efect asupra nici unui flag (indicator de stare). Aceste efecte sunt următoarele:

- Se şterge indicatorul carry (C)
- Se şterge indicatorul overflow (O)
- Se setează zero flag (Z) dacă rezultatul este zero, sau îl şterge în caz contrar
- Se copiază bitul mai "înalt" al rezultatului în indicatorul sign (S)
- Se setează bitul de paritate (P) conform cu *paritatea* rezultatului

Instrucţiunea NOT

Este o instrucţiune cu un singur operand (instrucţiune *unară*), cu forma generală:

NOT *destinatie*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi. Instrucţiunea are ca efect inversarea (negarea) tuturor biţilor operandului, adică aducerea în forma codului invers - complement faţă de 1.

Instrucţiunea AND

Este o instrucțiune cu doi operanzi (instrucțiune *binară*), cu forma generală:

AND *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are ca efect operația: <destinatie> == <destinatie> AND <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea TEST (AND "non-distructiv")

Este o instrucțiune cu doi operanzi (instrucțiune *binară*), cu forma generală:

TEST *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are același efect ca și instrucțiunea AND, cu deosebirea că nu se modifică operandul destinație, iar indicatorii de stare sunt modificați în același mod ca și în cazul instrucțiunii AND.

Instrucțiunea OR

Este o instrucțiune cu doi operanzi, cu forma generală:

OR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: <destinatie> == <destinatie> OR <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea XOR (SAU-Exclusiv)

Este o instrucțiune cu doi operanzi, cu forma generală:

XOR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: <destinatie> == <destinatie> XOR <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit. Funcția XOR, denumită SAU-Exclusiv (sau *anti-coincidență*) are valoarea logică 1 atunci când operanzii săi sunt diferiți (unul are valoarea 0 iar celălalt valoarea 1) și valoarea logică 0 când ambii operanzi au aceeași valoare (fie ambii au valoarea 0, fie ambii au valoarea 1).

Observație:

De cele mai multe ori, instrucțiunile AND și OR sunt folosit pe post de „mascare" a datelor; în acest sens, o valoarea de tip „mască" (*mask*) este utilizată pentru a forța anumiți biți să ia valoarea zero sau valoarea 1 în cadrul altei valori. O

astfel de „mască” logică are efect asupra anumitor biți, în timp ce pe alții îi lasă neschimbați. Exemple:

- Instrucțiunea **AND CL, 0Fh** – face ca cei mai semnificativi 4 biți să ia valoarea 0, în timp ce biții mai puțin semnificativi sunt lăsați neschimbați; astfel, dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **AND CL, 0Fh** va avea valoarea **0000 1101**.
- Instrucțiunea **OR CL, 0Fh** – face ca cei mai puțin semnificativi 4 biți să ia valoarea 1, în timp ce biții mai semnificativi să rămână nemodificați. Dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **OR CL, 0Fh** va avea valoarea **1001 1111**.

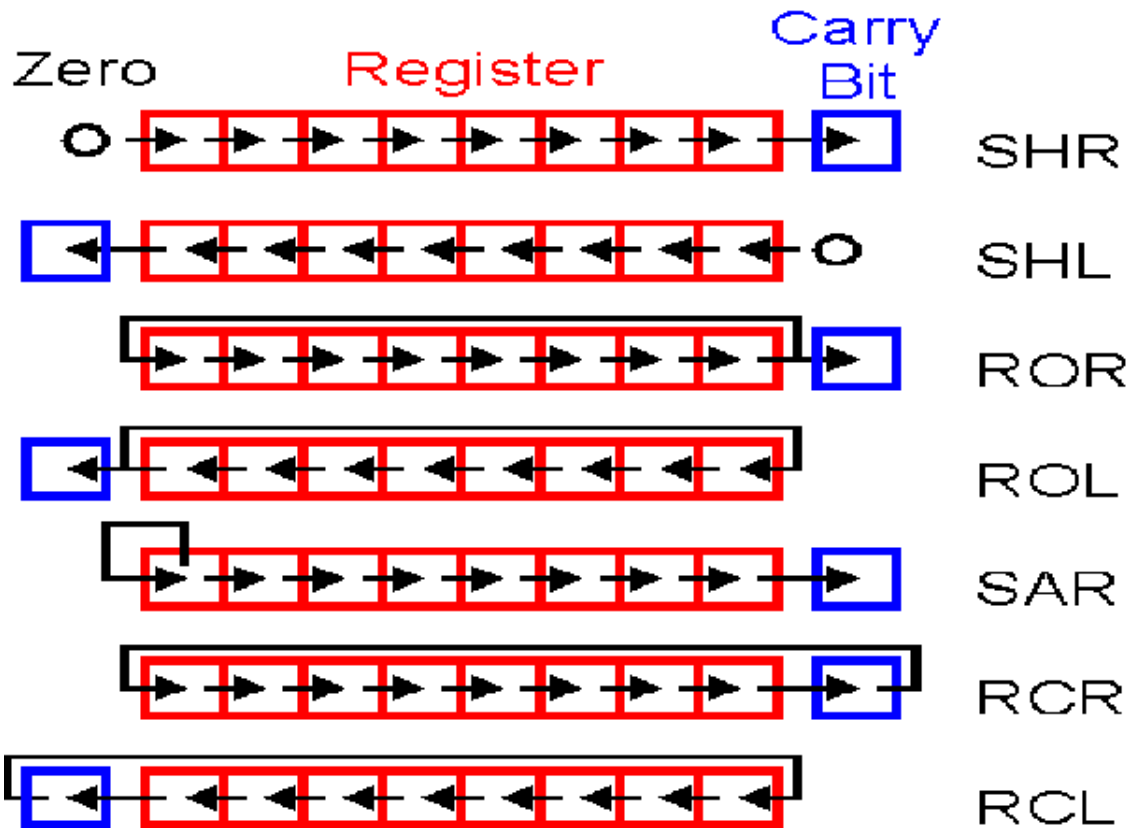


Figura 10. Instrucțiuni de deplasare şi de rotație

Instrucțiuni de deplasare şi de rotație

Acest tip de instrucțiuni (vezi figura **10**) permit realizarea operațiilor de deplasare şi de rotație la nivel de bit. Ele au doi operanzi, primul operand fiind cel asupra căruia se aplică operația de deplasare pe biți, iar cele de-al doilea (operandul *numărător* sau *contor*) semnifică numărul de biți cu care se face această deplasare. Operațiile se pot face de la dreapta spre stânga sau invers. Deplasarea înseamnă

translatarea tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în poziția rămasă liberă și cu pierderea biților din dreapta/stânga. Rotația presupune translatarea biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă. Sintaxa generală a instrucțiunilor de deplasare și rotație este următoarea:

INSTR <operand> , <contor>

Unde **INSTR** reprezintă numele instrucțiunii, **<operand>** reprezintă un registru sau o locație de memorie pe 8 sau 16 biți, iar **<contor>** semnifică numărul de biți cu care se face deplasarea, adică fie o constantă, fie registrul CL (care își confirmă astfel rolul de numărător).

Observație.

Totdeauna există două modalități de deplasare:

- Prin folosirea unui contor efectiv – de exemplu: SHL AX, 1
- Prin folosirea registrului CL pe post de contor – de exemplu: SHL AX, CL

Instrucțiunea SHL/SAL (Shift Left/Shift Arithmetic Left)

Această instrucțiune translatează biții operandului o poziție la stânga de câte ori specifică operandul numărător. Pozițiile rămase libere prin deplasarea la stânga sunt umplute cu zerouri la bitul cel mai puțin semnificativ, în timp ce bitul cel mai semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de înmulțire cu o putere a lui 2 (în funcție de numărul de biți pentru care se face deplasarea la stânga).

Exemple:

1. Înmulțirea lui AX cu 10 (1010 în binar) (înmulțim cu 2 și cu 8, apoi adunăm rezultatele)

```
shl   ax, 1      ; AX ori 2
mov   bx, ax     ; salvăm 2*AX în BX
shl   ax, 2      ; 2*AX(original) * 4 = 8*AX(original)
add   ax, bx     ; 2*AX + 8*AX = 10*AX
```

2. Înmulțirea lui AX cu 18 (10010 în binar) (înmulțim cu 2 și cu 16, apoi adunăm rezultatele)

```
shl   ax, 1      ; AX ori 2
```

```
mov   bx, ax        ; salvăm 2*AX
shl   ax, 3         ; 2*AX(original) ori 8 = 16*AX(original)
add   ax, bx        ; 2*AX + 16*AX = 18*AX
```

Instrucţiunea SHR (Shift Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai puţin semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *fără semn* la o putere a lui 2 (dacă deplasarea se face cu *o poziţie* la dreapta, operaţia este echivalentă cu o împărţire la 2, dacă deplasarea se face cu două poziţii, operaţia este echivalentă cu o împărţire la 2^2, etc.). Operaţia de împărţire se execută *fără semn*, completându-se cu un bit 0 dinspre stânga (bitul cel mai semnificativ).

Instrucţiunea SAR (Shift Arithmetic Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai semnificativ rămâne neschimbat, în timp ce bitul cel mai puţin semnificativ este copiat în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *cu semn* la o puterea a lui 2 (în funcţie de numărul de biţi cu care se face deplasarea la dreapta).

Instrucţiunea RCL (Rotate through Carry Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga prin intermediul lui CF (Carry Flag). Astfel, cel mai semnificativ bit trece din operand în CF, apoi se deplasează toţi biţii din operand cu o poziţie la stânga iar CF original trece în bitul cel mai puţin semnificativ din operand.

Instrucţiunea ROL (Rotate Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga. Astfel, cel mai semnificativ bit trece din operand în bitul cel mai puţin semnificativ.

Exemplu:

După execuţia instrucţiunilor:
ROL AX, 6
AND AX, 1Fh
Biţii 10-14 din AX se mută în biţii 0-4.

Instrucţiunea RCR (Rotate through Carry Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta prin intermediul lui CF (Carry Flag). Astfel, bitul din CF este scris înapoi în bitul cel mai semnificativ al operandului.

Instrucţiunea ROR (Rotate Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta. Bitul cel mai puţin semnificativ trece în bitul cel mai semnificativ.

Exemple:

```
MOV ax,3     ; Valori iniţiale              AX = 0000 0000 0000 0011
MOV bx,5     ;                              BX = 0000 0000 0000 0101
OR ax,9      ;      ax <- ax | 0000 1001    AX = 0000 0000 0000 1011
AND ax,10101010b  ; ax <- ax & 1010 1010    AX = 0000 0000 0000 1010
XOR ax,0FFh       ; ax <- ax ^ 1111 1111    AX = 0000 0000 1111 0101
NEG ax            ; ax <- (-ax)             AX = 1111 1111 0000 1011
NOT ax            ; ax <- (~ax)             AX = 0000 0000 1111 0100
OR ax,1           ; ax <- ax | 0000 0001    AX = 0000 0000 1111 0101
SHL ax,1     ; depl logică la stg cu 1 bit  AX = 0000 0001 1110 1010
SHR ax,1     ; depl logică la dr cu 1 bit   AX = 0000 0000 1111 0101
ROR ax,1     ; rotaţie stg (LSB=MSB)        AX = 1000 0000 0111 1010
ROL ax,1     ; rotaţie dr (MSB=LSB)         AX = 0000 0000 1111 0101
MOV cl,3     ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHR ax,cl    ; împărţim AX la 8             AX = 0000 0000 0001 1110
MOV cl,3     ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHL bx,cl    ; înmulţim BX cu 8             BX = 0000 0000 0010 1000
```

Instrucţiuni aritmetice

Instrucţiunea ADD (ADDition)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Cei doi operanzi nu pot fi însă în acelaşi timp locaţii de memorie. Rezultatul operaţiei este

următorul: <destinatie> == <destinatie> + <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Dacă apare ambiguitate la modul de exprimare al operanzilor (8 sau 16 biţi) se va folosi operatorul PTR.

Exemple:

```
ADD  AX, BX        ; adunare între regiştri – AX ← AX + BX
ADD  DL, 33h       ; adunare efectivă - DL ← DL + 33h
MOV DI, NUMB   ; adresa lui NUMB
MOV AL, 0          ; se şterge suma
ADD AL, [DI]       ; adună [NUMB]
ADD AL, [DI + 1] ; adună [NUMB + 1]
ADD word ptr [DI], -2     ; destinatie in memorie, sursa imediata
ADD byte ptr VAR, 5       ; fortarea instructiunii pe un octet, VAR fiind
                          ;declarat DW
```

Instrucţiunea INC (Increment addition)

Instrucţiunea INC are formatul general:

INC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biţi iar semnificaţia operaţiei este incrementarea valorii destinaţie cu 1. Toţi indicatorii de stare sunt afectaţi, cu excepţia lui CF (Carry Flag).

Exemplu:

```
MOV DI, NUMB       ; adresa lui NUMB
MOV AL, 0          ; şterge suma
ADD AL, [DI]       ; adună [NUMB]
INC   DI           ; DI = DI + 1
ADD AL, [DI]       ; adună [NUMB + 1]
```

Instrucţiunea ADC (ADdition with Carry)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată.

Instrucţiunea acţionează întocmai ca ADD, cu deosebirea că la rezultat este adăugat şi bitul CF. Este utilizat, de regulă, pentru a aduna numere mai mari de 16 biţi (8086-80286) sau mai mari de 32 de biţi la 80386, 80486, Pentium.

Exemplu:

Adunarea a două numere pe 32 de biţi se poate face astfel (BXAX) + (DXCX):

```
ADD AX, CX
ADC BX, DX
```

Instrucţiunea SUB (SUBstract)

Instrucţiunea SUB are formatul general:

SUB <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Rezultatul operaţiei este următorul: <destinatie> == <destinatie> - <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Scăderea poate fi văzută ca o adunare cu reprezentarea în complementul faţă de 2 al operandului sursă şi cu inversarea bitului CF, în sensul că, dacă la operaţie (adunarea echivalentă) apare transport, CF=0 şi dacă la adunarea echivalentă nu apare transport, CF=1.

Pentru instrucţiunile:

```
MOV CH, 22h
SUB CH, 34h
```

Rezultatul este **-12 (1110 1110)**, iar indicatorii de stare se modifică astfel:

ZF = 0 (rezultat diferit de zero)
CF = 1 (împrumut)
SF = 1 (rezultat negativ)
PF = 0 (paritate pară)
OF = 0 (fără depăşire)

Instrucţiunea DEC (DECrement substraction)

Instrucțiunea DEC are formatul general:

DEC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biți iar semnificația operației este decrementarea valorii destinație cu 1. Toți indicatorii de stare sunt afectați, cu excepția lui CF (Carry Flag).

Instrucțiunea SBB (SuBstract with Borrow)

Instrucțiunea SBB are formatul general:

SBB <destinatie>, <sursa>

Unde <*destinatie*> şi <*sursa*> pot fi registru sau operand în memorie, pe 8 sau pe 16 biți. Rezultatul operației este următorul: <destinatie> == <destinatie> - <sursa> - CF, deci la fel ca şi în cazul instrucțiunii SUB, dar din rezultat se scade şi bitul CF. Indicatorii de stare modificați în urma acestei operații sunt: AF, CF, PF, SF, ZF, OF. Această instrucțiune este utilizată, de regulă, pentru a scădea numere mai mari de 16 biți (la 8086 - 80286) sau de 32 de biți (la 80386, 80486, Pentium).

Exemplu

Scăderea a două numere pe 32 de biți se poate face astfel (BXAX) - (SIDI):

```
SUB   AX, DI
SBB   BX, SI
```

Exemple de programe

1. Program care citeşte un număr de la tastatură şi afişează dacă numărul este par sau nu:

```
; Programul citeste un numar si afiseaza un mesaj referitor la paritate
dosseg
.model small
.stack
.data

mesaj  db 13,10,'Introduceti numarul:(<=9)$'
mesg_par  db 13,10,'Numarul introdus este par!$'
mesg_impar  db 13,10,'Numarul introdus este impar!$'
```

```
.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          mov bx,2
          div bx ; se imparte AX la BX, catul va fi in AX, restul in DX
          cmp dx,0
          jnz impar
          mov ah,09
          mov dx,offset mesg_par
          int 21h
          jmp sfarsit
impar:  mov ah,09
          mov dx,offset mesg_impar
          int 21h
sfarsit:
          mov ah,4ch
          int 21h ; sfarsitul programului

END pstart
```

2. Program care calculează pătratul unui număr introdus de la tastatură.

```
; Programul calculeaza patratul unui numar (<=256) introdus de la tastatura
; Valoarea patratului se calculeaza in registrul AX (valoare maxima 2^16 = 65536)

dosseg
.model small
.stack
.data

nr DB 10,10 dup(0)
r DB 10, 10 dup(0)
```

```
mesaj  db 13,10,'Introduceti numarul:(<=256)$'
patrat db 13,10,'Patratul numarului este:$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov cl,nr[1]  ; incarc in CL numarul de cifre al numarului introdus
          inc cl        ; in sir se va merge pana la pozitia cl+1
          mov si,1      ; folosesc registrul SI pe post de contor
          xor ax,ax     ; initializez AX cu valoarea 0
          mov bl,10     ; se va inmulti cu valoarea 10 care este stocata in BL
inmultire:
          mul bl
          inc si
          mov dl,nr[si]
          sub dl,30h
          add ax,dx
          cmp si,cx
          jne inmultire

          mul ax

          xor si,si
          mov bx,10

cifra: ; aici incepe afisarea rezultatului din AX
          div bx
          add dl,30h
          mov r[si],dl
          inc si
          xor dx,dx
```

```
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset patrat
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter
        jmp sfarsit


        mov ah,9
        mov dx,offset patrat
        int 21h

sfarsit:
        mov ah,4ch
        int 21h ; stop program



        END pstart
```

3. Program care calculează valoarea unui număr ridicat la o putere. Atât numărul cât şi exponentul (puterea) sunt introduse de la tastatură.

```
; Programul calculeaza un numar ridicat la o putere

; Observatie. Deoarece rezultatul se calculeaza in registrul AX care este un
; registru pe  16 biti, valoarea maxima calculata corect este 2^16= 65536

.model small
.stack
.data

mesaj1  db 13,10,'Introduceti numarul:(<=9)$'
mesaj2  db 13,10,'Introduceti puterea:(<=9)$'
```

```
mesaj_final  db 13,10,'Rezultatul este: $'
mesaj_putere_0 db 13,10, 'Orice numar ridicat la puterea 0 este 1! $'

r db 30 dup(0) ; in variabila r se va stoca reultatul

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj1
        int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h ; se obtine valoarea numerica
                  ; scazandu-se codul lui 0 in ASCII (30H)
          push ax ; se salveaza valoarea lui ax in stiva

          mov ah,09
          mov dx,offset mesaj2
          int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
                  ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h   ; se obtine valoarea numerica
                        ; scazandu-se codul lui 0 in ASCII (30H)
          mov cx,ax     ; registrul CX contorizeaza numarul de inmultiri
          cmp cx,0
          jne putere_0
          mov ah,09
          mov dx, offset mesaj_putere_0
          int 21h
          jmp sfarsit

putere_0:
          pop bx          ;se salveaza in BX valoarea cu care inmulteste
          mov ax,0001
inmultire:
```

```
        mul bx
        loop inmultire

        xor si,si
        mov bx,10
cifra:
        div bx
        add dl,30h
        mov r[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset mesaj_final
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter



sfarsit:
        mov ah,4ch
        int 21h ; sfarsitul programului

END pstart
```

4. Program care verifică dacă un număr este palindrom (un număr se numeşte palindrom dacă scris de la dreapta la stânga sau invers are aceeaşi valoare).

```
; Programul verifica daca un numar sau sir de caractere este palindrom


dosseg
.model small
.stack
```

```
.data

nr DB 10,10 dup(0)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_nu db 13,10,'Numarul nu este palindrom!$'
mesaj_da db 13,10,'Numarul este palindrom!$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov si,1
          mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
          and cx,00FFh

          mov ax,cx
          mov bl,2
          div bl          ; in AL este catul impartirii lui AX la 2
          and ax,00FFh
          inc ax
          inc cx
          mov di,cx

urmatorul_caracter:
          inc si          ; SI creste de la inceputul sirului spre mijloc
          mov bl,nr[di]
          cmp nr[si],bl
          jne nu_este
          dec di          ; DI scade de la sfarsitul sirului spre mijloc
          cmp si,ax       ; in sir se va merge pana la pozitia cl+1
          jne urmatorul_caracter
          mov ah,9
```

```
        mov dx,offset mesaj_da
        int 21h
        jmp sfarsit

nu_este:
        mov ah,9
        mov dx,offset mesaj_nu
        int 21h


sfarsit:
        mov ah,4ch
        int 21h ; stop program

END pstart
```

5. Program care calculează suma cifrelor unui număr introdus de la tastatură.

```
; Programul calculeaza suma cifrelor unui numar introdus de la tastatura


dosseg
.model small
.stack
.data

nr DB 10,10 dup(?)
rezultat DB 10,10 dup(?)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_suma db 13,10,'Suma cifrelor numarului este: $'

.code

pstart:
      mov ax,@data
      mov ds,ax

      mov ah,09               ; aici se afiseaza mesajul initial de introducere
      mov dx,offset mesaj  ; a numarului
      int 21h
```

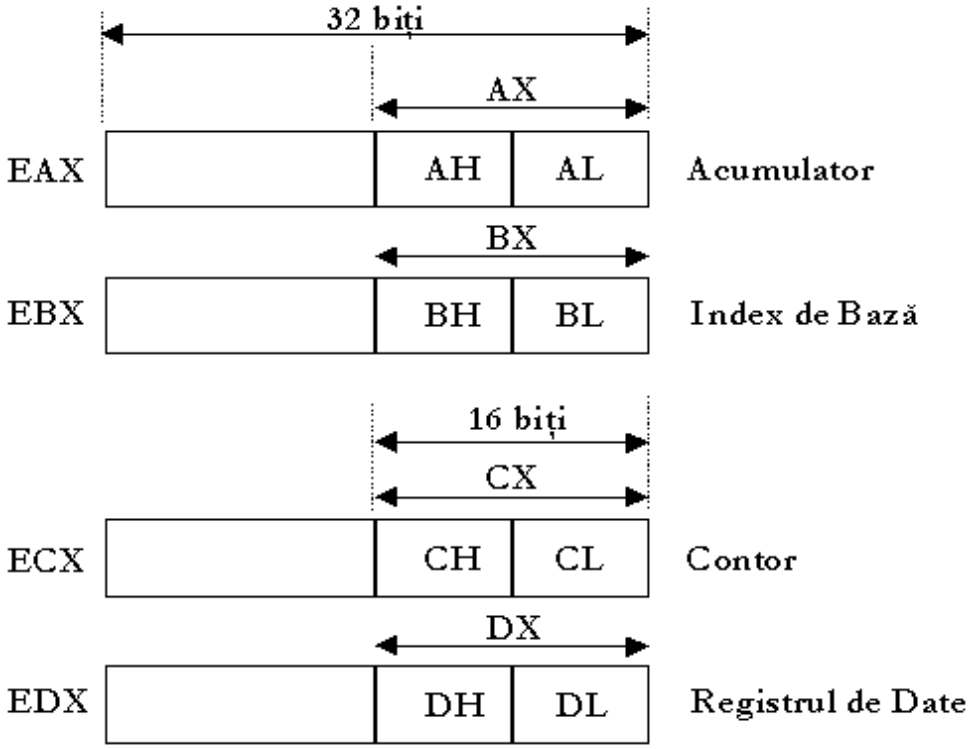
```
        mov ah,0ah          ; functia 10(0ah) citeste un sir de caractere de la
                            ; tastatura intr-o variabila de memorie
        mov dx,offset nr
        int 21h

        mov si,1
        mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
        and cx,00FFh
        inc cx          ; CX stocheaza acum ultima pozitie din sirul de cifre
        xor ax,ax       ; stocam rezultatul in AX, pe care il initializam cu zero

urmatorul_caracter:
        inc si          ; SI creste de la inceputul sirului spre sfarsit
        add al,nr[si]

        sub al,30h      ; scadem codul ASCII al lui zero

        cmp si,cx       ; in sir se va merge pana la pozitia cl+1
        jne urmatorul_caracter

        xor si,si       ; SI este indicele din sirul care va contine rezultatul

cifra: ; aici incepe afisarea rezultatului din AX

        mov bx,0ah
        div bx
        add dl,30h
        mov rezultat[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx,offset mesaj_suma
        int 21h

caracter:
        dec si
        mov ah,02           ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,rezultat[si] ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
```

```
        jne caracter

        mov ah,4ch
        int 21h                 ; terminarea programului

END pstart
```

Introducere în limbajul de asamblare

Ne vom referi în cele ce urmează la familia de microprocesoare intitulată iAPx86 ce stau la baza calculatoarelor IBM PC, începând de la procesoarele 8088 şi 8086, continuând cu 80286, 80386, 80486, Pentium, ş.a.m.d. Procesorul 8086 reprezintă, de fapt, baza familiei ce este cunoscută pe scurt sub denumirea de familia microprocesoarelor x86. De aceea se vor face referiri în continuare la această arhitectură (8086).

Elementele arhitecturale de bază ale microprocesorului



Notă:
Regiştrii pe 32 de biţi nu apar la 8086, 8088, 80286

Figura 1. Regiştrii de uz general – acumulator, index de bază, contor şi de date

Regiştrii microprocesorului

Regiştrii (sau registrele) microprocesorului reprezintă locaţii de memorie speciale aflate direct pe cip; din această cauză reprezintă cel mai rapid tip de memorie. Alt lucru deosebit legat de regiştri este faptul că fiecare dintre aceştia au un scop bine precizat, oferind anumite funcţionalităţi speciale, unice. Există patru mari

categorii de regiştri: regiştrii de uz general, registrul indicatorilor de stare (*flags*), regiştrii de segment şi registrul pointer de instrucţiune.

Regiştrii de uz general

Regiştrii de uz general (vezi figura 1 şi figura 2) sunt implicaţi în operarea majorităţii instrucţiunilor, drept operanzi sursă sau destinaţie pentru calcule, copieri de date, pointeri la locaţii de memorie sau cu rol de contorizare. Fiecare dintre cei 8 regiştri de uz general AX, BX, CX, DX, SP, BP, DI,SI sunt regiştri pe 16 biţi pentru microprocesorul 8086, iar de la procesorul 80386 încoace au devenit regiştri pe 32 de biţi, denumiţi, respectiv: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI (litera E provine de la *Extended – extins* în engleză). Mai mult, cei mai puţin semnificativi 8 biţi ai regiştrilor AX, BX, CX, DX formează respectiv regiştrii AL, BL, CL, DL (litera L provine de la *Low – jos* în engleză), iar cei mai semnificativi 8 biţi ai aceloraşi regiştri formează regiştrii AH, BH, CH, DH (litera H provine de la *High – înalt* în engleză) (figura 1).

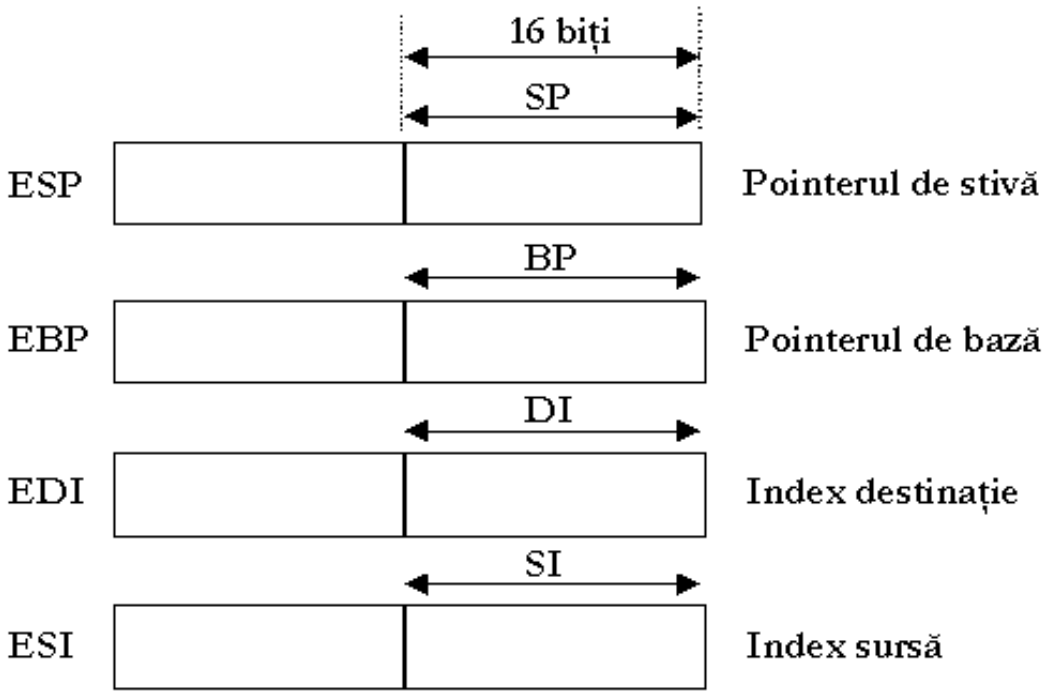


Figura 2. Regiştrii de uz general index şi pointer

Ne vom concentra în continuare atenţia asupra regiştrilor generali pe 16 biţi; fiecare dintre aceştia poate stoca o valoare pe 16 biţi, poate fi folosit pentru stocarea unei valori din memorie sau poate fi utilizat pentru operaţii aritmetice şi logice. Spre exemplu, următoarele instrucţiuni:

```
…
MOV  BX, 2
MOV  DX, 3
```

```
ADD  BX, DX
…
```

încarcă valoarea 2 în registrul BX, valoarea 3 în registrul DX, adună cele două valori iar rezultatul (5) este memorat în registrul BX. În exemplul anterior putem utiliza oricare dintre regiştrii de uz general în locul regiştrilor BX şi DX. În afara proprietăţii de a stoca valori şi de a folosi drept operanzi sursă sau destinaţie pentru instrucţiunile de manipulare a datelor, fiecare dintre cei 8 regiştri de uz general au propria "personalitate". Vom vedea în continuare care sunt caracteristicile specifice fiecăruia dintre regiştrii de uz general.

Registrul AX (EAX)

Registrul AX (EAX) este denumit şi registrul *acumulator*, fiind principalul registru de uz general utilizat pentru operaţii aritmetice, logice şi de deplasare de date. Totdeauna operaţiile de înmulţire şi împărţire presupun implicarea registrului AX. Unele dintre instrucţiuni sunt optimizate pentru a se executa mai rapid atunci când este folosit AX. În plus, registrul AX este folosit şi pentru toate transferurile de date de la/către porturile de Intrare/Ieşire. Poate fi accesat pe porţiuni de 8, 16 sau 32 de biţi, fiind referit drept AL (cei mai puţin semnificativi 8 biţi din AX), AH (cei mai semnificativi 8 biţi din AX), AX (16 biţi) sau EAX (32 de biţi). Prezentăm în continuare alte câteva exemple de instrucţiuni ce utilizează registrul AX. De remarcat este faptul că transferurile de date se fac pentru instrucţiunile (denumite şi *mnemonice*) Intel de la dreapta spre stânga, exact invers decât la Motorola (vom vedea şi alt exemplu asemănător la scrierea datelor în memorie sub format diferit la Motorola faţă de Intel), unde transferul se face de la stânga la dreapta.

Instrucţiunea: **MOV AX, 1234H** încarcă valoarea 1234H (4660 în zecimal) în registrul acumulator AX. După cum spuneam, cei mai puţini semnificativi 8 biţi ai registrului AX sunt identificaţi de AL (A-Low) iar cei mai semnificativi 8 biţi ai aceluiaşi registru sunt identificaţi ca fiind AH (A-High). Acest lucru este utilizat pentru a lucra cu date pe un octet, permiţând ca registrul AX să fie folosit pe postul a doi regiştri separaţi (AH şi AL). Aceeaşi regulă este valabilă şi pentru regiştrii de uz general BX, CX, DX. Următoarele trei instrucţiuni setează registrul AH cu valoarea 1, incrementează cu 1 această valoare şi apoi o copiază în registrul AL:

```
MOV  AH, 1
INC  AH
MOV AL, AH
```

Valoarea finală a registrului AX va fi 22 (AH = AL = 2).

Registrul BX (EBX)

Registrul BX (Base), sau registrul de bază poate stoca adrese pentru a face referire la diverse structuri de date, cum ar fi vectorii stocaţi în memorie. O valoare reprezentată pe 16 biţi stocată în registrul BX poate fi utilizată ca fiind o porţiune din adresa unei locaţii de memorie ce va fi accesată. Spre exemplu, următoarele instrucţiuni încarcă registrul AH cu valoarea din memorie de la adresa 21.

```
MOV  AX,  0
MOV  DS, AX
MOV  BX, 21
MOV  AH, [ BX ]
```

Se observă că am încărcat valoarea 0 în registrul DS înainte de a accesa locaţia de memorie referită de registrul BX. Acest lucru este datorat segmentării memoriei (segmentare discutată mai în detaliu în secţiunea consacrată regiştrilor de segment); implicit, atunci când este folosit ca pointer de memorie, BX face referire relativă la registrul de segment DS (adresa la care face referire este o adresă relativă la adresa de segment conţinută în registrul DS).

Registrul CX (ECX)

Specializarea registrului CX (Counter) este numărarea; de aceea, el se numeşte şi registrul contor. De asemenea, registrul CX joacă un rol special atunci când se foloseşte instrucţiunea LOOP. Rolul de contor al registrului CX se observă imediat din exemplul următor:

```
MOV CX, 5
start:
…
<instrucţiuni ce se vor executa de 5 ori>
…
SUB CX, 1
JNZ start
```

Deoarece valoarea iniţială a lui CX este 5, instrucţiunile cuprinse între eticheta start şi instrucţiunea JNZ se vor executa de 5 ori (până când registrul CX devine 0). Instrucţiunea SUB CX, 1 decrementează registrul CX cu valoarea 1 iar instrucţiunea JNZ start determină saltul înapoi la eticheta start dacă CX nu are valoarea 0. În limbajul microprocesorului există şi o instrucţiune specială legată de ciclare. Aceasta este instrucţiunea LOOP, care este folosită în combinaţie cu registrul CX. Liniile de cod următoare sunt echivalente cu cele anterioare, dar aici se utilizează instruxţiunea LOOP:

```
MOV CX, 5
start:
…
<instrucțiuni ce se vor executa de 5 ori>
…
LOOP start
```

Se observă că instrucțiunea LOOP este folosită în locul celor două instrucțiuni SUB şi JNZ anterioare; LOOP decrementează automat registrul CX cu 1 şi execută saltul la eticheta specificată (*start*) dacă CX este diferit de zero, totul într-o singură instrucțiune.

Registrul DX (EDX)

Registrul de uz general DX (Data register), denumit şi registrul de date, poate fi folosit în cazul transferurilor de date Intrare/Ieşire sau atunci când are loc o operație de înmulțire sau de împărțire. Instrucțiunea **IN AL, DX** copiază o valoare de tip Byte dintr-un port de intrare, a cărui adresă se află în registrul DX. Următoarele instrucțiuni determină scrierea valorii 101 în portul I/O 1002:

```
...
MOV  AL, 101
MOV  DX, 1002
OUT  DX, AL
…
```

Referitor la operațiile de înmulțire şi împărțire, atunci când împărțim un număr pe 32 de biți la un număr pe 16 biți, cei mai semnificativi 16 biți ai deîmpărțitului trebuie să fie în DX. După împărțire, restul împărțirii se va afla în DX. Cei mai puțin semnificativi 16 biți ai deîmpărțitului trebuie să fie în AX iar câtul împărțirii va fi în AX. La înmulțire, atunci când se înmulțesc două numere pe 16 biți, cei mai semnificativi 16 biți ai produsului vor fi stocați în DX iar cei mai puțin semnificativi 16 biți în registrul AX.

Registrul SI

Registrul SI (Source Index) poate fi folosit, ca şi BX, pentru a referi adrese de memorie. De exemplu, secvența de instrucțiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
MOV  AL, [ SI ]
```

Încarcă valoarea (pe 8 biţi) din memorie de la adresa 33 în registrul AL. Registrul SI este, de asemenea, foarte folositor atunci când este utilizat în legătură cu instrucţiunile dedicate tipului string (şir de caractere). Secvenţa următoare :

```
CLD
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
LODSB
```

nu numai că încarcă registrul AX cu valoarea de la adresa de memorie referită de registrul SI, dar adună, de asemenea, valoarea 1 la SI. Acest lucru este deosebit de eficient atunci când se accesează secvenţial o serie de locaţii de memorie, cum ar fi şirurile de caractere. Instrucţiunile de tip string se pot repeta de mai multe ori, astfel încât o singură instrucţiune poate avea ca efect sute sau mii de operaţii.

Registrul DI

Registrul DI (Destination Index) este utilizat în mod asemănător registrului SI. În secvenţa de instrucţiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  DI, 1000
ADD  BL, [ DI ]
```

se adună la registrul BL valoarea pe 8 biţi stocată la adresa 1000. Registrul DI este puţin diferit faţă de registrul SI în cazul instrucţiunilor de tip string; dacă SI este întotdeauna pe post de pointer sursă de memorie, registrul DI serveşte drept pointer destinaţie de memorie. Mai mult, în cazul instrucţiunilor de tip string, registrul SI adresează memoria relativ la registrul de segment DS, în timp ce DI conţine referiri la memorie relativ la registrul de segment ES. În cazul în care SI şi DI sunt utilizaţi cu alte instrucţiuni, ei fac referire la registrul de segment DS.

Registrul BP

Pentru a înţelege mai bine rolul regiştrilor BP şi SP, a sosit momentul să spunem câteva lucruri despre porţiunea de memorie denumită stivă (în engleză *stack*). Stiva (vezi figura 3) reprezintă o porţiune specială de locaţii adiacente din memorie. Aceasta este conţinută în cadrul unui segment de memorie şi identificată de un selector de segment memorat în registrul SS (cu excepţia cazului în care se foloseşte modelul nesegmentat de memorie în care stiva poate fi localizată oriunde în spaţiul de adrese liniare al programului). Stiva este o porţiune a memoriei unde valorile pot

fi stocate şi accesate pe principul LIFO (Last In – First Out), drept urmare ultima valoare stocată în stivă este prima ce va fi citită din stivă. De regulă, stiva este utilizată la apelul unei proceduri sau la întoarcerea dintr-un apel de procedură (principalele instrucţiuni folosite sunt CALL şi RET).

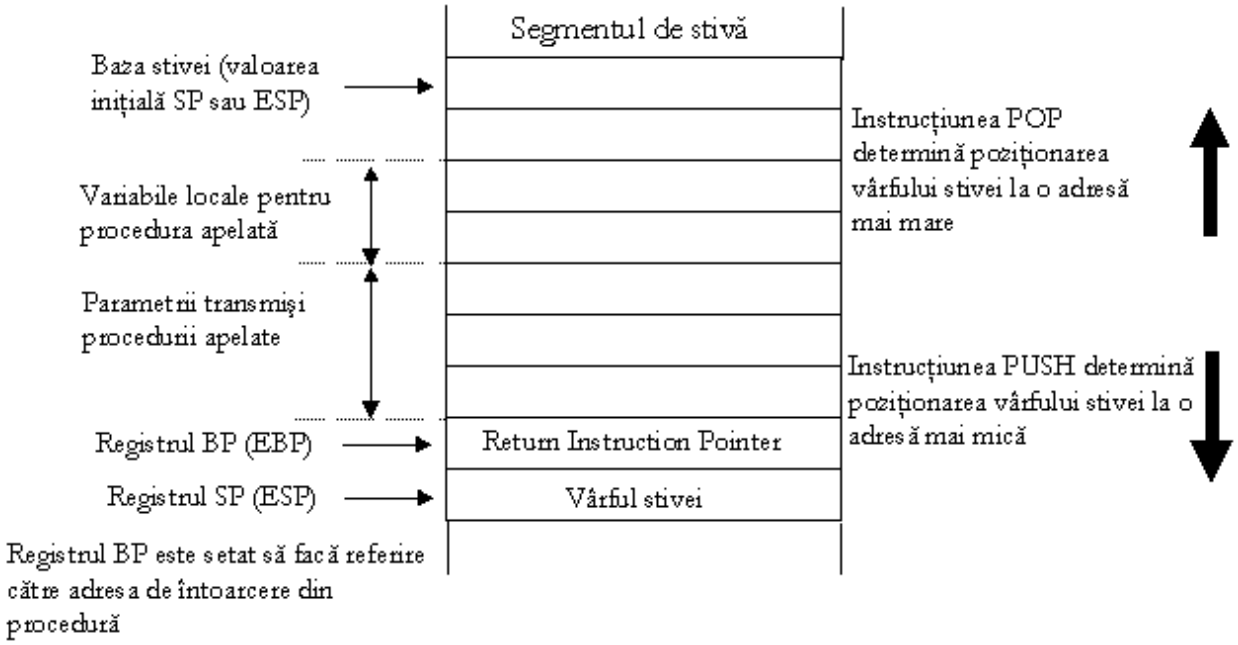


Figura 3. Structura stivei

Registrul pointer de bază, BP (Base Pointer) poate fi utilizat ca pointer de memorie precum regiştrii BX, SI şi DI. Diferența este aceea că, dacă BX, SI şi DI sunt utilizaţi în mod normal ca pointeri de memorie relativ la segmentul DS, registrul BP face referire relativ la segmentul de stivă SS. Principiul este următorul: o modalitate de a trece parametrii unei subrutine este aceea de a utiliza stiva (acest lucru se întâmplă în mod obişnuit în limbajele de nivel înalt, C sau Pascal, spre exemplu). Dacă stiva se află în porțiunea de memorie referită de registrul de segment SS (Stack Segment), datele se află în mod normal în segmentul de memorie referit de către DS, registrul segment de date. Deoarece BX, SI şi DI se referă la segmentul de date, nu există o modalitate eficientă de a folosi regiştrii BX, SI, DI pentru a face referire la parametrii salvaţi în stivă din cauză că stiva este localizată într-un alt segment de memorie. Registrul BP oferă rezolvarea acestei probleme asigurând adresarea în segmentul de stivă. Spre exemplu, instrucţiunile:

```
PUSH  BP
MOV  BP, SP
MOV  AX, [ BP+4 ]
```

fac să se acceseze segmentul de stivă pentru a încărca registrul AX cu primul parametru trimis de un apel C unei rutine scrise în limbaj de asamblare. În concluzie, registrul BP este conceput astfel încât să ofere suport pentru accesul la parametri, variabile locale şi alte necesităţi legate de accesul la porţiunea de stivă din memorie.

Registrul SP

Registrul SP (Stack Pointer), sau pointerul de stivă, reţine de regulă adresa de deplasament a următorului element disponibil în cadrul segmentului de stivă. Acest registru este, probabil, cel mai puţin „general" dintre regiştrii de uz general, deoarece este dedicat mai tot timpul administrării stivei.

Registrul BP face în fiecare clipă referire la vârful stivei – acest vârf al stivei reprezintă adresa locaţiei de memorie în care va fi introdus următorul element în stivă. Acţiunea de a introduce un nou element în stivă se numeşte „împingere" (în engleză *push*); de aceea, instrucţiunea respectivă poartă numele de PUSH. În mod asemănător, operaţia de scoatere a unui element din vârful stivei poartă, în engleză, numele de *pop*, iar instrucţiunea echivalentă operaţiei se numeşte POP. În figurile 3 şi 4 sunt ilustrate modificările survenite în conţinutul stivei şi al regiştrilor SP, BX şi CX ca urmare a execuţiei instrucţiunilor următoare (se presupune că registrul SP are iniţial valoarea 1000):

```
MOV  BX, 9
PUSH  BX
MOV  CX, 10
PUSH  CX
POP  BX
POP  CX
```

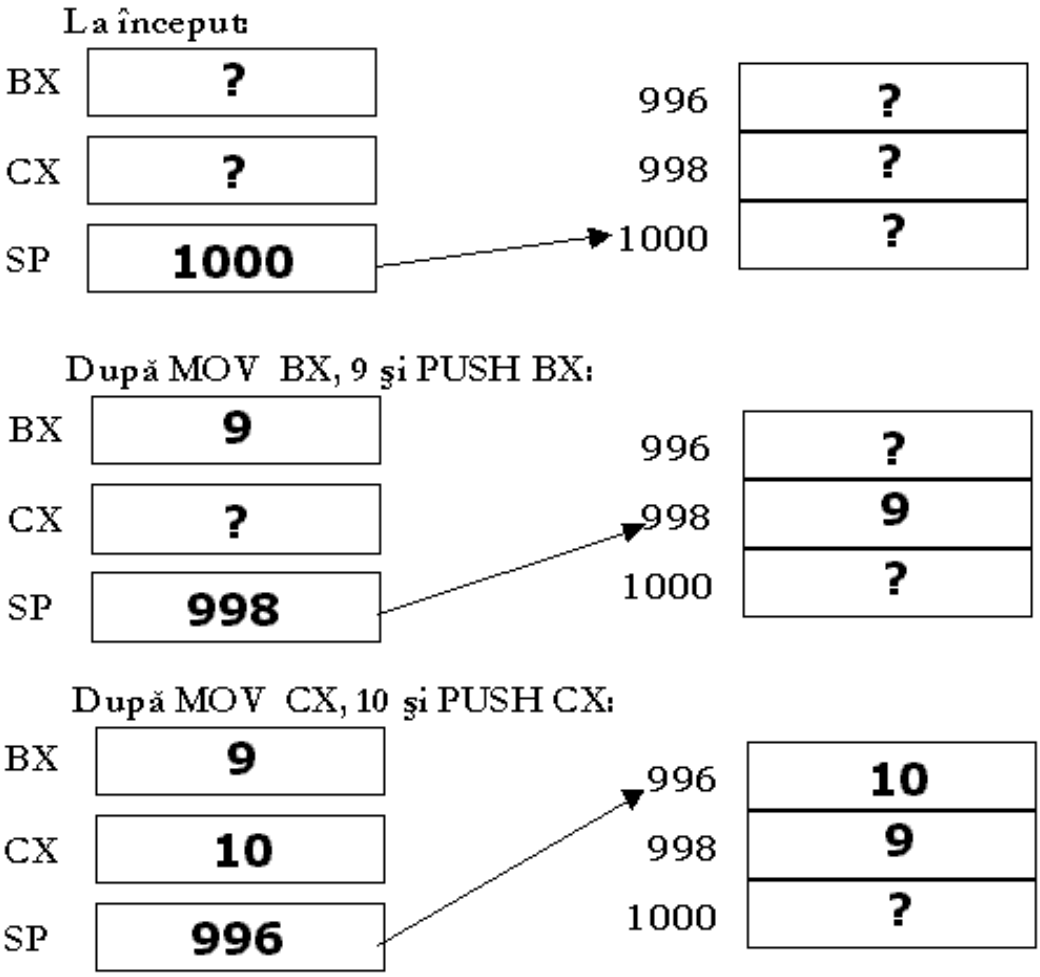


Figura 3. Modalitatea de funcţionare a stivei după execuţia primelor 4 instrucţiuni

Este permisă stocarea valorilor în registrul SP precum şi modificarea valorii sale prin adunare sau scădere la fel ca şi în cazul celorlalţi regiştri de uz general; totuşi, acest lucru nu este recomandat dacă nu suntem foarte siguri de ceea ce facem. Prin modificarea registrului SP, vom modifica adresa de memorie a vârfului stivei, ceea ce poate avea efecte neprevăzute, aceasta pentru că instrucţiunile PUSH şi POP nu reprezintă unicele modalităţi de utilizare a stivei. Indiferent dacă apelăm o subrutină sau ne întoarcem dintr-un astfel de apel de subrutină, fie procedură sau funcţie, în acest caz este folosită stiva. Unele resurse de sistem, precum tastatura sau ceasul de sistem, pot folosi stiva în momentul trimiterii unei întreruperi la microprocesor. Acest lucru presupune că stiva este folosită continuu, deci dacă se modifică registrul SP (adică adresa stivei), datele din noile locaţii de memorie nu vor mai fi cele corecte. În concluzie, registrul SP nu trebuie modificat în mod direct; el este modificat automat în urma instrucţiunilor POP, PUSH, CALL, RET. Oricare dintre ceilalţi regiştri de uz general pot fi modificaţi în mod direct în orice moment.

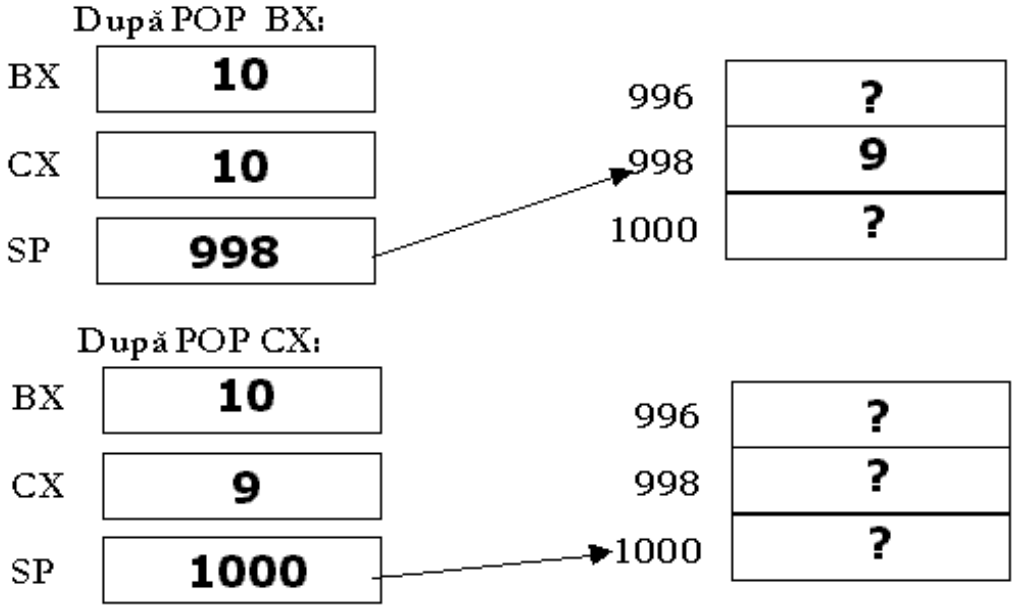


Figura 4. Funcţionarea stivei după ultimile două instrucţiuni POP

Registrul pointer de instrucţiuni (IP)

Registrul pointer de instrucţiuni (IP – Instruction Pointer, vezi figura 5) este folosit, întotdeauna, pentru a stoca adresa următoarei instrucţiuni ce va fi executată de către microprocesor. Pe măsură ce o instrucţiune este executată, pointerul de instrucţiune este incrementat şi se va referi la următoarea adresă de memorie (unde este stocată următoarea instrucţiune ce va fi executată). De regulă, instrucţiunea ce urmează a fi executată se află la adresa imediat următoare instrucţiunii ce a fost executată, dar există şi cazuri speciale (rezultate fie din apelul unei subrutine prin instrucţiunea CALL, fie prin întoarcerea dintr-o subrutină, prin instrucţiunea RET). Pointerul de instrucţiuni nu poate fi modificat sau citit în mod direct; doar instrucţiuni speciale pot încărca acest registru cu o nouă valoare. Registrul pointer de instrucţiune nu specifică pe de-a întregul adresa din memorie a următoarei instrucţiuni ce va fi executată, din aceeaşi cauză a segmentării memoriei. Pentru a aduce o instrucţiune din memorie, registrul CS oferă o adresă de bază iar registrul pointer de instrucţiune indică adresa de deplasament plecând de la această adresă de bază.

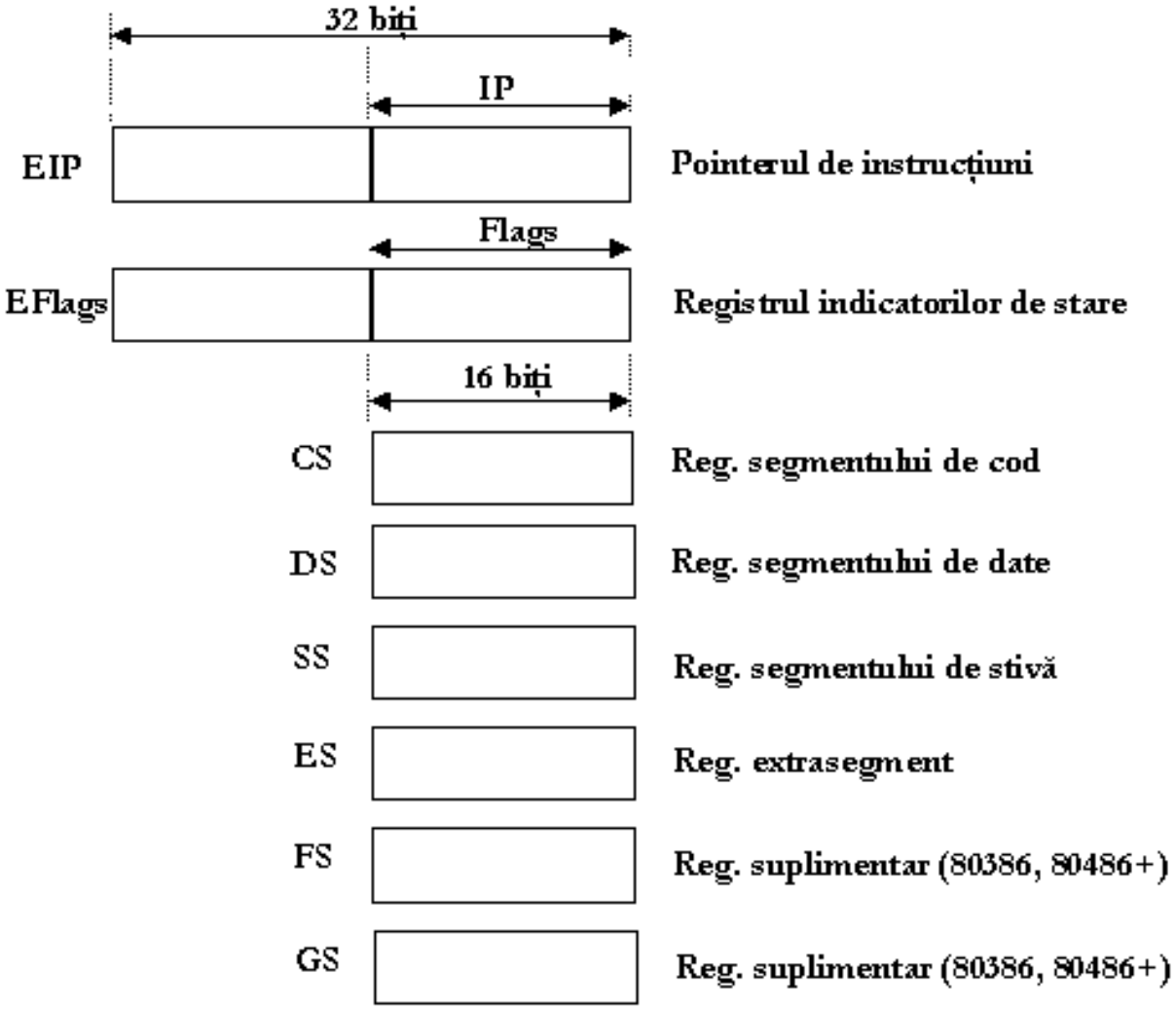


Figura 5. Regiştrii de segment, pointerul de instrucţiuni şi registrul indicatorilor de stare

Registrul indicatorilor de stare (FLAGS)

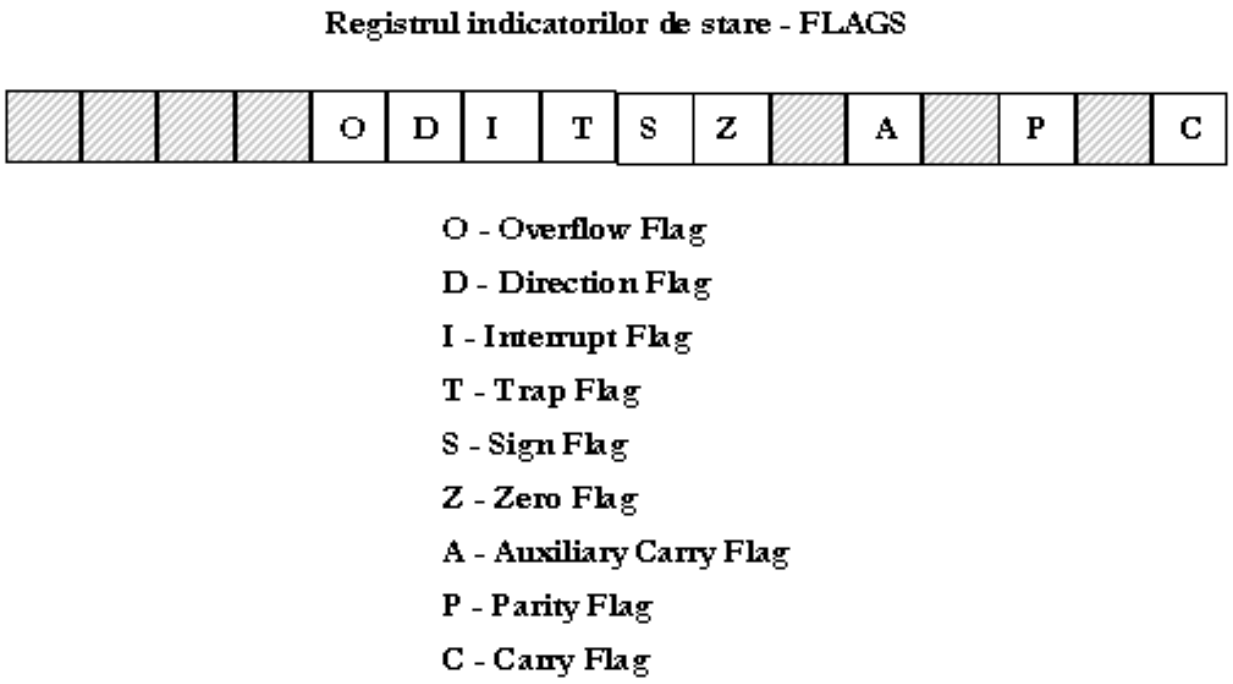


Figura 6. Registrul indicatorilor de stare - detaliu

Registrul indicatorilor de stare (FLAGS) pe 16 biţi conţine informaţii legate de starea microprocesorului precum şi de rezultatele ultimilor instrucţiuni executate. Un indicator de stare (*flag*) este în sine o locaţie de memorie de 1 bit ce indică starea curentă a microprocesorului şi modalitatea sa de operare. Un indicator se spune că "este setat" dacă are valoarea 1 şi "nu este setat" în caz contrar. Indicatorii de stare se modifică după execuţia unor instrucţiuni aritmetice sau logice. Exemple de indicatori de stare (vezi figura 6):

- C (Carry) indică apariţia unei cifre binare de transport în cazul unei adunări sau împrumut în cazul unei scăderi;
- O (Overflow) apare în urma unei operaţii aritmetice. Dacă este setat, înseamnă că rezultatul nu încape în operandul destinaţie;
- Z (Zero) indică faptul că rezultatul unei operaţii aritmetice sau logice ste zero;
- S (Sign) indică semnul rezultatului unei operaţii aritmetice;
- D (Direction) – când este zero, procesarea elementelor şirului se face de la adresa mai mică la cea mai mare, în caz contrar este invers;
- I (Interrupt) controlează posibilitatea microprocesorului de a răspunde la evenimente externe (apeluri de întrerupere);
- T (Trap) este folosit de programele de depanare (de tip debugger), activând sau nu posibilitatea execuţiei programului pas cu pas. Dacă este setat, UCP

întrerupe fiecare instrucțiune, lăsând programul depanator să execute programul respectiv pas cu pas;
- A (Auxiliary carry) suportă operații în codul BCD. Majoritatea programelor nu oferă suport pentru reprezentarea numerelor în acest format, de aceea se utilizează foarte rar;
- P (Parity) este stetat în conformitate cu paritatea biților cei mai puțin semnificativi ai unei operații cu date. Astfel, dacă rezultatul unei operații conține un număr par de biți 1, acest indicator este setat. Dacă numărul de biți 1 din rezultat este impar, atunci indicatorul PF este zero. Este folosit de regulă de programe de comunicații, dar Intel a introdus acest indicator nu pentru a îndeplini o anumită funcționalitate, ci pentru a asigura compatibilitatea cu vechile microprocesoare ale familiei x86.

Regiştrii de segment

Proprietățile regiştrilor de segment (vezi figura 5) sunt în strânsă legătură cu noțiunea de segmentare a memoriei. Premiza de la care se pleacă este următoarea: 8086 este capabil să adreseze 1MB de memorie, astfel că sunt necesare adrese pe 20 de biți pentru a cuprinde toate locațiile din spațiul de 1 MB de memorie. Totuşi, registrele utilizate sunt registre pe 16 biți, deci a trebuit să se găsească o soluție pentru această problemă. Soluția găsită se numeşte *segmentarea memoriei*; în acest caz memoria de 1MB este împărțită în 16 segmente de câte 64 KB (16*64 KB = 1024 KB = 1 MB).

Noțiunea de segmentare a memoriei presupune utilizarea unor adrese de memorie formate din două părți. Prima parte reprezintă adresa segmentului iar cea de-a doua porțiune reprezintă adresa de deplasament, sau offset-ul (figura 7).

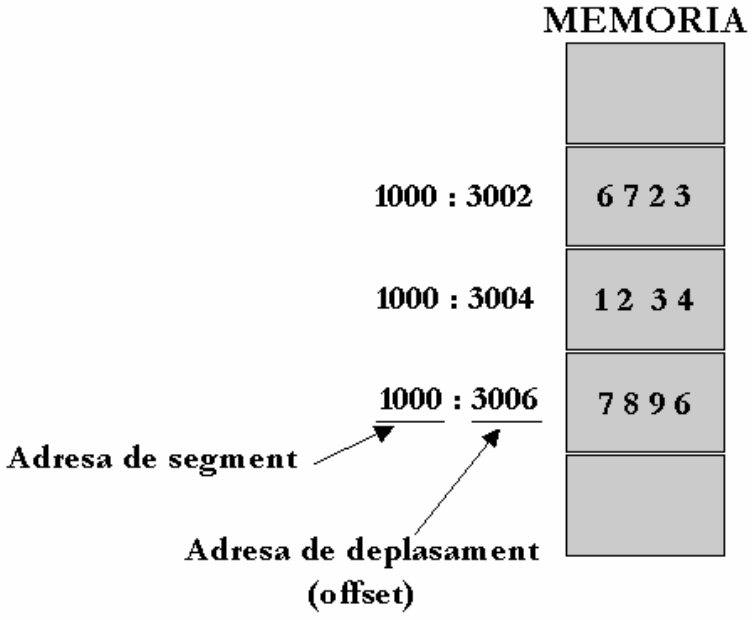


Figura 7. Cele două porțiuni ale unei adrese segmentate

Fiecare pointer de memorie pe 16 biți este combinat cu conținutul unui registru de segment pe 16 biți pentru a forma o adresă completă pe 20 de biți. Adresa de segment împreună cu adresa de deplasament sunt combinate în felul următor: valoarea de segment este deplasată la stânga cu 4 biți (înmulțită cu $16 = 2^4$) și apoi adunată cu valoarea adresei de deplasament. Adresa astfel construită se numește adresă efectivă; fiind o adresă pe 20 de biți poate accesa 2^{20} octeți de memorie, adică 1 MB de memorie. Construirea unei adrese efective este prezentată în figura 8.

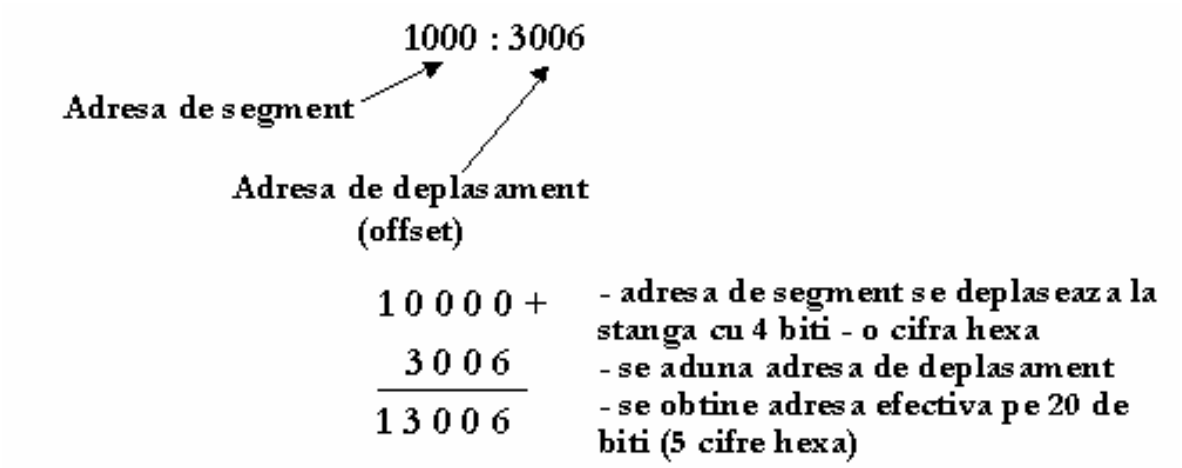


Figura 8. Exemplu de calcul al adresei efective

Registrul CS – acest registru face referire la începutul blocului de 64 KB de memorie în care se află codul programului (segmentul de cod). Microprocesorul 8086 nu poate aduce altă instrucțiune pentru execuție decât cea definită de CS. Registrul CS poate fi modificat de un număr de instrucțiuni, precum instrucțiuni de salt, apel sau de întoarcere. El nu poate fi încărcat în mod direct cu o valoare, ci doar prin intermediul unui alt registru general.

Registrul DS – face referire către începutul segmentului de date, unde se află mulțimea de date cu care lucrează programul aflat în execuție.

Registrul ES – face referire la începutul blocului de 64KB cunoscut sun denumirea de extra-segment. Acesta nu este dedicat nici unui scop anume, fiind disponibil pentru diverse acțiuni. Uneori acesta poate fi folosit pentru creearea unui bloc de memorie de 64 KB adițional pentru date. Acest extra-segment lucrează foarte bine în cazul instrucțiunilor de tip STRING. Toate instrucțiunile de tip STRING ce scriu în memorie folosesc adresarea ES : DI ca adresă de memorie.

Registrul SS – face referire la începutul segmentului de stivă, care este blocul de 64 KB unde se află stiva. Toate instrucțiunile ce folosesc implicit registrul SP (instrucțiunile POP, PUSH, CALL, RET) lucrează în segmentul de stivă deoarece registrul SP este capabil să adreseze memoria doar în segmentul de stivă.

Formatul general al unei instrucţiuni în limbaj de asamblare

O linie de cod scrisă în limbaj de asamblare are următorul format general:

<nume> <instructiune/directiva> <operanzi> <;comentariu>

unde:

- **<nume>** - reprezintă un nume simbolic opţional;
- **<instructiune/directiva>** - reprezintă mnemonica (numele) unei instrucţiuni sau a unei directive;
- **<operanzi>** - reprezintă o combinaţie de unul, doi sau mai mulţi operanzi (sau chiar nici unul), care pot fi constante, referinţe de memorie, referinţe de regiştri, şiruri de caractere, în funcţie de structura particulară a instrucţiunii;
- **<;comentariu>** - reprezintă un comentariu opţional ce poate fi plasat după caracterul „;” până la sfârşitul liniei respective de cod.

Nume de variabile şi etichete

Numele folosite într-un program scris în limbaj de asamblare pot identifica variabile numerice, variabile şir de caractere, locaţii de memorie sau etichete. Spre exemplu, următoarea secvenţă de cod, care calculează valoarea lui trei factorial (3!=1x2x3=6) cuprinde câteva nume de variabile şi etichete:

```
.MODEL  small
.STACK  200h
.DATA
Valoare_Factorial  DW  ?
Factorial  DW  ?
.CODE

Trei_Factorial  PROC
MOV  ax, @data
MOV ds, ax
MOV  [Valoare_Factorial], 1
MOV [Factorial], 2
MOV  cx, 2
Ciclare:
MOV ax, [Valoare_Factorial]
MUL [Factorial]
MOV  [Valoare_Factorial], ax
INC [Factorial]
LOOP Ciclare
```

```
RET
Trei_Factorial ENDP
END
```

Numele *Valoare_Factorial* şi *Factorial* sunt utilizate pentru definirea a două variabile de tip word (pe 16 biţi), *Trei_Factorial* identifică numele procedurii (subrutinei) ce conţine codul pentru calculul factorialului, permiţând apelul său din altă parte a programului. *Ciclare* reprezintă un nume de etichetă, identificând adresa instrucţiunii MOV ax, [Valoare_Factorial], astfel încât instrucţiunea LOOP folosită mai jos să poată face un salt înapoi la această instrucţiune. Numele de variabile pot conţine următoarele caractere: literele a-z şi A-Z, cifrele de la 0-9 precum şi caracterele speciale _ (*underscore* – liniuţă de subliniere), @ (*„at"* în engleză – citit şi „*a rond"* sau *„coadă de maimuţă"*), $ şi ?. Se poate folosi si caracterul punct (".") drept prim caracter al numelui unei etichete. Cifrele 0-9 nu pot fi utilizate pe prima poziţie a numelui; de asemenea, nu pot fi folosite nume care să conţină un singur caracter $ sau ?. Fiecare nume poate fi definit *o singură dată* (numele sunt *unice*) şi pot fi utilizate ca operanzi de oricâte ori se doreşte într-un program. Un nume poate să apară într-un program singur pe o linie (linia respectivă nu mai conţine altă instrucţiune sau directivă). În acest caz, valoarea numelui este dată de adresa instrucţiunii sau directivei de pe linia următoare din program. De exemplu, în secvenţa următoare:

```
...
JMP scadere
...
scadere:
SUB AX, CX
...
```

următoarea instrucţiune care va fi executată după instrucţiunea **JMP scadere** va fi instrucţiunea **SUB AX, CX**. Exemplul anterior este echivalent cu secvenţa:

```
...
JMP scadere
...
scadere: SUB AX, CX
...
```

Există unele avantaje atunci când scriem instrucţiunile pe linii separate. În primul rând, atunci când scriem un nume de etichetă pe o singură linie, este mai uşor să folosim nume lungi de etichete fără a strica „forma" programului scris în limbaj de asamblare. În al doilea rând, este mai uşor să adăugăm ulterior o nouă instrucţiune în dreptul etichetei dacă aceasta nu este scrisă pe aceeaşi linie cu instrucţiunea.

Numele variabilelor sau etichetelor folosite într-un program nu trebuie să se confunde cu numele *rezervate* de asamblor, cum ar fi numele de directive şi instrucţiuni, numele regiştrilor, etc. De exemplu, o declaraţie de genul:

```
...
ax  DW  0
BYTE:
...
```

nu poate fi acceptată, deoarece AX este numele registrului acumulator, AX, iar BYTE reprezintă un cuvânt cheie rezervat.

Orice nume de etichetă ce apare pe o linie fără instrucţiuni sau apare pe o linie cu instrucţiuni trebuie să aibă semnul „:" după numele ei. Totodată, se încearcă să se dea un nume sugestiv etichetelor din program. Fie următorul exemplu:

```
...
CMP AL, 'a'
JB  Nu_este_litera_mica
CMP AL, 'z'
JA  Nu_este_litera_mica
SUB AL, 20H         ; se transforma in litera mare
Nu_este_litera_mica:
…
```

comparativ cu:

```
...
CMP AL, 'a'
JB  x5
CMP AL, 'z'
JA  x5
SUB AL, 20H         ; se transforma in litera mare
x5:
…
```

Dacă în primul caz am folosit un nume sugestiv de etichetă (Nu_este_litera_mica), în cazul al doilea, identic din punct de vedere al funcţionalităţii cu primul, eticheta a fost denumită x5, absolut nesugestiv!

Observaţie:

Limbajul de asamblare nu este *case sensitive*. Aceasta semnifică faptul că, într-un program scris în limbaj de asamblare, numele de variabile, etichete, instrucţiuni, directive, mnemonice, etc., pot fi scrise atât cu litere mari cât şi cu litere mici, nefăcându-se diferenţa între ele (*Nu_este_litera_mica* este acelaşi lucru cu *nu_este_litera_mica* sau *Nu_Este_Litera_Mica*, etc.).

Directive de segment simplificate

Datorită faptului că regiştrii microprocesorului 8086 sunt regiştri pe 16 biţi, s-a impus folosirea unor segmente de memorie de câte 64Ko (maxim cât se poate adresa având la dispoziţie 16 biţi - 64Ko=2^16=65536). Într-un program scris în limbaj de asamblare (vom folosi în continuare prescurtarea ASM) există trei segmente: segmentul de cod, segmentul de date şi segmentul de stivă.

Directivele de segment (fie sub formă standard, fie sub formă simplificată) sunt necesare în orice program scris în limbaj de asamblare pentru a defini şi controla utilizarea segmentelor iar directiva END este folosită întotdeauna pentru a încheia codul programului.

Exemple de directive de segment simplificate sunt:

```
.STACK
.CODE
.DATA
.MODEL
DOSSEG
END
```

.STACK,.CODE,.DATA definesc, respectiv, segmentele de stivă, de cod şi de date.

De exemplu, **.STACK 200H** defineşte o stivă de 512 octeţi (în ASM valorile ce sunt încheiate cu litera H semnifică faptul că este vorba despre hexazecimal). O astfel de valoare pentru stivă este suficientă în mod normal; unele programe, însă (îndeosebi cele recursive) pot necesita dimensiuni mai mari ale stivei.

Directiva **.CODE** marchează începutul segmentului de cod.

Directiva **.DATA** marchează începutul segmentului de date, adică locul în care vom plasa variabilele de memorie. Reprezentativ aici este faptul că trebuie încărcat în mod explicit registrul de segment DS cu valoarea "@data" înaintea accesării locaţiilor de memorie în segmentul definit de .DATA. Având în vedere că un registru de segment poate fi încărcat fie dintr-un registru general fie dintr-o locaţie de memorie dar nu poate fi încărcat direct cu o constantă, registrul de segment DS este încărcat în general printr-o secvenţă de 2 instrucţiuni:

```
...
mov ax, @data
mov ds, ax
...
```

(se poate folosi şi alt registru general în locul lui AX).

Secvenţa anterioară semnifică faptul că DS se va referi către segmentul de date ce începe cu directiva .DATA.

Considerăm în continuare un exemplu de program ce afişează textul memorat în DataString pe ecran:

```
;Program p01.asm
.MODEL small          ;se specifică modelul de memorie SMALL
.STACK 200H           ;se defineşte o stivă de 512 octeţi
.DATA                 ;se specifică începutul segmentului de
                      ;date
DataString DB 'Hello!$'  ;se      defineşte      variabila
                      ;DataString, iniţializată cu valoarea
                      ;"Hello!"
.CODE                 ;începutul segmentului de cod al
                      ;programului
ProgramStart:         ;orice program are o etichetă de
                      ;început
mov bx,@data          ;secvenţa ce setează registrul DS să
                      ;facă referire la segmentul de date ce
                      ;începe cu .DATA
mov ds,bx
mov dx, OFFSET DataString ;se încarcă în DX adresa
;variabilei DataString
mov ah,09             ;codul funcţiei DOS de afişare a unui
                      ;string
int 21H               ;apelul DOS de afişare a string-ului
mov ah, 4cH           ;codul funcţiei DOS de terminare a
                      ;programului
int 21H               ;apelul DOS de terminare a programului
END ProgramStart      ;directiva de terminare a codului
                      ;programului
```

Explicaţii:

1. Se pot introduce comentarii într-un program ASM prin folosirea ";". Tot ce urmează după ";" şi până la sfârşitul liniei este considerat comentariu.

2. Nu are importanţă dacă programul este scris folosind litere mari sau mici (nu este "case sensitive").

3. Fără cele două instrucţiuni care setează registrul DS către segmentul definit de .DATA, funcţia de afişare a string-ului nu ar fi funcţionat cum trebuie. Variabila DataString se află în segmentul .DATA şi nu poate fi accesată dacă DS nu este poziţionat către acest segment. Acest lucru se explică în modul următor: atunci când facem apelul DOS de afişare a unui string, trebuie să parcurgem întreaga adresă de tipul segment:offset a string-ului în DS:DX. De aceea, de abia după ce am încărcat DS cu segmentul .DATA şi DX cu adresa (offset-ul) lui DataString avem o referinţă completă segment:offset către DataString.

Observaţii.

Nu trebuie să încărcăm în mod explicit registrul de segment CS deoarece DOS face acest lucru automat în momentul când rulăm un program. Astfel, dacă CS nu ar fi deja setat la momentul execuţiei primei instrucţiuni din program, procesorul nu ar şti unde să găsească instrucţiunea şi programul nu ar rula niciodată. În mod asemănător, registrul de segment SS este setat de DOS înainte de execuţia programului şi de regulă rămâne nemodificat pe perioada execuţiei programului.

Cu registrul de segment DS lucrurile stau altfel. În timp ce registrul CS se referă la intrucţiuni (cod), SS se referă ("pointează") la stivă, DS "pointează" la date. Programele nu manipulează direct instrucţiuni sau stive dar au de-a face în mod direct cu date. De asemenea, programele vor acces la date situate în segmente diferite în orice moment. Se poate dori încărcarea în DS a unui segment, accesarea datelor din acel segment şi apoi încărcarea lui DS cu un alt segment pentru a accesa un bloc diferit de date. În programe mici sau medii nu vom avea nevoie de mai mult de un segment de date dar programe mai complexe folosesc deseori segmente de date multiple.

Următorul program va afişa un caracter pe ecran, folosind încărcarea registrului ES în locul lui DS.

```
;Program p02.asm
.MODEL small
.STACK 200H
.DATA
OutputChar DB 'B'         ;definirea variabilei OutputChar
                          ;iniţializată cu valoarea "B"
.CODE
ProgramStart:
mov dx, @data
mov es, dx           ;spre deosebire de programul anterior,
                     se foloseşte ES pentru specificarea
                     segmentului de date
mov bx, offset OutputChar     ;se încarcă BX cu adresa
                              ;variabilei OutputChar
mov dl, es:[bx]           ;se încarcă AL cu valoarea de la
                          ;adresa explicită es:[bx]
                          ;(adresare indexată)
mov ah,02                 ;codul funcţiei DOS de afişare a
                          ;unui caracter
int 21H              ;apelul DOS de execuţie a afişării
mov ah, 4cH          ;codul funcţiei DOS de terminare a
                     ;programului
int 21H              ;apelul DOS de terminare a programului
END ProgramStart     ;directiva de terminare a codului
                     ;programului
```

DOSSEG este directiva ce face ca segmentele dintr-un program să fie grupate conform convenţiilor Microsoft de adresare a segmentelor.

Directiva **.MODEL**

Este directiva ce specifică modelul de memorie pentru un program ASM ce foloseşte directive de segment simplificate.

Definiţii: "near" înseamnă adresa (offset-ul) pe 16 biţi din cadrul aceluiaşi segment, în timp ce "far" înseamnă o adresă completă de tip segment:offset, din cadrul altui segment decât cel curent.

Modelele de memorie ce se pot specifica prin intermediul directivei .MODEL sunt:

- ***tiny*** - atât codul cât şi datele programului încap în acelaşi segment de 64Ko. Atât codul cât şi datele sunt de tip near.
- ***small*** - codul programului trebuie să fie într-un singur segment de 64Ko şi datele într-un bloc separat de 64Ko; codul şi datele sunt near
- ***medium*** - codul programului poate fi mai mare decât 64Ko dar datele trebuie să fie într-un singur segment de 64 Ko. Codul este far, datele sunt near.
- ***compact*** - codul programului poate fi într-un singur segment, datele pot fi mai mari de 64 Ko. Codul este near, datele sunt far.
- ***large*** - atât codul cât şi datele pot depăşi 64Ko, dar nici un masiv de date nu poate depăşi 64 Ko. Atât codul cât şi datele sunt far.
- ***huge*** - atât codul cât şi datele pot depăşi 64Ko şi masivele de date pot depăşi 64 Ko. Atât codul cât şi datele sunt far. Pointerii la elementele dintr-un masiv sunt far.

În continuare sunt prezentate câteva exemple legate de modalităţile de declarare a variabilelor şi de adresare a memoriei.

```
var1 DW 01234h      ;se defineste o variabila word cu
                    ;valoarea 1234h
var2 DW 01234       ;se defineste o variabila word cu
                    ;valoarea zecimala 1234 (4D2 in hexa)
var3 RESW 1         ;se rezerva spatiu pentru o variabila
                    ;word (de valoare 0)
var4 DW ABCDh       ;atribuire ilegala!

 mesajsco2 DB 'SCO 2 este cursul preferat!'

...start:

  mov ax,cs         ; setarea  segmentului de date
  mov ds,ax         ; DS=CS
```

```
     ; orice referinta de memorie se presupune ca este relativa
la segmentul DS

        mov ax,[var2]           ; AX <- var2
                                ; == mov ax,[2]

        mov si,var2       ;se  foloseste  SI  ca  pointer  catre
                          var2 (cod C echivalent SI=&var2)

        mov ax,[si]       ;se  citeste  din  memorie  valoarea  lui
                          ;var2 (*(&myvar2))
                          ;(referinta indirecta)

        mov bx,mesajsco2 ; BX este pointer la un string
                          ; (cod C echivalent:  BX=&mesajsco2)

        dec BYTE [bx+1]  ; transforma 'C' in 'B' !

        mov si, 1         ; Foloseste SI cu rol de index
        inc byte [mesajsco2+SI]  ; == inc byte [SI + 8]
                                 ; == inc byte [9]

     ; Memoria poate fi adresata folosindu-se 4 registri:
     ; SI  -> Implica DS
     ; DI  -> Implica DS
     ; BX  -> Implica DS
     ; BP  -> Implica SS !  (nu este foarte des utilizat)
     ;
     ;Exemple:

     mov ax,[bx]     ; ax <- word in memorie referit de BX
     mov al,[bx]     ; al <- byte in memorie referit de BX
     mov ax,[si]     ; ax <- word referit de SI
     mov ah,[si]     ; ah <- byte referit de SI
     mov cx,[di]     ; di <- word referit de DI

     mov ax,[bp]     ; AX <- [SS:BP] Operatie cu stiva!

     ; In plus, sunt permise BX+SI si BX+DI:

     mov ax,[bx+si]
     mov ch,[bx+di]

     ; Deplasamente pe 8 sau 16 biti:

     mov ax,[23h]         ; ax <- word in memorie DS:0023
     mov ah,[bx+5]        ; ah <- byte in memorie [DS:BX+5]
     mov ax,[bx+si+107]   ; ax <- word la adresa [DS:BX+SI+107]
     mov ax,[bx+di+47]    ; ax <- word la adresa [DS:BX+DI+47]
```

```
; ATENTIE: copierea din memorie in memorie este ilegala!
;Totdeauna trebuie sa se treaca valoarea copiata printr-un
;registru

mov [bx],[si]   ;Ilegal
mov [di],[si]   ;Ilegal

; Caz special: operatiile cu stiva!

pop word [var]          ; var <- [SS:SP]
```

Adrese de memorie şi valori

Un program scris în limbaj de asamblare se poate referi fie la o adresă de memorie (OFFSET = DEPLASAMENT), fie la o valoare stocată de variabilă în memorie. Din păcate, limbajul de asamblare nu este nici strict, nici intuitiv cu privire la modurile în care aceste două tipuri de referire sunt făcute şi, drept urmare, referirile la OFFSET sau la valoare sunt deseori confundate. În figura 9 sunt ilustrate conceptele de adresă de deplasament (offset) şi valoare stocată în memorie.

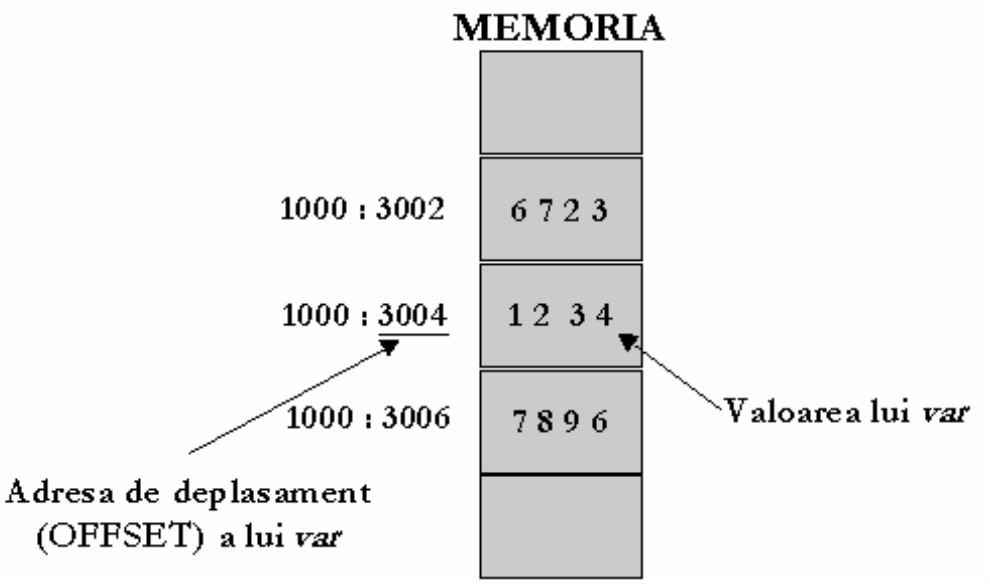


Figura 9. Ilustrarea noţiunilor de adresă de deplasament şi valoare stocată în memorie

Deplasamentul unei variabile de memorie *var* de dimensiune word este valoarea constantă 5004H, obţinută cu operatorul OFFSET. Spre exemplu, instrucţiunea:

MOV BX, OFFSET var

Încarcă valoarea 5004H în registrul BX. Valoarea 5004H nu se modifică; ea este construită în cadrul instrucţiunii. Valoarea lui *var* este 1234H, citită din memorie la adresa dată de offset-ul 5004H din segmentul de date. O modalitatea de citire a acestei valori este de a încărca registrele BX, SI, DI sau BP cu offset-ul lui *var* şi apoi folosirea registrului respectiv pentru adresarea memoriei. Instrucţiunile:

```
MOV BX, OFFSET var
MOV AX, [ BX ]
```

Au ca efect încărcarea valorii lui *var* (1234H) în registrul AX.
De asemenea, se poate încărca valoarea lui *var* direct în AX folosind:

```
MOV AX, var
Sau
MOV AX, [ var ]
```

În timp ce valoarea deplasamentului rămâne constantă, valoarea 1234H nu este permanent asociată cu *var*. De exemplu, instrucţiunile:

```
MOV [ var ], 5555H
MOV AX, [ var ]
```

Au ca efect încărcarea valorii 5555H în registrul AX.
Cu alte cuvinte, în timp ce deplasamentul lui *var* este o valoare constantă ce descrie o adresă fixă dintr-un segment de date, valoarea variabilei *var* este un număr ce poate fi modificat şi care se află memorat la adresa (de memorie) respectivă. Instrucţiunile:

```
MOV [ var ], 1
ADD [ var ], 2
```

Modifică valoarea lui var la 3, dar instrucţiunea:

ADD OFFSET var, 2 este echivalentă cu **ADD 5002H, 2**, ceea ce este un lucru fără sens, deoarece este imposibil să se însumeze o constantă cu alta.

O problemă ce poate apărea adesea în timpul programării este aceea a omiterii lui OFFSET; de exemplu, dacă scriem **MOV SI, var** atunci când, de fapt, dorim încărcarea în SI a deplasamentului lui *var*. Nu va fi semnalată nici o eroare în acest caz, având în vedere că *var* este o variabilă de tip word. Totuşi, în momentul execuţiei programului, registrul SI va fi încărcat cu valoarea lui *var* (1234H), în loc de OFFSET, ceea ce poate conduce la rezultate imprevizibile. În acest caz, referirile la constantele de adresă vor fi precedate de OFFSET iar referirile la valori din memorie să fie cuprinse între paranteze drepte („[” şi „]”), eliminând astfel ambiguitatea.

Instrucţiuni ale microprocesorului Intel

Microprocesoarele din familia Intel x86 dispun de o serie impresionantă de instrucţiuni, asemeni tuturor procesoarelor din clasa procesoarelor CISC (Complex Instruction Set Computer). Instrucţiunile se pot împărţi în: instrucţiuni logice, aritmetice, de transfer şi de control. Prezentăm în continuare câteva exemple din fiecare clasă de instrucţiuni.

Instrucţiuni logice

Instrucţiunile logice implementează funcţiile logice de bază, pe un octet sau pe cuvânt. Ele acţionează bit cu bit, deci se aplică funcţia logică respectivă tuturor biţilor sau perechilor de biţi corespunzători operanzilor. Instrucţiunile logice sunt următoarele:

- **NOT**: A =~A
- **AND**: A &= B
- **OR**: A |= B
- **XOR**: A ^= B
- **TEST**: A & B

De regulă, instrucţiunile logice au efect asupra indicatorilor de stare, cu excepţia instrucţiunii NOT, care nu are efect asupra nici unui flag (indicator de stare). Aceste efecte sunt următoarele:

- Se şterge indicatorul carry (C)
- Se şterge indicatorul overflow (O)
- Se setează zero flag (Z) dacă rezultatul este zero, sau îl şterge în caz contrar
- Se copiază bitul mai "înalt" al rezultatului în indicatorul sign (S)
- Se setează bitul de paritate (P) conform cu *paritatea* rezultatului

Instrucţiunea NOT

Este o instrucţiune cu un singur operand (instrucţiune *unară*), cu forma generală:

NOT *destinatie*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi. Instrucţiunea are ca efect inversarea (negarea) tuturor biţilor operandului, adică aducerea în forma codului invers - complement faţă de 1.

Instrucţiunea AND

Este o instrucțiune cu doi operanzi (instrucțiune *binară*), cu forma generală:

AND *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are ca efect operația: <destinatie> == <destinatie> AND <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea TEST (AND "non-distructiv")

Este o instrucțiune cu doi operanzi (instrucțiune *binară*), cu forma generală:

TEST *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are același efect ca și instrucțiunea AND, cu deosebirea că nu se modifică operandul destinație, iar indicatorii de stare sunt modificați în același mod ca și în cazul instrucțiunii AND.

Instrucțiunea OR

Este o instrucțiune cu doi operanzi, cu forma generală:

OR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: <destinatie> == <destinatie> OR <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea XOR (SAU-Exclusiv)

Este o instrucțiune cu doi operanzi, cu forma generală:

XOR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: <destinatie> == <destinatie> XOR <sursa>. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit. Funcția XOR, denumită SAU-Exclusiv (sau *anti-coincidență*) are valoarea logică 1 atunci când operanzii săi sunt diferiți (unul are valoarea 0 iar celălalt valoarea 1) și valoarea logică 0 când ambii operanzi au aceeași valoare (fie ambii au valoarea 0, fie ambii au valoarea 1).

Observație:

De cele mai multe ori, instrucțiunile AND și OR sunt folosit pe post de „mascare" a datelor; în acest sens, o valoarea de tip „mască" (*mask*) este utilizată pentru a forța anumiți biți să ia valoarea zero sau valoarea 1 în cadrul altei valori. O

astfel de „mască” logică are efect asupra anumitor biți, în timp ce pe alții îi lasă neschimbați. Exemple:

- Instrucțiunea **AND CL, 0Fh** – face ca cei mai semnificativi 4 biți să ia valoarea 0, în timp ce biții mai puțin semnificativi sunt lăsați neschimbați; astfel, dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **AND CL, 0Fh** va avea valoarea **0000 1101**.
- Instrucțiunea **OR CL, 0Fh** – face ca cei mai puțin semnificativi 4 biți să ia valoarea 1, în timp ce biții mai semnificativi să rămână nemodificați. Dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **OR CL, 0Fh** va avea valoarea **1001 1111**.

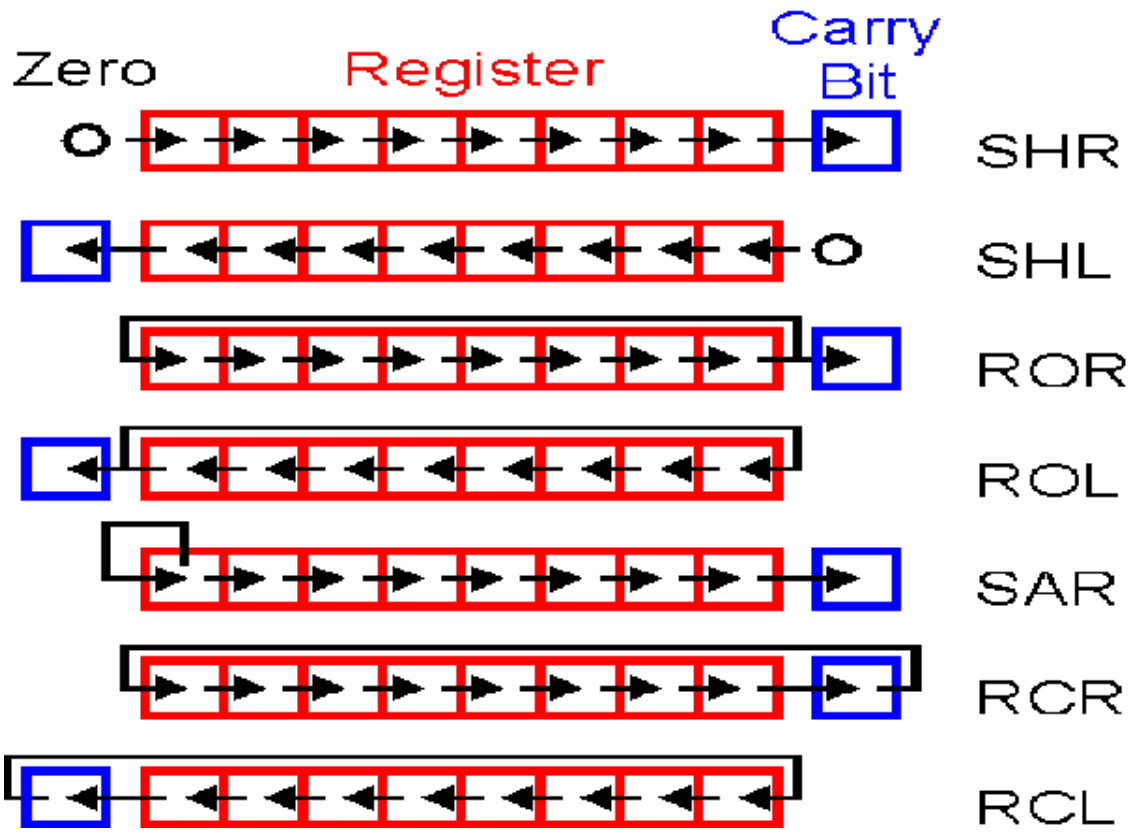


Figura 10. Instrucțiuni de deplasare şi de rotație

Instrucțiuni de deplasare şi de rotație

Acest tip de instrucțiuni (vezi figura **10**) permit realizarea operațiilor de deplasare şi de rotație la nivel de bit. Ele au doi operanzi, primul operand fiind cel asupra căruia se aplică operația de deplasare pe biți, iar cele de-al doilea (operandul *numărător* sau *contor*) semnifică numărul de biți cu care se face această deplasare. Operațiile se pot face de la dreapta spre stânga sau invers. Deplasarea înseamnă

translatarea tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în poziția rămasă liberă și cu pierderea biților din dreapta/stânga. Rotația presupune translatarea biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă. Sintaxa generală a instrucțiunilor de deplasare și rotație este următoarea:

INSTR <operand> , <contor>

Unde **INSTR** reprezintă numele instrucțiunii, **<operand>** reprezintă un registru sau o locație de memorie pe 8 sau 16 biți, iar **<contor>** semnifică numărul de biți cu care se face deplasarea, adică fie o constantă, fie registrul CL (care își confirmă astfel rolul de numărător).

Observație.

Totdeauna există două modalități de deplasare:

- Prin folosirea unui contor efectiv – de exemplu: SHL AX, 1
- Prin folosirea registrului CL pe post de contor – de exemplu: SHL AX, CL

Instrucțiunea SHL/SAL (Shift Left/Shift Arithmetic Left)

Această instrucțiune translatează biții operandului o poziție la stânga de câte ori specifică operandul numărător. Pozițiile rămase libere prin deplasarea la stânga sunt umplute cu zerouri la bitul cel mai puțin semnificativ, în timp ce bitul cel mai semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de înmulțire cu o putere a lui 2 (în funcție de numărul de biți pentru care se face deplasarea la stânga).

Exemple:

1. Înmulțirea lui AX cu 10 (1010 în binar) (înmulțim cu 2 și cu 8, apoi adunăm rezultatele)

```
shl   ax, 1      ; AX ori 2
mov   bx, ax     ; salvăm 2*AX în BX
shl   ax, 2      ; 2*AX(original) * 4 = 8*AX(original)
add   ax, bx     ; 2*AX + 8*AX = 10*AX
```

2. Înmulțirea lui AX cu 18 (10010 în binar) (înmulțim cu 2 și cu 16, apoi adunăm rezultatele)

```
shl   ax, 1      ; AX ori 2
```

```
mov   bx, ax        ; salvăm 2*AX
shl   ax, 3         ; 2*AX(original) ori 8 = 16*AX(original)
add   ax, bx        ; 2*AX + 16*AX = 18*AX
```

Instrucţiunea SHR (Shift Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai puţin semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *fără semn* la o putere a lui 2 (dacă deplasarea se face cu *o poziţie* la dreapta, operaţia este echivalentă cu o împărţire la 2, dacă deplasarea se face cu două poziţii, operaţia este echivalentă cu o împărţire la 2^2, etc.). Operaţia de împărţire se execută *fără semn*, completându-se cu un bit 0 dinspre stânga (bitul cel mai semnificativ).

Instrucţiunea SAR (Shift Arithmetic Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai semnificativ rămâne neschimbat, în timp ce bitul cel mai puţin semnificativ este copiat în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *cu semn* la o puterea a lui 2 (în funcţie de numărul de biţi cu care se face deplasarea la dreapta).

Instrucţiunea RCL (Rotate through Carry Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga prin intermediul lui CF (Carry Flag). Astfel, cel mai semnificativ bit trece din operand în CF, apoi se deplasează toţi biţii din operand cu o poziţie la stânga iar CF original trece în bitul cel mai puţin semnificativ din operand.

Instrucţiunea ROL (Rotate Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga. Astfel, cel mai semnificativ bit trece din operand în bitul cel mai puţin semnificativ.

Exemplu:

După execuţia instrucţiunilor:
ROL AX, 6
AND AX, 1Fh
Biţii 10-14 din AX se mută în biţii 0-4.

Instrucţiunea RCR (Rotate through Carry Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta prin intermediul lui CF (Carry Flag). Astfel, bitul din CF este scris înapoi în bitul cel mai semnificativ al operandului.

Instrucţiunea ROR (Rotate Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta. Bitul cel mai puţin semnificativ trece în bitul cel mai semnificativ.

Exemple:

```
MOV ax,3     ; Valori iniţiale              AX = 0000 0000 0000 0011
MOV bx,5     ;                              BX = 0000 0000 0000 0101
OR ax,9      ;      ax <- ax | 0000 1001    AX = 0000 0000 0000 1011
AND ax,10101010b  ; ax <- ax & 1010 1010    AX = 0000 0000 0000 1010
XOR ax,0FFh       ; ax <- ax ^ 1111 1111    AX = 0000 0000 1111 0101
NEG ax            ; ax <- (-ax)             AX = 1111 1111 0000 1011
NOT ax            ; ax <- (~ax)             AX = 0000 0000 1111 0100
OR ax,1           ; ax <- ax | 0000 0001    AX = 0000 0000 1111 0101
SHL ax,1     ; depl logică la stg cu 1 bit  AX = 0000 0001 1110 1010
SHR ax,1     ; depl logică la dr cu 1 bit   AX = 0000 0000 1111 0101
ROR ax,1     ; rotaţie stg (LSB=MSB)        AX = 1000 0000 0111 1010
ROL ax,1     ; rotaţie dr (MSB=LSB)         AX = 0000 0000 1111 0101
MOV cl,3     ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHR ax,cl    ; împărţim AX la 8             AX = 0000 0000 0001 1110
MOV cl,3     ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHL bx,cl    ; înmulţim BX cu 8             BX = 0000 0000 0010 1000
```

Instrucţiuni aritmetice

Instrucţiunea ADD (ADDition)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Cei doi operanzi nu pot fi însă în acelaşi timp locaţii de memorie. Rezultatul operaţiei este

următorul: <destinatie> == <destinatie> + <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Dacă apare ambiguitate la modul de exprimare al operanzilor (8 sau 16 biţi) se va folosi operatorul PTR.

Exemple:

```
ADD  AX, BX        ; adunare între regiştri – AX ← AX + BX
ADD  DL, 33h       ; adunare efectivă - DL ← DL + 33h
MOV DI, NUMB   ; adresa lui NUMB
MOV AL, 0          ; se şterge suma
ADD AL, [DI]       ; adună [NUMB]
ADD AL, [DI + 1] ; adună [NUMB + 1]
ADD word ptr [DI], -2     ; destinatie in memorie, sursa imediata
ADD byte ptr VAR, 5       ; fortarea instructiunii pe un octet, VAR fiind
                          ;declarat DW
```

Instrucţiunea INC (Increment addition)

Instrucţiunea INC are formatul general:

INC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biţi iar semnificaţia operaţiei este incrementarea valorii destinaţie cu 1. Toţi indicatorii de stare sunt afectaţi, cu excepţia lui CF (Carry Flag).

Exemplu:

```
MOV DI, NUMB       ; adresa lui NUMB
MOV AL, 0          ; şterge suma
ADD AL, [DI]       ; adună [NUMB]
INC   DI           ; DI = DI + 1
ADD AL, [DI]       ; adună [NUMB + 1]
```

Instrucţiunea ADC (ADdition with Carry)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată.

Instrucţiunea acţionează întocmai ca ADD, cu deosebirea că la rezultat este adăugat şi bitul CF. Este utilizat, de regulă, pentru a aduna numere mai mari de 16 biţi (8086-80286) sau mai mari de 32 de biţi la 80386, 80486, Pentium.

Exemplu:

Adunarea a două numere pe 32 de biţi se poate face astfel (BXAX) + (DXCX):

```
ADD AX, CX
ADC BX, DX
```

Instrucţiunea SUB (SUBstract)

Instrucţiunea SUB are formatul general:

SUB <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Rezultatul operaţiei este următorul: <destinatie> == <destinatie> - <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Scăderea poate fi văzută ca o adunare cu reprezentarea în complementul faţă de 2 al operandului sursă şi cu inversarea bitului CF, în sensul că, dacă la operaţie (adunarea echivalentă) apare transport, CF=0 şi dacă la adunarea echivalentă nu apare transport, CF=1.

Pentru instrucţiunile:

```
MOV CH, 22h
SUB CH, 34h
```

Rezultatul este **-12 (1110 1110)**, iar indicatorii de stare se modifică astfel:

ZF = 0 (rezultat diferit de zero)
CF = 1 (împrumut)
SF = 1 (rezultat negativ)
PF = 0 (paritate pară)
OF = 0 (fără depăşire)

Instrucţiunea DEC (DECrement substraction)

Instrucțiunea DEC are formatul general:

DEC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biți iar semnificația operației este decrementarea valorii destinație cu 1. Toți indicatorii de stare sunt afectați, cu excepția lui CF (Carry Flag).

Instrucțiunea SBB (SuBstract with Borrow)

Instrucțiunea SBB are formatul general:

SBB <destinatie>, <sursa>

Unde <*destinatie*> şi <*sursa*> pot fi registru sau operand în memorie, pe 8 sau pe 16 biți. Rezultatul operației este următorul: <destinatie> == <destinatie> - <sursa> - CF, deci la fel ca şi în cazul instrucțiunii SUB, dar din rezultat se scade şi bitul CF. Indicatorii de stare modificați în urma acestei operații sunt: AF, CF, PF, SF, ZF, OF. Această instrucțiune este utilizată, de regulă, pentru a scădea numere mai mari de 16 biți (la 8086 - 80286) sau de 32 de biți (la 80386, 80486, Pentium).

Exemplu

Scăderea a două numere pe 32 de biți se poate face astfel (BXAX) - (SIDI):

```
SUB   AX, DI
SBB   BX, SI
```

Exemple de programe

1. Program care citeşte un număr de la tastatură şi afişează dacă numărul este par sau nu:

```
; Programul citeste un numar si afiseaza un mesaj referitor la paritate
dosseg
.model small
.stack
.data

mesaj  db 13,10,'Introduceti numarul:(<=9)$'
mesg_par  db 13,10,'Numarul introdus este par!$'
mesg_impar  db 13,10,'Numarul introdus este impar!$'
```

```
.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          mov bx,2
          div bx ; se imparte AX la BX, catul va fi in AX, restul in DX
          cmp dx,0
          jnz impar
          mov ah,09
          mov dx,offset mesg_par
          int 21h
          jmp sfarsit
impar:  mov ah,09
          mov dx,offset mesg_impar
          int 21h
sfarsit:
          mov ah,4ch
          int 21h ; sfarsitul programului

END pstart
```

2. Program care calculează pătratul unui număr introdus de la tastatură.

```
; Programul calculeaza patratul unui numar (<=256) introdus de la tastatura
; Valoarea patratului se calculeaza in registrul AX (valoare maxima 2^16 = 65536)

dosseg
.model small
.stack
.data

nr DB 10,10 dup(0)
r DB 10, 10 dup(0)
```

```
mesaj  db 13,10,'Introduceti numarul:(<=256)$'
patrat db 13,10,'Patratul numarului este:$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov cl,nr[1]  ; incarc in CL numarul de cifre al numarului introdus
          inc cl        ; in sir se va merge pana la pozitia cl+1
          mov si,1      ; folosesc registrul SI pe post de contor
          xor ax,ax     ; initializez AX cu valoarea 0
          mov bl,10     ; se va inmulti cu valoarea 10 care este stocata in BL
inmultire:
          mul bl
          inc si
          mov dl,nr[si]
          sub dl,30h
          add ax,dx
          cmp si,cx
          jne inmultire

          mul ax

          xor si,si
          mov bx,10

cifra: ; aici incepe afisarea rezultatului din AX
          div bx
          add dl,30h
          mov r[si],dl
          inc si
          xor dx,dx
```

```
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset patrat
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter
        jmp sfarsit


        mov ah,9
        mov dx,offset patrat
        int 21h

sfarsit:
        mov ah,4ch
        int 21h ; stop program



        END pstart
```

3. Program care calculează valoarea unui număr ridicat la o putere. Atât numărul cât şi exponentul (puterea) sunt introduse de la tastatură.

```
; Programul calculeaza un numar ridicat la o putere

; Observatie. Deoarece rezultatul se calculeaza in registrul AX care este un
; registru pe  16 biti, valoarea maxima calculata corect este 2^16= 65536

.model small
.stack
.data

mesaj1  db 13,10,'Introduceti numarul:(<=9)$'
mesaj2  db 13,10,'Introduceti puterea:(<=9)$'
```

```
mesaj_final  db 13,10,'Rezultatul este: $'
mesaj_putere_0 db 13,10, 'Orice numar ridicat la puterea 0 este 1! $'

r db 30 dup(0) ; in variabila r se va stoca reultatul

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj1
        int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h ; se obtine valoarea numerica
                  ; scazandu-se codul lui 0 in ASCII (30H)
          push ax ; se salveaza valoarea lui ax in stiva

          mov ah,09
          mov dx,offset mesaj2
          int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
                  ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h   ; se obtine valoarea numerica
                        ; scazandu-se codul lui 0 in ASCII (30H)
          mov cx,ax     ; registrul CX contorizeaza numarul de inmultiri
          cmp cx,0
          jne putere_0
          mov ah,09
          mov dx, offset mesaj_putere_0
          int 21h
          jmp sfarsit

putere_0:
          pop bx          ;se salveaza in BX valoarea cu care inmulteste
          mov ax,0001
inmultire:
```

```
        mul bx
        loop inmultire

        xor si,si
        mov bx,10
cifra:
        div bx
        add dl,30h
        mov r[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset mesaj_final
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter



sfarsit:
        mov ah,4ch
        int 21h ; sfarsitul programului

END pstart
```

4. Program care verifică dacă un număr este palindrom (un număr se numeşte palindrom dacă scris de la dreapta la stânga sau invers are aceeaşi valoare).

```
; Programul verifica daca un numar sau sir de caractere este palindrom


dosseg
.model small
.stack
```

```
.data

nr DB 10,10 dup(0)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_nu db 13,10,'Numarul nu este palindrom!$'
mesaj_da db 13,10,'Numarul este palindrom!$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov si,1
          mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
          and cx,00FFh

          mov ax,cx
          mov bl,2
          div bl          ; in AL este catul impartirii lui AX la 2
          and ax,00FFh
          inc ax
          inc cx
          mov di,cx

urmatorul_caracter:
          inc si          ; SI creste de la inceputul sirului spre mijloc
          mov bl,nr[di]
          cmp nr[si],bl
          jne nu_este
          dec di          ; DI scade de la sfarsitul sirului spre mijloc
          cmp si,ax       ; in sir se va merge pana la pozitia cl+1
          jne urmatorul_caracter
          mov ah,9
```

```
        mov dx,offset mesaj_da
        int 21h
        jmp sfarsit

nu_este:
        mov ah,9
        mov dx,offset mesaj_nu
        int 21h


sfarsit:
        mov ah,4ch
        int 21h ; stop program

END pstart
```

5. Program care calculează suma cifrelor unui număr introdus de la tastatură.

```
; Programul calculeaza suma cifrelor unui numar introdus de la tastatura


dosseg
.model small
.stack
.data

nr DB 10,10 dup(?)
rezultat DB 10,10 dup(?)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_suma db 13,10,'Suma cifrelor numarului este: $'

.code

pstart:
      mov ax,@data
      mov ds,ax

      mov ah,09               ; aici se afiseaza mesajul initial de introducere
      mov dx,offset mesaj  ; a numarului
      int 21h
```

```
        mov ah,0ah          ; functia 10(0ah) citeste un sir de caractere de la
                            ; tastatura intr-o variabila de memorie
        mov dx,offset nr
        int 21h

        mov si,1
        mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
        and cx,00FFh
        inc cx          ; CX stocheaza acum ultima pozitie din sirul de cifre
        xor ax,ax       ; stocam rezultatul in AX, pe care il initializam cu zero

urmatorul_caracter:
        inc si          ; SI creste de la inceputul sirului spre sfarsit
        add al,nr[si]

        sub al,30h      ; scadem codul ASCII al lui zero

        cmp si,cx       ; in sir se va merge pana la pozitia cl+1
        jne urmatorul_caracter

        xor si,si       ; SI este indicele din sirul care va contine rezultatul

cifra: ; aici incepe afisarea rezultatului din AX

        mov bx,0ah
        div bx
        add dl,30h
        mov rezultat[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx,offset mesaj_suma
        int 21h

caracter:
        dec si
        mov ah,02           ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,rezultat[si] ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
```

```
        jne caracter

        mov ah,4ch
        int 21h                 ; terminarea programului

END pstart
```

Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный университет леса»

Факультет электроники и системотехники
Кафедра электроники и микропроцессорной техники

И.В. Антошина, Ю.Т. Котов

Микропроцессоры и микропроцессорные системы

(аналитический обзор)

Москва 2005 г.

Рекомендовано к изданию редакционно-издательским советом университета в качестве учебного пособия

Разработано в соответствии с Государственным образовательным стандартом ВПО 2000 г. На основе примерных программ дисциплин «Микропроцессорные системы» (спец. 220100 (230101) Вычислительные машины, комплексы, системы и сети), «Микропроцессорная техника в приборостроении» (спец. 551500 (200100) Приборостроение), «Микропроцессоры в системах управления».

Рецензент: доцент кафедры прикладной математики В.Г. Пугин

Пособие содержит сведения о принципах построения, особенностях функционирования современных микропроцессоров и микропроцессорных систем на их основе

От авторов

Исходные материалы для данной работы были заимствованы нами, в основном, из публикаций А. Андреева, А. Богданова, А. Власова, Д. Волкова, А. Галушкина, А. Геворкяна, А. Горбаня, Ю. Горбачева, В. Дьяконова, А. Дудкина, Е. Зудиловой, В. Корнеева, А. Кузнецова, М. Кузьминского, В. Мареева, Ю. Полякова, А. Пылкина, Д. Россиева, Е. Станковой, В. Степанова, А. Шадского, В. Шахнова, приведенных на различных сайтах в Internet. Мы не претендуем на новизну материалов в данной работе и не можем дать гарантии в полной достоверности полученных выводов ввиду очевидной неполноты исходного материала. Одновременно выражаем благодарность всем вышеперечисленным авторам за приведенные в Internet материалы и надеемся, что отсутствие ссылок по тексту на их работы не будет принято ими как нарушение нами их авторских прав.

Оглавление

Сокращения:

ПК – персональный компьютер
МП - микропроцессор
МПС – микропроцессорная система
УВВ – устройство ввода - вывода
АЛУ – арифметико-логическое устройство
ПО – программное обеспечение
ВУ – внешнее устройство
ВС – вычислительная система
БИС (СБИС) – большая (сверхбольшая) интегральная схема
СОЗУ – сверхбыстрое оперативное запоминающее устройство
ЦПЭ – центральный процессорный элемнт
ОЗУ – оперативное запоминающее устройство
ЗУ – запоминающее устройство
ЭВМ – электронная вычислительная машина
РОН – регитр общего назначения
ШД – шина данных
ЦП – центральный процессор
ША – шина адреса
ПЗУ – постоянное запоминающее устройство
АЛУ – арифметико – логическое устройство
ПДП – прямой доступ к памяти
ОС – операционная система
МПК БИС – микропроцессорный комплектк БИС
ЦПЭ – центральный процессорный элемент
КВВ – контроллер ввода - вывода
КПр – контроллер прерываний
ИНС – искусственная нейронная сеть
ССП – слово состояния процессора

Введение

Бурные темпы развития цифровых методов обработки информации влекут разработку и всеобщее внедрение в практику вычислений и управления производством микропроцессорных средств (персональных компьютеров и соответствующего периферийного оборудования) обработки информации. Их аппаратурная реализация, включающая микропроцессоры, контроллеры, системные платы, шины, накопители, системы вывода видео- и аудиоинформации и т.д., во многом обеспечивает заданные уровни вычислительной мощности и функциональных возможностей систем, использующих эти средства.

Разработкой аппаратурных средств занимаются множество фирм. Использование при разработке различных конструктивно-технологических принципов порождает большое разнообразие вариантов их построения для построения высокопроизводительных вычислительных систем (ВС), основу которых составляют микропроцессорные средства обработки информации (микропроцессорные системы – МПС).

При разработке МПС основным практически всегда стоит вопрос выбора оптимального состава аппаратурных средств и, прежде всего, микропроцессоров (МП), обеспечивающих получение максимально возможной эффективности работы системы. В реальных условиях поиск соответствующих материалов для решения данного вопроса почти всегда оставался сложным из-за недостаточности публикаций обобщающего характера. В данной работе сделана попытка провести обзор наиболее известных существующих микропроцессоров и общих принципов построения систем на их основе.

1. Области применения микропроцессоров

При построении различных микропроцессорных систем учету подлежат различные технические и производственно-технологические факторы, влияющие на эффективность использования систем в аппаратуре. Состав аппаратуры МПС должен обеспечивать:

- простое наращивание разрядности и производительности,
- возможность широкого распараллеливания вычислительного процесса,
- эффективную обработку алгоритмов решения различных задач,
- простоту технической и математической эксплуатации.

Сама МПС, будучи оснащенной разнообразными устройствами ввода - вывода (УВВ) информации, может применяться в качестве законченного изделия. Однако часто к МПС необходимо подавать сигналы от множества измерительных датчиков и исполнительных механизмов какого - либо сложного объекта управления или технологического процесса. В этом случае уже образуется сложная вычислительная система, центром которой является МП.

Простые в архитектурном исполнении микропроцессоры применяются для измерения временных интервалов, управления простейшими вычислительными операциями (в калькуляторах), работой кино-, фото-, радио- и телеаппаратуры. Они используются в системах охранной и звуковой сигнализации, приборах и устройствах бытового назначения. Бурно развивается производство электронных игр с использованием микропроцессоров. Они порождают не только интересные средства развлечения, но и дают возможность проверять и развивать приемы логических заключений, ловкость и скорость реакции.

Видеоигры можно отнести к приложениям, требующим использования компьютеров с ограниченным набором функций. Сегодня игровые приставки потребляют наибольшее количество, если не считать ПК, 32 - разрядных микропроцессоров. Наибольшее применение здесь получили МП Intel, Motorola. В устройстве PlayStation фирмы Sony используется 32 - разрядный процессор MIPS, а в видеоприставке Nintendo 64 — даже 64 - разрядный чип

того же производителя. Продукты компании Sega с видеоиграми Saturn и Genesis вывели RISC - процессоры серии SH фирмы Hitachi на третье место в мире по объему продаж среди 32 - разрядных систем.

Хорошие перспективы сулит 32 - разрядным процессорам рынок персональных электронных секретарей (PDA) и электронных органайзеров. Современные электронные органайзеры - яркий пример интегрированных приложений, ведь для них практически не существует независимых поставщиков программного обеспечения. С другой стороны, PDA типа Newton фирмы Apple, по сути, не что иное, как новая вычислительная платформа, будущее которой зависит от разработчиков программного обеспечения (ПО).

До настоящего времени успехом среди электронных органайзеров пользуются устройства с ограниченным набором функций. Тем не менее, дальнейшее совершенствование технологии может вывести эти «ручные» компьютеры в абсолютные лидеры, которые по объемам продаж в натуральном выражении должны обойти ПК.

Важной функцией МП является предварительная обработка информации с внешних устройств (ВУ), преобразования форматов данных, контроллеров электромеханических внешних устройств. В аппаратуре МП дает возможность производить контроль ошибок, кодирование - декодирование информации и управлять приемо-передающими устройствами. Их применение позволяет в несколько раз сократить необходимую ширину телевизионного и телефонного каналов, создать новое поколение оборудования связи.

Использование МП в контрольно-измерительных приборах и в качестве контрольных средств радиоэлектронных систем дает возможность проводить калибровку, испытание и поверку приборов, коррекцию и температурную компенсацию, контроль и управление измерительными комплексами, преобразование и обработку, индикацию и представление данных, диагностику и локализацию неисправностей.

С помощью микропроцессорных средств можно решать сложные технические задачи по разработке различных систем сбо-

ра и обработки информации, где общие функции сводятся к передаче множества сигналов в один центр для оценки и принятия решения. Например, в бортовых системах летательных аппаратов за время полета накапливается большое количество информации от различных источников, требующих зачастую незамедлительной ее обработки. Это осуществляется централизованно с помощью вычислительной системы на основе бортовой МПС.

Обобщая рассмотренные примеры использования МП, можно выделить четыре основных направления их применения:

- встроенные системы контроля и управления;
- локальные системы накопления и обработки информации;
- распределенные системы управления сложными объектами,
- распределенные высокопроизводительные системы параллельных вычислений.

Встроенные системы контроля и управления. Управляющие встроенные МПС предназначены для решения локальных задач управления объектами и могут выполнять функции контроллеров устройств, подключаемых к МПС более высоких контуров управления или быть центром управляющих систем нижних контуров управления.

Использование МПС даже в простейшей схеме управления принципиально изменяет качество функционирования обслуживаемых им устройств. Она позволяет оптимизировать режимы работы управляемых объектов или процессов и за счет этого получать прямой и/или косвенный технико-экономический эффект.

Прямой технико-экономический эффект выражается в экономии потребляемой энергии, повышении срока службы и снижении расхода материалов и оборудования. Косвенный технико-экономический эффект связан со снижением требований к обслуживающему персоналу и повышением производительности.

Опыт показывает, что практически во всех случаях использование МПС только за счет экономии электроэнергии обеспечивается ее окупаемость за 1 - 1.5 года. Управление оборудованием на основе встроенных систем контроля и управления создает реальные предпосылки создания полностью автоматизированных производств.

Использование МПС повышает качество работы и произво-

дительность оборудования, существенно снижает требования к персоналу, работающему на нем. Цифровое управление отдельными единицами оборудования на различных уровнях позволяет легко собирать информацию (или вызвать ее) с нижних на верхние уровни иерархической системы управления.

Локальные системы накопления и обработки информации. Уровень управления современным предприятием или учреждением требует наличия для любого специалиста или руководителя достаточно большого объема специфичной информации. Это может быть обеспечено за счет применения локальных микропроцессорных вычислительных систем.

Локальные, т. е. расположенные на рабочем месте, МПС накопления и обработки информации экономически и технически просто осуществляют информационное обеспечение потребителей. Объединение локальных систем между собой в сеть и дистанционное подключение этой сети к центральной ЭВМ с громадным информационным архивом позволяют создать завершенную автоматизированную систему информационного обеспечения.

Внешние устройства локальных МПС могут встраиваться в корпус ЭВМ. Их устройства образуют комплект, минимально необходимый для проведения вычислительных работ и обработки данных. В комплект сложных локальных МПС, ориентированных на решение инженерных и научных задач, могут входить разнообразные внешние устройства, например, печати, визуального отображения, внешней памяти, комплексирования, пульты операторов общего назначения и т. д.

Распределенные системы управления сложными объектами. Альтернативой широко распространенным системам с центральным процессором становятся распределенные микропроцессорные управляющие системы. В этом случае микропроцессоры и связанные с ними схемы обработки данных физически располагаются вблизи мест возникновения информации, образуя локальные МПС. Такое построение системы позволяет вести обработку информации на месте ее возникновения, например, вблизи двигателей, рулей управления, тормозной системы и т. д. В этом случае связь системы с центральной системой обработки и накопления данных и создает пространственно - распределенную систему управления.

В распределенных системах достигается значительный рост быстродействия получения и обработки входной информации, экономия в количестве и распределении линий связи, повышается живучесть, существенно развиваются возможности оптимизации режимов управления и функционирования.

Распределенные высокопроизводительные системы параллельных вычислений. МПС открыли новые возможности решения сложных вычислительных задач, алгоритмы вычисления которых допускают распараллеливание, т. е. одновременные (параллельные) вычисления на многих микропроцессорах.

Системы параллельных вычислений на основе десятков, сотен и даже тысяч одинаковых или специализированных на определенные задачи микропроцессоров при значительно меньших затратах дают такую же производительность, как и вычислительных системах на основе мощных процессоров конвейерного типа. Создание МПС с большим количеством специализированных по функциональному назначению процессоров позволяет проектировать мощные ВС нового типа по сравнению с традиционными развитыми большими вычислительными системами.

2. Виды архитектур микропроцессоров

Термин «архитектура» носит двойной смысл. В первом случае под архитектурой понимается архитектура набора команд, исполняемых микропроцессором. Во втором случае архитектура охватывает понятие организации системы, включающее структуру памяти, системной шины, организацию ввода/вывода и т.п. Применительно к вычислительным системам термин «архитектура» может быть определен как распределение функций, реализуемых системой, между ее уровнями.

Так, например, архитектура первого уровня определяет, какие функции по обработке данных выполняются МП в целом, а какие возлагаются на внешний мир (пользователей, операторов, администраторов баз данных и т.д.). МП взаимодействует с внешним миром через набор интерфейсов: языков (оператора, программирования, описания, манипулирования базой данных,

управления заданиями) и системных программ (служебных, редактирования, сортировки, сохранения и восстановления информации).

Архитектура второго уровня может разграничивать определенные уровни внутри программного обеспечения. Например, уровень управления логическими ресурсами может включать реализацию таких функций, как управление базой данных, файлами, виртуальной памятью, сетевой телеобработкой. К уровню управления физическими ресурсами относятся функции управления внешней и оперативной памятью, управления процессами, выполняющимися в системе.

Следующий, третий, уровень отражает основную линию разграничения системы, а именно границу между системным программным обеспечением и аппаратурой. Эту идею можно развить и дальше и говорить о распределении функций между отдельными частями физической системы. Например, некоторый интерфейс определяет, какие функции реализуют центральные процессоры, а какие - процессоры ввода/вывода.

Архитектура четвертого уровня определяет разграничение функций между процессорами ввода/вывода и контроллерами внешних устройств. В свою очередь можно разграничить функции, реализуемые контроллерами и самими устройствами ввода/вывода (терминалами, модемами, накопителями на магнитных дисках и лентах). Архитектура таких уровней часто называется архитектурой физического ввода/вывода.

При создании МП используются три наиболее широко применяемых вида архитектур, созданных за время их развития: регистровая, стековая и ориентированная на оперативную память.

Регистровая архитектура (архитектура типа «регистр - регистр») микропроцессора определяет наличие достаточно большого набора регистров внутри больших интегральных схем (БИС) микропроцессора. Этот набор регистров образует поле сверхбыстрой оперативной памяти (СОЗУ) с произвольной записью и выборкой информации.

В микропроцессорах с регистровой архитектурой рабочие области регистров размещаются в логических частях процессоров. Однако малая плотность логических схем по сравнению с плотно-

стью схем памяти ограничивает возможность регистровой архитектуры. МП с архитектурой, ориентированной на память, обеспечивают быстрое подключение к рабочим областям, когда необходимо заменять контексты. Смена контекстов осуществляется изменением векторов трех регистров - счетчика команд, регистров состояния и указателя рабочей области. Достоинство этой архитектуры в отношении смены контекстов связано с выполнением только одной команды для передачи полного вектора контекста.

Микропроцессоры с регистровой архитектурой имеют высокую эффективность решения научно - технических задач, поскольку высокая скорость работы СОЗУ позволяет эффективно использовать скоростные возможности арифметик - логического блока. Однако при переходе к решению задач управления эффективность таких микропроцессоров падает, так как при переключениях программ необходимо разгружать и загружать регистры СОЗУ.

Стековая архитектура микропроцессора дает возможность создать поле памяти с упорядоченной последовательностью записи и выборки информации. Эта архитектура эффективна для организации работы с подпрограммами, когда возникает постоянная необходимость перехода от текущей программы к подпрограмме, обслуживающей какое - либо ВУ, и возврат в текущую программу. Хранение адресов возврата позволяет организовать в стеке эффективную обработку последовательностей вложенных подпрограмм.

Основным недостатком МП этого типа является то, что стек, реализованный на кристалле микропроцессора, как правило имеет малую информационную емкость. При работе он быстро переполняется, приводя к возможности нарушения работы системы. Построение же стека большой емкости требует значительных ресурсов кристалла. Поэтому наилучшими характеристиками обладают МП, в которых стек реализуется вне микропроцессора - в оперативной памяти (оперативном запоминающем устройстве – ОЗУ).

Архитектура микропроцессора, ориентированная на оперативную память, обеспечивает высокую скорость работы и большую информационную емкость рабочих регистров и стека

при их организации в ОЗУ. В МП с такой архитектурой все обрабатываемые числа после операции в микропроцессоре выводятся из микропроцессора и вновь возвращаются в память, что и дало ей такое название.

При оценке быстродействия МП типа «память - память» необходимо учитывать физическую реализацию как элементов, так и связей между ними. Высокая скорость срабатывания логических элементов интегральных схем не всегда может обеспечить высокую скорость работы МП, поскольку большие значения индуктивно - емкостных параметров связей на печатных платах не позволяют передавать сигналы без искажения. Высокий уровень технологии современных МП до долей микрон существенно уменьшило размеры БИС, снизило паразитные параметры связей. Поэтому стало возможным физически отделить блок регистров и стек от арифметико-логического блока и обеспечить при этом их высокоскоростную совместную работу. При создании однокристальных МП регистровые СОЗУ и ОЗУ МПС имеют практически одни и те же параметры. Повышение скорости работы ОЗУ позволяет удалить набор регистров и стек из кристалла микропроцессора и использовать освободившиеся ресурсы для развития системы команд, средств прерывания, многоразрядной обработки. Организация рабочих регистров и стека в ОЗУ ведет к уменьшению скорости передачи информации, однако при этом повышается общая эффективность такого решения за счет большой информационной емкости полей регистровой и стековой памяти, а также возможности развития системы команд и прерываний.

Архитектура микропроцессора, ориентированная на оперативную память, обеспечивает экономию площади кристалла МП. В этом случае на кристалле размещается только регистр - указатель начального файла набора регистров. Адресация остальных регистров осуществляется указанием в команде специальным указателем - кодом смещения. Доступ к рабочим регистрам в этом случае замедляется, поскольку приходится совершать сопряженное с затратами времени кольцевое «путешествие» из процессора во внекристальную память, где размещаются рабочие регистры. Однако контекстное переключение в микропроцессоре с такой архитектурой происходит быстро, поскольку при прерывании необхо-

димо только изменить значение содержимого регистра - указателя рабочей области памяти.

Другая отличительная особенность архитектуры МП, ориентированной на оперативную память - двухадресный формат команд. В этих МП нет специального накапливающего регистра, выполняющего функции подразумеваемой ячейки результата для всех двухоперандных команд. Результат формируется в соответствии с алгоритмом, приведенном для примера на рис. 1,а, где операция сложения содержимого двух ячеек памяти с номерами X и Y осуществляется по команде «сложить XY».

Поскольку в архитектуре типа «память - память» любая ячейка памяти может содержать либо исходный операнд, либо операнд-результат, то эта операция выполняется по одной команде. В то же время в процессорах с одноадресной регистровой архитектурой для достижения той же самой цели приходится использовать две команды:

- команду пересылки операнда Y во внутренний регистр Рг,
- команду сложения содержимого внутреннего регистра Рг с содержимым ячейки памяти X и пересылки результата в ячейку X (рис. 1,б).

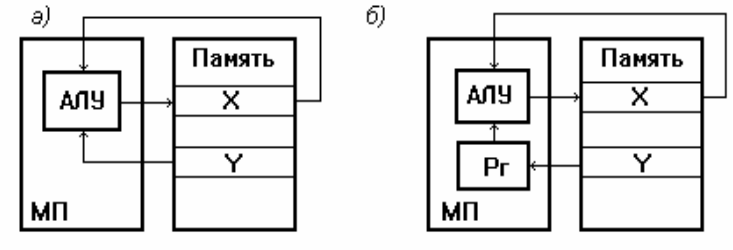


Рис. 1

В первом случае при компиляции программ для компиляторов высокоуровневых языков существенно упрощается задача присвоения значений переменным и, благодаря этому, получаются более короткие модули объектных программ.

Использование возможностей быстрой смены контекстов и фактически неограниченной рабочей области в МП с архитектурой, ориентированной на оперативную память, позволяет им легко

находить применение в МПС, работающим в реальном масштабе времени.

К достоинствам архитектуры МП, ориентированной на оперативную память, относится возможность развития системы, позволяющая снизить время разработки ПО. Здесь под развитием понимается способность систем внедрять в виде функциональных модулей программные, программно - аппаратурные и даже аппаратурные средства, которые можно использовать в системе по мере совершенствования аппаратурных средств и накопления опыта.

Распределенные системы управления часто требуют применения полуавтономных контроллеров, которые должны вписываться в определенные иерархические структуры. При этом архитектура МП, ориентированная на память, обеспечивает естественный и эффективный интерфейс между контроллерами, расположенными на одном иерархическом уровне, и процессорами управления, расположенными на более высоком иерархическом уровне, а структура связей между контроллерами может быть обеспечена за счет развитых информационных магистралей.

3. История развития микропроцессоров

3.1. Микропроцессоры первого поколения

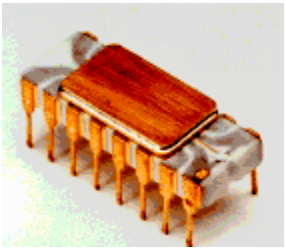
Микропроцессор обязан своему появлению внедрению в начале 70-х годов в производство микроэлектронной элементной базы, основу которой составляют интегральные схемы (ИС). Разработкой и производством МП в то время занимались ряд фирм США, среди которых наиболее совершенной технологией на фоне других обладала фирма Intel. Поэтому историю развития рассмотрим на примере развития МП данной фирмы.

До начала 70-х годов вычислительные машины были доступны весьма ограниченному кругу специалистов, а их применение, как правило, оставалось окутанным завесой секретности и мало известным широкой публике. Однако в 1971 г. произошло событие, которое резко изменило ситуацию и превратило вычислительную машину в повседневный рабочий инструмент десятков миллионов людей.

В 1971 году фирма Intel из небольшого американского городка Санта-Клара (шт. Калифорния) создала новый полупроводниковый прибор, получивший название «микропроцессор». В 1968 г. Гордон Мур и Боб Нойс, одни из тех, кто закладывал фундамент известной полупроводниковой компании Fairchild Semiconductor, основали фирму Intel Corporation. Первой идеей нового предприятия было создание полупроводниковых запоминающих устройств, призванных заменить ЗУ на магнитных сердечниках. Поскольку к концу 60-х годов память этого типа практически исчерпала весь свой потенциал развития, проблема была весьма актуальной, а ее разработка сулила немалые прибыли. И хотя в данной области Intel добилась заметных успехов, тем не менее мировую славу ей принесли совсем другие изделия.

Поворотным моментом в истории компании стал 1969 г., когда был получен заказ на создание ряда специализированных микросхем для калькуляторов от ныне уже несуществующей японской фирмы Busicom. В результате его реализации был разработан кристалл в сопровождении соответствующих средств поддержки.

МП i4004. 15 ноября 1971 г. Intel приступила к поставкам первого в мире микропроцессора Intel 4004 - именно такое обозначение получил первый прибор, послуживший отправной точкой абсолютно новому классу полупроводниковых устройств.



Кристалл представлял собой 4-разрядный процессор и изготавливался по передовой в те годы p-канальной МОП-технологии с проектными нормами 10 мкм. Электрическая схема прибора насчитывала 2300 транзисторов. Микропроцессор работал на тактовой частоте 750 кГц при длительности цикла команды 10,8 мкс.

МП i4004 имел адресный стек (счетчик команд и три регистра стека типа LIFO - Last In First Out), блок регистров общего назначения - РОН (регистры сверхоперативной памяти, или регистровый файл), 4-разрядное параллельное АЛУ, аккумулятор, регистр команд с дешифратором команд и схемой управления, а также схему связи с периферийными устройствами. Все эти функ-

циональные узлы объединялись между собой 4-разрядной шиной данных (ШД).

Память команд достигала 4Кбайт (для сравнения: объем ЗУ мини-ЭВМ в начале 70-х годов редко превышал 16 Кбайт), а регистровый файл центрального процессора (ЦП) насчитывал шестнадцать 4-разрядных регистров, которые можно было использовать и как восемь 8-разрядных (восемь 4-разрядных пар). Такая организация РОН сохранена и в последующих микропроцессорах фирмы Intel. Три регистра стека обеспечивали три уровня вложения подпрограмм.

МП i4004 монтировался в пластмассовый или металлокерамический корпус типа DIP (Dual In-line Package) всего с 16 выводами.

В систему его команд входило 46 инструкций. По своему функциональному составу она была универсальной, т. е. рассчитана на широкий круг решаемых задач и разрабатываемых приложений. Первоначальное назначение кристалла наложило определенный отпечаток на состав системы команд, поэтому присутствие в ней ряда инструкций, в частности десятичной коррекции, а также наличие соответствующих аппаратных средств не вызывает особого удивления.

Вместе с тем кристалл располагал весьма ограниченными средствами ввода/вывода, а в системе команд отсутствовали операции логической обработки данных (И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ), в связи с чем их приходилось реализовывать с помощью специальных подпрограмм, что в некоторых случаях чрезмерно усложняло создаваемое ПО.

МП i4004 не имел возможности останова (команды HALT) и обработки прерываний. Это объясняется тем, что в калькуляторах, где поначалу и планировалось использовать прибор, особой необходимости в этих средствах нет.

Цикл команды процессора состоял из восьми тактов задающего генератора. Как уже отмечалось, МП i4004 монтировался в корпус всего с 16 выводами - самый распространенный (а значит, и самый дешевый) тип корпуса в начале 70-х годов. А поскольку в распоряжении инженеров оказался узкий интерфейс с "внешним миром", то пришлось пойти на применение мультиплексированной

шины адреса (ША) и данных, причем 12 - разрядный адрес выдавать порциями по четыре разряда, что, конечно, не могло не сказаться на длительности машинного цикла. Прием команды по такому интерфейсу требовал еще двух тактов. На исполнение же самой инструкции из восьми тактов процессор затрачивал лишь три.

Компанией было разработано и выпущено целое семейство БИС, в которое вошли постоянное запоминающее устройство (ПЗУ) 4001, ОЗУ 4002, регистр сдвига 4003 и ряд других вспомогательных микросхем. Поскольку все они были рассчитаны на совместное использование, разработка аппаратных средств системы заметно упрощалась.

Опыт использования первого МП показал, что такие факторы, как отсутствие средств обработки прерываний, наличие трех уровней вложения подпрограмм и необходимость реализации логических операций И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ с помощью специальных подпрограмм, далеко не всегда удовлетворяют разработчиков. Указанные недостатки привели к созданию нового МП i4040.

МП i4040. В i4040 сохранены все функциональные возможности предшествующей модели и существенно улучшены как технические, так и программные средства. Система команд пополнилась 14 инструкциями, включая выполнение логических операций И и ИЛИ; кроме того, в процессор были введены средства останова и обработки прерываний.

Претерпела некоторые изменения и архитектура устройства. Адресный стек процессора увеличен с трех до семи регистров, а количество РОН возросло с 16 до 24, причем их разбили на две области, выбираемые при помощи специальных команд. Отчасти такая организация обусловлена тем, что процессор теперь мог обращаться к двум блокам памяти команд объемом 4 Кбайт и за каждым из них программист мог закрепить свою область регистров. Наряду с этим восемь РОН были всегда доступны для использования. В итоге получилась достаточно гибкая и удобная структура, позволявшая разрабатывать самостоятельные программные модули, способные взаимодействовать через общую часть регистрового файла.

Обработка одноуровневых прерываний является одним из наиболее существенных новшеств МП i4040. Эта функция превратила его в полноценный процессор и сделала возможным использование в системах реального масштаба времени. Благодаря применению сигнала «останов» стала реальностью синхронизация работы процессора с некоторыми внешними событиями.

Несмотря на то что тактовая частота и машинный цикл i4040 не претерпели изменений, производительность МП возросла за счет использования более совершенной архитектуры и эффективной системы команд. 60 инструкций, ориентированных на широкий спектр решаемых задач, обработка прерываний, до 8 Кбайт памяти команд, а также возможность быстрого перевода систем на базе i4004 на новый процессор вывели i4040 в безусловные лидеры рынка 4-разрядных устройств.

МП i8008. Intel с 1 апреля 1972 г. начала поставки первого в отрасли 8-разрядного прибора Intel 8008. Он был разработан для нужд американской фирмы Computer Terminals Corporation of Texas, позднее известной как Datapoint.

Проектирование i8008 шло практически параллельно с работами над i4004. Кристалл изготавливался по p-канальной МОП-технологии с проектными нормами 10 мкм и содержал 3500 транзисторов. Процессор работал на частоте 500кГц при длительности машинного цикла 20 мкс (10 периодов задающего генератора).

В отличие от своих предшественников новый МП допускал применение комбинации ПЗУ и ОЗУ. Помимо увеличения разрядности и перехода на использование общего поля памяти для команд и данных, структура процессора претерпела еще ряд существенных изменений. Прежде всего это коснулось регистрового файла и устройства управления. По сравнению с i4004 число РОН уменьшилось вдвое (с 16 до 8), причем два регистра в основном использовались для хранения адреса при косвенной адресации памяти.

В связи с этим следовало бы ожидать снижения производительности, которого на самом деле не произошло, поскольку операции с памятью i8008 выполнял быстрее предыдущих моделей благодаря меньшему количеству состояний в машинном цикле и отсутствию необходимости исполнения минимум трех подготови-

тельных команд (как в i4004 и i4040) при обращении к ОЗУ или ПЗУ.

Вместе с тем объем блока регистров был ограничен возможностями технологии, которая в то время еще не позволяла размещать на кристалле большие регистровые структуры (в МП i8008 блок РОН был реализован в виде динамической памяти).

Почти вдвое (с восьми до пяти состояний) сократилась длительность машинного цикла. Теперь процессор выполнял команды за один - три машинных цикла, а некоторые инструкции - за один цикл из трех состояний. Для синхронизации работы МП с медленными устройствами был введен сигнал готовности (READY).

Разработчики технических средств на базе i8008 не были ограничены жесткими требованиями в отношении быстродействия микросхем памяти и периферийных устройств и могли использовать те ИС, которые наиболее полно соответствовали конкретной системе. В ряде случаев это приводило к ощутимому сокращению стоимости оборудования.

Система команд первого 8-разрядного МП насчитывала 65 инструкций, причем значительно увеличилось число команд условных переходов, а также логических инструкций и команд сдвига. Новый кристалл мог адресовать память объемом до 16 Кбайт (объем ЗУ для МП типа i4040 не превышал 8 Кбайт). Его производительность по сравнению с 4-разрядными системами возросла в 2,3 раза.

Процессор с такими параметрами уже можно было применять для построения контрольно - испытательного оборудования, прецизионной измерительной техники и сложных промышленных контроллеров систем управления технологическими процессами.

Однако i8008 имел свои недостатки. Объем и организация стека остались такими же, как и у i4040, и реализация операций с ним по - прежнему возлагалась на программиста. Узкий интерфейс с "внешним миром" ограничил количество управляющих сигналов процессора: в результате специалистам Intel пришлось использовать их шифрацию, что повлекло за собой необходимость установки дополнительного внешнего оборудования для формирования сигналов управления. В среднем для сопряжения процессора с па-

мятью и устройствами ввода/вывода требовалось около 20 схем средней степени интеграции.

Вскоре после выхода i8008 появилась его усовершенствованная версия i8008-1. Модернизированный вариант работал уже на частоте 800 кГц при длительности машинного цикла 12,5 мкс. Увеличение в 1,5 раза производительности центрального процессора наряду с большим (по тому времени) объемом оперативной памяти послужило лучшей рекомендацией для активного использования кристалла в различных областях, начиная от промышленности и медицины и кончая военной электроникой и торговлей. По мере расширения сферы влияния МП и усложнения систем на его базе возросли и требования к нему со стороны проектировщиков оборудования.

Несмотря на значительный успех разработанного кристалла среди проектировщиков систем, ПО к этому времени уже с трудом вписывалось в 16 Кбайт, да и производительность прибора начинала не удовлетворять многих разработчиков. Кроме того, некоторые области применения настойчиво требовали расширения не только количества, но и номенклатуры периферийных устройств. Системщики уже с трудом могли обходиться без такой традиционной для мэйнфреймов и мини-ЭВМ периферии, как дисплеи, принтеры, накопители на магнитной ленте и дисках и т. п. Стало очевидно, что технические характеристики изделия превратились в фактор, сдерживающий его дальнейшее распространение.

Возможности p-канальной МОП-технологии для создания сложных высокопроизводительных МП были уже практически исчерпаны, поэтому направление главного удара перенесли на технологию n-МОП. Перед проектировщиками стояли не менее сложные проблемы - разработка эффективной системы команд, рассчитанной на широкий круг решаемых задач, при сохранении программной совместимости с предыдущей моделью, расширение объема адресуемой памяти, поддержка интенсивного ввода/вывода без существенной потери производительности процессора, совершенствование подсистемы обработки прерываний. Указанные причины привели к созданию нового прибора – i8080.

МП i8080 - триумф 8-разрядных систем, который появился 1 апреля 1974 г. Благодаря использованию технологии n-МОП с

проектными нормами 6 мкм, на кристалле удалось разместить 6 тыс. транзисторов. При этом геометрические размеры самого кристалла по сравнению с i8008 увеличились незначительно. Следовательно, процент выхода годных изделий и ряд экономических показателей производства, включая себестоимость, удалось сохранить на достаточно высоком уровне. Тактовая частота процессора была доведена до 2 МГц, что в 2,5 раза превышало аналогичный параметр для i8008, а длительность цикла команды составила уже 2 мкс.

Несмотря на чисто внешнее сходство структур i8080 и i8008, схема нового процессора существенно отличалась от предшествующей модели. Объем памяти, адресуемой процессором, был увеличен в четыре раза и достиг 64 Кбайт (кстати, в то время ОЗУ такой емкости предлагали потребителям минимальные конфигурации многих мини-ЭВМ). В сочетании с эффективным механизмом обработки прерываний это давало им возможность широкого применения нового МП в сложных системах сбора и обработки информации различного назначения, особенно функционирующих в реальном масштабе времени. За счет использования корпуса с 40 выводами удалось разделить адресную и информационную шины процессора, в результате отпала необходимость применения дополнительных внешних схем для разделения потоков адресов и данных. Общее же количество микросхем, требовавшихся для построения системы в минимальной конфигурации, сократилось с 20 до 6, т. е. более чем в три раза. В регистровый файл были введены указатель стека, активно используемый при обработке прерываний, а также два программно-недоступных регистра для внутренних пересылок.

Поскольку предыдущий МП i8008 имел большой успех и для был наработан достаточно большой объем ПО, то сохранение разработчиками программной совместимости i8080 и i8008 было вполне естественным и разумным шагом. Именно поэтому в состав РОН нового процессора были включены основные рабочие регистры предыдущей модели. Правда, полной совместимости с i8008 достичь не удалось, так как процедуры обращения к подпрограммам и инструкции ввода/вывода МП i8080 в значительной степени отличались от соответствующих процедур и операций

кристалла i8008, и при переводе систем со старого процессора на новый в некоторых случаях программы приходилось полностью перерабатывать.

Включение в систему команд ряда инструкций, адресующих память с использованием трех пар регистров (в i8008 для этого выделялась одна пара), придало дополнительную гибкость. Реализация же блока РОН на основе статической, а не динамической памяти дала дополнительную экономию площади кристалла для размещения других схем процессора. Исключение аккумулятора из регистрового файла и введение его в состав арифметико - логического устройства упростило схему управления внутренней шиной, поскольку при этом отпала необходимость в ее использовании для передачи данных между сверхоперативной памятью и арифметико-логическим устройством (АЛУ) во время выполнения арифметических и логических операций.

Новым веянием в архитектуре МП стало использование многоуровневой системы прерываний по вектору. Такое техническое решение позволило довести общее число источников прерываний в системе до 256. Правда, до появления специализированных БИС контроллеров прерываний схема формирования векторов прерываний требовала применения до десяти дополнительных чипов средней степени интеграции.

Освобождение центрального процессора от управления ВУ и обмен данными между памятью системы и периферией, минуя ЦП, были уже достаточно давно и успешно реализованы в универсальных ЭВМ (IBM System 360 и др.). Таким образом, появление в кристалле i8080 механизма прямого доступа к памяти (ПДП) при работе с ВУ можно смело считать первым (но далеко не последним) ударом микропроцессоров по большим системам. ПДП открыл зеленую улицу для применения в микроЭВМ таких сложных устройств, как накопители на магнитных дисках и лентах, а также дисплеи на ЭЛТ, которые и превратили микроЭВМ в полноценную вычислительную систему.

Начиная с первого кристалла, Intel стала выпускать не отдельные чипы, а семейства БИС, рассчитанные на совместное использование. Помимо МП, в новый набор микросхем вошли ИС системных генератора и контроллера. Вскоре их пополнили БИС

контроллера ПДП и контроллера прерываний. Благодаря хорошо продуманному составу комплекта, проектирование МПС на его базе в ряде случаев упростилось.

Следует отметить, что в эти годы разработчики систем все большее внимание стали уделять развитию мультипроцессорным универсальным МПС, которые в ту пору еще не стали привычным атрибутом вычислительных центров. Поэтому в начале 1976 г. стартовали работы по созданию 16-разрядного прибора, который впоследствии получил обозначение i8086.

МП i8086. Конечной целью нового проекта было получение 16-разрядного микропроцессора с производительностью, на порядок превышающей аналогичный параметр кристалла i8080 и позволяющего создавать многопроцессорные системы. Поставленная задача решалась за счет дальнейшего совершенствования архитектурных концепций, положенных в основу его предшественника. Был разработан сложный и исключительно удачный процессор в очень сжатые сроки.

Новый кристалл был анонсирован 8 июня 1978 г. Прибор изготавливался по высококачественной трехмикронной МОП-технологии с кремниевыми затворами (H-MOS), позволившей разместить на кристалле 29 тыс. транзисторов. Высокое быстродействие элементов (задержка 2 нс/вентиль) обеспечило тактовую частоту процессора 5 МГц, а 16-разрядная архитектура и 200-нс машинный цикл - производительность процессора, превышающую аналогичный параметр i8080 на порядок величины.

Программная совместимость с i8080 была, пожалуй, единственной, но вместе с тем и исключительно важной характеристикой, которая объединяла 86-й кристалл с его предшественниками. Структура процессора оказалась полностью пересмотренной. Прежде всего, прибор был разбит на два функциональных блока - операционный и интерфейсный, которые могли работать одновременно. В результате исполнение одной команды совмещалось во времени с выборкой следующей инструкции или данных из памяти. Более того, в МП появился регистровый файл команд, что давало дополнительную экономию времени при обращениях к памяти. Алгоритм работы операционного и интерфейсного блоков позволял вести обработку команд, находящихся в конвейере команд

регистрового файла, одновременно с их вводом из программной памяти.

Возможность адресации 1 Мбайт ОЗУ и сегментация памяти могут быть отнесены к одним из наиболее существенных новшеств, предложенных инженерами Intel. В частности, сегментация памяти и большое число уровней прерываний были ориентированы на работу систем в многозадачном режиме, весьма актуальном для приложений управления.

Большая емкость ОЗУ позволяла перевести проекты построения сложных операционных и прикладных систем из области теории в сферу практической реализации. Наряду с поддержкой ввода/вывода по каналу прямого доступа к памяти i8086 обеспечивал адресацию до 64К портов программно-управляемого ввода/вывода. Это снимало практически любые ограничения при формировании крупных систем сбора и обработки информации.

Микропроцессор имел два режима работы - минимальный и максимальный. Первый рассчитан на его использование в однопроцессорных системах и предполагал работу кристалла без БИС контроллера шины. Максимальный режим был ориентирован на применение МП в многопроцессорных системах и требовал наличия указанного контроллера. Таким образом, один и тот же процессор с одинаковым успехом мог применяться в системах различного класса.

Система команд процессора содержала 147 инструкций. Она позволяла решать задачи управления практически любой сложности. Появление среди них таких операций, как умножение и деление 16-разрядных чисел со знаком и без знака, команд обработки массивов данных, а также программно-управляемых прерываний дает все основания назвать этот кристалл универсальным, рассчитанным на использование не только в сложных контроллерах, но и в качестве центрального процессора ЭВМ общего назначения.

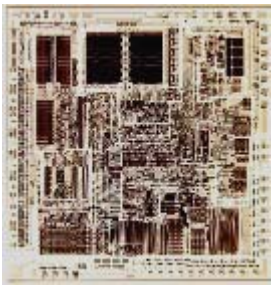
МП вышел в мощном сопровождении средств поддержки: вспомогательных БИС, средств разработки и отладки аппаратуры и системного ПО.

Использование микросхем i8086 в персональных компьютерах IBM предопределило дальнейшее развитие корпорации Intel

как разработчика и изготовителя универсальных процессоров общего назначения.

Вычислительная мощь 16-разрядных приборов была поддержана арифметическим сопроцессором i8087, который позволил превратить МПС в достаточно мощный инструмент и для решения задач вычислительного характера. Более того, теперь и разработчики систем управления на базе 86-го МП получили возможность использовать интенсивную арифметическую обработку информации, для которой ранее служили мини-ЭВМ.

3.2. Микропроцессоры второго поколения



МП i80286 второго поколения был разработан к 1 февраля 1982 г. Оставшись 16-разрядным прибором, по производительности новый ЦП в 3 - 6 раз превзошел своего предшественника при тактовой частоте первой модификации 8 МГц. Благодаря использованию многовыводного корпуса разработчики смогли применить схему с раздельными шинами адресов и данных.

24 разряда адреса позволили обращаться к физической памяти объемом до 16 МБайт. Встроенная система управления памятью и средства ее защиты открывали широкие возможности использования МП в многозадачных средах. Кроме того, аппаратура i80286 обеспечивала работу с виртуальной памятью объемом до 1 Гбайт. Для поддержки устройства управления памятью система команд пополнилась еще 16 инструкциями.

МП имел два режима работы - реальный и защищенный. В первом случае он воспринимался как быстрый МП i8086 с несколько расширенной системой команд и прекрасно подходил тем потребителям, для которых, помимо скоростных характеристик, жизненно важным было сохранение существующего задела ПО.

Работа в защищенном режиме позволяла использовать преимущества МП в полном объеме, и, прежде всего, большой объем основной памяти, дающий возможность работать ему в многозадачном варианте. Ведь основная проблема многозадачности была

в том, что предыдущие модели МП исполняемые программы могли быть записаны по любому адресу памяти, даже в занятые ячейки памяти ранее исполнявшимися программами. Операционная система и другие приложения при этом были не защищены: в любой момент исполняемая программа могла затереть эти места в памяти и система не смогла бы в дальнейшем вести достоверные расчеты.

Со стороны разработчиков программных продуктов были попытки создать операционную систему, которая сама бы контролировала все действия программ. Но для этого пришлось отказаться от компиляции приложений в готовые машинные коды - они стали интерпретируемыми, а производительность упала раз в двадцать. Стало ясно, что без аппаратной акселерации контроля, т. е. без защищенного режима процессора не обойтись.

Суть работы защищенного режима состоит в следующем. Все свои команды процессор выполнял точно так же как и в реальном режиме, но программистам пришлось использовать понятие «логического адреса». Логический адрес состоял из 32 бит: селектора (16 бит) и смещения (16 бит) (рис. 2). При этом в сегментных регистрах теперь хранился не сегмент, а селектор. Селектор - это индекс в таблице дескрипторов.

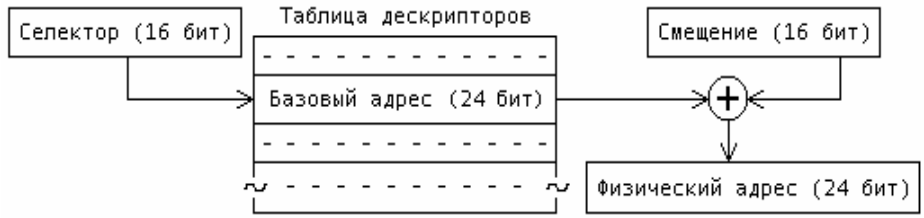


Рис. 2

Запись в таблице дескрипторов содержала всю необходимую информацию о некотором блоке памяти: его базовый адрес, размер всего блока, его тип (код или данные) и сведения о приоритете программы - владельца. Таким образом, каждый дескриптор полностью описывал один сегмент программы. Размер этого дескриптора был одинаков как для 286-х, так и для 386-х машин - 64 бита или 8 байт, но у 286-го старшие 16 бит не использовались.

Существовала одна глобальная и несколько локальных таблиц. Глобальная присутствовала всегда и хранила информацию о сегментах операционной системы. Локальные таблицы были для всех остальных программ. Управление памятью в защищенном режиме всегда было связано с конкретной операционной системой и ее версией. В операционной системе (ОС) OS/2 2.0 каждой программе были доступны глобальная и локальная (своя) таблицы дескрипторов. Всем приложениям в ОС Windows 3.0 давалась одна общая локальная таблица.

Всеми преимуществами МП решила воспользоваться IBM, применив процессор в новой модели ПК типа AT.

К сожалению, защищенный режим 286-го обладал и недостатками: несмотря на возможность адресовать 16 Мб памяти, максимальный размер сегмента остался по-прежнему равным 64 Кб, затрудняя программистам работу с большими массивами данных.

Режим работы с виртуальной памятью имел недостаток. Он заключался в том, что отсутствовал «прозрачный» для приложений способ перемещения данных операционной системой из памяти на жесткий диск - для реализации этого программам приходилось прибегать к разным ухищрениям вроде «запирания» и «отпирания» указателей на блок памяти.

В защищенном режиме отсутствовала совместимость с программами, написанными для реального режима MS-DOS. Переход из реального режима в защищенный был односторонним, для обратного перехода требовалась перезагрузка системы.

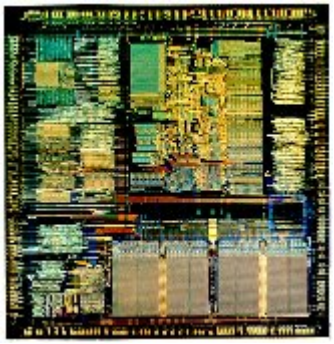
Указаные недостатки и высокая потребность в высокопроизводительных МП стимулировали усилия специалистов Intel по разработке прибора следующего поколения. Увеличение тактовой частоты 286-го процессора сверх достигнутого предела в 16 МГц давалось уже слишком дорого, а кроме того, никак не устраняло узкого места системы, которым оставалась оперативная память. Помимо прочего, 286-й решил далеко не все проблемы, характерные для многозадачных сред.

У инженеров Intel было два пути кардинального повышения производительности процессора: 32-разрядная архитектура прибора и совершенствование тракта процессор - память. При этом эффективное функционирование МП под управлением многоза-

дачных ОС требовало усовершенствования устройства управления памятью.

3.3. Микропроцессоры третьего поколения

МП i80386 третьего поколения был представлен 17 октября 1985 г. Использование КМОП-технологии с проектными нормами 1 мкм и двумя уровнями металлизации позволило разместить на кристалле 275 тыс. транзисторов и реализовать полностью 32-разрядную архитектуру МП.



32 разряда адреса обеспечили адресацию физической памяти объемом до 4 Гбайт и виртуальной памяти емкостью до 64 Тбайт. Встроенная в МП система управления памятью и защиты включала регистры преобразования адреса, механизмы защиты оперативной памяти и улучшенные аппаратные средства поддержки многозадачных ОС.

Помимо работы с виртуальной памятью допускались операции с памятью, имевшей страничную организацию. Предварительная выборка команд, буфер на 16 инструкций, конвейер команд и аппаратная реализация функций преобразования адреса значительно уменьшили среднее время выполнения команды.

Благодаря этим архитектурным особенностям, процессор мог выполнять 3 - 4 млн. команд в секунду, что примерно в 6 - 8 раз превышало аналогичный показатель для МП i8086. Безусловно, новый прибор остался совместимым со своими предшественниками на уровне объектных кодов.

Одной из наиболее любопытных особенностей рассматриваемой разработки компании было использование высокоскоростной кэш-памяти, позволившей существенно повысить производительность систем на базе 386-го процессора (еще один атрибут универсальных машин, который стал применяться в микропроцессорных системах). Для управления работой этой памятью была разработана БИС высокопроизводительного контроллера кэш-

памяти типа i82385, с помощью которой формировался двухвходовой множественный ассоциативный кэш. Указанная БИС обеспечивала управление памятью емкостью до 32 Кбайт и высокий коэффициент удачных обращений.

Для реализации работы с числами с плавающей точкой был разработан математический сопроцессор, который выпускался в виде отдельного кристалла i80387, дополняя вычислительную мощь МП.

Особый интерес представляли три режима работы кристалла - реальный, защищенный и режим виртуального МП i8086. В первом обеспечивалась совместимость на уровне объектных кодов с устройствами i8086 и i80286, работающими в реальном режиме. При этом архитектура i80386 была почти идентична архитектуре 86-го процессора, для программиста же он вообще представлялся как МП i8086, выполняющий соответствующие программы с большей скоростью и обладающий расширенными системой команд и регистрами.

Одно из основных ограничений реального режима на практике было связано с предельным объемом адресуемой памяти, равным 1 Мбайт. От него свободен защищенный режим, позволяющий воспользоваться всеми преимуществами архитектуры нового МП. Размер адресного пространства в этом случае увеличивался до 4 Гбайт, а объем поддерживаемых программ - до 64 Тбайт.

Производителям ПО это позволяло задействовать достаточно гибкие методы разработки и создавать более крупные программные пакеты. Для конечных пользователей выполнение приложений, рассчитанных на работу в реальном и защищенном режимах, происходило без каких-либо функциональных отличий, поскольку управление обоими режимами базировалось на средствах ОС и специальном прикладном ПО. Однако системы защищенного режима обладали более высоким быстродействием и возможностями организации истинной многозадачности.

Наконец, режим виртуального МП открывал возможность одновременного исполнения ОС и прикладных программ, написанных для МП i8086, i80286 и i80386. Поскольку объем памяти, адресуемой 386-м процессором, не ограничен значением 1 Мбайт,

он позволял формировать несколько виртуальных сред i8086. Немаловажно, что эти среды могли порождаться в одно и то же время, а механизм защищенного режима обеспечивал ОС и ее прикладным задачам использование различных областей памяти. Благодаря таким возможностям аппаратуры, можно было вместо нескольких МП типа i8086 использовать один процессор i80386, сохранив львиную долю имевшегося ПО.

Примерно в этот же период IBM и Microsoft приступили к разработке новой многозадачной ОС с графическим интерфейсом пользователя.

Стремление удовлетворить запросы потребителей всех категорий привело Intel к созданию клона 386-го МП с 16- разрядной внешней шиной данных (при сохранении внутренней 32-разрядной архитектуры). Существующий прибор получил обозначение i80386SX и был анонсирован 16 июня 1988 г., а уже менее чем через полгода пользователям были предложены первые ПК на его основе. Поскольку эти модели стоили дешевле компьютеров с МП 80386DX, многие потребители вполне справедливо рассматривали их как начальную ступень в применении вычислительной техники.

В конце 80-х годов степень интеграции микросхем приближалась к 1 млн. транзисторов на кристалле и 10 апреля 1989 г. Intel объявила о начале выпуска 32-разрядного прибора четвертого поколения - i80486, ставшего после устройств i8080 и i8086 еще одним долгожителем.

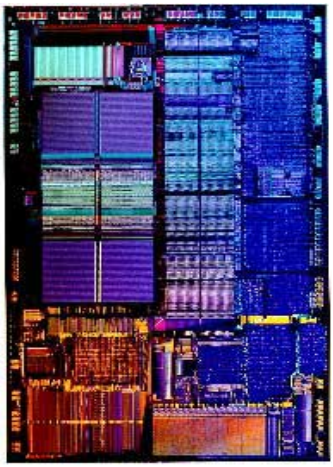
3.4. Микропроцессоры четвертого поколения

МП i80486. Архитектура нового МП отчасти напоминала строение своего предшественника, но вместе с тем имела и ряд коренных отличий. 1,2 млн. транзисторов позволили разработчикам реализовать на кристалле быстродействующую кэш-память (L1) и математический сопроцессор. Такое техническое решение свело к возможному минимуму число чипов на плате и самым благоприятным образом сказалось на стоимости готовых систем.

В 486-м процессоре кэш-память имела объем 8 Кб и была предназначена для одновременного хранения данных и инструкций. Кэш-память имела 4-канальную наборно-ассоциативную ар-

хитектуру и работала на уровне физических адресов памяти. Она содержала 128 наборов по 4 строки размером по 16 байт. Кэш-память умела работать только со строками, и если процессор требовал какой-нибудь байт, отсутствующий в кэше, то кэш-контроллер загружал из ОЗУ всю 16-байтную строку, содержащую необходимый байт.

Выбор строки для замещения производился по алгоритму «псевдо-LRU», для этого каждому набору строк отводилось по 3 бита статистики использования. Алгоритм LRU (Least Recently Used) основан на поиске элемента, к которому дольше всего не было обращений. При каждом обращении к строке кэш-контроллер увеличивал на 1 соответствующий счетчик LRU. Приставка «псевдо» означает лишь несовершенство механизма работы, ведь под счетчик отводилось всего 3 бита, что дает всего 8 состояний счетчика (2^3). После 8-го обращения к строке счетчик обнулится и соответствующая строка из самой «необходимой» станет самой «не необходимой» и будет прямым кандидатом на замещение.



Кэш-память первых 486-х работала в режиме Write Through (сквозная запись). В этом случае при записи данных тратилось дополнительное время на их запись во внешнюю память (даже если они присутствовали в кэше). Эта давало возможность ускорить чтение данных, но скорость записи, при этом, не ускоряется.

В следующих модификациях 486-х процессоров (некоторые 486DX2 и все 486DX4) был реализован принцип Write Back. В этом варианте запись данных, если их старая копия уже присутствовала в кэш-памяти, производилась только в кэш-память, а запись в ОЗУ откладывалась.

Процессор i486 мог использовать и внешнюю кэш-память (L2), расположенную вне кристалла микросхемы процессора. В 486-м, как видно, появилось 2-х уровневой кэш. Очевидно, что даже если оба кэш работают на одной частоте, кэш-память L1

функционирует быстрее второго. Это связано с тем, что при чтении данных из кэш-памяти L2 процессор все равно вынужден делать несколько пустых тактов, хотя и меньше, чем при чтении из ОЗУ.

Объем L2 составлял от 256 до 512 Кб. В системных платах 386-х моделей L2 обычно не превышал 128 Кб (типичный объем - 64 Кб). В марте 1994-го Intel, выпустив 486DX4, увеличила объем L1 до 16 Кб, при этом он по-прежнему оставался общим для данных и для команд.

МП функционировал в трех режимах и был ориентирован на многозадачные среды. За счет интеграции математического сопроцессора в БИС, а также модернизации его архитектуры производительность на задачах вычислительного характера возросла в 3 - 4 раза. Общая же производительность 486-го превышала аналогичный параметр своего предшественника в 4 - 5 раз.

Ровно через два года после выпуска i80486 появилась упрощенная версия кристалла (без сопроцессора), получившая обозначение i80486SX. Дальнейшее совершенствование пошло по пути увеличения тактовой частоты: были представлены версии на 50, 66, 75 и 100 МГц.

3.5. Микропроцессоры пятого поколения

Pentium P5. Выпуск высокопроизводительных МП Pentium P5 началось 22-го марта 1993 года. Это был первый процессор с двухконвейерной структурой. Он имел тактовые частоты 60 и 66 МГц. Частота шины совпадала с тактовой частотой процессора.

Процессоры содержали более 3.1 млн. транзисторов и выпускались по технологии 0.80 мкм, а позже – 0.60 мкм. Размер L1 составлял 16 Кб - 8 Кб на данные и 8 Кб на инструкции. L2 размещался на материнской плате и мог иметь объем до 1 Мб. Процессор выпускался для разъема Socket 4.

Как показали результаты эксплуатации МП, простое наращивание тактовой частоты неоднозначно влияло на увеличение его

производительности. В первую очередь это связано с большим разбросом во времени выполнения различных программ (приложений), связанных с разницей в производительности их различных компонентов. Так, например, разброс высоких и низких значений производительности относительно среднего значения компьютеров, построенных на процессорах 80486 и Pentium P5, составляет ориентировочно от 10 % до 20 % (рис. 3).

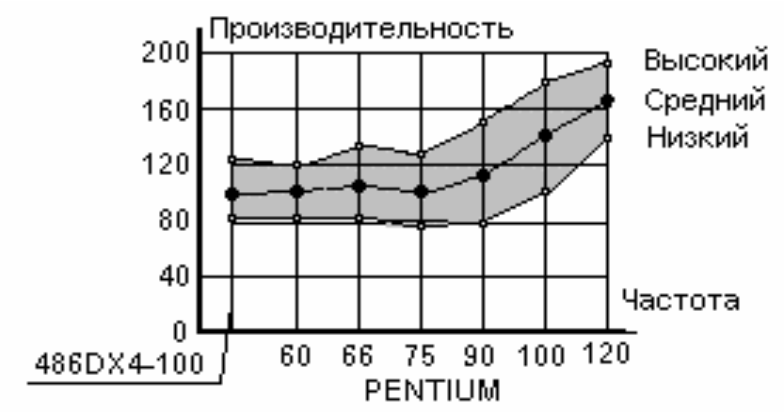


Рис. 3

Оценка производительности здесь проводилась по индексу iCOMP (Intel Comparative Microprocessor Perfomance), который учитывает четыре главных аспекта производительности процессора при 32- и 16 - разрядных операциях: с целыми числами, числами с плавающей точкой, графикой и видео.

Поэтому разработчики Intel основное внимание уделяли не только повышению тактовой частоты МП, но и совершенствованию его архитектуры. Эта тенденция сохранилась у разработчиков МП и в настоящее время.

В марте 1994 года Intel выпустила МП **Pentium P54**. Процессор имел частоты от 75 до 200 МГц. Частота шины 50-66 МГц. Размер L1 остался прежним – 16 Кб (8 Кб на данные и 8 Кб на инструкции). L2 был расположен вне кристалла и мог иметь объем до 1 Мб. При производстве этого процессора Intel применяет более совершенный техпроцесс 0.50 мкм. Процессор содержал более 3.3

млн. транзисторов. Выпускался для разъема Socket 5, позднее Socket 7.

Pentium MMX (**P55**) был выпущен 8 января 1997 года. Он пришел на смену МП P54 в связи с появлением все большего числа мультимедийных приложений. В нем был реализован новый набор из 57 команд MMX (Multi Media eXtention), существенно увеличивающий производительность компьютера при работе с этими приложениями (от 10 до 60 %, в зависимости от оптимизации).

МП выпускался с тактовыми частотами 166, 200 и 233 МГц. Тактовая частота шины составляла 66 МГц. По сравнению с Pentium P54 в нем был вдвое увеличен размер L1, который составил 32 Кб. Как и в предыдущих версиях L1 был разбит на два блока по 16 Кб для хранения данных и для инструкций.

L2 находился на материнской плате и мог иметь объем до 1 Мб. Процессоры выпускались по 0.35 мкм технологии и состояли из 4.5 млн. транзисторов. Он рассчитан на использование с разъемом Socket 7.

3.6. Микропроцессоры шестого поколения

Pentium PRO (P6) был выпущен 1 ноября 1995 года. От предыдущего поколения их отличало применение технологии динамического исполнения инструкций - изменения порядка их исполнения, и архитектура двойной независимой шины. Добавилась еще одна шина, которая соединила процессор с L2, встроенным в ядро. В результате этого впервые был применен L2, работающий на частоте процессора. Первоначальный размер L2 имел 256 Кб, позже достиг 1024 Кб. Максимальный размер – 2048 Кб. L1 остался прежним: 8 Кб + 8 Кб.

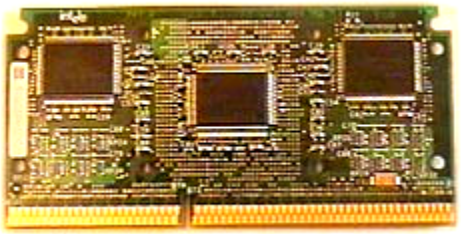
МП имел тактовые частоты 150, 166, 180, 200 МГц.

Процессоры Pentium PRO выпускались в корпусах SPGA (Staggered Pin Grid Array) с матрицей штырьковых выводов. В одном корпусе было установлено два кристалла – ядро процессора и L2 собственного изготовления. Устанавливался в Socket 8 с возможностью объединить до 4-х процессоров для симметричной мультипроцессорной обработки. Шина работала на частоте 60-66 МГц.

При 32-битных вычислениях и многозадачности значительно превосходил по производительности предыдущие версии Pentium, но в 16-битных приложениях проигрывал ему.

Процессор с тактовой частотой 150 МГц производился с использованием техпроцесса 0.60 мкм, более поздние модели – 0.35 мкм. Кристалл самого процессора состоял из более чем 5.5 млн. транзисторов, кэш-память содержала от 15.5 до 31 млн. транзисторов.

Pentium II (**Klamath**) появились 7 мая 1997 года. Эти процессоры объединили архитектуру Pentium PRO и технологию MMX. По сравнению с Pentium Pro удвоен размер L1 (16 Кб + 16 Кб). В процессоре была использована новая технология корпусов - картридж с печатным краевым разъемом, на который выведена системная шина: SECC (Single Edge Contact Cartridge). Выпускался в конструктиве Slot 1. На картридже размером 14×6.2×1.6 см установлена микросхема ядра процессора, несколько микросхем, реализующих L2, и вспомогательные дискретные элементы (резисторы и конденсаторы).



Такой подход можно считать шагом назад – у Intel уже была отработана технология встраивания в ядро кэша второго уровня. Но таким образом можно было использовать микросхемы памяти сторонних производителей. В свое время, Intel считала такой подход перспективным на ближайшие 10 лет, хотя через непродолжительное время отказалась от него.

В то же время сохранилась независимость шины L2, которая тесно связана с ядром процессора собственной локальной шиной. Частота этой шины была вдвое меньше частоты ядра. Так что Pentium II имел большую L2, работающую на половинной частоте процессора.

Pentium II насчитывал около 7.5 млн. транзисторов только в процессорном ядре и выполнялся по технологии 0.35 мкм. Он имел тактовые частоты ядра 233, 266 и 300 МГц при частоте системной шины 66 МГц. При этом L2 работал на половинной частоте ядра и имел объем 512 Кб.

Для этих процессоров был разработан Slot 1, по составу сигналов схожий с Socket 8 для Pentium Pro. Однако Slot 1 позволяет объединять лишь пару процессоров для реализации симметричной мультипроцессорной системы, либо системы с избыточным контролем функциональности.

26 января 1998 году вышел процессор из линейки **Pentium II (Deschutes)**. От Klamath отличался более тонким технологическим процессом – 0.25 мкм и частотой шины 100 МГц. Имел тактовые частоты 350, 400, 450 МГц. Выпускался в конструктиве SECC, который в старших моделях был сменен на SECC2 - кэш с одной стороны от ядра, а не с двух, как в стандартном Deschutes и измененное крепление кулера.

Процессор состоял из 7.5 млн. транзисторов и выпускался для разъема Slot 1.

Pentium II OverDrive – процессор, вышедший 11 августа 1998 года, был предназначен для замены Pentium PRO на старых материнских платах. Он носил кодовое имя **P6T**. Имел частоту 333 МГц. Кэш первого уровня – 16 Кб на данные + 16 Кб на инструкции, кэш второго уровня имел размер 512 Кб, был интегрирован в ядро и работал на частоте процессора. Шина тактировалась частотой 66 МГц.

МП содержал 7.5 млн. транзисторов и производился по техпроцессу 0.25 мкм.

Микропроцессоры Celeron (Covington) стали новой веткой в направлении технологии микропроцессоров, направленной

на удешевление своей продукции. Он был выпущен как альтернативный вариант Pentium II, имевший довольно высокое соотношение «цена-производительность». МП был выпущен 15 апреля 1998 года и работал на тактовой частоте 266 МГц.

Этот процессор по числу устройств в нем был «усеченным» Pentium II. Celeron был построен на базе ядра Deschutes и не имел кэш-памяти второго уровня. Это привело к снижению его производительности, но и существенно снизило его стоимость. Celeron работал на шине 66 МГц и повторял все основные характеристики своего предка – Pentium II Deschutes: L1 – 16 Кб + 16 Кб, MMX, техпроцесс 0.25 мкм. 7.5 млн. транзисторов. Процессор выпускался без защитного картриджа - конструктив – SEPP (Single Edge Pin Package). Разъем - Slot 1.

Начиная с частоты 300 МГц, появились процессоры **Celeron (Mendocino)** с интегрированным в ядро L2, работающим на частоте процессора, размером 128 Кб. Он вышел 8 августа 1998.

Благодаря высокоскоростному L1 имел хорошую производительность, сравнимую с Pentium II (при условии одинаковой частоты системной шины). Выпускались с тактовыми частотами от 300 до 533 МГц. 30 ноября 1998 года.

До 433 МГц выпускался в двух конструктивах: SEPP и PPGA. Некоторое время параллельно существовали Slot-1 (266 - 433 МГц) и Socket-370 (300A - 533 МГц) варианты, в конце концов, первый был вытеснен последним.

Celeron (Mendocino) был шагом к Pentium III, но, работая на шине 66 МГц, не мог показать все преимущества интегрированного высокоскоростного L1. Так как L1 был интегрирован в ядро, значительно увеличилось количество транзисторов, из которых состоит процессор - 19 млн. Техпроцесс остался прежним – 0.25 мкм.

Для мощных систем Intel выпустил 29 июня 1998 года МП **Pentium II Xeon** - серверный вариант процессора Pentium II, пришедший на смену Pentium PRO. Он производился на ядре Deschutes и отличался от Pentium II более быстрой и более емкой (есть варианты с 1 или 2 Мб) кэш-памятью второго уровня. Вы-

пускался в конструктиве SECC для Slot 2. Это тоже краевой разъем, но с 330 контактами, регулятором напряжения VRM, запоминающим устройством EEPROM. Способен работать в мультипроцессорных конфигурациях.

L2, как и в Pentium PRO, полноскоростной. Только здесь он находится на одной плате с процессором, а не интегрирован в ядро. L1 – 16 Кб + 16 Кб. Частота шины – 100 МГц. Поддерживал набор инструкций MMX.

Процессор работал на частотах 400 и 450 МГц. Выпускался с применением техпроцесса 0.25 мкм. и содержал 7.5 млн. транзисторов.

Pentium III (Katmai) был разработан к 26 февраля 1999 года и мало чем отличался от Pentium II. Он работал на такой же шине с первоначальной частотой 100 МГц, позже появились модели, работающие на шине 133 МГц. МП выпускался в конструктиве SECC 2 и был рассчитан на установку в разъем Slot 1.

Кэш-память осталась прежней: L1 – 16 Кб + 16 Кб, а L2 – 512 Кб. Они были размещены на процессорной плате и работали на половинной частоте процессора.

Главным отличием МП является расширение набора SIMD-инструкций - SSE (Streaming SIMD Extensions). Также расширен набор команд MMX и усовершенствован механизм потокового доступа к памяти.

Процессор работал на частотах 450-600 МГц, содержал 9.5 млн. транзисторов. Выпускался с применением техпроцесса 0.25 мкм.

Pentium III (Coppermine) был выпущен 25 октября 1999 года. По сути, именно Coppermine является новым процессором, а не доработкой Deschutes. Новый процессор имел полноскоростной интегрированный в ядро L2 размером 256 Кб (Advanced Transfer Cache).

Выпускался с использованием техпроцесса 0.18 мкм. Изменение технологии с 0.25 до 0.18 мкм позволило разместить на ядре большее число транзисторов и теперь их стало 28 млн. Правда, ос-

новная масса нововведенных транзисторов относится к интегрированному L2. Заметим, что L1 кэш остался без изменений.

МП поддерживал наборы команд MMX и SSE. Сначала выпускался в конструктиве SECC 2, но так как кэш был встроен в ядро процессора, процессорная плата оказалась ненужной, и только повышала стоимость процессора. Поэтому вскоре процессоры стали выходить в конструктиве FC-PGA (Flip-Chip PGA). Как и Celeron Mendocino, они работали в разъеме Socket 370.

Coppermine был последним процессором для Slot 1. Работал с шиной, имевшей частоту тактирования 100 и 133 МГц.

Pentium III (Tualatin) пришел на смену Coppermine 21 июня 2001 года. В это время на рынке уже присутствовали первые процессоры Pentium 4, и новый процессор был предназначен для испытания новой 0.13 мкм. технологии, а также для того чтобы заполнить нишу высокопроизводительных процессоров, так как производительность первых Pentium 4 была довольно низкой.

Tualatin - это изначальное название глобального проекта Intel по переводу производства процессоров на 0.13-микронную технологию. Сами процессоры с новым ядром стали первыми продуктами, появившимися в рамках этого проекта.

Изменений в самом ядре немного - добавилась только технология "Data Prefetch Logic". Она повышала производительность, предварительно загружая данные, необходимые приложению в кэш. Разъем для нового процессора остался прежним - Socket 370, а вот конструктив сменился на FC-PGA 2, который использовался в процессорах Pentium 4. От старого FC-PGA он в первую очередь отличается тем, что ядро было покрыто теплорассеивающей пластиной, которая также защищает его от повреждения при установке радиатора.

МП Tualatin работали на шине с частотой 133 МГц и состояли из 44 млн. транзисторов. Поддерживали наборы инструкций MMX и SSE. Процессор работал на частотах от 1 ГГц до 1.33 ГГц (Desktop Tualatin), и от 1.13 ГГц до 1.4 ГГц (серверный вариант).

Celeron (Coppermine Lite) были разработаны 29 марта 2000 года с целью, чтобы не терять позиций на рынке бюджетных процессоров. Теперь это были абсолютно другие процессоры – Intel повторил опыт создания первых процессоров с названием Celeron: использовал ядро процессора Pentium III с обрезанным до 128 кб L2 и медленной шиной 66 МГц.

Как видно из названия, процессор выполнен на ядре Coppermine с вдвое уменьшенным L2. Как и Pentium III Coppermine, новый Celeron, имел набор дополнительных команд SSE, быструю встроенную L1 и производится по той же технологической норме (0.18 мкм.), отличаясь только объемом L2 - 128 Кб против 256 Кб у Pentium III. Работает в том же разъеме Socket 370.

Первые процессоры появились с частотой 566 МГц и работали на шине 66 МГц. Позже, 3 января 2001 года, с выходом 800 МГц версии, Celeron перешел на более быструю 100 МГц шину. Максимальная частота этих процессоров составляла 1100 МГц. Кэш первого уровня: 32 Кб (16 Кб на данные и 16 Кб на инструкции). Процессор состоял из 28.1 млн. транзисторов.

2-го октября 2001 года, Intel переводит процессор **Celeron** на новое ядро – **Tualatin**. Еще никогда Celeron не был так близок к процессору Pentium. От Pentium III он отличался лишь более медленной 100 МГц шиной. В общем, оставив неизменным объем L2 и снизив частоту FSB до 100 МГц. Процессоры выпускались с тактовыми

частотами от 900 МГц до 1400 МГц, состояли из 44 млн. транзисторов, поддерживали MMX, SSE. Техпроцесс 0.13 мкм. Выпускались в конструктиве FC-PGA 2, для разъема Socket 370.

Pentium III (Tanner) был построен на базе Pentium III Katmai. Содержал 512, 1024 или 2048 Кб полноскоростной кэш памяти второго уровня. L1 - 16 Кб + 16 Кб. Выпускался с частотами 500 и 550 МГц с применением 0.25 мкм. техпроцесса и состоял из 9.5 млн. транзисторов. Работал на 100 МГц системной шине. Выпускался в конструктиве SECC для Slot 2. Был предназначен для использования в двух-, четырех-, восьмипроцессорных (и более) серверах и рабочих станциях.

С переходом Pentium III на новое ядро 25 октября 1999 года появилась и модификация МП **Xeon (Cascades).** По сути, это было модернизированное ядро Coppermine. Процессор имел от 256 КБ до 2048 Кб кэш памяти второго уровня, работал на частотах системной шины 100 и 133 МГц (в зависимости от версии). Выпускались процессоры с частотами от 600 до 900 МГц. Процессоры с частотой 900 МГц из первых партий перегревались и их поставки были временно приостановлены. Как и предшественник, Xeon Cascades был рассчитан на установку в разъем Slot 2. Выпускался с применением 0.18 мкм. техпроцесса и состоял из 28.1 млн. транзисторов

Pentium 4 с **NetBurst Micro-Architecture** были предназначены для работы на частотах порядка нескольких гигагерц, Intel увеличило длину конвейера Pentium 4 до 20 ступеней (Hyper Pipelined Technology) за счет чего удалось даже при технологических нормах 0,18 мкм добиться работы процессора на частоте в 2 ГГц. Однако из-за такого увеличения длины конвейера время выполнения одной команды в процессорных тактах также сильно увеличивается. Поэтому компания провела доработку алгоритмов предсказания переходов (Advanced Dynamic Execution).

L1 в процессоре претерпела значительные изменения. В отличие от Pentium III, она могла хранить и команды, и данные.

Pentium 4 имел всего 8 Кб кэш данных. Команды, сохраняются в так называемом Trace Cache. Там они хранятся уже в декодированном виде, т.е. в виде последовательности микроопераций, поступающих для выполнения в исполнительные устройства процессора. Емость этого кэша составляет 12000 микроопераций.

В новом процессоре был расширен набор команд - SSE2. К 70 инструкциям SSE, добавились еще 144 новые инструкции. Одной из множества инноваций была совершенно новая 100 МГц шина, передающая по 4 пакета данных за такт - QPB (Quad Pumped Bus), что дает результирующую частоту 400 МГц.

Первым из линейки Pentium 4 был МП **Willamette 423**. Появившись 20 ноября 2000 года с частотами 1.4 и 1.5 ГГц, эти процессоры, изготовленные с применением техпроцесса 0.18 мкм, достигли частоты 2 ГГц.

Процессор устанавливался в новый разъем Socket 423 и выпускался в конструктиве FC-PGA 2. Он состоял из 42 млн. транзисторов.

Кэш 2-го уровня остался прежнего объема - 256 Кб. Ширина шины L2 составляет 256 бит, но латентность кэш-памяти уменьшилась в два раза, что позволило добиться пропускной способности кэша в 48 Гб при частоте 1.5 ГГц.

Так как архитектура нового процессора была ориентирована в первую очередь на рост частоты, то первые процессоры Pentium 4 показали крайне низкую производительность. В большинстве задач 1.4 ГГц процессор уступал Pentium III Coppermine, работающему на частоте 1000 МГц.

27 августа 2001 года, появились МП **Willamette** предназначенные для установки в новый разъем - Socket 478. Процессор повторял все характеристики своего предка, за исключением конструктива - mPGA и разъема Socket 478.

Размеры процессора уменьшились благодаря тому, что теперь выводы сделаны непосредственно под ядром процессора.

Этот процессор, как и предшественник, работал на частотах от 1.4 до 2.0 ГГц.

Pentium 4 Northwood – так называется следующее ядро, на котором выпускались процессоры Pentium 4. Переход на 0.13 мкм. техпроцесс позволил еще больше наращивать тактовую частоту, и увеличить кэш второго уровня до 512 Кб. Увеличилось и количество транзисторов, которые составляют процессор – теперь их стало 55 млн. Естественно, что осталась поддержка наборов инструкций MMX, SSE и SSE2.

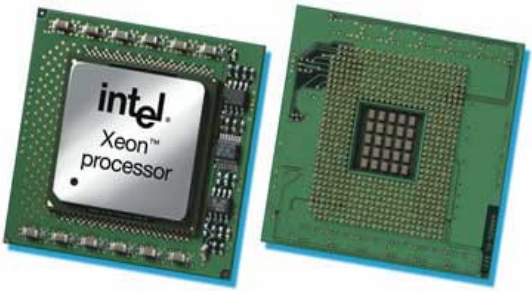
Первые процессоры на ядре Northwood появились 7 августа 2001 года с частотой 2.0 ГГц и частотой системной шины 400 МГц (4×100 МГц). МП Northwood, работают на частотах от 1.6 до 3.2 ГГц.

6-го мая 2002 года, Intel выпустила процессор на базе ядра Northwood с частотой системной шины 533 МГц (4×133 МГц) и тактовой частотой 2.26 ГГц. Так как модели с частотой шины 400 МГц выпускались с частотами до 2.6 ГГц, то и тут была применена буквенная маркировка.

14 апреля 2003 года выпускается процессор на все том же ядре Northwood, но уже с частотой системной шины 800 МГц (4×200 МГц) и тактовой частотой 3.0 ГГц. Позже, процессоры с 800 МГц системной шиной стали выпускаться с меньшими частотами – от 2.4 ГГц.

Pentium 4 XEON были представлены Intel 21 мая 2001 года, который базировался на ядре Willamette. Процессор выпущен в трех вариантах: 1.4 ГГц, 1.5 ГГц и 1.7 ГГц. Ядро процессора почти полностью идентично обычной версии Pentium 4 за исключением незначительных деталей. Это означает, что новый Xeon имеет все то, что есть в Pentium 4 – как достоинства новой архитектуры, так и ее недостатки.

Первые модели Xeon выпускались с применением 0.18 мкм. техпроцесса, с ядром, практически полностью повторявшим Pentium 4 Willamette и носившем кодовое имя **Foster**. Процессор выпускался с тактовыми частотами до 2,0 ГГц. Он состоял из 42 млн. транзисторов.



Кэш память первого уровня, как и у всех процессоров линейки Pentium 4, с архитектурой NetBurst, 8 Кб кэш данных. Кэш второго уровня – 256 Кб с улучшенной передачей данных (256 Кб Advanced Transfer Cache). Также как в Pentium 4 Willamette, в новом Xeon применена 400 МГц системная шина (4×100 МГц) которая синхронно работает с двумя каналами памяти на частоте 400 МГц.

Исторически, линейки процессоров Intel Xeon (то есть Pentium II Xeon, Pentium III Xeon) всегда использовали отличный от обычных версий процессора конструктив. В то время как процессоры Pentium II и Pentium III выпускались в 242-контактном Slot 1 варианте, то их Xeon версии использовали 330-контактный разъем Slot-2. Большинство добавочных ножек использовалось для снабжения кристалла дополнительной энергией.

С двумя мегабайтами L2 Pentium III Xeon потреблял больше энергии, чем его 256-килобайтный собрат. Аналогичная ситуация произошла и с новым Xeon. Если первые процессоры Pentium 4 Willamette, используют 423-контактный разъем, то в Xeon применяется 603-контактный интерфейс, предназначенный для использования в разъеме Socket 603.

Процессор мог работать только в одно- или двухпроцессорных конфигурациях.

9 января 2002 года появляются процессоры Xeon, сделанные на базе ядра Northwood с применением 0.13 мкм. техпроцесса, и оснащенные 512 Кб кэш памяти второго уровня. Кодовое название ядра – **Prestonia**.

От своего предшественника Xeon Foster отличается только увеличенной кэш-памятью и более совершенным техпроцессом.

Процессоры работали на частотах от 1.8 ГГц, до 3.0 ГГц и состояли из 55 млн. транзисторов. В процессорах с ядром Prestonia впервые появилась поддержка Hyper-Threading.

12 марта 2002 года, выходит процессор **Xeon MP**. Изготовлен с применением 0.18 мкм. и оснащен 256 Кб кэш памяти второго уровня. Основное отличие от процессоров Xeon Foster - возможность работать в многопроцессорных системах. Они работали на частотах от 1.4 до 1.6 ГГц. В этих процессорах осуществлена поддержка технологии Hyper-Threading (HT).

Сущность этой технологии заключается в том, что один физический процессор с Hyper-Threading видится системой как два, что позволяет оптимизировать загрузку его ресурсов и повысить производительность. В каждый момент времени только часть ресурсов процессора используется при выполнении программного кода. Неиспользуемые ресурсы также можно загрузить работой - например, задействовать для параллельного выполнения еще одного приложения (либо другого потока этого же приложения).

HT – это не настоящая многопроцессорность, ведь количество блоков непосредственно исполняющих команды не изменилось. Повысился лишь коэффициент их использования. Поэтому, чем лучше оптимизирована конкретная программа под HT, тем выше будет выигрыш в производительности.

По данным Intel, преимущество от HT может достигать 30%, в то время как блоки, ее реализующие, занимают менее 5% общей площади кристалла Pentium 4. Впрочем, даже идеально оптимизированные приложения могут, к примеру, обращаться к данным, которых нет в кэш-памяти процессора, заставляя его простаивать. Если сама архитектура NetBurst была рассчитана на повышение количества мегагерц, то Hyper-Threading наоборот, рассчитан на повышение выполняемой работы за один такт.

Одной из причин достаточно позднего представления Hyper-Threading в Pentium 4 (поддержка существует не только в ядре Northwood, но даже в Willamette, однако была заблокирована) являлась относительно небольшая распространенность Windows XP – единственной ОС семейства Windows, полноценно поддерживающей новую технологию. Также технологию должен поддерживать чипсет и BIOS системной платы.

Технологию Hyper-Threading поддерживает процессор Pentium 4 3.06 ГГц с частотой системной шины 533 МГц, а также все процессоры с частотой шины 800 МГц.

4 ноября 2002 года появляются процессоры Xeon MP, изготовленные с применением 0.13 мкм. техпроцесса. Эти процессоры, работающие на частотах 1.5 ГГц, 1.9 ГГц и 2.0 ГГц отличаются от своего собрата Xeon Prestonia не только возможностью работы в многопроцессорных конфигурациях, но и наличием интегрированной L3 размером 1 или 2 Мб. Благодаря этому увеличилось количество транзисторов, составляющих процессор до 108 млн.

18 ноября 2002 года появились процессоры Xeon работающие на 533 МГц (4 × 133 МГц) системной шине. Эти процессоры сделаны на ядре Prestonia, с применением 0.13 мкм. техпроцесса и состоят из 108 млн. транзисторов. Кэш - память второго уровня – 512 Кб, третьего уровня - 1 Мб. Процессоры Xeon на 533 МГц шине выпускаются с тактовыми частотами от 2.0 ГГц до 3.06 ГГц (вышел 10 марта 2003).

Pentium 4 HT начали выпускаться 14 ноября 2002 года. Он имеет частоту 3.06 ГГц, а системная шина тактировалась частотой 533 МГц с поддержкой технологии Hyper-Threading.

Celeron (Willamette 128) был выпущен с целью вытеснения с рынка процессоров для Socket 370, а также, желая занять нишу бюджетных процессоров (где до этого был Celeron Tualatin.

Ядро Willamette 128 архитектурно ничем не отличается от ядра Pentium 4 Willamette. Организация кэш - памяти и алгоритмы его работы не изменились, единственное отличие заключается в размере - 128 Кб кэш - памяти второго уровня вместо 256 Кб в оригинальном Pentium 4 Willamette.

Естественно, сохранен и форм-фактор Socket 478. 15 мая 2002 года появляется первый процессор с названием Celeron, построенный на базе Pentium 4, с частотой 1.7 ГГц. Позже, 12 июня 2002 года появляется версия на 1.8 ГГц.

Новый Celeron, как и раньше, использует 100 МГц системную шину, правда теперь уже с передачей 4-х сигналов за такт. Учетверенная 100 МГц системная шина наконец-то решает старую проблему Celeron - недостаток пропускной способности FSB.

Celeron выполнен с применением 0.18 мкм. техпроцесса. Состоит из 42 млн. транзисторов. Выпускается с частотами 1.7 и 1.8 ГГц.

Следующее ядро процессора Celeron, это **Northwood** (с урезанной до 128 Кб кэш - памятью второго уровня). Первым процессором на этом ядре был Celeron 2.0 ГГц, который вышел 18 сентября 2002 года. Он, как и Celeron Willamette 128, полностью повторяет характеристики старшего брата Pentium 4 Northwood, за исключением шины, рассчитанной исключительно на 400 МГц (4 × 100 МГц) и кэш - памятью второго уровня размером 128 Кб.

Применение 0.13 мкм. техпроцесса дает преимущество в виде хорошей разгоняемости. У ядра Northwood хороший частотный потенциал, поэтому запас для разгона есть.

В конце 2003 года Intel представила новое ядро для своих процессоров – **Prescott**. Эти процессоры изготовлены с применением 0.09 мкм. (90 нм) технологии. Ядро Prescott состоит из 125 млн. транзисторов, содержит 1 Мб кэш- - память второго уровня, увеличена кэш - память первого уровня до 32 Кб. Ядро обладает поддержкой технологии Hyper-Threading 2, дальнейшее развитие «многопроцессорности» в одном чипе.

В МП добавлен новый набор инструкций (или расширен уже присутствующий), включающий 15 новых инструкций по переводу чисел с плавающей запятой в целые, арифметику комплексных чисел, специальные команды для декодирования видео, SIMD-инструкции для формата с плавающей запятой и процесс синхронизации потоков.

Первые процессоры с этим ядром предназначены для работы на частотах 3.2 и 3.4 ГГц. Их корпуса совместимы с корпусами процессоров Pentium 4 Northwood.

На базе нового ядра продолжен выпуск процессоров линейки Celeron. Чипы **Celeron** на ядре **Prescott** быстрее предшественников на Northwood не только за счет возросшей тактовой частоты ядра. Они поддерживают системную шину с частотой 533 МГц, а объем их кэш - памяти увеличен со 128 до 256 кб. Celeron на ядре Prescott имеют частоты 2.8 и 3.06 ГГц.

Pentium 4 Extreme Edition оснащен технологией Hyper-Threading, работает на системной шине 800 МГц, имеет тактовую частоту ядра 3.2 ГГц. Но главным его отличием от предшествующих Pentium 4 стало наличие интегрированной в кристалл кэш-памяти третьего уровня L3 объемом 2 Мб. Эта кэш-память дополняет стандартный кэш L2 512 кбайт и работает также на частоте ядра процессора (правда, с гораздо большей латентностью, поскольку она асинхронная и призвана ускорять работу с данными из наиболее часто используемых областей системной памяти). Таким образом, Pentium 4 Extreme Edition имеет кэш-память объемом 2.5 Мб. А также является единственным desktop процессором с кэшем третьего уровня, интегрированным в ядро.

Процессор Pentium 4 Extreme Edition позиционируется Intel главным образом для игрового рынка, хотя не исключено и его применение в производительных рабочих станциях. Процессор использует ядро от мультипроцессорных Xeon MP с интегрированной кэш-памятью L3. Его немного изменили с целью поддержки системной шины 800 МГц, уменьшения энергопотребления и др. и упаковали в стандартный корпус от Pentium 4.

В настоящее время Intel ведет активные работы по созданию следующих поколений кристаллов с проектными технологическими нормами 65 нм. Также ведутся разработки и есть работающие чипы, изготовленные с применением не только 0.065 мкм. техпроцесса, но и 45 нм, 32 нм и даже 22 нм.

За Prescott планируется выпуск МП на ядре **Tejas** с шиной 1066 МГц. На его основе будут представлены восемь различных процессоров с тактовыми частотами от 6 до 9.2 ГГц. После этого компания планирует представить ядро **Nehalem**, использующее

системную шину 1200 МГц и позволяющее получить рабочую частоту свыше 10 ГГц. Nehalem будет основан на совершенно новой архитектуре. Это будет не модернизированный Pentium 4, как Prescott и Tejas. В нем будет применена система аппаратной защиты LaGrande, и по некоторым данным, использована более совершенная технология многопоточной обработки. Число транзисторов в чипе составит порядка 150-250 миллионов

4. Классификация микропроцессоров

Микропроцессор как функциональное устройство ЭВМ обеспечивает эффективное автоматическое выполнение операций обработки цифровой информации в соответствии с заданным алгоритмом. Для решения широкого круга задач в различных областях применений микропроцессор должен обладать алгоритмически полной системой команд (операций).

Теоретически показано, что минимальная алгоритмически полная система команд процессора состоит из одной или нескольких универсальных команд. Однако использование процессоров с минимальными по числу операций системами команд ведет к неэкономичному использованию информационных емкостей памяти и значительным затратам времени на выполнение «длинных» программ. Поэтому обычно в МП встраиваются аппаратурные средства, позволяющие реализовать многие десятки и сотни команд. Такие развитые системы команд дают возможность обеспечить компактную запись алгоритмов и соответственно эффективные программы.

При проектировании МП решаются задачи определения наборов команд, выполняемых программным или аппаратурным способом на основе заданной системы микрокоманд. Аппаратурная реализация сложных команд дает возможность увеличить быстродействие микропроцессора, но требует значительных аппаратурных ресурсов кристалла интегральной схемы МП. Программная реализация сложных команд позволяет обеспечивать программирование сложных задач, изменять количество и особенности исполнения сложных команд. Однако скорость исполнения про-

граммных команд ниже скорости исполнения аппаратурно-реализованных команд.

Практически во всех современных МПС используются сложные развитые системы команд. Их ядро, состоящее из набора универсальных команд, реализуется аппаратурным способом в центральном МП. Кроме того, специализированные части наборов системы команд реализуются вспомогательными или периферийными микропроцессорами. Эти расширяющие возможности обработки данных специальные арифметические или логические МП позволяют ускорить выполнение определенных команд и тем самым сократить время исполнения программ.

Для описания МП как функциональных устройств необходимо охарактеризовать формат обрабатываемых данных и команд, количество, тип и гибкость команд, методы адресации данных, число внутренних регистров общего назначения и регистров результата, возможности организации и адресации стека, параметры виртуальной памяти и информационную емкость прямо адресуемой памяти. Большое значение имеют средства построения системы прерываний программ, построения эффективных систем ввода — вывода данных и развитого интерфейса.

МП могут быть реализованы на различной физической основе: на электронной, оптоэлектронной, оптической, биологической и даже на пневматической или гидравлической.

По назначению различают универсальные и специализированные микропроцессоры.

Универсальные МП предназначены для решения широкого круга задач. При этом их эффективная производительность слабо зависит от проблемной специфики решаемых задач В системе команд МП заложена алгоритмическая универсальность, означающая, что выполняемый машиной состав команд позволяет получить преобразование информации в соответствии с любым заданным алгоритмом.

К универсальным МП относятся и секционные микропроцессоры, поскольку для них система команд может быть оптимизирована в каждом частном проекте создания секционного микропроцессора.

Эта группа МП наиболее многочисленна, в нее входят такие комплекты как К580, Z80, Intel 80×86, К582, К587, К1804, К1810 и др.

Специализированные МП предназначены для решения определенного класса задач, а иногда только для решения одной конкретной задачи. Их существенными особенностями являются простота управления, компактность аппаратурных средств, низкая стоимость и малая мощность потребления.

Специализированные МП имеют ориентацию на ускоренное выполнение определенных функций, что позволяет резко увеличить эффективную производительность при решении только определенных задач.

Среди специализированных микропроцессоров можно выделить различные микроконтроллеры, ориентированные на выполнение сложных последовательностей логических операций; математические МП, предназначенные для повышения производительности при выполнении арифметических операций за счет, например матричных методов их выполнения; МП для обработки данных в различных областях применений и т. д.

С помощью специализированных МП можно эффективно решать новые сложные задачи параллельной обработки данных. Например, они позволяют осуществить более сложную математическую обработку сигналов, чем широко используемые методы корреляции, дают возможность в реальном масштабе времени находить соответствие для сигналов изменяющейся формы путем сравнения их с различными эталонными сигналами для эффективного выделения полезного сигнала на фоне шума и т.д.

По виду обрабатываемых входных сигналов различают цифровые и аналоговые микропроцессоры.

Сами МП являются цифровыми устройствами обработки информации. Однако в ряде случаев они могут иметь встроенные аналого-цифровые и цифро-аналоговые преобразователи. Поэтому входные аналоговые сигналы передаются в МП через преобразователь в цифровой форме, обрабатываются и после обратного преобразования в аналоговую форму поступают на выход.

С архитектурной точки зрения такие микропроцессоры представляют собой аналоговые функциональные преобразователи

сигналов. Они выполняют функции любой аналоговой схемы (например, производят генерацию колебаний, модуляцию, смещение, фильтрацию, кодирование и декодирование сигналов в реальном масштабе времени и т. д., заменяя сложные схемы, состоящие из операционных усилителей, катушек индуктивности, конденсаторов и т.д.). При этом применение аналогового МП значительно повышает точность обработки аналоговых сигналов и их воспроизводимость, а также расширяет функциональные возможности за счет программной “настройки” цифровой части микропроцессора на различные алгоритмы обработки сигналов.

Обычно в составе однокристальных аналоговых МП имеется несколько каналов аналого-цифрового и цифро-аналогового преобразования. В аналоговом микропроцессоре разрядность обрабатываемых данных достигает 24 бит и более. Большое значение уделяется увеличению скорости выполнения арифметических операций.

Отличительная черта аналоговых МП - это способность к переработке большого объема числовых данных, т. е. к выполнению операций сложения и умножения с большой скоростью, при необходимости даже за счет отказа от операций прерываний и переходов. Аналоговый сигнал, преобразованный в цифровую форму, обрабатывается в реальном масштабе времени и передается на выход обычно в аналоговой форме через цифро-аналоговый преобразователь. При этом согласно теореме Котельникова частота квантования аналогового сигнала должна вдвое превышать верхнюю частоту сигнала.

Одним из направлений дальнейшего совершенствования аналоговых МП является повышение их универсальности и гибкости. Поэтому вместе с повышением скорости обработки большого объема цифровых данных будут развиваться средства обеспечения развитых вычислительных процессов обработки цифровой информации за счет реализации аппаратурных блоков прерывания программ и программных переходов.

По количеству выполняемых программ различают одно- и многопрограммные микропроцессоры.

В однопрограммных МП выполняется только одна программа. Переход к выполнению другой программы происходит после завершения текущей программы.

В много- или мультипрограммных МП одновременно выполняется несколько (обычно несколько десятков) программ. Организация мультипрограммной работы микропроцессорных управляющих систем, например, позволяет осуществить контроль за состоянием и управлением большим числом источников или приемников информации.

По числу БИС в микропроцессорном комплекте различают однокристальные, многокристальные и многокристальные секционные МП.

Процессоры даже самых простых ЭВМ имеют сложную функциональную структуру, содержат большое количество электронных элементов и множество разветвленных связей. Реализовать принципиальную схему обычного процессора в виде одной или нескольких БИС практически невозможно из-за специфических особенностей БИС (ограниченность количества элементов, сложность выполнения разветвленных связей, сравнительно небольшое число выводов корпуса). Поэтому необходимо изменять структуру процессора так, чтобы полная принципиальная схема или ее части имели количество элементов и связей, совместимое с возможностями БИС. При этом МП приобретают внутреннюю магистральную структуру, т. е. в них к единой внутренней информационной магистрали подключаются все основные функциональные блоки (арифметико-логический, рабочих регистров, стека, прерываний, интерфейса, управления и синхронизации и др.).

Для обоснования классификации МП по числу БИС надо распределить все аппаратурные блоки процессора между основными тремя функциональными частями: операционной, управляющей и интерфейсной. Сложность операционной и управляющей частей процессора определяется их разрядностью, системой команд и требованиями к системе прерываний; сложность интерфейсной части - разрядностью и возможностями подключения других устройств ЭВМ (памяти, внешних устройств, датчиков и исполнительных механизмов и др.). Интерфейс процессора содер-

жит несколько десятков шин информационных магистралей данных, адресов и управления.

Однокристальные МП получаются при реализации всех аппаратурных средств процессора в виде одной БИС или СБИС. По мере увеличения степени интеграции элементов в кристалле и числа выводов корпуса параметры однокристальных МП улучшаются. Однако их возможности ограничены аппаратурными ресурсами кристалла и корпуса. Поэтому более широко распространены многокристальные, а также многокристальные секционные МП.

Для получения многокристального МП необходимо провести разбиение его логической структуры на функционально законченные части и реализовать их в виде БИС (СБИС). Функциональная законченность БИС многокристального МП означает, что его части выполняют заранее определенные функции и могут работать автономно, а для построения развитого процессора не требуется организации большого количества новых связей и каких-либо других электронных ИС (БИС). (Типичный пример - МПК БИС серии К581).

На рис. 4,а показано функциональное разбиение структуры МП при создании трехкристального микропроцессора (пунктирные линии), содержащие БИС операционного (ОП), БИС управляющего (УП) и БИС интерфейсного (ИП) процессоров.

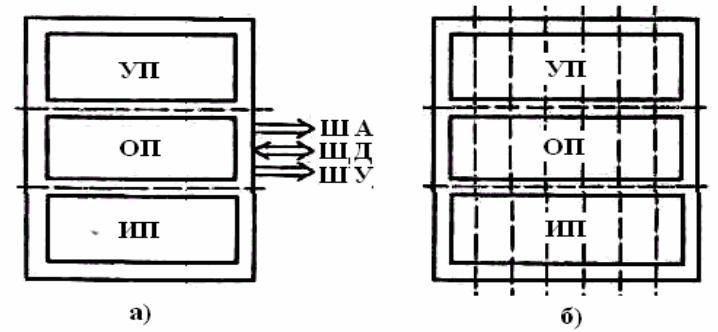


Рис. 4

Операционный процессор ОП служит для обработки данных, управляющий процессор УП выполняет функции выборки, декодирования и вычисления адресов операндов и также генери-

рует последовательности микрокоманд. Автономность работы и большое быстродействие БИС УП позволяет выбирать команды из памяти с большей скоростью, чем скорость их исполнения БИС ОП. При этом в УП образуется очередь еще не исполненных команд, а также заранее подготавливаются те данные, которые потребуются ОП в следующих циклах работы.

Такая опережающая выборка команд экономит время ОП на ожидание операндов, необходимых для выполнения команд программ. Интерфейсный процессор ИП позволяет подключить память и периферийные средства к МП; по существу, является сложным контроллером для устройств ввода — вывода информации. БИС ИП выполняет также функции канала прямого доступа к памяти.

Выбираемые из памяти команды распознаются и выполняются каждой частью МП автономно, и поэтому может быть обеспечен режим одновременной работы всех БИС МП, т. е. конвейерный поточный режим исполнения последовательности команд программы (выполнение последовательности с небольшим временным сдвигом). Такой режим работы значительно повышает его производительность.

Многокристальные секционные МП получаются в том случае, когда в виде БИС реализуются части (секции) логической структуры процессора при функциональном разбиении ее вертикальными плоскостями (рис.4,6). Для построения многоразрядных МП при параллельном включении секций БИС МП в них добавляются средства «стыковки».

Для создания высокопроизводительных многоразрядных МП требуется столь много аппаратурных средств, не реализуемых в доступных БИС, что может возникнуть необходимость еще в функциональном разбиении структуры МП горизонтальными плоскостями. В результате рассмотренного функционального разделения структуры МП на функционально и конструктивно законченные части создаются условия реализации каждой из них в виде БИС. Все они образуют комплект секционных БИС МП.

Таким образом, микропроцессорная секция — это БИС, предназначенная для обработки нескольких разрядов данных или выполнения определенных управляющих операций. Секционность

БИС МП определяет возможность «наращивания» разрядности обрабатываемых данных или усложнения устройств управления микропроцессором при «параллельном» включении большего числа БИС.

С момента создания однокристальные МП развились от простых специализированных 4-разрядных до 32-разрядных процессоров. Трехкристальные МП имеют разрядность до 32 бит и параметры, сравнимые с параметрами старших моделей рядов мини-ЭВМ и средних ЭВМ общего применения.

Многокристальные секционные МП имеют разрядность от 2—4 до 8—16 бит и позволяют создавать разнообразные высокопроизводительные процессоры ЭВМ.

Однокристальные и трехкристальные БИС МП, как правило, изготовляют на основе микроэлектронных технологий униполярных полупроводниковых приборов, а многокристальные секционные БИС МП — на основе технологии биполярных полупроводниковых приборов.

Использование многокристальных микропроцессорных высокоскоростных биполярных БИС, имеющих функциональную законченность при малой физической разрядности обрабатываемых данных и монтируемых в корпус с большим числом выводов, позволяет организовать разветвление связи в процессоре, а также осуществить конвейерные принципы обработки информации для повышения его производительности.

По структурному признаку различают МП с фиксированной разрядностью и МП с наращиваемой разрядностью (секционные МП).

МП с фиксированной разрядностью имеют строго определенную разрядность обрабатываемых слов, величина которой определяется разрядностью МП. МП с наращиваемой разрядностью позволяют на их основе секциями увеличивать число разрядов МПС до требуемой величины, что, как правило, используется при построении миниЭВМ и больших ЭВМ вычислительного типа.

По виду алгоритма работы управляющего устройства МП подразделяют на два вида:

- МП с жестким алгоритмом управления, реализуемым схемно (МП с фиксированным набором команд),

- МП с алгоритмом управления, реализуемым программным путем в виде последовательности микроопераций (МП с микропрограммным управлением). Здесь система команд определена не жестко, а зависит от микропрограммы, записанной в ПЗУ, входящей в состав устройства управления. Использование микропрограммного управления дает возможность получить необходимый набор команд, например, для воспроизведения (эмуляции) набора команд другого МП.

По разрядности обрабатываемой информации МП могут быть 4, 8, 12, 16, 24, 32 - разрядными. На практике наибольшее распространение имеют 32 - разрядные МП (Pentium, Celeron, AMD). Все большее применение находят 64-разрядные МП фирмы AMD.

По характеру временной организации работы МП делятся на синхронные и асинхронные.

В синхронных МП начало и конец выполнения каждой операции задаются устройством управления, то есть фаза начала и конца выполнения команды строго привязана к временной оси.

В асинхронных МП начало выполнения следующей операции начинается сразу же после окончания выполнения предыдущей операции**.**

По количеству одновременно выполняемых программ различают одно- и многопрограммные МП.

В однопрограммных МП на текущий момент времени выполняется только одна программа. Переход к выполнению другой программы происходит либо по завершению этой программы, либо по специальной команде условного или безусловного перехода, либо по прерыванию.

В многопрограммных МП одновременно может выполняться несколько программ, то есть обеспечивается мультипрограммный режим работы системы.

По виду технологии изготовления разрабатываются и выпускаются БИС МП:

- по униполярной технологии - p - канальные (p - МОП), n - канальные (n - МОП) и комплиментарные (КМОП) БИС;

- по биполярной технологии - БИС на базе транзисторно-транзисторной логики (ТТЛ), в том числе и с диодами Шотки

(ТТЛШ);

- по эмиттерно-связанной логики (ЭСЛ);
- по интегральной инжекционной логики ($И^2Л$).

Вид технологии изготовления БИС во многом определяет степень интеграции микросхем, быстродействие, энергопотребление, помехозащищенность и стоимость МП. По комплексу этих признаков можно отдать предпочтение МП, выполненным по n - МОП и КМОП технологиям, обеспечивающих высокую плотность компоновки, высокое быстродействие и относительно малую стоимость. ЭСЛ и ТТЛШ технологии обеспечивают МП самое высокое быстродействие, но микропроцессорные БИС (МП БИС) при этом отличаются самой низкой плотностью компоновки и высоким энергопотреблением. МП на основе $И^2Л$ технологии обладают усредненными характеристиками. По плотности компоновки они уступают n - МОП, по быстродействию - ЭСЛ и ТТЛШ, а по стоимости - n - МОП и p - МОП МП. Вместе с тем, p - МОП технология обеспечивает МП наиболее низкую стоимость, но его быстродействие при этом является также наиболее низким.

В процессе развития микропроцессорных средств, кроме микропроцессорных БИС, были разработаны различные интегральные микросхемы, выполняющие различные функции и позволяющие в совокупности построить микроЭВМ требуемой структуры. Эти микросхемы совместно с МП БИС образуют микропроцессорный комплект (МПК БИС), который может быть определен как совокупность конструктивно и электрически совместимых интегральных схем, предназначенных для построения МП, микроЭВМ и других вычислительных устройств с определенным составом и требуемыми технологическими характеристиками.

Основу любого МПК БИС образует базовый комплект интегральных микросхем, который предназначен для построения МПС и может состоять из БИС однокристального или из нескольких корпусов многокристального МП. Базовый комплект, как правило, дополняется другими типами интегральных схем, на которых реализуются запоминающие устройства, устройства сопряжения с объектом и различные устройства ввода - вывода. Эти микросхемы в общем случае могут иметь другой номер серии или даже иной тип корпуса.

Минимальный набор микросхем из состава МПК БИС, позволяющих построить конкретный тип вычислительного устройства, называется микропроцессорным набором интегральных схем.

5. Особенности архитектуры 32-разрядных МП

5.1. Микропроцессоры с RISC – архитектурой

В развитии архитектур МП наблюдается два подхода. Первый из них относится к более ранним моделям процессоров и носит название МП с CISC (Complete Instruction Set Computer) архитектурой - процессоры с полным набором инструкций. К ним относится семейство процессоров 80 × 86. Состав и назначение их регистров существенно неоднородны, широкий набор команд усложняет декодирование инструкций, на что расходуются аппаратные ресурсы. Возрастает число тактов, необходимое для выполнения инструкций.

Процессоры 80 × 86 имеют весьма сложную систему команд, что еще довольно терпимо при использовании ее в 8 - и 16 – разрядных МП. В начале 80-х годов архитектура CISC стала серьезным препятствием на пути реализации идеи «один процессор в одном кристалле», поскольку для работы с «традиционным» расширенным списком команд требуется очень сложное устройство центрального управления (обычно - микропрограммное), занимающее до 60% всей площади кристалла.

В процессорах семейства 80 × 86, начиная с i80486, применяется комбинированная архитектура - CISC-процессор имеет RISC-ядро. Архитектура RISC (Reduced Instruction Set Computer — компьютер с сокращенным набором инструкций) была впервые реализована в 1979 г. в миникомпьютере IBM801. В ней воплотились три основных принципа:

- ориентация системы на поддержку языка высокого уровня с помощью развитого компилятора;
- использование примитивного набора инструкций, который полностью реализуется аппаратными средствами;
- организация памяти и ввода—вывода, которая позволяет выполнять процессором большинство инструкций за один такт.

Первые микропроцессоры с архитектурой RISC были разработаны и изготовлены в начале 80-х годов в Калифорнийском (г. Беркли) и Стандфордском университетах. Разработчики этих МП ставили перед собой задачу достижения наивысшей производительности при наименьшей сложности. В ходе ее решения сложились два подхода.

Первый заключается в снижении числа обращений в память за счет увеличения емкости регистрового файла и организации его в виде перекрывающихся регистровых окон. Архитектура, созданная на этой основе, была впервые реализована в МП RISC I, разработанном в г. Беркли (берклийская архитектура).

Другой подход заключается в устранении задержек конвейера за счет переупорядочения инструкций и интенсивного использования регистров МП при помощи оптимизирующего компилятора. Архитектура, реализующая этот способ, была разработана и впервые применена в г. Стэнфорде (станфордская архитектура).

5.1.1. Общие принципы построения

Рабочие станции и серверы, созданные на базе концепции RISC, завоевали лидирующие позиции благодаря своим исключительным характеристикам. Дело дошло до предсказаний скорого отмирания более традиционных CISC-систем. Чисто академический интерес середины 80-х годов к архитектуре RISC в начале 90-х годов сменился бурным ростом производства промышленных RISC-систем. Практически все ведущие производители - IBM, Hewlett-Packard, DEC, Silicon Graphics - создали процессоры с RISC-архитектурой и выпустили на рынок новые семейства рабочих станций и серверов на их базе. Более того, RISC-системы вышли за границы узких профессиональных приложений и находят все большее признание среди средних пользователей.

В теории цифровых логических систем есть известная аксиома, которая гласит, что любой компьютер в принципе может быть построен с использованием всего одного типа элементов - вентиля "И - НЕ / ИЛИ - НЕ". Однако никому из разработчиков машин 60-х и 70-х годов не приходило в голову отказаться от каталога из десятков и сотен логических микросхем и спроектиро-

вать компьютер на одном типе вентиля.

Никому, кроме Сеймура Крея. Результат известен: суперкомпьютер CRAY-1, созданный в рекордно короткие сроки, оказался меньше и быстрее всех своих предшественников. Нечто подобное произошло и в процессе становления RISC-архитектуры. Идея, заложенная в основу RISC-архитектуры, состояла в следующем: оставить в системе команд всего несколько десятков наиболее употребимых и наиболее универсальных инструкций, исключив сложные и редко используемые.

Результатом должно было стать существенное упрощение центрального управления, а значит, высвобождение части поверхности кристалла процессора для размещения более мощных средств обработки данных. Так возникла философия RISC-архитектуры – «меньше команд - выше скорость», которая основывается на двух фундаментальных постулатах:

- скорость компьютерной обработки определяется не столько быстродействием аппаратных средств, сколько хорошим взаимодействием программного обеспечения и аппаратуры

- за скорость всегда надо платить усложнением либо аппаратуры, либо программных средств, либо того и другого.

Их реализация давно интересовала разработчиков МПС. Еще задолго до разделения компьютеров на RISC- и CISC-семейства было освоено два способа повышения скорости вычислений – «быстрые» технологии и параллелизм обработки. На пути ускорения обработки данных в принципе хорошо известны: схемы на арсениде галлия примерно в четыре раза производительнее схем на кремниевой основе, насыщенная логика по быстродействию уступает, оптимизация откомпилированного кода теоретически позволяет в 2 - 4 раза сократить время выполнения программы и т. д.

Концепция RISC - архитектуры базируется на почти очевидной логической формуле: если «быстрые» технологии и параллельная обработка для всего списка команд недостижимы из-за высокого уровня затрат, то надо ускорять только часто выполняемые операции, а редко применяемыми и сложными следует пожертвовать ради повышения общей производительности. Заметная разница между RISC-компьютерами 80-х и 90-х годов и CISC- ма-

шинами 60-х годов заключается в числе аппаратных шагов, приходящихся на инструкцию. В RISC - процессоре одна инструкция выполняется за один шаг, тогда как в CISC та же инструкция может вызвать сотни и тысячи аппаратных действий.

Конечно, программирование с помощью подобных насыщенных операций позволяет получить компактный исполняемый модуль, но возникает естественный вопрос: «Что лучше - короткая программа с медленными инструкциями или длинная программа с быстрыми инструкциями?". Ответ на него помогли дать прикладные исследования в лабораториях фирмы IBM. Было сконструировано подмножество языка PL/1 под названием PL/8 и написан компилятор с оптимизацией кода для гипотетического компьютера, система команд которого использовала короткие инструкции типа «регистр-регистр». Имитация работы этого компьютера проводилась на мэйнфрейме IBM/370 модели 168. Этот эксперимент дал весьма впечатляющий результат: С большинством наиболее часто употребляемых программных операторов компьютер справилась в 2 - 3 раза быстрее, чем IBM 370/168, запрограммированная при помощи стандартного варианта языка PL/1.

Для последующих семейств мэйнфреймов производства IBM данное соотношение несколько уменьшилось за счет конвейеризации процессора и большего объема кэш-памяти, однако принципиальный вывод из эксперимента в IBM не потерял своей значимости: отказ от применения редких инструкций и оптимизация использования регистров ускоряют вычислительный процесс более чем в два раза.

Итак, исключение из системы команд редко применяемых инструкций и ориентация аппаратных и программных средств на операции типа «регистр-регистр» открывают широкие возможности для экономии оборудования без существенной потери производительности. Но как только на кристалле процессора оказалось свободное место, сразу же нашлись желающие занять его под более мощные средства обработки. Так, в 1985 г. фирма Acorn Corporation of England выпустила 32-разрядный RISC-процессор ARM, примерно эквивалентный по степени интеграции 8-разрядному CISC-процессору Intel 8080 (около 25 тыс. транзисторов), но со значительно большим быстродействием.

Правда, произошло это уже после того, как были сформулированы основные законы RISC-архитектуры. Законы RISC в самом начале 80-х годов почти одновременно завершились теоретические исследования в области RISC-архитектуры, проводившиеся в Калифорнийском университете (г. Беркли), Станфордском университете и в корпорации IBM. Именно тогда были сформулированы четыре основных принципа RISC-архитектуры:

1. Каждая команда независимо от ее типа выполняется за один машинный цикл, длительность которого обратно пропорциональна тактовой частоте процессора и должна быть максимально короткой. Стандартом для RISC - процессоров считается длительность машинного цикла, равная времени сложения двух целых чисел (для современного уровня развития технологии эта величина составляет от 3 до 10 нс).

2. Все команды должны иметь одинаковую длину и использовать минимум адресных форматов; это резко упрощает логику центрального управления процессором. Другим важным следствием принципа простоты адресации является то, что RISC - процессор способен выбирать очередную команду в темпе обработки, т. е. одну команду за один цикл.

3. Обращение к памяти происходит только при выполнении операций записи и чтения, модификация операндов в памяти возможна лишь с помощью команды «запись», вся обработка данных осуществляется исключительно в регистровой структуре процессора.

4. Система команд должна обеспечивать поддержку языков высокого уровня. Имеется в виду подбор системы команд, наиболее эффективной для различных языков программирования.

Само собой разумеется, что четыре перечисленных базовых принципа RISC-архитектуры не существуют вне основного закона RISC: система команд должна содержать минимум наиболее часто используемых и наиболее простых инструкций.

Конечно, в компьютерной практике можно найти немало примеров широкого толкования принципов RISC, однако один закон RISC-архитектуры соблюдается всеми разработчиками неукоснительно - обработка данных должна вестись только в рамках регистровой структуры и только в формате команд «регистр- ре-

гистр».

Регистры - основное достоинство и главная проблема RISC. Все существующие RISC-процессоры базируются на единственном типе обработки данных в формате «регистр-регистр», а точнее, «регистр-регистр-регистр»: R1:=R2,R3. Это позволяет без существенных затрат времени выбрать операнды из адресуемых оперативных регистров и записать в регистр результат операции.

Кроме того, трехместные операции дают компилятору большую гибкость по сравнению с типовыми двухместными операциями формата «регистр-память» архитектуры CISC. В сочетании с быстродействующей арифметикой RISC-операции типа «регистр-регистр» становятся очень мощным средством повышения производительности процессора. Проблема заключается в том, что в процессе выполнения задачи RISC-система неоднократно вынуждена обновлять содержимое регистров процессора, причем за минимальное время, чтобы не вызвать длительных простоев арифметического устройства (а это прямые потери производительности). Для CISC-систем подобной проблемы не существует, поскольку модификация регистров может происходить на фоне обработки команд формата «память – память».

Существует два подхода к решению проблемы модификации регистров в RISC - архитектуре: аппаратный, предложенный в проектах RISC-1и RISC-2 университета в Беркли, и программный, разработанный специалистами IBM и Станфордского университета. Принципиальная разница между ними заключается в том, что аппаратное решение основано на стремлении уменьшить время вызова процедур за счет установки дополнительного оборудования процессора, тогда как программное базируется на возможностях компилятора и является более экономичным с точки зрения аппаратуры центрального процессора.

В RISC-архитектуре используется механизм переключения множественных перекрывающихся регистровых окон - MORS (Multiple Overlapping RegisterSets), иногда называемый структурой регистрового файла Rolodex. Механизм MORS послужил основой архитектуры RISC-1, в соответствии с которой процессор содержит 138 регистров для хранения данных. Из них десять, именуемых глобальными, всегда «видны» программе; их основное назна-

чение - хранение данных, являющихся общими для всех процессов в текущем контексте программы. Остальные 128 регистров разбиты на восемь перекрывающихся окон по 22 регистра. В каждый момент времени программа, исполняемая на RISC-1, «наблюдает» десять глобальных регистров и одно целое окно, т. е. всего 32 регистра.

Идея структуры MORS заключается в минимизации затрат процессорного времени при обращении к процедурам. Для этого каждое из восьми окон связано с конкретной процедурой, а регистры окна разделены на верхние, локальные и нижние. При вызове процедуры B из процедуры A активное окно регистрового файла смещается на шесть позиций так, что верхние регистры процедуры A перекрываются нижними регистрами процедуры B.

Перекрывающиеся зоны окон - это физически одни и те же регистры, доступные обеим процедурам. Они используются для передачи параметров, адресов возврата и позволяют обращаться к процедуре, не обмениваясь данными с оперативной памятью. Таким образом, вызов процедуры реализуется не сложнее, чем, скажем, суммирование регистровых операндов. К тому же эта операция выполняется практически моментально: для обращения к процедуре или для возврата в точку вызова достаточно переместить указатель активного окна регистрового файла. В этом состоит важнейшая особенность архитектуры RISC-1.

Вероятно, именно благодаря своей логической стройности архитектура RISC-1 послужила основой для разработки массовых процессоров Pyramid и SPARC, правда, с небольшими изменениями в организации регистрового файла Rolodex (в SPARC программа «видит» окно из тех же 32 регистров, но количество глобальных, верхних, локальных и нижних регистров одинаково - по восемь в каждой зоне).

Однако структура MORS обладает двумя недостатками – оптимальное размещение процедур по окнам регистрового файла является далеко не тривиальной задачей для ОС, а выбранное число из восьми процедур, сохраняемых в регистровом файле, представляется, скорее, эмпирическим значением. Во всяком случае можно найти множество примеров, когда задача включает существенно большее количество процедур и при этом возникает реаль-

ная проблема модификации одного или нескольких окон для активизации процедур, сохраняемых в оперативной памяти.

В компьютере RISC-1 ситуация, когда требуется выполнить вызов очередной вложенной процедуры, а все окна регистрового файла заняты, разрешается с помощью логики процессора, которая формирует специальную программную ситуацию. При этом процессор инициирует программу ОС, высвобождающую одно или несколько регистровых окон, т. е. передает (trap) содержимое регистров в оперативную память. В случае применения RISC-1 в качестве машины общего назначения такое решение казалось весьма приемлемым, поскольку обычно ситуация trap возникает в одном из ста обращений к процедуре.

Но для работы в реальном времени один процент случаев оказывается недопустимо большой величиной. Действительно, если прерывание происходит в момент, когда все регистровые окна заняты, то инициируется выполнение процедуры trap и время реакции становится недетерминированным - ситуация, крайне опасная для систем реального времени. Если к тому же потребуется контекстное переключение от одной задачи к другой, то придется передать в память от одного до восьми окон в зависимости от текущего состояния прерываемой программы. А это, в свою очередь, означает, что время контекстного переключения будет изменяться в широких пределах: от 60 до 840 машинных циклов (прерывание, выполняемое внутри регистрового файла RISC-1, занимает не более трех циклов).

Именно такой разброс и является неприемлемым для систем реального времени, в которых период реакции должен быть строго детерминированным. Попытки решения данной проблемы привели к совершенствованию процедуры trap в проекте RISC-2 университета в Берклии в проекте Omega. Существенное отличие названных проектов от RISC-1 состоит во включениив архитектуру компьютера динамического механизма быстрого сохранения регистровых окон в специальной памяти.

5.1.2. Берклийская архитектура

Согласно статистике, 50—70% используемых операндов

составляют локальные переменные и параметры процедур. Их размещение в регистровом файле МП позволяет существенно снизить число обращение в память. В RISC МП с берклийской архитектурой регистры группируются в несколько банков, чтобы для каждой процедуры процессор мог назначить свой набор регистров, переключение которого осуществляется модификацией аппаратного указателя.

Рассмотрим организацию регистрового файла (табл. 1) на примере МП RISC II, разделенного на виртуальные регистровые окна емкостью 32 регистра каждое (рис.5).

Таблица 1

Параметры	RISC I	RISC II	SOAR	SPARC	Am29000
Число вирт. рег. окон	6	8	8	7	8
Число локальных рег. в вирт. окне	14	22	16	24	16
Глубина перекрытия вирт. рег. окон	4	6	8	8	0
Число глоб. регистров	18	10	8	8	64
Общее число рег. в рег. файле	78	138	72	120	192

Регистры 26—31 (верхние) содержат параметры, переданные от вызывающей процедуры. Регистры 16—25 (локальные) используются для хранения локальных скалярных переменных, 10—15 (нижние) — для хранения переменных и параметров, передаваемых вызываемой процедуре. Регистры 0—9 предназначены для хранения глобальных переменных.

Таким образом, возможно обращение каждой процедуры к 32 регистрам. Соседние регистровые банки, используемые вызывающей и вызываемой процедурами, перекрываются, так что параметры могут быть переданы процедуре без какого-либо перемещения данных. На каждое обращение к процедуре назначается новый набор регистров R10...R31. При этом нижние регистры вызывающей процедуры становятся верхмими вызываемой, поскольку они совмещены физически.

Таким образом, без перемещения информации параметры,

хранящиеся в регистрах 10—15 вызывающей процедуры, появляются в регистрах 26—31 вызываемой. Рис. 2 иллюстрирует этот подход для случая, когда процедура A вызывает процедуру B, а та, в свою очередь, вызывает C.

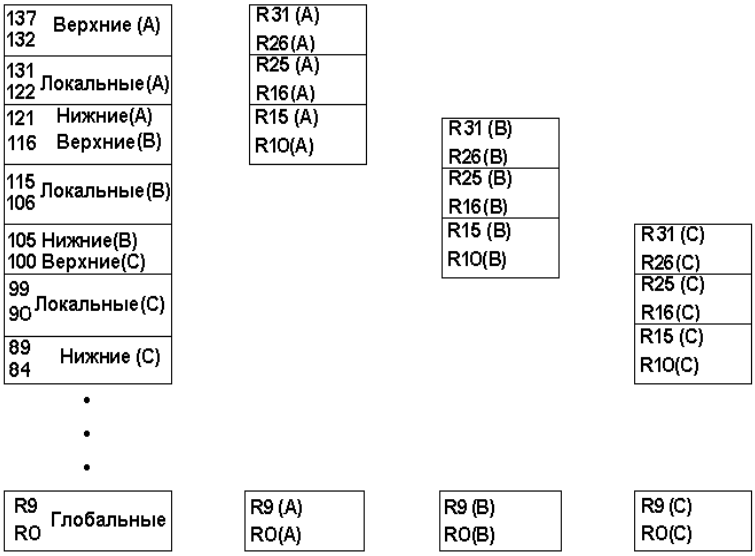


Рис. 5

Во многих программах глубина вложенности процедур превышает число регистровых окон, поэтому в МП с берклийской архитектурой освобождение регистровых банков при переполнении регистрового файла осуществляется путем передачи их содержимого в память. Если глубина вложенности выходит за пределы логических наборов регистров, содержащихся в регистровом файле, то начинается программная или аппаратная обработка прерывания. Содержимое нескольких регистров передается в память, в отдельной области которой организован стек переполнения регистров. Сигналы переполнения/недополнения перемещают указатель стека на его вершину.

Эффективность такой организации обращения к процедурам зависит от частоты появления переполнений/недополнений, в

большей степени связанной с логическими изменениями глубины стека, нежели с его абсолютной глубиной.

Исследования показали, что при восьми регистровых банках переполнения/недополнения возникают менее чем в 1% обращений к процедурам. Для того, чтобы к переменным в регистрах можно было обращаться с помощью указателей, все регистры отображаются в адресное пространство регулярной памяти.

5.1.3. Станфордская архитектура

Скорость обработки инструкций в МП с конвейерной архитектурой существенно снижается из-за возникновения конфликтных ситуаций следующего типа:

- программа осуществляет переход, для которого требуются очистка конвейера и загрузка его новыми инструкциями (зависимость по адресу);
- инструкции запрашивают информацию, которая еще не получена от обрабатываемых в конвейере инструкций (зависимость по данным);
- инструкциям в конвейере одновременно требуются обращения к одному и тому же ресурсу — шине памяти, регистру или АЛУ.

Возможны различные случаи задержки конвейера, содержащего пять ступеней обработки инструкции (рис. 6): выборку инструкции (IF), дешифрацию инструкции (ID), выборку операнда (OF), вычисление (OE) и запоминание результата (OS). В первом случае (см. рис. 6,а) выполняется инструкция безусловного перехода JMP, что задерживает конвейер до тех пор, пока не завершится ее обработка, включающая четыре этапа:

- дешифрацию инструкции JMP,
- выборку операнда из программного счетчика PC,
- модификацию содержимого PC,
- запись результата обратно в программный счетчик; задержка в этом случае длится четыре такта.

Во втором случае (рис. 6,б] инструкция INC A не может выбрать операнд A, пока предыдущая инструкция ADD B, C, A не завершит запись результата операции в регистр A; задержка длит-

ся два такта.

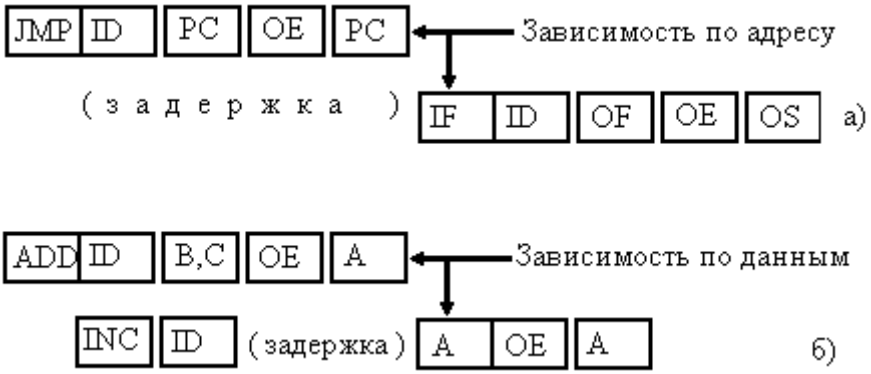


Рис. 6

Такие конфликтные ситуации - зависимость по адресу и по данным, свойственны всем МП с конвейерной обработкой инструкций. Традиционно эти проблемы решаются аппаратно за счет опережающей выборки инструкций в точке перехода или за счет блокировки конвейера в случае возникновения конфликтной ситуации.

Станфордская архитектура предусматривает устранение задержек конвейера при помощи оптимизирующего компилятора, переупорядочивающего инструкции так, чтобы они не зависели одна от другой при обработке их в конвейере. При этом вводится инструкция задержанного перехода (табл. 2), которая применяется также и в некоторых RISC МП с берклийской архитектурой. Задержанный переход выполняется так, что инструкция, следующая за инструкцией перехода, выполняется до передачи управления в точку перехода. В это время процессор имеет возможность выбрать инструкцию по адресу перехода и загрузить ее в конвейер.

Традиционные МП реализуют этот фрагмент программы как обычный переход - инструкции выполняются в последовательности: 100, 101,102, 105 и т.д. Чтобы получить такой эффект в RISC МП, необходимо ввести в задержанный переход инструкцию NOP, не выполняющую операций. Тогда последовательность станет такой: 100, 101, 102, 103, 106 и т.д.; обрабатывается она опти-

мизирующим компилятором, который по возможности переупорядочивает ее так чтобы максимально использовать цикл после задержанного перехода. Последовательность исполнения инструкций принимает вид: 100, 101, 102, 105 и т.д. Поскольку инструкция, следующая за инструкцией перехода ,выполняется всегда и переход по адресу 101 не зависит от выполнения инструкции ADD по адресу 102, последовательность эта эквивалентна исходному фрагменту программы.

Таблица 2

Адрес	Обычный переход	Задержанный переход	Оптимиз. задержанный переход
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JMP 105
102	JMP 105	JMP 106	ADD 1,A
103	ADD A,B	NOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

Концепция задержанного перехода применяется в ряде случаев для выполнения инструкций обращения в память за один такт. Непосредственный способ выполнения этих инструкций требует двух тактов: один необходим для вычисления адреса, второй — собственно для обращения в память. Модифицированные инструкции обращения в память называются задержанными загрузками и предполагают некоторые изменения аппаратуры: память и регистры МП должны быть оснащены дополнительными портами. Зависимость по данным, которая может возникнуть при выполнении задержанной загрузки, устраняется при помощи оптимизирующего компилятора.

Для RISC МП со станфордской архитектурой характерно небольшое число регистров общего назначения, используемых для хранения локальных переменных и параметров проце-

дур. Распределение регистров между переменными выполняется оптимизирующим компилятором.

5.1.4. Применение RISC - архитектуры в 32-разрядных МП

Каждой из рассмотренных разновидностей архитектуры присущи как достоинства, так и недостатки, поэтому обе они получили примерно одинаковое развитие и распространение.

Первые МП с архитектурой RISC (RISC I и RISC II) были разработаны и изготовлены в начале 80-х годов в Калифорнийском университете (г. Беркли). Они имели небольшой набор инструкций и простые режимы адресации. Все арифметические и логические инструкции выполнялись над регистровыми операндами, и только две — LOAD и STORE — использовались для обращения в память. На площади кристалла, освободившейся от схемы дешифрации сложных команд и формирователя последовательности микрокоманд, присущих традиционной архитектуре МП, размещен большой регистровый файл. Организация его в виде перекрывающихся регистровых окон позволяла каждой процедуре назначать новый набор регистров и передавать данные от одной процедуры к другой простым изменением аппаратного указателя. Регистровые окна, впервые примененные в берклийских МП обеспечивают эффективную поддержку вложенных процедур, сокращают число обращений в память и значительно повышают производительность МП.

В 1983 г. в Станфордском университете был разработан и изготовлен МП MIPS (Microprocessor without Interlocked Pipeline Stages - микропроцессор без блокировок конвейера) . Целью проекта MIPS было создание МП, свободного от задержек конвейера, связанных: с выполнением переходов, зависимостью инструкций по данным и одновременным обращением нескольких инструкций, обрабатываемых в конвейере, к одному ресурсу. Благодаря разработке и применению оптимизирующего компилятора в МП MIPS удалось преодолеть недостатки берклийсиой архитектуры, обусловленные, прежде всего, необходимостью использования большого регистрового файла, занимающего до 60% площади кристалла.

Архитектура МП SOAR (Smalltalk On A RISC) является развитием берклийской ветви RISC. Он предназначен для поддержки языка высокого уровня Smalltalk-80 и функционирования в составе рабочей станции Sun. SOAR поддерживает два типа данных: 31-разрядные переменные со знаком и 28-разрядные указатели. Каждое слово данных содержит признак типа (тег), который проверяется одновременно с выполнением арифметических операций. Если оба операнда оказываются целыми, то через один такт получается правильный результат. В противном случае SOAR обращается к программам обработки указателей. Другой особенностью его является использование инструкций групповой загрузки—хранения, позволяющих передавать содержимое восьми регистров за девять тактов вместо шестнадцати (один такт тратится на выборку инструкции и восемь — на обращения в память).

МП ARM (Acorn RISC Machine) разработан специально для персонального компьютера модели В фирмы Acorn Computers и воплощает основные принципы станфордской архитектуры. Он обладает небольшим набором аппаратно-реализованных инструкций, имеющих одинаковый 32-разрядный формат. Для обеспечения быстрой реакции на прерывания разработчики ARM исключили из его набора команд длинные операции, которые невозможно прервать. Как и МП SOAR, ARM поддерживает инструкции групповой загрузки—хранения, позволяющие повысить скорость обмена между МП и памятью. В отличие от других RISC МП, в ARM не используются задержанные переходы. Для устранения задержек конвейера при выполнении переходов в каждую инструкцию введено 4-разрядное поле кода условия; выполняется оно только в том случае, если установлен код соответствующего условия.

МП ROMP (Research/Office division Microprocessor) - микропроцессор отделения по автоматизации научных и Учрежденческих работ) корпорации IBM. Он предназначен для работы в качестве центрального процессора в персональном компьютере RT PC (RISC Technology Personal Computer), реализует станфордскую архитектуру и оснащен развитым оптимизирующим компилятором.

Особенностью ROMP является длинное 40 - разрядное адресное слово и соответствующее ему адресное пространство 1Т

байт.

При разработке ROMP большое внимание уделялось повышению скорости обмена между процессором и памятью. Для этого используется аппаратно управляемый буфер преобразованных адресов, применявшийся в ранних моделях IBM. Он выполняет эффективное преобразование виртуальных адресов в физические. Кроме того, каждое слово данных в ROMP снабжается тегом, по которому определяется получатель данных, поэтому любой элемент данных может перебраться, как только он будет готов.

Центральный процессорный элемент (ЦПЭ) R2000 фирмы Mips Computer Systems составляет основу для построения супермини компьютеров этой фирмы. Он имеет станфордскую архитектуру, работает в программной среде ОС Umips, совместимой с ОС Unix, языков высокого уровня Си, Фортран-77 и Паскаль. Для управления внешней кэш-памятью используется буфер преобразованных адресов. R2000 работает в составе трехкристального набора, в который входят также сопроцессор для операций с плавающей точкой и буфер записи, предназначенный для согласования скоростей работы ЦПЭ и динамической памяти.

ЦПЭ MD 484 фирмы McDonnell Douglas изготавливается по GaAs-технологии и имеет расчетную производительность 100 млн. инструкций в 1 с. Благодаря станфордской архитектуре, MD 484 содержит небольшое число компонентов (23,2 тыс. транзисторов и 10,4 тыс. резисторов) и поддерживает набор инструкций МП MIPS. Он имеет развитую систему прерываний, включающую прерывания из-за неправильной адресации, внутреннего переполнения и системных обращений. Для передачи инструкций и данных используются три двунаправленные 32-разрядные шины, каждая линия которых заряжается с помощью большого n-канального транзистора. Высокая производительность ЦПЭ обеспечивается за счет сопроцессора для операций с плавающей точкой и устройства управления памятью, входящих в МП набор.

МП SPARC (Scalable Processor ARChitecture) фирмы Sun Microsystems воплощает основные принципы берклийской архитектуры: он содержит большой регистровый файл емкостью 120 32-разрядных регистров и использует задержанные переходы. МП

реализован на вентильной матрице фирмы Fujitsu, выполненной по 1,5 мкм КМОП-технологии и содержащей 20 тыс. вентилей. МП SPARC предназначен для работы в програмной среде ОС Unix в составе рабочей станции Sun-4.

МП Am29000 фирмы AMD отличается наивысшей производительностью среди серийных RISC МП и реализует усовершенствованную версию берклийской архитектуры. В него входит регистровый файл емкостью 192 регистра, разделенный на банки по 16 регистров; использует он задержанные переходы для устранения задержек конвейера, вызванных зависимостью по адресу, и выполняет команды обращения в память за один такт благодаря применению концепции задержанной загрузки. Кроме того, для ускорения обмена между МП и внешней памятью в Am29000 введен буфер преобразованных адресов.

Трехкристальный МП набор Clipper C100 корпорации Fairchild позволяет реализовать суперЭВМ на одной плате. Архитектура Clipper C100 имеет некоторые черты RISC, например аппаратную реализацию большинства инструкций, однако ее нельзя отнести ни к одному из рассмотренных выше классов. Этот МП способен выполнять все арифметические операции с плавающей точкой по стандарту IEEE 754, он поддерживает десять типов данных и девять режимов адресации, а его система прерываний включает 256 векторных прерываний и 128 системных обращений.

Набор инструкций Clipper C100 содержит 101 аппаратно-реализованную инструкцию и 67 макроинструкций, выполняющих преобразование чисел из формата с фиксированной точкой в формат с плавающей точкой и обратно, обработку символьных строк, хранение - восстановление регистров, обработку прерываний и другие операции. Для поддержки макроинструкций используется микропрограммное ПЗУ емкостью 1К 48-разрядных слов. Вычислительный модуль Clipper C100 работает в программной среде ОС Unix и языков высокого уровня Си, Фортрая и Паскаль.

МП система MC88000 фирмы Motorola включает процессор MC88100 и два кристалла памяти MC88200.

MC88100 содержит регистровый файл небольшой емкости, блоки обработки чисел в формате с фиксированной и плавающей точкой, а также до шести заказных блоков специальных функций.

МП имеет четыре порта ввода - вывода и поэтому может быть использован для построения мультипроцессорных систем.

5.1.5. Особенности интеграции элементов RISC-архитектуры в процессорах серии x86

Организация первых моделей процессоров - i8086/8088 - была направлена, в частности, на сокращение объёма программ, отличавшихся малой оперативной памятью. Расширение спектра операций, реализуемых системой команд, позволило уменьшить размер программ, трудоёмкость их написания и отладки, но повысило трудоёмкость их разработки.

Последнее проявилось в удлинении сроков разработки CISC-процессоров и проявлении различных ошибок в их работе. Кроме того, нерегулярность потока команд ограничила развитие топологии временным параллелизмом обработки инструкций на конвейере «выборка команды, дешифрация команды, выборка данных, вычисление - запись результата».

Эти недостатки обусловили необходимость разработки альтернативной архитектуры, нацеленной, прежде всего, на снижение нерегулярности потока команд уменьшением их общего количества. Это было реализовано в RISC-процессорах.

Сокращение нерегулярности потока команд позволило обогатить топологию RISC-процессоров пространственным параллелизмом, специализированными аппаратными АЛУ, независимыми кэш данных и команд, раздельными шинами ввода-вывода. Последние, в частности, увеличили длину конвейеров команд. Всё это повысило и производительность - увеличением числа операций, выполняемых за один такт, и быстродействие - сокращением пути транзактов - RISC-процессоров. При этом срок разработки данных чипов свидетельствует о том, что её трудоёмкость меньше, чем в случае CISC-процессоров.

На мировых рынках CISC-процессоры представлены, в основном, клонами процессоров Intel серии x86, производимыми AMD, Cyrix, а RISC -чипами Alpha, PowerPC, SPARC. Уступая во многом последним, процессоры x86 сохранили лидерство на рынке персональных систем лишь благодаря совместимостис про-

граммным обеспечением младших моделей. В свою очередь, достоинства RISC-процессоров укрепили их позиции на рынке высокопроизводительных машин.

Несмотря на формальное разделение «сфер влияния», между представителями этих архитектур в начале 90-х годов началась острая конкуренция за улучшение характеристик. В первую очередь, производительности и её отношения к трудоёмкости разработки процессоров.

Первыми к этому пришли разработчики Intel, реализовавшие в i80486 пространственный параллелизм вычислений с фиксированной и плавающей запятой. Поддержка каждого АЛУ своей шиной данных/команд и регистровым блоком повысила производительность i80486 одновременным выполнением указанных команд. Кроме того, интеграция кэш-памяти и очереди команд позволила поднять частоту ядра процессора в 2-3 раза в сравнении с системной шиной. Однако совместное размещение данных и команд ограничило эффективность кэш необходимостью его полной перезагрузки после выполнения команд переходов.

Для устранения недостатка в Pentium реализованы раздельные кэш для команд и данных, позволяющие после переходов перезагружать лишь команды- такое решение называется Гарвардской архитектурой, а также предсказание переходов, снижающее частоту перезагрузок. Последнее достигается предварительной загрузкой в кэш команд с обоих разветвлений. Введение второго целочисленного тракта, состоящего из АЛУ, адресного блока, шин данных/команд, и работающего на общий блок регистров, повысило производительность поддержкой параллельной обработки целочисленных данных. Развитием данной тенденции стало обогащение Pentium MMX мультимедийным трактом, образованным АЛУ, шинами данных/команд и регистровым файлом.

При этом в случае выборки двух целочисленных команд, зависящих по данным, каждая из них выполняется последовательно, что снижает эффективность работы процессора. Частично поправило ситуацию создание оптимизирующих рекомпиляторов, например, Pen-Opt фирмы Intel, разделяющих по возможности такие команды.

Реализация описанного управления обработкой команд

CISC-формата вызвала дополнительный рост трудоёмкости разработки Pentium в сравнении с i8086/i80486, что привело не только к увеличению её реального срока на 27% в сравнении с ожидаемым, но и к проявлению ошибок в первых моделях данного процессора

Учтя это, компания Intel реализовала в Pentium Pro RISC-подобную организацию вычислений. Интерпретация команд 80×86 внутренними - RISC86 - инструкциями VLIW-формата помимо снижения нерегулярности их потока, обеспечила синхронную загрузку четырёх операционных - по два с плавающей и фиксированной запятой - АЛУ этого чипа. Термин VLIW расшифровывается как "очень длинное командное слово" (Very Long Instruction Word). Инструкции этого формата содержат команды для всех параллельных АЛУ.

Обогащение управления обработкой предвыборкой данных и команд, предполагаемых к обработке в ближайшие 20 тактов, повысило регулярность загрузки вычислительных трактов. В свою очередь, осуществление предвыборки из интегрированного на кристалле кэш второго уровня, обслуживаемого раздельными шинами «интерфейс-кэш» и «кэш-АЛУ» и работающего на частоте АЛУ, повысило быстродействие подготовки команд в сравнении с внешними кэш. Дополнительное повышение производительности Pentium Pro обеспечило увеличение длины команд до 11 ступеней введением ступеней трансляции и предвыборки. Кроме того, интеграция кэш второго уровня позволила умножать частоту ядра в 5-6 раз.

В архитектуре P6 RISC - решения впервые в семействе 80×86 перестали быть лишь дополнением исконных CISC - средств повышения производительности - роста разрядности, отложенной записи шины и других. Поэтому частица PRO в названии первого процессора этой серии обозначает «полноценная RISC-архитектура» (Precision RISC Organization).

Топологические новинки Pentium II - интеграция тракта MMX, мультипроцессорный интерфейс Xeon, вынесение кэш второго уровня на кристалл в корпусе чипа, как и полное устранение кэш второго уровня в Celeron, не имеют в данном случае качественной роли и направлены на оптимизацию отношения характери-

стик этих процессоров, к их цене.

Снижение трудоёмкости и длительности разработки аппаратно – программных реализаций алгоритмов работы Pentium Pro, позволило достигнуть роста характеристик сочетанием преимуществ RISC&CISC архитектур.

Сказанное иллюстрирует и организация современных RISC-процессоров. Их отличает, в данном случае, развитие систем команд с целью сохранения иерархической совместимости и снижения трудоёмкости разработки программ. Это сближает технологии обработки команд процессорами упомянутых архитектур. Например, SuperSparc взяли от последних моделей 80×86 предсказание переходов и предварительную интерпретацию кода.

Таким образом, развиваясь, каждая из рассмотренных архитектур, «отказавшись» от своих черт - CISC от скалярности вычислений, RISC от «простоты» системы команд, приобрела лучшие характеристики её представителей. Это подтверждает и процессор Merced, разработанный Intel и Hewlett Packard.

Имеющиеся сведения позволяют предположить, что его архитектура продолжит тенденции Pentium Pro по оптимизации обработки внутренних VLIW – подобных команд реализацией эффективных архитектурных решений при одновременной оптимизации преобразования «внешних» инструкций. Особо отмечаются намерения создания двух вариантов этого чипа, различающихся лишь множеством этих инструкций. Первый совместим с CISC-семейством 80×86, второй - с RISC-процессорами Alpha.

Merced в известной степени прекратил соперничество CISC и RISC, в ходе которого представители данных архитектур улучшили свои характеристики. Это позволило предположить, что дальнейшее развитие массовых процессоров пройдёт по пути развития топологических и микропрограммных решений вычислительного ядра RISC - организации при одновременном повышении возможностей CISC - подобной системы команд.

5.2. МП с традиционной архитектурой

5.2.1. Intel Pentium 4

Одной из особенностей архитектуры процессора Pentium 4 является гарвардская внутренняя структура, реализуемая путем разделения потоков команд и данных, поступающих от системной шины через блок внешнего интерфейса и размещённую на кристалле процессора общую кэш-память 2-го уровня L2 (рис. 7).

Блок внешнего интерфейса реализует обмен процессора с системной шиной, к которой подключается память, контроллеры ввода / вывода и другие активные устройства системы. Обмен по системной шине осуществляется с помощью 64-разрядной двунаправленной шины данных, 41-разрядной шины адреса (33 адресных линии и 8 линий выбора байтов), обеспечивающей адресацию до 64 Гб внешней памяти.

Архитектура МП является суперскалярной, что обеспечивает одновременное выполнение нескольких команд в параллельно работающих исполнительных устройствах. Суперскалярность архитектуры реализуется путем организации исполнительного ядра процессора в виде ряда параллельно работающих блоков. Арифметико-логические блоки ALU производят обработку целочисленных операндов, которые поступают из заданных регистров блока регистров замещения (БРЗ). В эти же регистры заносится и результат операции. При этом проверяются также условия ветвления для команд условных переходов и выдаются сигналы перезагрузки конвейера команд в случае неправильно предсказанного ветвления. Исполнительное ядро работает с повышенной скоростью выполнения операций.

Адреса операндов, выбираемых из памяти, вычисляются блоком формирования адреса (БФА), который реализует интерфейс с кэш-памятью данных L1 ёмкостью 8 Кбайт. В соответствии с заданными в декодированных командах способами адресации формируются 48 адресов для загрузки операндов из памяти в регистр БРЗ и 24 адреса для записи из регистра в память. При этом БФА формирует адреса операндов для команд, которые ещё не поступили на выполнение.

При обращении к памяти БФА одновременно выдаёт адреса двух операндов: один для загрузки операнда в заданный регистр БРЗ, второй - для пересылки результата из БРЗ в память. Таким

образом реализуется процедура предварительного чтения данных для последующей их обработки в исполнительных блоках, которая называется спекулятивной выборкой.

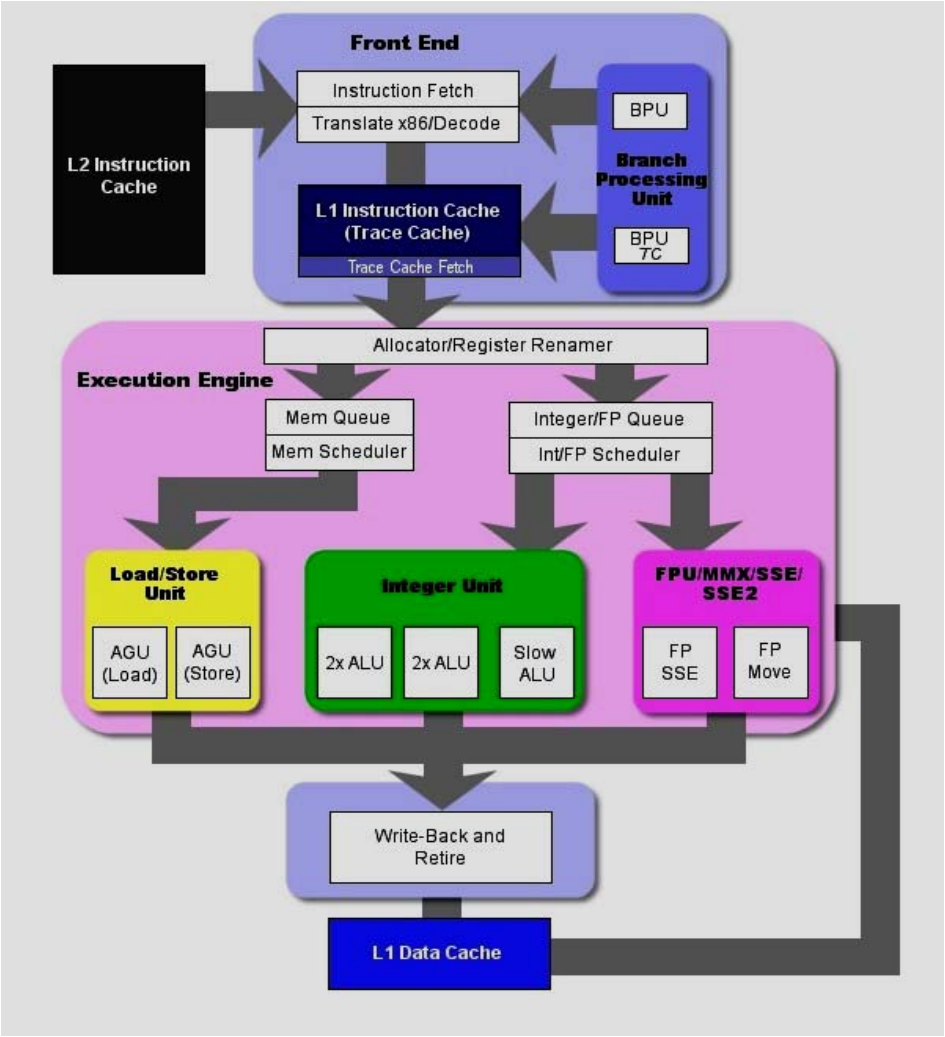


Рис. 7

Аналогичным образом организуется параллельная работа блоков SSE, FPU, MMX, которые используют отдельный набор регистров и блок формирования адресов операндов.

Хотя конвейер Pentium 4 является намного более длинным, он выполняет те же функции что и конвейеры других процессоров.

Из-за сложности архитектуры на рис. 7 не изображена каждая из ступеней конвейера. Тем не менее, связанные ступени сгруппированы воедино, чтобы можно было представить всю схему процессора и схему потока команд.

Особое внимание стоит уделить тому, что кэш-память L1 разделена и кэш инструкций находится фактически на препроцессоре. Он называется отслеживающим кэшем (Trace Cache) и является одной из важных инноваций в Pentium 4. Эта кэш-память оказывает сильное влияние и на конвейер, и на основной поток инструкций.

Если рассмотреть процессоры Pentium III или Athlon, то можно отметить, что инструкции поступают в их декодер из кэш-памяти инструкций, в декодере они разбиваются на меньшие части, более единообразные, с которыми проще работать, - микрокоманды. Фактически, эти инструкции применяются при внеочередном выполнении команд, исполнительный модуль выполняет их планирование, исполнение и сброс. Такое разбиение случается всякий раз, когда процессор выполняет инструкцию, поэтому на эту операцию в начале конвейера отводится несколько ступеней (на рис. 8 и 9 эти ступени объединены, на самом же деле выборка инструкций занимает несколько ступеней, транслирование - несколько ступеней, декодирование - несколько, и т.д.).

Если взять фрагмент кода, повторно выполняющийся всего несколько раз по ходу программы, то для него такая потеря нескольких тактов мало что означает. Но для фрагмента кода, где инструкции исполняются тысячи и тысячи раз (например, в цикле в мультимедийном приложении, выполняющем несколько операций над большим файлом), количество повторных трансляций и декодирований может отнимать ощутимые ресурсы. Для того, чтобы избежать таких циклов, процессор Pentium 4 не осуществляет повторного разбиения 80x86-инструкций на микрокоманды при их выполнении.

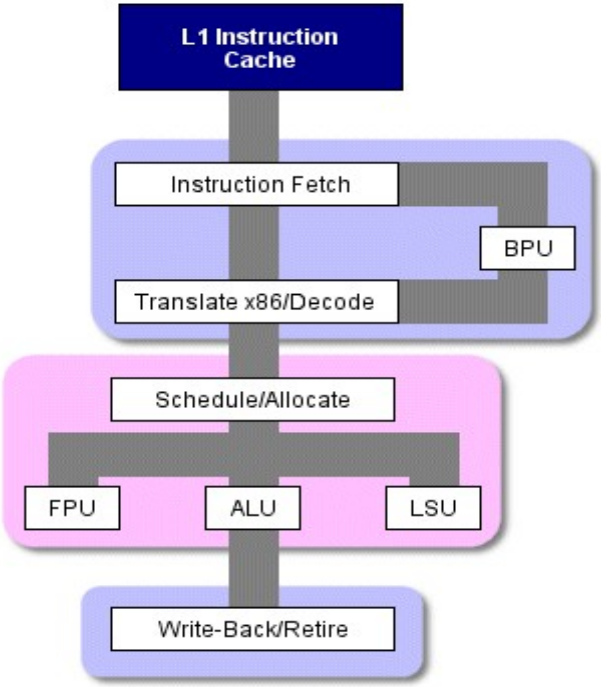


Рис. 8

Кэш инструкций в Pentium 4 принимает транслированные и декодированные микрокоманды, готовые к передаче на внеочередное выполнение, и формирует из них мини-программы, назовем их трейс-последовательностями (traces). Именно эти мини-программы (а не 80x86-код, созданный компилятором) и выполняет Pentium 4 в том случае, если происходит попадание в кэш-память L1 (процент попадания - 90%). До тех пор, пока требуемый код находится в кэш-памяти L1. Схема выполнения представлена на рис. 9.

По мере выполнения препроцессором накопленных трейс-последовательностей, отслеживающий кэш посылает до трех микрокоманд за такт напрямую на внеочередной модуль выполнения, ведь процессору уже не нужно проводить команды через логику трансляции или декодирования. И только в случае промаха кэш-

памяти L1 препроцессор нарушит этот порядок и начнёт выбирать и декодировать инструкции из кэш-памяти L2.

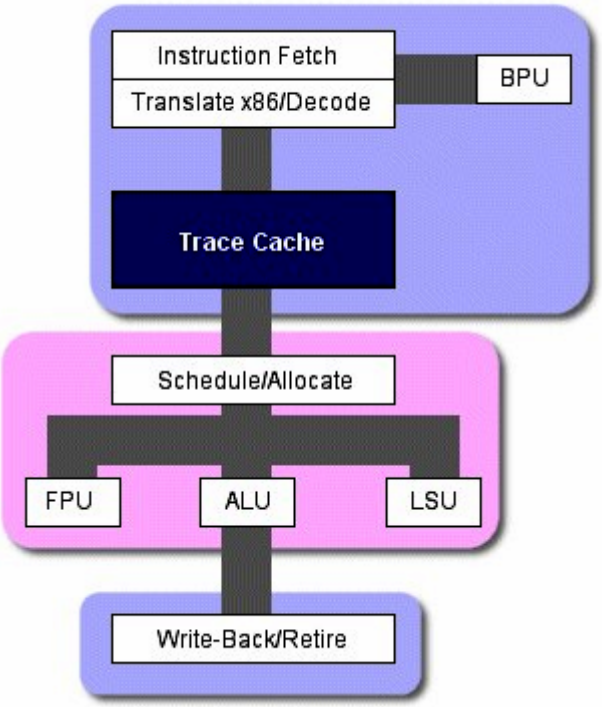


Рис. 9

В этом случае к началу основного конвейера добавляется ещё восемь ступеней. Как видно, кэш с отслеживанием может избавить от довольно большого количества тактов при выполнении программы.

Отслеживающий кэш может работать в двух режимах. Исполнительный режим (execute mode) был только что рассмотрен. Здесь отслеживающий кэш снабжает логику выполнения инструкциями. В этом режиме он обычно и работает. Когда наступает промах кэш-памяти L1, он переходит в так называемый режим построения отслеживаемых сегментов (trace segment build mode) В этом режиме препроцессор выбирает 80x86-инструкции из кэш-памяти L2, транслирует их в микрокоманды, создает отслеживае-

мый сегмент, который затем перемещается в отслеживающий кэш и далее выполняется.

Из рис. 9 видно, что когда работает отслеживающий кэш, устройство предсказания ветвлений не участвует в работе, равно как не работают и ступени выборки/декодирования инструкций. На самом деле отслеживаемый сегмент - это нечто большее, чем просто фрагмент транслированного и декодированного кода 80x86, выданного компилятором и полученного препроцессором из кэш-памяти L2. В действительности, при создании мини-программы отслеживающий кэш все же использует предсказание ветвлений. Он может добавить в мини-программу (где содержится предназначенный для выполнения код) код, который только предполагается к выполнению при предсказании ветвления. Поэтому если имеется фрагмент x86-кода с ветвлением, отслеживающий кэш построит трейс-последовательность из инструкций до ветвления, включая саму инструкцию ветвления. Затем он продолжит спекулятивно строить мини-программу вдоль предсказанной ветви (рис. 10).

Такое спекулятивное выполнение даёт отслеживающему кэшу два больших преимущества по сравнению с обычным кэшем инструкций. Во-первых, в стандартном процессоре для работы устройства предсказания ветвлений требуется некоторое время. При обработке условной инструкции ветвления устройство ветвления (BPU) должно определить, какую из ветвей нужно спекулятивно выполнять, найти адрес кода после ветвления и т.д. Весь этот процесс добавляет, по крайней мере, еще один такт задержки для каждой условной инструкции ветвления. Такая задержка часто не может быть заполнена выполнением другого кода, что приводит к появлению нежелательного пузырька (пустая ячейка в конвейере - pipeline bubble). В случае же использования отслеживающего кэша, код после ветвления уже готов к выполнению сразу же после инструкции ветвления, поэтому показанных задержек не возникает.

Второе преимущество также связано с возможностью хранения спекулятивных ветвей. Когда стандартный кэш инструкций L1 считывает строку кэш-памяти, он прекращает считывание при попадании на инструкцию ветвления, поэтому оставшаяся часть

строки остается пустой. Если инструкция ветвления находится вначале строки кэш-памяти L1, то в считанной строчке будет находиться только одна эта инструкция. При использовании отслеживающего кэша считанные строчки могут содержать как инструкции ветвления, так и спекулятивный код после них. Таким образом, в 6-командных строчках не возникает потерянного места.

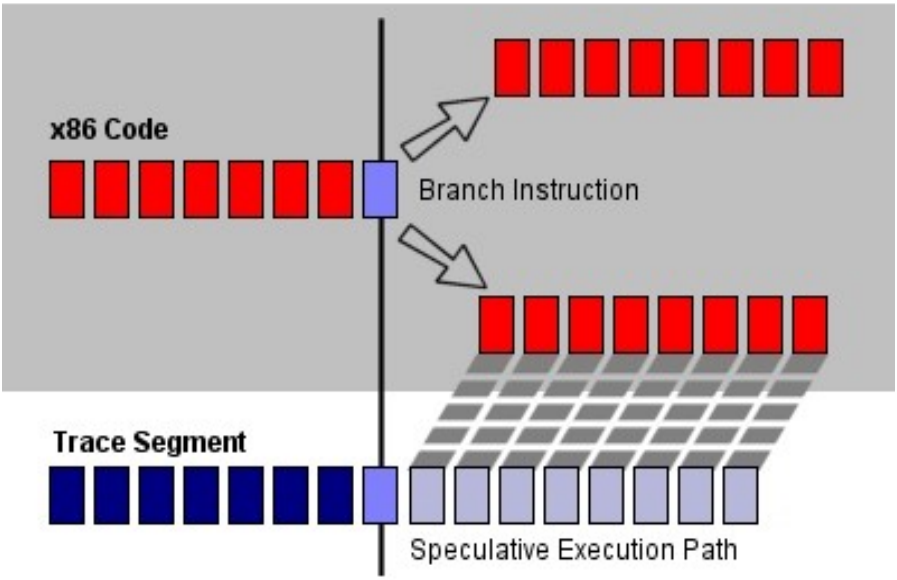


Рис.10.

Кстати, большинство компиляторов сталкиваются именно с описанными двумя проблемами: с задержками в инструкциях ветвления и с неполными строками из кэш-памяти. Как видно, отслеживающий кэш по-своему позволяет решать эти проблемы. Если программы оптимизированы с учетом этих возможностей, то они будут быстрее выполняться.

Ещё один интересный эффект, производимый отслеживающим кэшем на препроцессор Pentium 4 заключается в том, что пропускная способность ступеней транслирования и декодирования 80x86-команд не зависит от пропускной способности ступени диспетчеризации. Если вспомнить процессор AMD K7, то он рас-

ходует множество транзисторов на усиленный блок декодирования 80x86-макрокоманд, что позволяет за цикл декодировать достаточно много громоздких 80x86-инструкций в микрокоманды для загрузки исполнительного модуля. В случае же с Pentium 4 наличие отслеживающего кэша означает, что большая часть кода забирается из него уже в виде готовых микрокоманд, так что здесь отпадает надобность в трансляторах и декодерах с высокой пропускной способностью.

Процессор начинает декодирование только лишь в случае промаха кэш-памяти L1. Поэтому он разработан таким образом, чтобы декодировать только одну 80x86-инструкцию за такт. Это составляет всего треть от максимальной теоретической пропускной способности декодера Athlon, но отслеживающий кэш в Pentium 4 позволяет ему достичь или даже обойти производительность Athlon (2,5 диспетчеризации за такт).

Стоит обратить внимание и на то, как отслеживающий кэш обращается с очень длинными 80x86-инструкциями из нескольких циклов. Большинство 80x86-инструкций декодируются примерно в две или три микрокоманды. Но встречаются и такие инструкции, которые декодируются в сотни микрокоманд, например, инструкции по строковой обработке. Как и в Athlon, в Pentium 4 существует специальное ПЗУ микрокода, которое обрабатывает эти громоздкие инструкции, что позволяет разгрузить аппаратный декодер для работы только с небольшими, быстрыми инструкциями.

Каждый раз, когда встречается громоздкая инструкция, ПЗУ находит готовую последовательность микрокоманд и выдаёт их дальше в по очереди. Чтобы не засорять отслеживающий кэш этими длинными последовательностями микрокоманд, разработчики поступили следующим образом: как только при создании отслеживаемого сегмента отслеживающий кэш встречает такую большую 80x86-инструкцию, вместо того, чтобы разбивать её на последовательность микрокоманд, он вставляет в отслеживаемый сегмент метку (tag), которая указывает на место в ПЗУ, содержащее последовательность микрокоманд данной инструкции. Позднее, в режиме выполнения, когда отслеживающий кэш будет передавать поток инструкций на ступень выполнения, при попадании на такую метку он временно приостановит работу и на время пе-

редаст управление потоком инструкций ПЗУ микрокода. Здесь уже ПЗУ будет выдавать в поток инструкций требуемую последовательность микрокоманд (как определено меткой). После этого, оно возвратит управление обратно, и отслеживающий кэш продолжит передавать инструкции. Исполнительному модулю безразлично, откуда поступает поток инструкций (из отслеживающего кэша или из ПЗУ). Для него все это выглядит как непрерывный поток команд.

Единственным недостатком отслеживающего кэша является его размер: он слишком мал. Точные размеры его неизвестны. Он может содержать до 12 тысяч микрокоманд. Intel уверяет, что это примерно эквивалентно обычному кэшу команд на 16-18 тысяч инструкций. Но так как отслеживающий кэш работает совсем иначе, нежели стандартный кэш инструкций L1, то для того, чтобы оценить, как его размер влияет на производительность всей системы, нельзя обойтись простым сравнением его размера с кэш-памятью другого процессора.

Общая архитектура процессора определяет комплекс средств, предоставляемых пользователю для решения различных задач. Эта архитектура задаёт базовую систему команд процессора и реализуемых способов адресации, набор программно-доступных регистров (регистровая модель), возможные режимы работы процессора и обращения к памяти и внешним устройствам (организация памяти и реализация обмена по системной шине), средства обработки прерываний и исключений.

В процессоре Pentium 4 реализуется архитектура IA-32 (Intel Architecture-32), общая для всех 32-разрядных микропроцессоров Intel, начиная с 80386. Модели Pentium II Xeon и Pentium III Xeon ориентированы на работу в высокопроизводительных мультипроцессорных системах (серверах, рабочих станциях). Для этих же приложений в 2001 году выпущена модификация процессора Pentium 4 с поддержкой мультипроцессорного режима работы (на ядре Foster).

В процессе развития IA-32 производилось расширение возможностей обработки данных, представленных в различных форматах (рис. 11). Процессоры 80386 выполняли обработку только целочисленных операндов. Для обработки чисел с плавающей точ-

кой использовался внешний сопроцессор 80387, подключаемый к микропроцессору. В состав процессоров 80486 и последующих моделей Pentium введён специальный блок FPU (Floating-Point Unit), выполняющий операции над числами с "плавающей точкой". В процессорах Pentium MMX была впервые реализована групповая обработка нескольких целочисленных операндов разрядностью 1, 2, 4 или 8 байт с помощью одной команды. Такая обработка обеспечивается введением дополнительного блока MMX.

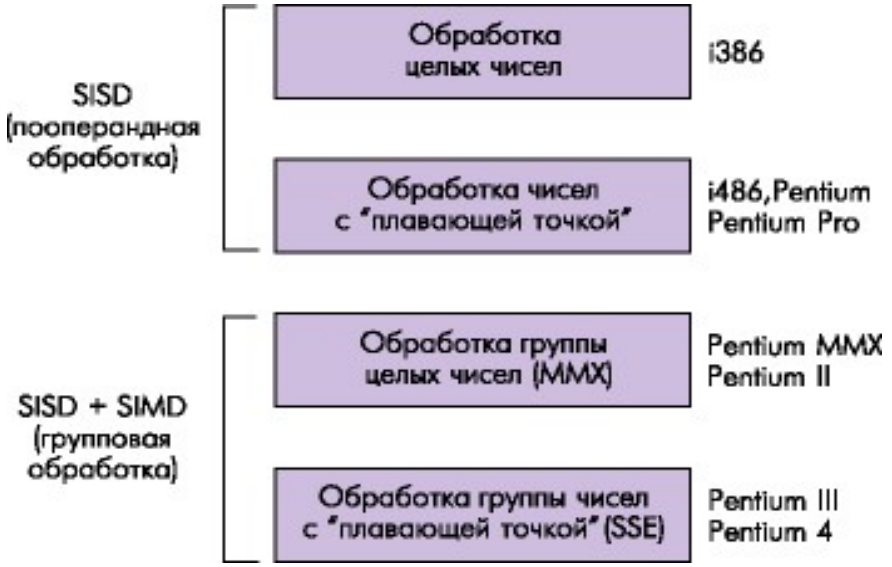


Рис. 11

Название блока отражает его направленность на обработку видео- и аудиоданных, когда одновременное выполнение одной операции над несколькими операндами позволяет существенно повысить скорость обработки изображений и звуковых сигналов. Начиная с модели Pentium III, в процессоры вводится блок SSE (Streaming SIMD Extension) для групповой обработки чисел с плавающей точкой.

Таким образом, если первые модели процессоров Pentium выполняли только пооперандную обработку данных по принципу

"одна команда - одни данные" (SISD), то, начиная с процессора Pentium MMX, реализуется также их групповая обработка по принципу "одна команда - много данных" (SIMD).

Соответственно, расширяется и набор регистров процессора, используемых для промежуточного хранения данных (рис. 12). Кроме 32-разрядных регистров для хранения целочисленных операндов, процессоры Pentium содержат 80-разрядные регистры, которые обслуживают блоки FPU и MMX. При работе FPU регистры ST0-ST7 образуют кольцевой стек, в котором хранятся числа с плавающей точкой, представленные в формате с расширенной точностью (80 разрядов). При реализации MMX-операций они используются как 64-разрядные регистры MM0-MM7, где могут храниться несколько операндов (8 8-разрядных, 4 16-разрядных, 2 32-разрядных или один 64-разрядный), над которыми одновременно выполняется поступившая в процессор команда (арифметическая, логическая, сдвиг и т.д.).

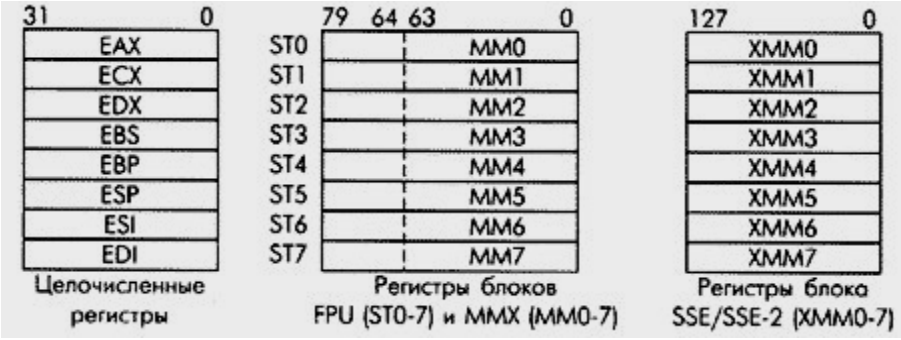


Рис.12

Блок SSE2, введённый в состав процессора Pentium 4, значительно расширяет возможности обработки нескольких операндов по принципу SIMD, по сравнению с блоком SSE в модели Pentium III. Этот блок реализует 144 новые команды, обеспечивающих одновременное выполнение операций над несколькими операндами, которые располагаются в памяти и в 128-разрядных регистрах XMM0-XMM7. В регистрах могут храниться и одновре-

менно обрабатываться 2 числа с плавающей точкой в формате двойной точности (64 разряда) или 4 числа в формате одинарной точности (32 разряда). Этот блок может также одновременно обрабатывать целочисленные операнды: 16 8-разрядных, 8 16-разрядных, 4 32-разрядных или 2 64-разрядных. В результате производительность процессора Pentium 4 при выполнении таких операций оказывается вдвое выше, чем Pentium III.

Операции SSE2 позволяют существенно повысить эффективность процессора при реализации трехмерной графики и интернет-приложений, обеспечении сжатия и кодирования аудио- и видео-данных. Что касается базового набора команд и используемых способов адресации операндов, то они практически полностью совпадают с набором команд и способов адресации в предыдущих моделях Pentium. Процессор обеспечивает реальный и защищённый режимы работы, реализует сегментную и страничную организации памяти. Таким образом, пользователь имеет дело с хорошо знакомым набором регистров и способов адресации, может работать с базовой системой команд и известными вариантами реализации прерываний и исключений, которые характерны для всех моделей семейства Pentium.

Pentium 4 является первым IA-32 (32-bit Intel Architecture) процессором, использующим не P6 архитектуру. Сегодня эта архитектура получает название с использованием терминологии - Intel NetBurst.

Первой особенностью NetBurst архитектуры является то, что Intel называет гиперконвейерной технологией, что является несколько причудливым термином для 20-ти шагового конвейера Pentium 4. Эти 20 шагов или стадий – вдвое длиннее P6 конвейера, которым оснащен Pentium III и в четыре раза длиннее, чем P5 конвейер. Как известно, более длинный конвейер имеет свои “за” и “против”.

20-ти шаговый конвейер Pentium 4, позволяет ему работать на более высокой тактовой частоте. По этой причине Pentium 4 будет дебютировать на скорости 1.4 ГГц и выше. Но 20-ти шаговый конвейер Pentium 4 приводит к уменьшению значения IPC (инструкций за такт).

Имеется множество путей восполнения низкого IPC. Один из них, наиболее очевидный, заключается в простом увеличении тактовой частоты, что Intel и сделала. Нет сомнения, что на любом современном эталонном тесте 1ГГц Pentium III по-сравнению с гипотетическим 1ГГц Pentium 4 показал бы значительно больший результат, потому что выполняет больше инструкций за такт, чем Pentium 4.

В современных процессорах предусмотрены средства увеличения эффективности конвейеров засчет предсказания хода программы. Когда процессор правильно предсказывает следующую команду, все идет согласно плану, но когда предсказание сделано неверно, цикл обработки должен начаться с начала. Из-за этого процессор с 10 шаговым конвейером имеет несколько меньший штраф за неправильный переход, чем процессор с 20 шаговым конвейером.

Для уменьшения недостатков более длинного конвейера, архитектура NetBurst имеет несколько особенностей.

Как уже указывалось ранее, АЛУ Pentium 4 работает на удвоенной тактовой частоте. Это означает, что АЛУ 1.4ГГц Pentium 4 работает на 2.8ГГц, а у 1.5ГГц Pentium 4 на 3.0ГГц. Считается, что это дает Pentium 4 явное преимущество в производительности в целочисленных операциях. Однако практика показала, что основной причиной удвоенной частоты АЛУ является восполнение более низкий IPC NetBurst архитектуры.

Другая особенность связана со снижением влияния более длинного конвейера заключается в том, что Intel называет Execution Trace Cache. Декодер любого 80x86 процессора (модуль, который берет выбранные инструкции и декодирует их в форму, понятную вычислительным модулям) является одним из самых медленных модулей. Execution Trace Cache действует как посредник между стадией декодирования и первой стадией выполнения. Trace cache по существу кэширует декодированные micro-ops (инструкции после того, как они были выбраны и декодированы, т.е. полностью готовы к выполнению) так, чтобы вместо прохождения процесса выборки и декодирования при выполнении новой команды Pentium 4 мог обратиться к trace cache, получить декодированные micro-ops и начинать выполнение.

Это помогает уменьшать штраф, связанный с неправильно предсказанным переходом в длинном конвейере Pentium 4. Другой особенностью trace cache является, то, что он кэширует micro-ops в предсказанном пути выполнения, означая, что, если Pentium 4 выбрал 3 инструкции из trace cache, то они уже представлены в порядке выполнения. Это добавляет некоторый потенциал для неправильного предсказания пути выполнения кэшируемых micro-ops, однако Intel уверена, что это будет компенсироваться новыми алгоритмами предсказания, используемых в Pentium 4.

Intel отказалась от обычного метода определения размера кэш-памяти, по крайней мере для Execution Trace Cache. Вместо этого заявлено, что trace cache может кэшировать приблизительно 12K micro-ops. В дополнение к Execution Trace Cache, Pentium 4 имеет 8KB L1 Data Cache. Очевидно, что это меньше 16KB L1 Data Cache Pentium III. Такой размер кэш-памяти был сделан для достижения лучшего отношения цены и производительности для Pentium 4.

Pentium 4 также имеет 256KB L2, работающий на основной тактовой частоте процессора. Этот кэш имеет большую ширину полосы частот, чем текущий 256KB L2 в Pentium III. Основной причиной является работа на большей тактовой частоте, а также передача данных на каждом такте.

В терминах полосы частот, доступной L2, гипотетический Pentium III работающий на тактовой частоте 1.5ГГц имел бы скорость передачи 24GB/s. Pentium 4 на той же тактовой частоте имеет 48GB/s. L1 в Pentium 4 (включая Execution Trace Cache) дублируется в L2.

5.2.2. AMD Athlon

Микроархитектура Athlon представлена на рис.13. С целью увеличения пропускной способности декодеров, которые перекодируют 80x86-команды во внутренние макрооперации, в блок кэш-памяти команд первого уровня добавлена специальная кэш-память предварительного декодирования. Эти макрооперации представляют собой по сути RISC-команды, которые, собственно, и испол-

няются. Подобная схема перекодирования из 80x86 в RISC используется и в Pentium III, и в предыдущих процессорах AMD, и в некоторых других 80x86-совместимых процессорах.

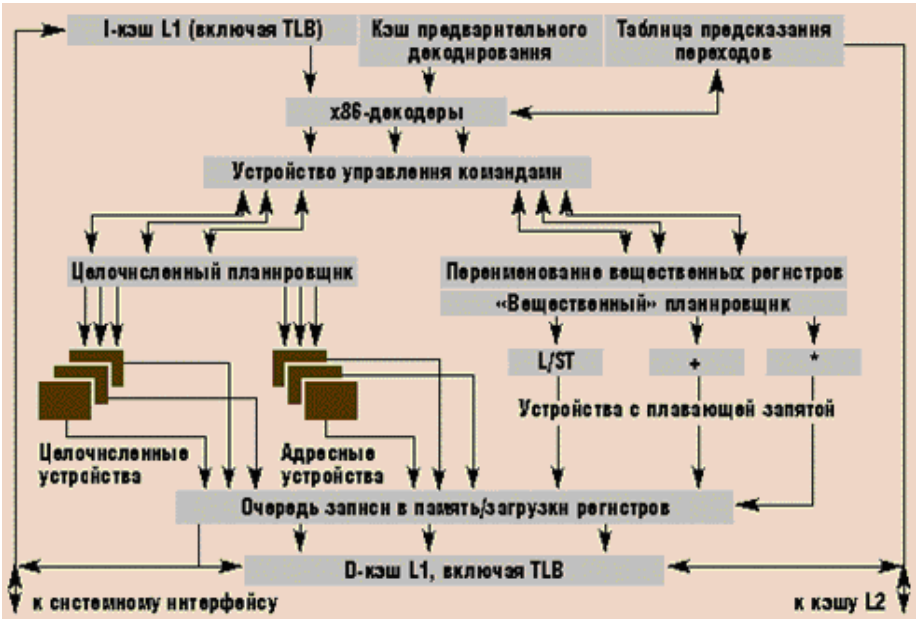


Рис.13

В Athlon таких декодеров три, а число выдаваемых на выполнение за такт команд больше, чем в Pentium III. Команды в декодеры попадают из кэш команд первого уровня. Он является двухканальным, а его емкость составляет 64 Кбайт, что в четыре раза выше, чем в Pentium III. Кроме того, блок кэш команд в AMD содержит два уровня буферов быстрой переадресации TLB: первого уровня — на 24 строки, а второго — на 256.

Емкость кэш данных первого уровня в Athlon также в четыре раза больше, чем в Pentium III, и составляет 64 Кбайт. Он является двухканальным и включает двухуровневый блок TLB. Кэш данных имеет восемь банков, что позволяет одновременно загружать в регистры или писать из них в кэш до двух 64-разрядных величин. Команды из кэш команд поступают в декодеры, а оттуда —

в блок управления командами, емкость которого 72 строки.

Athlon является суперскалярным микропроцессором с внеочередным спекулятивным выполнением команд. Большая емкость очереди команд позволяет Athlon эффективно использовать свои ресурсы: число функциональных исполнительных устройств в нем больше, чем в Pentium III.

Совокупность вышеуказанных факторов уже объясняет, почему производительность Athlon выше, чем в Pentium III. В Athlon имеется три целочисленных устройства и три устройства с плавающей запятой. Кроме того, Athlon содержит три адресных устройства. Все исполнительные устройства способны работать во внеочередном режиме. Для этого в арифметических устройствах имеются «планировщики», содержащие очереди команд емкостью 18 (для целочисленных устройств) и 36 (для вещественных устройств) строк соответственно.

Следует отметить, что в составе МП имеются следующие устройства с плавающей запятой: сумматор, умножитель и модуль загрузки регистров/записи в память. Они могут работать параллельно, поэтому производительность микропроцессора (в MFLOPS) в два раза выше, чем его тактовая частота (1,3 GFLOPS при 650 МГц), и в два раза выше, чем у Pentium III при той же частоте. Кроме того, умножитель Athlon - это настоящий конвейер. Физических регистров с плавающей запятой в Athlon - 88, что позволяет использовать технологию переименования регистров.

Athlon не только суперскалярный, но еще так называемый и суперконвейерный микропроцессор. С одной стороны, большое число ступеней конвейеров (10 — в целочисленном и 15 — в вещественном конвейере) позволяет легче поднимать тактовую частоту. С другой стороны, это вызывает проблему заполнения конвейеров: если они не заполняются, производительность падает. Наиболее «опасными» будут при этом программы нерегулярного характера с большим числом условных переходов, которые трудно динамически предсказывать. Кстати, в Athlon блок динамического предсказания переходов включает таблицу предыстории на 2048 строк. Такой большой объем позволяет добиться очень высокого качества предсказания переходов.

В Athlon расширена система команд 3D–Now!. Их теперь

45, из них 24 - новых, в том числе: 12 команд целочисленной математики для обработки видео и распознавания речи; 7 команд пересылки данных, ориентированных на программы, подобные Internet-приложениям, работающим с графическими данными; 5 новых команд для цифровой обработки сигналов

Кроме рассмотренных выше блоков микропроцессора, Athlon имеет встроенное управление внешней кэш-памятью второго уровня и сопряжение с системной шиной. Интеграция в микропроцессор функций управления внешней кэш-памятью позволяет Athlon иметь L2, расположенный на выделенной шине с программируемой частотой. Поддерживается совместимость с индустриальными стандартами SRAM, в том числе DDR и SDR. Кроме того, этот блок содержит память тегов для L2 наиболее популярного размера 512 Кбайт, при этом емкость L2 может составлять до 8 Мбайт.

Системный интерфейс Athlon обеспечивает соединения «точка-точка», то есть фактически мы имеем дело с коммутатором, а не с общей системной шиной, как у Pentium III. Важным преимуществом коммутаторов является то, что в отличие от системной шины они не имеют конфликтов и обеспечивают гарантированный уровень пропускной способности. Это особенно важно для многопроцессорных SMP-систем.

Системный интерфейс шириной 8 байт может работать на частотах от 200 до 400 МГц, что обеспечивает гораздо более высокую пропускную способность, чем у шины Pentium III. Однако, по некоторым данным, на тестах пропускной способности оперативной памяти (STREAM) Athlon лишь незначительно опережает Pentium III. Важным преимуществом системного интерфейса Athlon является расщепленная обработка транзакций (до 24 на процессор против 4 в Pentium III). При этом пакетный протокол может передавать блоки в 64 байт против 32 байт у Pentium III.

Максимальная поддерживаемая емкость оперативной памяти составляет у Athlon 7 Тбайт против 64 Гбайт в Pentium III; впрочем, это отличие вряд ли имеет сегодня практическое значение. Учитывая высокую пропускную способность системного интерфейса, он явно проектировался в расчете на использование с Athlon технологии RAMBUS. Увеличилась и надежность: теперь

как шина внешней кэш-памяти, так и системный интерфейс используют ECC-коды.

Athlon имеет площадь 128 кв. мм и производится по 0,18-микронной технологии с шестислойной металлизацией. Он использует разъем типа Slot A, механически совместимый со Slot 1 и близкий к применяемому в Alpha EV6.

5.2.3. MC88110 компании Motorola

Процессор 88110 относится к разряду суперскалярных RISC-процессоров. Основные особенности этого процессора связаны с использованием принципов суперскалярной обработки, двух восьмипортовых регистровых файлов, десяти независимых исполнительных устройств, больших по объему внутренних кэш-памятей и широких магистралей данных.

На рис. 14 представлена блок-схема процессора, содержащего 1.3 миллиона вентилей. Центральной частью этой архитектуры является шина операндов (в реализации это шесть 80-битовых шин), соединяющая регистровые файлы и исполнительные устройства.

Процессор имеет 10 исполнительных устройств, которые работают одновременно и независимо, и два регистровых файла. Файл регистров общего назначения имеет 32-битовую организацию. Расширенные регистры плавающей точки имеют 80-битовую организацию. Эти регистровые файлы снабжены шестью портами чтения и двумя портами записи каждый.

Внешняя шина процессора имеет отдельные линии данных (64 бит) и адреса (32 бит), что позволяет реализовать быстрые групповые операции перезагрузки внутренней кэш-памяти. Внешняя шина имеет также специальные сигналы управления, обеспечивающие аппаратную поддержку когерентности кэш-памяти в мультипроцессорных конфигурациях.

В процессоре имеются две двухканальные множественно-ассоциативные кэш-памяти емкостью по 8 Кбайт (для команд и для данных). Они имеют физическую адресацию.

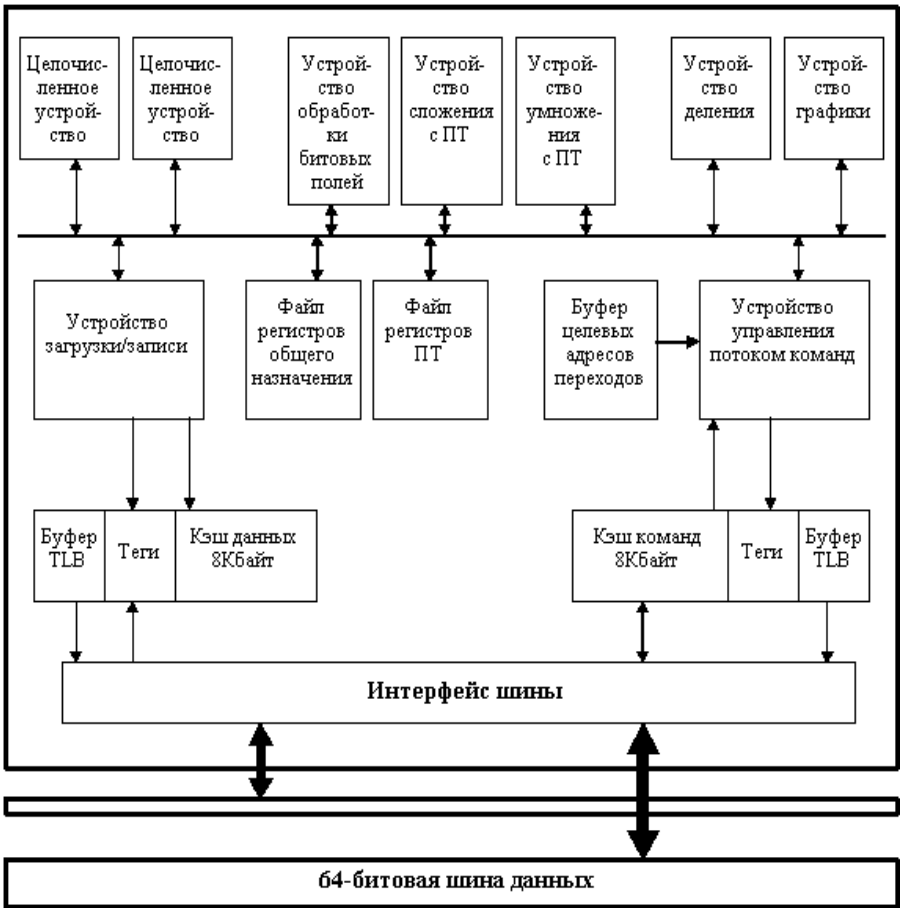


Рис. 14

Все операции по перезагрузке кэш-памяти выполняются в режиме групповой пересылки данных, при этом первым пересылается требуемое слово. Когерентность кэш данных обеспечивается аппаратным протоколом наблюдения за шиной с четырьмя состояниями (MESI). Для увеличения производительности в кэш памяти данных применяется стратегия задержанного обратного копирования.

Суперскалярная архитектура процессора базируются на реализации возможности завершения команд не в порядке их поступления для выполнения, что позволяет существенно увеличить производительность, однако приводит к проблемам организации точного прерывания. Эта проблема решается в процессоре 88110 с помощью так называемого буфера истории, который хранит старые значения регистров при выполнении и завершении операций не в предписанном программой порядке, и позволяет аппаратно восстановить необходимое состояние в случае прерывания.

В процессоре предусмотрено несколько способов ускорения обработки условных переходов. Один из них, предсказание направления перехода, позволяет компилятору сообщить процессору предпочтительное направление перехода.

Для выполняемых переходов используется буфер целевых адресов перехода емкостью 32 строки, позволяющий быстро выбрать две команды по целевому адресу перехода. Механизм предсказания направления переходов позволяет одновременно выполнять эти команды и оценивать условие перехода. Для предсказанного направления перехода разрешено спекулятивное (условное) выполнение команд. Если направление перехода предсказано неверно, исходное состояние процессора восстанавливается с помощью буфера истории. Выполнение программы в этом случае будет продолжено с "правильной" команды.

В каждом такте процессор может выдавать на выполнение две команды. В большинстве случаев выдача команд осуществляется в порядке, предписанном программой. Команды записи и условных переходов могут посылаться на буферные станции резервирования, из которых они в дальнейшем будут выданы на выполнение. Команды загрузки могут накапливаться в очереди. Таким образом эти команды не блокируют выдачу второй команды из пары.

Большое количество исполнительных устройств позволяет осуществлять одновременную выдачу двух команд во многих ситуациях: 2 целочисленные команды, 2 команды с плавающей точкой, 2 графические команды или любая комбинация перечисленных команд.

В устройстве загрузки/записи реализован буфер загрузки FIFO на четыре строки и три станции резервирования операций записи, что позволяет иметь в каждый момент времени до 4 отложенных команд загрузки и до трех команд записи. Выполнение этих команд внутри устройства может переупорядочиваться для обеспечения большей эффективности.

При построении многопроцессорной системы все процессоры и основная память размещаются на одной плате. Для обеспечения хорошей производительности системы каждый процессор в такой конфигурации снабжается кэш-памятью второго уровня емкостью 256 Кбайт. Протокол поддержания когерентного состояния кэш-памяти (протокол наблюдения) базируется на методике записи с аннулированием, гарантирующей размещение модифицированной копии строки кэш памяти только в одной из кэш системы. Протокол позволяет нескольким процессорам иметь одну и ту же копию строки кэш памяти. При этом, если один из процессоров выполняет запись в память (общую строку кэш памяти), другие процессоры уведомляются о том, что их копии являются недействительными и должны быть аннулированы.

5.3. Микропроцессоры с масштабируемой архитектурой

5.3.1. SuperSPARC

Масштабируемая процессорная архитектура компании Sun Microsystems (SPARC - Scalable Processor Architecture) является наиболее широко распространенной RISC-архитектурой, отражающей доминирующее положение компании на рынке UNIX-рабочих станций и серверов. Процессоры с архитектурой SPARC лицензированы и изготавливаются по спецификациям Sun несколькими производителями, среди которых следует отметить компании Texas Instruments, Fujitsu, LSI Logic, Bipolar International Technology, Philips и Cypress Semiconductor. Эти компании осуществляют поставки процессоров SPARC не только самой Sun Microsystems, но и другим известным производителям вычисли-

тельных систем, например, Solbourne, Toshiba, Matsushita, Tatung и Cray Research.

Первоначально архитектура SPARC была разработана с целью упрощения реализации 32-битового процессора. В последствии по мере улучшения технологии изготовления интегральных схем она постепенно развивалось и в настоящее время имеется 64-битовая версия этой архитектуры.

В отличие от большинства RISC архитектур, SPARC использует регистровые окна, которые обеспечивают удобный механизм передачи параметров между программами и возврата результатов. Архитектура SPARC была первой коммерческой разработкой, реализующей механизмы отложенных переходов и аннулирования команд. Это давало компилятору большую свободу заполнения времени выполнения команд перехода командой, которая выполняется в случае выполнения условий перехода и игнорируется в случае, если условие перехода не выполняется.

Первый процессор SPARC был изготовлен компанией Fujitsu на основе вентильной матрицы, работающей на частоте 16.67 МГц. На основе этого процессора была разработана первая рабочая станция Sun-4 с производительностью 10 MIPS, объявленная осенью 1987 года (до этого времени компания Sun использовала в своих изделиях микропроцессоры Motorola 680X0). В марте 1988 года Fujitsu увеличила тактовую частоту до 25 МГц создав процессор с производительностью 15 MIPS.

Позднее компания Sun умело использовала конкуренцию среди компаний-поставщиков интегральных схем (LSI Logic, Cypress и Texas Instruments), выбирая наиболее удачные разработки для реализации своих изделий SPARCstation 1, SPARCstation 1+, SPARCstation IPC, SPARCstation ELC, SPARCstation IPX, SPARCstation 2 и серверов серий 4XX и 6XX. Тактовая частота процессоров SPARC была повышена до 40 МГц, а производительность - до 28 MIPS.

Дальнейшее увеличение производительности процессоров с архитектурой SPARC было достигнуто за счет реализации в кристаллах принципов суперскалярной обработки компаниями Texas Instruments и Cypress. Процессор SuperSPARC компании Texas Instruments стал основой серии рабочих станций и серверов

SPARCstation/SPARCserver 10 и SPARCstation/SPARCserver 20. Имеется несколько версий этого процессора, позволяющего в зависимости от смеси команд обрабатывать до трех команд за один машинный такт, отличающихся тактовой частотой. Процессор SuperSPARC (рис. 15) имеет сбалансированную производительность на операциях с фиксированной и плавающей точкой. Он имеет внутреннюю кэш-память емкостью 36 Кб (20 Кб - кэш команд и 16 Кб - кэш данных), раздельные конвейеры целочисленной и вещественной арифметики и при тактовой частоте 75 МГц обеспечивает производительность около 205 MIPS. Процессор SuperSPARC применяется также в серверах SPARCserver 1000 и SPARCcenter 2000 компании Sun.

Конструктивно кристалл монтируется на взаимозаменяемых процессорных модулях трех типов, отличающихся наличием и объемом кэш-памяти второго уровня и тактовой частотой. Модуль M-bus SuperSPARC, используемый в модели 50, содержит 50-МГц SuperSPARC процессор с внутренней кэш-памятью емкостью 36 Кб (20 Кб кэш команд и 16 Кб кэш данных). Модули M-bus SuperSPARC в моделях 51, 61 и 71 содержат по одному SuperSPARC процессору, работающему на частоте 50, 60 и 75 МГц соответственно, одному кристаллу кэш-контроллера (так называемому SuperCache), а также внешнюю кэш-память емкостью 1 Мб.

Модули M-bus в моделях 502, 612, 712 и 514 содержат два SuperSPARC процессора и два кэш-контроллера каждый, а последние три модели и по одному 1 Мб внешней кэш-памяти на каждый процессор. Использование кэш-памяти позволяет модулям CPU работать с тактовой частотой, отличной от тактовой частоты материнской платы. Пользователи всех моделей поэтому могут улучшить производительность своих систем заменой существующих модулей CPU, вместо того, чтобы производить upgrade всей материнской платы.

Компания Texas Instruments разработала также 50 МГц процессор MicroSPARC с встроенной кэш-памятью емкостью 6 Кб. Он ранее широко использовался в дешевых моделях рабочих станций SPARCclassic и SPARCstation LX, а в настоящее время применяется лишь в X-терминалах.

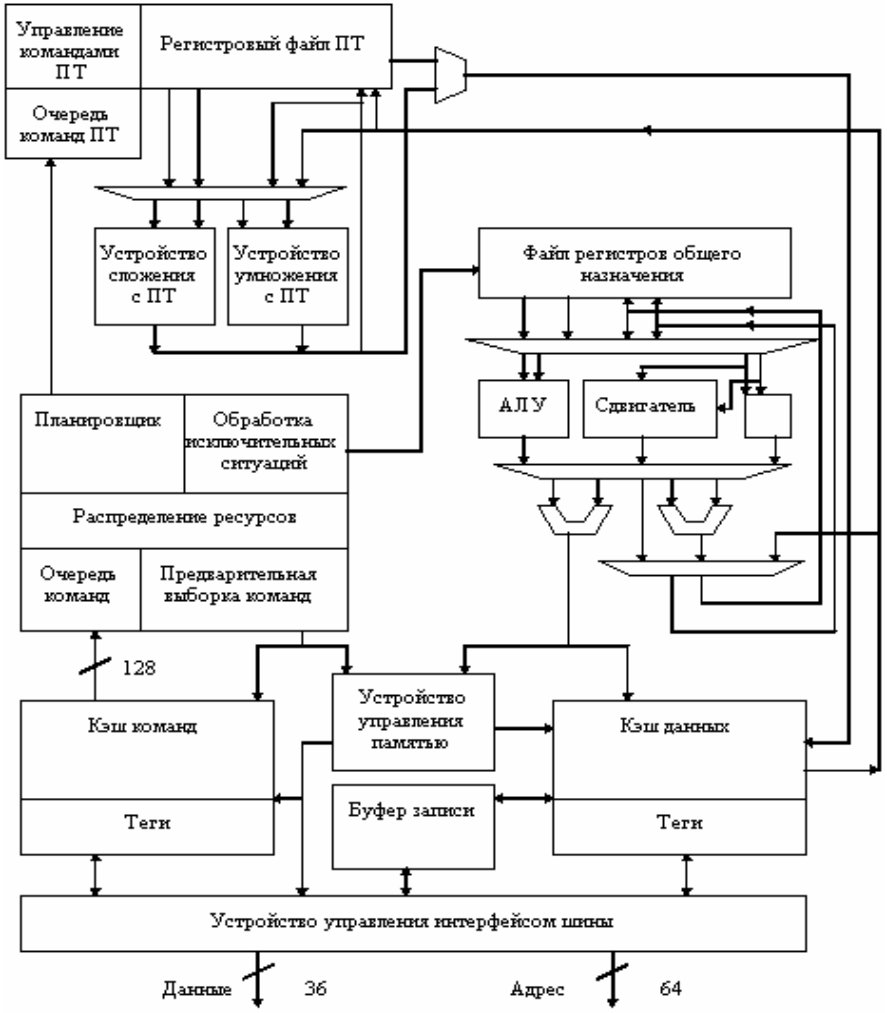


Рис. 15

Sun совместно с Fujitsu создали также новую версию кристалла MicroSPARC II с встроенной кэш-памятью емкостью 24 Кб. На его основе построены рабочие станции и серверы SPARCstation/SPARCserver 4 и SPARCstation/SPARCserver 5, работающие на частоте 70, 85 и 110 МГц.

Хотя архитектура SPARC остается доминирующей на рынке процессоров RISC, особенно в секторе рабочих станций, повышение тактовой частоты процессоров происходило более медленными темпами по сравнению с повышением тактовой частоты других архитектур процессоров. Чтобы ликвидировать это отставание, а также в ответ на появление на рынке 64-битовых процессоров компания, Sun проводила в жизнь программу модернизации. В соответствии с этой программой Sun довела тактовую частоту процессоров MicroSPARC до 100 МГц в 1994 году (процессор MicroSPARC II с тактовой частотой 70, 85 и 110 МГц уже используется в рабочих станциях и серверах SPARCstation 5) и до 125 МГц (процессор MicroSPARC III). В конце 1994 - начале 1995 года на рынке появились микропроцессоры hyperSPARC и однопроцессорные и двухпроцессорные рабочие станции с тактовой частотой процессора 100 и 125 МГц. К середине 1995 года тактовая частота процессоров SuperSPARC доведена до 90 МГц (60 и 75 МГц версии этого процессора в настоящее время применяются в рабочих станциях и серверах SPARCstation 20, SPARCserver 1000 и SPARCcenter 2000 компании Sun и 64-процессорном сервере компании Cray Research). Во второй половине 1995 года появились 64-битовые процессоры UltraSPARC I с тактовой частотой от 167 МГц, в конце 1995 - начале 1996 года - процессоры UltraSPARC II с тактовой частотой от 200 до 275 МГц, а в 1997/1998 годах - процессоры UltraSPARC III с частотой 500 МГц.

5.3.2. MicroSPARC-II

Эффективная с точки зрения стоимости конструкция не может полагаться только на увеличение тактовой частоты. Экономические соображения заставляют принимать решения, основой которых является массовая технология. Системы microSPARC обеспечивают высокую производительность при умеренной тактовой частоте путем оптимизации среднего количества команд, выполняемых за один такт. Это ставит вопросы эффективного управления конвейером и иерархией памяти. Среднее время обращения к памяти должно сокращаться, либо должно возрастать среднее количество команд, выдаваемых для выполнения в каждом такте,

увеличивая производительность на основе компромиссов в конструкции процессора.

MicroSPARC-II (рис. 16) является представителем семейства микропроцессоров SPARC. Основное его назначение - однопроцессорные низкостоимостные системы. Он представляет собой высокоинтегрированную микросхему, содержащую целочисленное устройство, устройство управления памятью, устройство плавающей точки, раздельную кэш-память команд и данных, контроллер управления микросхемами динамической памяти и контроллер шины SBus.

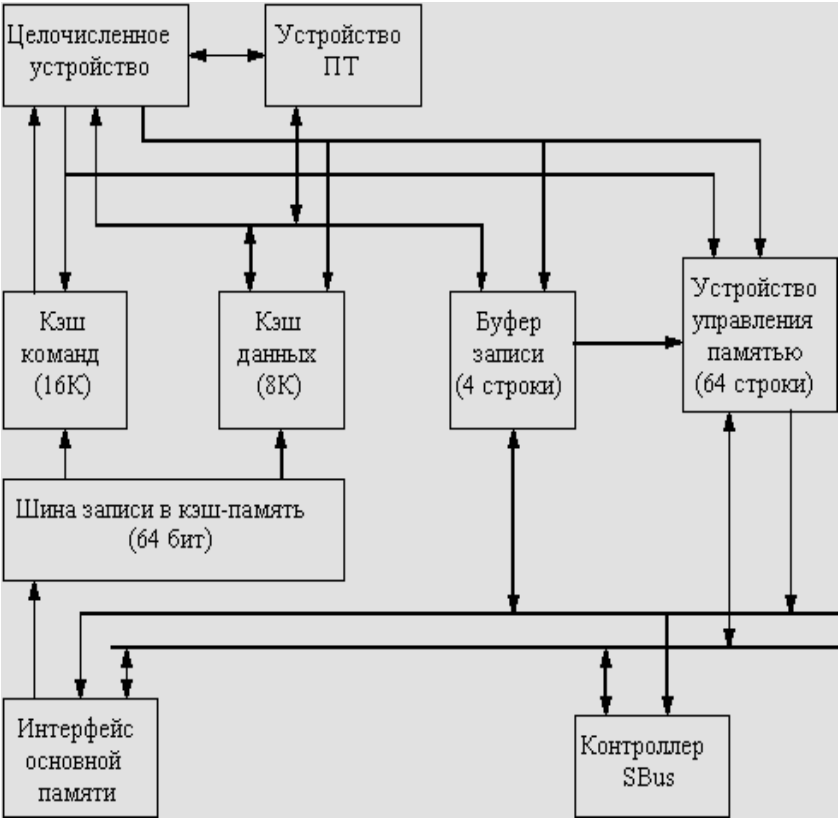


Рис. 16

Основными свойствами целочисленного устройства microSPARC-II являются:

- пятиступенчатый конвейер команд;
- предварительная обработка команд переходов;
- поддержка потокового режима работы кэш-памяти команд и данных;
- регистровый файл емкостью 136 регистров (8 регистровых окон);
- интерфейс с устройством плавающей точки;
- предварительная выборка команд с очередью на четыре команды.

Целочисленное устройство использует пятиступенчатый конвейер команд с одновременным запуском до двух команд. Устройство плавающей точки обеспечивает выполнение операций в соответствии со стандартом IEEE 754.

Устройство управления памятью выполняет четыре основных функции. Во-первых, оно обеспечивает формирование и преобразование виртуального адреса в физический. Эта функция реализуется с помощью ассоциативного буфера TLB. Кроме того, устройство управления памятью реализует механизмы защиты памяти. И, наконец, оно выполняет арбитраж обращений к памяти со стороны ввода/вывода, кэша данных, кэша команд и TLB.

Процессор microSPARC II имеет 64-битовую шину данных для связи с памятью и поддерживает оперативную память емкостью до 256 Мбайт. В процессоре интегрирован контроллер шины SBus, обеспечивающий эффективную с точки зрения стоимости реализацию ввода/вывода.

6. Особенности архитектуры 64 – разрядных МП

6.1. Itanium 2 Intel

Процессор разрабатывался с нуля, причем, параллельно сразу в двух версиях: инженерами Intel и Hewlett-Packard. Впрочем, в основе обоих чипов лежали, естественно, одни и те же идеи, поскольку создавались они все же совместно, и должны были оба стать родоначальниками одного и того же семейства. Цементи-

рующим составом были, естественно, единая идеология, пришедшая на смену CISC - EPIC (Explicitly Parallel Instruction Computing), и новая архитектура - IA-64, включающая в себя набор инструкций, описание регистров, и прочие подобные вещи. Впрочем, архитектура как раз - вещь изменчивая, достаточно вспомнить как отличаются между собой такие CISC процессоры, как 8086 и i80486, оба созданные на базе 80x86.

Точно так же и с Merced и McKinley, Itanium и Itanium 2 - оба построены на базе одной идеологии, но в разных разновидностях архитектуры. В свое время та же история, в общем то, была и с Pentium и Pentium Pro. Впрочем, общие черты были и у тех, есть и у этих, за это "отвечает" EPIC. В первую очередь речь идет о полноценной масштабной суперскалярности, то есть, способности выполнять одновременно несколько инструкций. Для чего, естественно, процессор содержит исполнительные модули - для операций с целыми числами, с числами с плавающей запятой, и т.д.

В отличие от Pentium и его последователей, разбирающихся в коде самостоятельно, EPIC-процессоры сильно полагаются на компилятор, который должен сам проанализировать код на предмет нахождения оптимальных мест для распараллеливания его выполнения, и снабдить процессор этой информацией. Поэтому и используется «explicitly» - процессор не должен сам пытаться понять, что можно исполнять параллельно, а что нет, и т.д. - все это ему уже заранее объяснит компилятор. Плюс, мощные механизмы по предсказанию переходов, предварительному выполнению кусков кода, предварительной загрузке данных, и тому подобные вещи - загрузка исполнительных блоков должна быть распределена максимально равномерно.

Кардинально решен вопрос с регистрами, количество которых увеличено в несколько раз: у Itanium их количество составляет 128 общего назначения (рис.17), 128 - для хранения чисел с плавающей запятой, 8 регистров переходов, и 64, отвечающих за работу механизмов предсказания. Здесь все очевидно - такого количества регистров, да еще реально 64-битных, хватит для хранения любых требуемых чисел для любого разумного количество исполнительных модулей. У Itanium, первого представителя семейства, таких регистров всего пять - два целочисленных, два для операций

с памятью и четыре - для операций с плавающей точкой. Физическая память адресуется 44-бит числами, что на самом деле ограничивает ее объем "всего лишь" 17.6 Терабайт, блоки для операций с плавающей точкой работают с числами в 82-бит представлении.

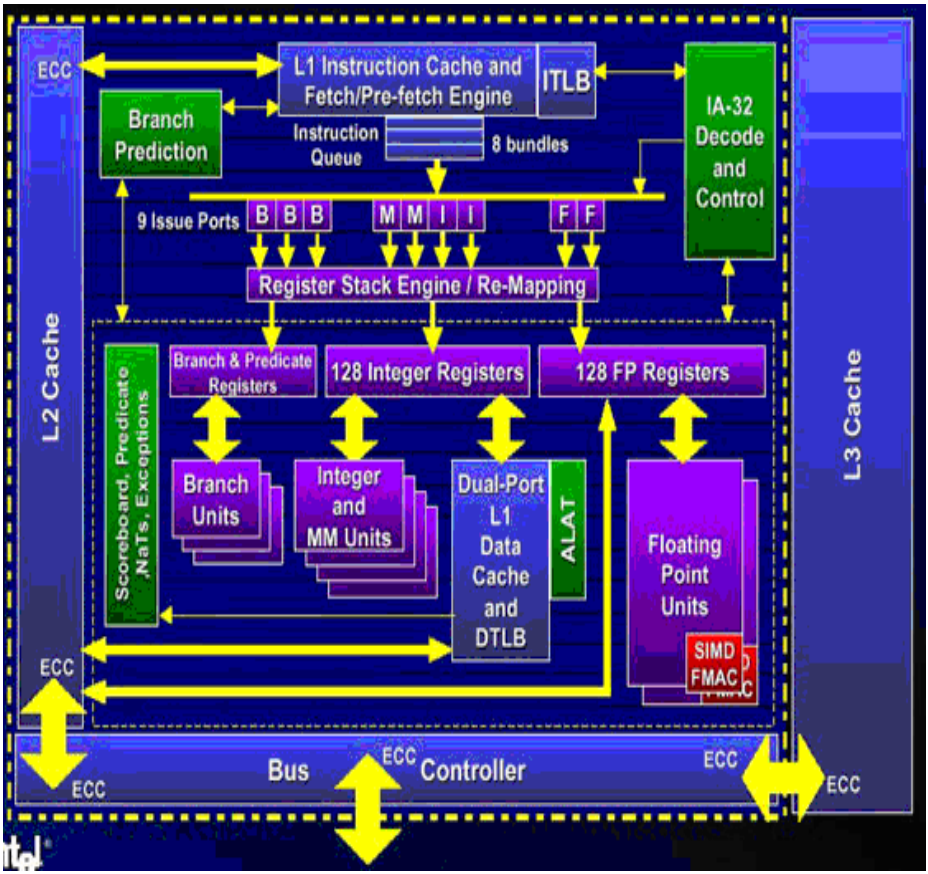


Рис.17

От идеи реализовать 32-бит 80x86 ядро в аппаратном виде Intel отказался, сочтя это слишком неэффективным использованием площади кристалла. Так что для того, чтобы получить возможность исполнения Itanium 80x86 кода, была создана система трансляции, которая на лету преобразует 80x86 код в IA-64.

Очевидно, что при прочих равных, производительность подобного решения будет ниже, чем чистого x86, работающего на той же частоте. Впрочем, никто и не ждал от Itanium скоростного исполнения x86 программ - поддержка этой архитектуры относится скорее к издержкам переходного периода. Тем не менее, факт остается фактом: это семейство для решения 32-бит задач не приспособлено. Впрочем, вряд ли кто-то будет использовать Itanium для подобных целей при наличии полноценного 64-битного ПО..

Вдобавок, сам по себе Itanium был в значительной степени пилотным проектом, как и Pentium Pro, так что процессор вообще стоит рассматривать больше как демонстрацию возможностей архитектуры. Характерный штрих - чипсет для Itanium, 460GX, поддерживает в качестве памяти всего лишь PC100 SDRAM, это кое-что говорит о скорости, с которой способен переваривать данные процессор. С другой стороны, однако, в какой-то мере не слишком быстрый интерфейс с оперативной памятью компенсируется очень большой кэш-памятью L3 - 2 или 4 Мбайт, работающей на полной частоте процессора (733 или 800 МГц) с пропускной способностью до 12.8 Гбайт/с.

Еще одной задачей Itanium было решить вопрос с компиляторами - ведь EPIC-процессоры, как уже упоминалось, очень сильно от них зависят. В отличие от компиляторов для 80x86 процессоров, которые на их производительность почти не влияли, здесь компиляторы являются полноправными партнерами процессора - ведь они снабжают его крайне необходимой для работы информацией, и от того, насколько качественной она будет, будет зависеть скорость исполнения этой программы процессором.

Itanium 2 является уже куда более коммерчески интересным продуктом. Созданный Hewlett-Packard, набившей руку на создании 64-бит процессоров серии PA-RISC, чип получился куда более совершенным. С несколько меньшим количеством L3 (1.5 или 3 Мбайт) и несколько более

высокой частотой, 900 МГц или 1 ГГц, он обеспечивает в полтора-два раза большую производительность на тех же задачах, что и Itanium. Он является, фактически, первым представителем архитектуры IA-64.

Дальше планируется еще большее распараллеливание максимально модным на сегодняшний день путем: процессор должен будет перейти на два физических ядра, что позволит практически удвоить производительность по достаточно приемлемой цене - по крайней мере, результат получится куда более дешевым, чем если бы того же количества исполнительных модулей, регистров, и т.д., пытались достичь на едином кристалле.

6.2. Athlon 64 AMD

В первую очередь заметим, что процессор Athlon 64 – это именно тот 64-битный процессор для настольных систем, который изначально планировала выпустить AMD. Впоследствии, в свете выхода скоростных процессоров Pentium 4, появления в них 800-мегагерцовой шины и технологии Hyper-Threading, AMD в срочном порядке решила нацелить на рынок настольных систем и однопроцессорный Opteron, дав ему имя Athlon 64 FX. Однако Athlon 64 FX в силу своего серверного происхождения оказался дорогим и малораспространенным. По настоящему же продвинуть архитектуру AMD64 для массового использования должен именно Athlon 64.

Ниже представлена таблица 3 со спецификациями 64 - разрядных МП Athlon 64 3200+, Athlon 65 FX-51 и Athlon XP 3200+:

Таблица 3

Характеристика	Athlon 64 FX-51	Athlon 64 3200+	Athlon XP 3200+
Частота	2.2 ГГц	2.0 ГГц	2.2 ГГц
Технология производства	0.13 мкм	0.13 мкм	0.13 мкм
Число транзисторов	105.9 млн.	105.9 млн.	54.3 млн.
Площадь ядра	193 кв.мм	193 кв.мм	101 кв.мм

Номинальное напряжение	1.5В	1.5В	1.65В
Встроенный контроллер памяти	Двуканальный, 128-битный	Одноканальный, 64-битный.	Нет
Поддержка ECC	+	+	-
L1	128 Кбайт (по 64 Кбайта на код и данные)	128 Кбайт (по 64 Кбайта на код и данные)	128 Кбайт (по 64 Кбайта на код и данные)
L2	1024 Кбайт (эксклюзивный)	1024 Кбайт (эксклюзивный)	512 Кбайт (эксклюзивный)

* Заметим, что память в Athlon 64 и Athlon 64 FX тактуется относительно частоты ядра, поэтому реальные частоты для памяти в этом случае составляют 129.4, 157.1 и 200 МГц.

Фактически, Athlon 64 отличается от своего старшего собрата Athlon 64 FX, помимо формы и размеров корпуса, только лишь контроллером памяти. Хотя, при этом, и тот и другой процессоры производятся из одних и тех же кристаллов. Контроллер памяти в Athlon 64 одноканальный и в этом заключается как его слабость, так и преимущество по сравнению с Athlon 64 FX. Недостаток одноканального контроллера памяти в Athlon 64 очевиден: это более низкая теоретическая пропускная способность.

Учитывая, что Athlon 64 способен работать с DDR400 памятью, максимальная пропускная способность встроенного в CPU контроллера памяти составляет 3.2 Гбайт в секунду. Это в два раза меньше, чем аналогичная характеристика Athlon 64 FX. Преимущество же контроллера памяти Athlon 64 заключается в том, что он, в отличие от контроллера Athlon 64 FX, поддерживает обычные нерегистровые модули памяти. Такие модули по сравнению с регистровыми более дешевы, имеет более агрессивные тайминги и работают быстрее, даже при одинаковых с регистровыми модулями настройках. То есть при более низкой пропускной способности, обеспечиваемой контроллером памяти Athlon 64, подсистема па-

мяти, его использующая, имеет более низкую латентность, что мы и покажем ниже.

AMD Athlon 64 по внешнему виду похож на Opteron и Athlon 64 FX.

Отличия обнаруживаются только лишь в маркировке и в меньшем числе ножек на обратной стороне, поскольку процессоры Athlon 64 устанавливаются в материнские платы с Socket 754 и не совместимы с Socket 940 платами, предназначенными для CPU семейств Athlon 64 FX и Opteron.

Помимо перечисленных выше особенностей, есть в новых процессорах Athlon 64 и еще одна. Эти процессоры обладают поддержкой технологии Cool'n'Quiet, фактически пришедшей в них из мобильных вариантов МП. По сути, Cool'n'Quiet представляет собой некое подобие технологии энергосбережения PowerNow!, уже давно используемой в мобильных МП от AMD. Но теперь эта технология, наконец, пришла и в настольные процессоры компании. Поддержка Cool'n'Quiet – еще одно преимущество Athlon 64 над Athlon 64 FX/Opteron, не имеющих пока никаких подобных технологий. Компания AMD достаточно давно уделяет пристальное внимание понижению уровня тепловыделения своих настольных процессоров.

Надо сказать, что в этом компания уже давно превосходит Intel: старшие модели процессоров AMD при максимальной нагрузке выделяют значительно меньше тепла, чем старшие модели Pentium 4. Также, в процессорах применяются технологии, понижающие тепловыделение и при низкой нагрузке. Еще МП семей-

ства Athlon XP имели возможность перехода в «ждущий режим» (Halt/Stop Grant) при выполнении команды HALT, что выливалось в понижение температуры процессора при его загрузке ниже 100%. Однако теперь AMD пошла еще дальше. В новых процессорах Athlon 64 реализована еще более интеллектуальная схема понижения тепловыделения.

В дополнение к состояниям Halt/Stop Grant, Athlon 64 умеет сбрасывать свою тактовую частоту и напряжение питания для еще более сильного снижения тепловыделения. В работе с использованием этой технологии тактовой частотой МП управляет драйвер процессора, который сбрасывает или повышает ее, основываясь на данных о его загрузке. Действительно, если процессор полностью справляется с возлагаемой на него работой и его загрузка сильно меньше 100%, то можно без ущерба для функционирования системы в целом снизить его тактовую частоту: на работе системы это никак не скажется. Например, при простоях, работе в офисных приложениях, просмотре видео, дефрагментации дисков и в подобных задачах мощности процессора в полной мере не используются. Именно в таких случаях процессорный драйвер переводит Athlon 64 на меньшую тактовую частоту. Когда же от процессора требуется полная отдача, например, в играх, при решении вычислительных задач, в задачах кодирования данных и т.п., частота процессора поднимается до номинала. Именно таким образом и работает технология Cool'n'Quiet.

На практике это выглядит следующим образом. В обычных условиях, при минимальной загрузке МП процессорный драйвер сбрасывает частоту Athlon 64 3200+ со штатных 2 ГГц до 800 МГц. Напряжение питания процессора при этом понижается до 1.3В. Как видим, снижение тактовой частоты обеспечивается за счет уменьшения множителя процессора до 4х. Это, кстати, обуславливает и тот факт, что процессоры Athlon 64 3200+ поставляются с незафиксированным коэффициентом умножения. В таком режиме процессор продолжает работать до тех пор, пока его загрузка не превысит 70-80%. В частности, мы смогли запустить одновременно дефрагментацию диска, проигрывание файлов с расширением mp3 (аудифайлов) и просмотр MPEG-4 (видеофайлов)

ролика, в то время как процессор продолжал работать на частоте 800 МГц.

Когда же загрузка процессора Athlon 64 при частоте 800 МГц превышает допустимый предел, МП переводится драйвером в следующее состояние, при котором частота Athlon 64 3200+ составляет 1.8 ГГц, а напряжение питания 1.4В. Достигается это вновь за счет уменьшения множителя, на этот раз до 9х. И только если в данном случае нагрузка процессора вновь оказывается чрезмерно высокой, драйвер переводит МП в штатный режим: частота 2 ГГц, напряжение питания – 1.5В.

Отметим, что в режимах с пониженным питанием и частотой тепловыделение процессора Athlon 64 3200+ резко падает. Для сравнения приведем таблицу 4 с тепловыделением этого процессора в основных режимах.

Таблица 4

Частота	2000 МГц	1800 МГц	800 МГц
Напряжение	1.5 В	1.4 В	1.3 В
Типичное тепловыделение	89 Вт	66 Вт	35 Вт
Типичное тепловыделение в состоянии Halt/Stop Grant	2.2 Вт	2.2 Вт	2.2 Вт

Таким образом, использование технологии Cool’n’Quiet позволяет значительно снизить температуру процессора не только в моменты простоя, но и во время выполнения ряда задач, не требующих от МП максимальной производительности. Что немаловажно, быстродействие МП в задачах, требовательных к процессорным ресурсам, при этом совершенно не снижается. В итоге, при применении систем охлаждения с вентиляторами с переменной скоростью, использование технологии Cool’n’Quiet может позволить значительно снизить уровень шума.

6.3. UltraSPARC III Sun

Процессор UltraSPARC III создавался для того, чтобы заменить микропроцессоры второго поколения семейства UltraSPARC. Он имеет производительность, в 2-3 раза превышающую показатели 300-мегагерцовых процессоров UltraSPARC II. Помимо этого, архитектура процессора разрабатывалась с учетом использования его в масштабируемых многопроцессорных системах.

UltraSPARC III представляет собой высокопроизводительный суперскалярный микропроцессор, предназначенный для использования в мощных рабочих станциях и серверах. Он ориентирован на работу в масштабируемых микропроцессорных комплексах.

На рис.18 приведена структурная схема процессора. В его состав входят следующие основные функциональные блоки:

- шесть исполнительных устройств (4 целочисленных и 2 плавающих);
- кэш-памяти команд и данных объемом 32 КБ и 64 КБ соответственно;
- кэш предвыборки и записи объемом по 2 КБ каждый;
- системный интерфейс;
- контроллер основной памяти;
- контроллер L2;
- таблица тэгов L2.

Функции исполнительных устройств распределяются следующим образом. Целочисленные: 2 АЛУ (Integer ALU, ALU/Load), специализированное исполнительное устройство (Load/Store) и устройство обработки ветвлений (Brach Unit). Плавающие: устройства плавающей арифметики/графики (FP adder, FP mul/div/sqrt). Integer ALU, ALU/Load отвечают за выполнение целочисленных операций. Блок Integer ALU содержит фактически два устройства, каждое из которых может выполнять все однотактные операции. Блок ALU/Load выполняет многотактные команды (умножение, деление) и рассчитывает адреса в командах предварительной выборки. В этот блок команды поступают на выполнение по одной, в то время как всего микропроцессор может

выдавать на выполнение до четырех целочисленных команд за такт, одна из которых должна быть командой загрузки или условной пересылки. Функциональных устройств для операций с плавающей запятой тоже два; одна операция сложения и одна операция умножения могут выдаваться на выполнение в каждом такте. В этих устройствах выполняются также и "графические" команды UltraSPARC III.

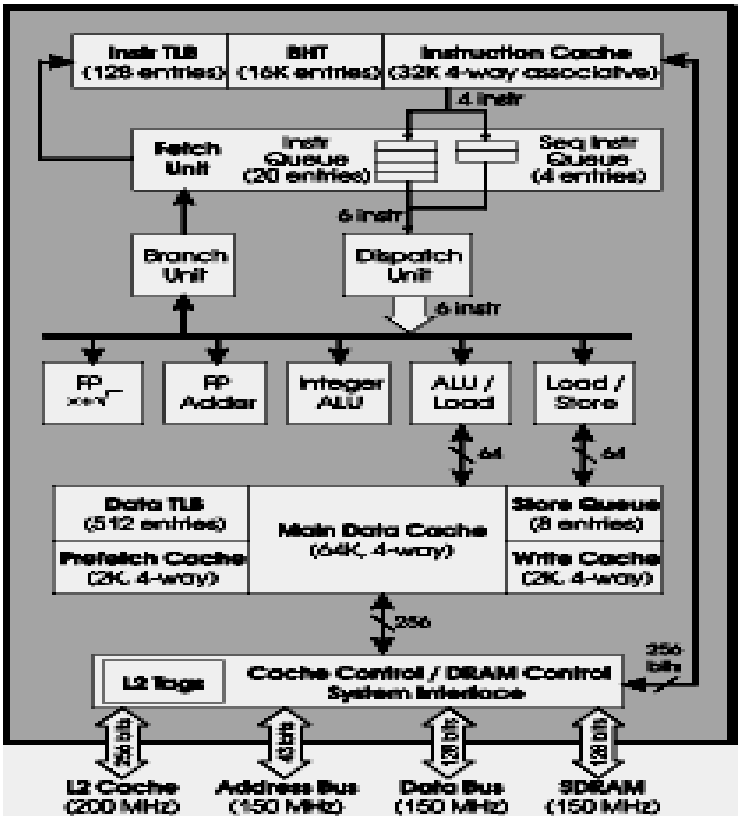


Рис. 18

В UltraSPARC III предусмотрены четырехканальный частично-ассоциативный D-кэш (кэш данных) первого уровня емкостью 64 Кбайт и четырехканальный частично-ассоциативный I-

кэш (кэш инструкций) первого уровня емкостью 32 Кбайт; оба кэш неблокирующиеся.

Следует обратить внимание на большую емкость D-кэш. Кэш данных первого уровня имеет двухтактный доступ, что обусловлено, по всей видимости, высокой частотой процессора (600 МГц).

Кроме кэш-памяти первого уровня, в UltraSPARC III имеется также кэш предварительной выборки (prefetch) емкостью 2 Кбайт, также являющийся четырехканальным частично-ассоциативным. В нем может поместиться до восьми результатов запросов на предварительную выборку. Использование механизма предварительной выборки позволяет избежать блокировки кэш первого уровня, которая могла бы возникнуть из-за превышения порога для числа непопаданий в кэш. Кэш предварительной выборки поддерживает одновременное выполнение двух команд загрузки регистров с общей задержкой всего три такта; за 14 тактов процессора из кэш второго уровня в кэш предварительной выборки можно записать 64 байт данных.

В обоих кэш первого уровня содержатся также буферы быстрой переадресации (I-TLB и D-TLB соответственно), ускоряющие преобразование 64-разрядных виртуальных адресов в 43-разрядные физические. Емкость I-TLB составляет 128 строк, а D-TLB - 512 строк. Кроме 128-строчного буфера, I-TLB включает еще 16-строчный полностью ассоциативный буфер. Надо сказать, что в последнее время разработчики RISC-процессоров уделяют больше внимания буферам TLB; увеличение их емкости предусмотрено архитектурами HP PA-8x00, SGI/MIPS R1x000 и Alpha. В процессоре Alpha 21264 буферы I-TLB и D-TLB имеют емкость по 128 строк.

Еще один тип кэш-памяти UltraSPARC III - это четырехканальный частично-ассоциативный кэш записи емкостью 2 Кбайт. В рассматриваемом микропроцессоре команды записи в память перед выполнением помещаются в очередь записи длиной восемь строк. Запись может производиться либо в D-кэш первого уровня, либо в память более низкого уровня. В последнем случае данные как раз и помещаются сперва в кэш записи. В отличие от D-кэш,

использующего алгоритм сквозной записи, здесь применяется алгоритм обратной записи.

Применение очереди записи и соответствующего кэш способствует объединению нескольких запросов на запись в более крупные блоки. Кроме того, если данные, которые еще находятся в очереди записи, необходимо загрузить в регистр, они загружаются прямо из очереди. В результате, по оценкам Sun, график записи в кэш второго уровня удается сократить на порядок.

Теги кэш второго уровня в UltraSPARC III расположены на основной микросхеме процессора. Как считают разработчики Sun Microsystems, если бы память тегов размещалась непосредственно на микросхеме кэш-памяти, то задержка при непопадании в кэш второго уровня возросла бы на 10 тактов, а пропускная способность уменьшилась втрое.

Память тегов кэш второго уровня в UltraSPARC III достаточна для поддержки кэш емкостью до 8 Мбайт. Для современных процессоров это максимальная величина. Однако у интеграции в микропроцессор памяти тегов есть и оборотная сторона: память тегов занимает на кристалле почти такую же площадь, как I-кэш первого уровня.

Основу архитектуры любого процессора составляет конвейер. В UltraSPARC III увеличено число стадий конвейера до 14 (рис.19).

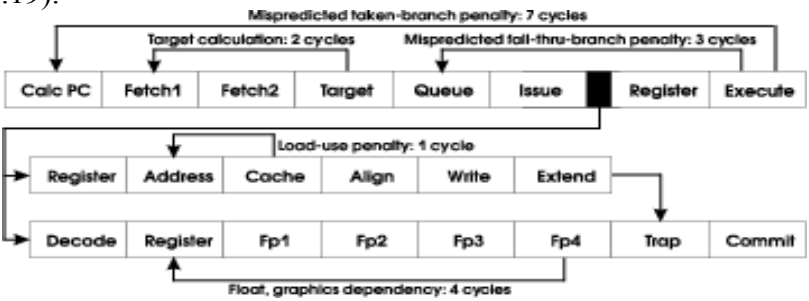


Рис. 19

Это один из самых длинных конвейеров среди всех современных универсальных микропроцессоров архитектуры RISC. Общепризнанное слабое место подобного подхода - проблема заполнения длинных конвейеров. Так, связанная с неправильным

предсказанием перехода задержка в UltraSPARC III составляет 7 тактов. Почти половина длины конвейера (6 ступеней) отводится на подготовку к выполнению команд, столько же - на исполнение команд; две последние ступени - завершающие.

Исполнительная часть конвейера состоит из двух частей: целочисленной и плавающей. Обе части имеют одинаковую длину, что упрощает согласование их работы (позволяет выдавать результаты вычислений в порядке их запуска на исполнение). Аналогичное решение (выровненные конвейеры целочисленной и плавающей арифметики) успешно использовалось в предшествующих поколениях микропроцессоров UltraSPARC.

Большое количество ступеней конвейера можно объяснить существенным повышением тактовой частоты процессора. Более короткие фазы выполнения позволяют избежать длинных связей на кристалле, которые при столь малых технологических нормах начинают вносить заметную дополнительную задержку. Те операции, которые не вписываются в один такт, как, например, выборка команд из кэш-памяти, разбиваются на более мелкие процедуры и выполняются за 2 такта.

Следует отметить еще одну особенность, связанную с исполнительной частью целочисленного конвейера. Для выполнения целочисленных команд отводится 4 такта, реально на это уходит меньше времени (команды АЛУ выполняются за 1 такт). Однако, выровненный конвейер позволяет получить на выходе результаты в том порядке, в котором они поступили на исполнение.

В отличие от многих других современных процессоров, использующих механизм произвольного запуска команд на исполнение, то есть не в порядке их расположения в программе, UltraSPARC III действует строго по порядку. По утверждениям разработчиков, это позволяет сократить объем логики управления в ядре процессора, которая при разработке ее топологии требует большого объема ручной работы, что благоприятно сказывается на быстродействии.

Таким образом, в новом процессоре, как и в его предшественниках, запуск команд на исполнение происходит в порядке их расположения в программе. Процессор позволяет одновременно запустить на исполнение до шести команд за такт (по числу ис-

полнительных устройств); правда, средневзвешенное число одновременно запускаемых команд равно четырем.

Команды выбираются из I-кэш первого уровня и помещаются в очередь команд (буфер) длиной 20 строк (Instruction Queue), откуда группами направляются в соответствующие исполнительные устройства. Максимальное число команд в группе - 6. Все команды в группе получают идентификационный код, в соответствии с которым на выходе из конвейера будут сложены их результаты.

Выше уже упоминалось, что время выполнения большинства целочисленных команд существенно меньше длины исполнительной части целочисленного конвейера, и что фиксированная длина конвейера обеспечивает сохранение порядка поступления команд. Однако, это не означает, что результаты становятся доступны только по достижению выхода конвейера. На самом деле полученные данные могут быть использованы другими командами, находящимися на стадии выполнения, уже на следующем такте после получения результата. Это возможно благодаря наличию рабочего регистрового файла, в котором хранятся все промежуточные результаты вычислений и из которого, по завершению исполнения группы команд, эти результаты переписываются в архитектурный регистровый файл. Таким образом, реальная эффективность конвейера заметно повышается.

Устройство предсказания ветвлений является непременным атрибутом любого современного высокопроизводительного микропроцессора, однако, в каждом конкретном случае оно реализовано по-своему. Основная задача данного устройства состоит в сокращении накладных расходов из-за нарушений в работе конвейера при ветвлении программ. Если взглянуть на конвейер UltraSPARC III, то хорошо видно, сколько придется заплатить за неправильно предсказанный переход. Штрафные санкции составят 7 дополнительных тактов (кстати, у Alpha 21264 — столько же при меньшей длине конвейера).

При решении данной задачи разработчики процессора решили использовать достаточно простой одноуровневый механизм (в отличие от двухуровневого адаптивного механизма в Alpha 21264). Он представляет собой таблицу на 16 К значений (ее раз-

мер увеличен), содержащую информацию об уже происшедших ветвлениях и обеспечивает точность предсказаний на уровне 95% на тестах SPEC95, что близко к аналогичным показателям для Alpha 21264 и AMD K6. Надо отметить, что в UltraSPARC III буфер адресов перехода отсутствует, поэтому при выполнении правильно предсказанного перехода возникает дополнительная задержка.

Помимо механизма предсказания ветвлений в процессоре используется еще и стек адресов возврата на 8 значений, запоминающий адреса возврата при вызове подпрограмм, а также очередь последовательных команд (Sequential Instruction Queue), которая хранит до четырех команд, следующих за командой ветвления, но соответствующих альтернативному пути. В случае, когда предсказанное ветвление окажется неверным, команды из этой очереди сразу направляются на исполнение, экономя несколько тактов.

Производительность вычислительной системы зависит от многих факторов, среди которых быстродействие процессора - не самый главный. Очень многое определяет то, как он взаимодействует с другими компонентами системы.

Первое, что сразу обращает на себя внимание, это большое количество внешних интерфейсов у процессора. Их три: 128-разрядный канал обмена с основной памятью (Main Memory), 256-разрядный канал обмена с L2 и 128-разрядный системный интерфейс.

Достаточно очевидно, что такая многошинная архитектура способствует более эффективной работе многопроцессорной системы в целом. Правда, это новое качество недешево обходится (это и дополнительные расходы на разработку нового корпуса с рекордным числом выводов — 1200, и проблемы повышенного энергопотребления как самого процессора, так и микросхем чипсета и т.п.). Рассмотрим подробнее каждый из перечисленных интерфейсов.

Поскольку архитектура процессора подразумевает единоличное владение памятью, то есть отсутствие непосредственного доступа к ней со стороны каких-либо других устройств, появляется возможность достижения максимальной пропускной способности данного канала, для чего предусмотрен накристальный кон-

троллер памяти. Такое решение имеет следующие положительные стороны.

Во-первых, отпадает необходимость в дополнительных внешних компонентах, то есть упрощается сопряжение процессора и памяти. Это к тому же приводит еще и к снижению стоимости.

Во-вторых, возрастает пропускная способность, поскольку производительность канала зависит только от параметров памяти. Так, при использовании SDRAM с тактовой частотой 150 МГц пропускная способность составит 2.4 ГБ/с.

По своей реализации данный канал напоминает описанный выше. Однако, здесь есть ряд принципиальных особенностей.

Во-первых, канал имеет более высокую разрядность — 256 бит. На сегодняшний день UltraSPARC III, пожалуй, единственный микропроцессор, имеющий такую широкую шину данных кэш второго уровня. При использовании микросхем статической памяти (SRAM), работающих на частоте 200 МГц, пропускная способность данного канала способна достичь 6.4 ГБ/с. И это не предел.

Во-вторых, по аналогии с накристальным SDRAM-контроллером канала основной памяти, данный интерфейс имеет накристальную таблицу тэгов вторичного кэш. Размер таблицы составляет 90 КБ, и этого достаточно для поддержания кэш-памяти объемом до 8 МБ.

Основным достоинством такого решения является то, что работа с таблицей осуществляется на частоте процессора, то есть результат обращения к кэш становится известен гораздо раньше, чем в случае внекристального расположения таблицы тэгов. Соответственно, при непопадании в кэш процедура инициализации обращения к основной памяти начинается на несколько тактов раньше. Аналогично обстоит дело и с поддержкой когерентности кэш в многопроцессорных системах.

Канал записи состоит из трех основных частей: очереди на 8 слов (Store Queue), кэш-памяти данных первого уровня (L1 Data Cache) и кэш-памяти записи (Write Cache). Сразу же отметим, что кэш имеют различные механизмы обновления: L1 кэш данных - сквозной записи, а кэш записи - отложенный. Далее будет понятно, зачем это нужно.

Сначала сохраняемая информация записывается в очередь. Это происходит во время выполнения команды сохранения. Затем, после завершения команды, данные записываются в L1 и, одновременно, в кэш записи. При этом, если происходит непопадание в L1, то его содержимое не обновляется. В противном случае из-за сквозного режима обновления данной кэш-памяти происходило бы постоянное обращение ко вторичному кэш. Таким образом, кэш-память записи как бы дополняет и дублирует L1, но только в процессе записи.

По утверждениям разработчиков, использование такой организации канала записи позволяет сократить трафик на шине вторичной кэш-памяти на 90%.

Системный интерфейс по своим характеристикам аналогичен каналу основной памяти. Из специфических механизмов, свойственных только ему, следует отметить поддержку многопроцессорности (до четырех процессоров в конфигурации с общей шиной и более четырех при иерархической структуре шин).

В настоящее время фирмой Sun предлагается множество вариантов UltraSparc, как одного из самого успешных процессоров с RISC-архитектурой. Этот процессор можно найти во встраиваемых системах высших моделей, но его основное назначение – высокопроизводительные рабочие станции и серверы. Архитектура процессоров этого семейства использует регистровые окна, а не стек блоков памяти, чем подобна EPIC-архитектуре фирмы Intel.

Последняя модель семейства – UltraSpark IV – первый представитель процессоров, выполненных на основе концепции "организации многопоточной архитектуры на уровне микросхемы" (Chip Multithreading – CMT) в рамках инициативы "производительные вычисления" (Throughput Computing). Процессор UltraSpark IV совместим на уровне двоичных кодов со Spark-процессорами предыдущих поколений и поддерживает 8 и 9 версии ОС Solaris v.8 и 9.

UltraSpark IV содержит двухпроцессорное ядро на базе конвейерной (глубина конвейера каждого ядра – 14 стадий) архитектуры процессора предыдущего поколения UltraSpark III. 16-входной буфер команд выдает конвейеру на выполнение четыре команды/такт, а каждый суперскалярный процессор ядра выполня-

ет четыре команды/такт. Шести исполнительным устройствам параллельного действия (двум целочисленным устройствам, одному выполнения условного перехода, одному устройству загрузки регистров/записи в память и двум функциональным устройствам с плавающей точкой) выдаются на выполнение шесть команд за такт.

L1 кэш каждого ядра содержит 64-Кбайт памяти данных, 32-Кбайт – команд, 32 Кбайт – записи в память и 2 Кбайт – предварительной выборки. Как и все современные 64-бит микропроцессоры, UltraSpark IV имеет внутрисхемный контроллер внешней оперативной памяти (статического ДОЗУ). Нити (процессы), выполняемые UltraSpark IV, совместно используют адресную шину и шину данных для получения доступа к ячейкам L2 кэш (типа СОЗУ) объемом 8 Мбайт, к контроллеру оперативной памяти и соединительной шине типа Sun Fireplane с максимальной пропускной способностью 2,4 Гбайт/с. Максимальная мощность, потребляемая схемой, составляет 108 Вт при напряжении питания 1,35 В и частоте 1,2 ГГц.

Выполнен микропроцессор по 0,13-мкм КМОП-технологии с семислойной медной металлизацией фирмы Texas Instruments (с которой Sun решает проблемы производства разрабатываемых изделий). Он содержит 66 млн. транзисторов. Монтируется в 1368-выводной типа LGA.

Ключ к успеху процессоров семейства UltraSpark – операционная система Solaris фирмы Sun Microsystems. Чтобы в дальнейшем реализовывать все возможности этой ОС на более высоком уровне, необходимо совершенствовать поддерживающее ее "железо". И фирма не отказывается от этой задачи, ставя целью создание процессоров с более радикальной многопоточной архитектурой, более чем в 30 раз превосходящих по быстродействию современные процессоры для корпоративных систем.

Фирмой планируется создание процессора (кодовое название Niagara) с восемью процессорными ядрами на кристалле, одновременно обрабатывающими в целом до 32 нитей. Можно с уверенностью сказать, что компания Sun Microsystems хорошо понимает, что понадобится в ближайшие годы ей и ее заказчикам.

6.4. Alpha 21264 DEC

Развитие МП Alpha сначала шло в направлении первоочередного роста тактовой частоты при относительно простой микроархитектуре МП. Наиболее ярко это выразилось в Alpha 21064. Микроархитектура 21164 сильно усложнилась, а сам МП далеко обогнал всех оппонентов по тактовой частоте и пиковой производительности при работе с вещественной арифметикой. При этом это процессор имеет существенно более простое внутреннее строение, чем суперскалярные микропроцессоры с внеочередным выполнением команд - HP PA-8x00 или SGI/MIPS R10000. Однако, в Alpha 21264 разработчики также обратилась к внеочередному суперскалярному выполнению команд.

Это подтверждает, что наиболее перспективна сложная микроархитектура МП, а не простой суперконвейерный подход, при котором проще поднимать тактовую частоту. Из современных процессоров RISC, если не считать IBM P2SC с сильно отличным от других строением, только UltraSPARC и Alpha 21164 не поддерживают внеочередное выполнение команд. С другой стороны, нельзя сказать, что компания DEC шла по неверному пути, когда стремилась в первую очередь увеличивать тактовую частоту, сохраняя относительно более простую архитектуру МП Alpha. Доказательством этого служит тот факт, что Alpha все это время была лидером производительности.

Микроархитектура Alpha 21264 представлена на рис.20.

Собственно, ключевые особенности, с которыми "играют" разработчики современных высокопроизводительных RISC-процессоров, относятся к трем основным областям: кэш-память; внеочередное выполнение в ФИУ; алгоритмы предсказания переходов.

Очевидно, что при высоких тактовых частотах возникают проблемы как для внутренней, так и для внешней кэш-памяти. Что касается внутреннего кэш, то в Alpha 21164 это привело к созданию уникального интегрированного на чипе двухуровневой кэш, включающей прямоадресуемые I- и D-кэш L1 емкостью 8 Кбайт каждый, плюс 3-канальный частично-ассоциативный кэш L2 емкостью 96 Кбайт.

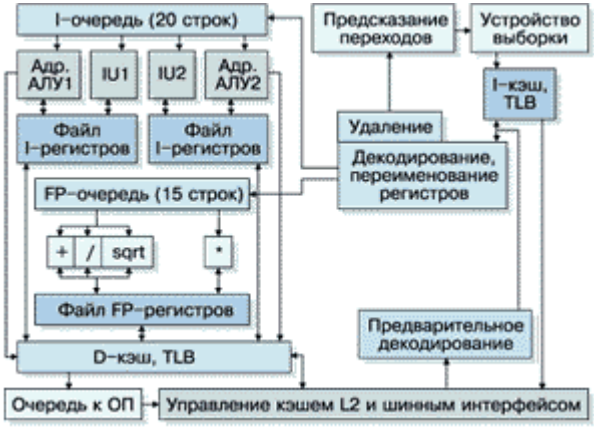


Рис. 20

Подобную конструкцию не повторили ни в одном из других известных микропроцессоров. Размер L1 ограничен требованиями доступа к нему за 2 нс (для 21164/500 МГц). Процент непопаданий в такой кэш относительно велик (следует учесть, что он прямоадресуемый, что несколько увеличивает вероятность непопадания по сравнению с многоканальным частично-ассоциативным кэш), а для обращения в L2 требуется уже 6 тактов. Кроме L1 и L2, для достижения высокой производительности применяется внешний кэш L3 большой емкости.

В Alpha 21264 применены 2-канальные частично-ассоциативные I- и D-кэш емкостью 64 Кбайт каждый. Очевидно, что вероятность попадания в такой кэш гораздо выше. Элиминируется и обмен данными между кэш внутри чипа 21164. Емкость этого кэш в 2 раза больше, чем в SGI/MIPS R10000. Наибольшую емкость кэш на чипе сегодня имеет IBM P2SC - 128 Кбайт данных плюс 32 Кбайт для команд. Однако этот микропроцессор вообще не имеет вторичного кэш.

Микропроцессоры DEC имеют непосредственно взаимодействующие блоки микропроцессора, располагающиеся рядом. D-кэш должен работать как с целочисленными ФИУ и ФИУ с плавающей запятой, так и с системной шиной. За увеличение емкости первичного кэш в 21264 пришлось заплатить увеличением време-

ни доступа до 2 тактов. Двухтактный кэш L1 используется и в HP PA-8000, но в качестве внешнего.

Очевидно, что увеличение в Alpha 21264 по сравнению с Alpha 21164 на 1 такт времени доступа в кэш L1 полностью компенсируется его большей емкостью и одноуровневой организацией. DEC оценивает уменьшение общей пиковой производительности из-за этого дополнительного такта в 4%. Альтернативные подходы привели бы к большему ее снижению.

D-кэш L1 в Alpha 21264 является двухпортовым и позволяет получать 2 независимых 64-разрядных результата за такт. Это достигается путем запуска нового доступа каждые полтакта - D-кэш работает на частоте 1 ГГц (в предположении 500 МГц частоты самого 21264). До 8 непопаданий в кэш L1 чипа 21264 не блокируют его работу.

В отличие от Alpha 21164, в Alpha 21264 внешний кэш L2 использует выделенную (собственную) шину, подобно MIPS R10000, HP PA-8000/8200 или Intel Pentium Pro/Pentium II. Это очень важное усовершенствование. В Alpha 21164 внешний кэш применяет для передачи данных 128-разрядную системную шину, что повышает нагрузку на шину в многопроцессорных системах и может отрицательно сказываться на масштабируемости. Кэш L2 в 21264 может работать на частотах 2/3, 1/2, 1/3 и 1/4 от тактовой частоты процессора, но не выше 333 МГц.

Общей проблемой высокочастотных микропроцессоров на сегодня является отсутствие микросхем SRAM для кэш с большой тактовой частотой. Дешевые системы на базе 21264 будут использовать 133 МГц-кэш SRAM, подобный применяемому в Intel Pentium II/266. Однако пропускная способность L2 в Alpha 21264 в 2 раза выше (до 2.1 Гбайт/с), чем у Pentium II - разрядность шины равна 128 против 64 в Pentium II. К моменту начала поставок Alpha 21264 были выпущены схемы SRAM с частотой 200-250 МГц, чему отвечает пропускная способность кэша 3.2-4.0 Гбайт/с. Отметим, что SGI/MIPS R10000 работает с 200 МГц внешним кэшем как раз на 128-разрядной выделенной шине.

DEC совместно с фирмами Motorola и Samsung участвовала в разработке нового типа SRAM - "dual data". Они аналогичны 5 нс- SRAM, но за 1 такт через выходные шины отсылают данные

сразу в 2 транзакции. Работа на частоте 167 МГц в этом случае эквивалентна 333 МГц-передаче данных. Такой кэш позволяет достигнуть максимальной для 21264 пропускной способности - 5.3 Гбайт/с. Большее значение, 5.8 Гбайт/с, имеет HP PA-8000.

Использование дешевой кэш-памяти с частотой в 1/4 от процессорной понижает производительность при работе с целыми числами на 5%, а с плавающей точкой - не менее чем на 20 %. Задержка при обращении во внешний кэш составляет 12 тактов, а для 133 МГц кэш - 14 тактов. При этом, как и внутренний кэш, кэш L2 является неблокирующимся.

Одновременно с доступом в D-кэш L1 в Alpha 21264 осуществляется преобразование виртуальных адресов в физические с использованием буфера TLB емкостью 128 строк, что в 2 раза больше, чем в SGI/MIPS R10000; HP PA-8000 имеет TLB на 96 строк, PA-8200 - на 120 строк.

В 21264 применено внеочередное выполнение команд (не в соответствии с порядком команд в коде программы) и переименование регистров на лету. Если Alpha 21164 имеет 32 целочисленных регистра и 32 регистра с плавающей запятой, то в 21264 есть 80 целочисленных физических регистров (имеется даже 2 копии этого файла) и 72 физических регистра с плавающей запятой. Внеочередное выполнение команд может иметь место как для очереди целочисленных команд, так и для очереди команд с плавающей запятой.

Чип Alpha 21264 способен выполнять до 6 команд за такт (поддерживаемый уровень - 4 команды за такт). В это число включены также команды загрузки регистров/записи в память. Команды выбираются из I-кэша, куда они попадают частично декодированными. Предварительное декодирование создает 3-разрядное поле, по которому определяется, к какому ФИУ следует направить команду: целочисленному ФИУ общего назначения, (целочисленному) адресному АЛУ или ФИУ с плавающей запятой. Это 3-разрядное поле хранится вместе с командами в I-кэше.

Alpha 21264 имеет 6 ФИУ, из них 4 целочисленных: 2 - общего назначения, и 2 - адресных АЛУ. Последние отвечают за выполнение всех команд загрузки/записи в память и могут также выполнять простые арифметические и логические целочисленные

операции. 2 ФИУ общего назначения производят арифметические и логические операции над целыми числами, операции сдвига и перехода. Одно из них умеет умножать, а другое - выполнять новые команды из мультимедийного расширения набора команд MVI (Motion-Video Instructions).

После окончательного декодирования "целочисленные" команды отправляются в очередь емкостью 20 строк. Простые целочисленные операции, например, сложение, могут выполняться в любых целочисленных ФИУ. С целью упрощения логики уже при постановке в очередь решается, направлять ли команду в целочисленные ФИУ общего назначения или в адресные АЛУ. Каждый такт команды из очереди, операнды которых доступны, арбитрируются на предмет направления в ФИУ. Выборка команд осуществляется не обязательно в соответствии с их расположением в программе.

Одновременно может выполняться 4 операции над целыми числами. Это требует файла целочисленных регистров с 8 портами чтения и 6 портами записи. Вместо этого разработчики по технологическим причинам продублировали файл целочисленных регистров. Каждая копия файла имеет 4 порта чтения и 6 портов записи. Каждому файлу ставится в соответствие свое целочисленное ФИУ общего назначения и АЛУ. Такой комплект оборудования DEC называла "кластером". Файлы регистров остаются синхронизированными, но это требует одного дополнительного такта для записи из АЛУ одного кластера в файл регистров другого кластера. По оценкам DEC, кластеризация понижает Пт в среднем на 1%. Последовательность взаимозависимых команд имеет тенденцию попадать в один и тот же кластер благодаря "сдвигу" доступности операндов на 1 такт.

ФИУ с плавающей запятой в Alpha 21264 всего 2. Их конвейеры имеют длину 4 такта, что больше, чем у HP PA-8000 (3 такта) и MIPS R10000 (2 такта). Представление об основных стадиях конвейеров МП 21264 можно получить из рис. 21. Alpha 21264 может поддерживать одновременное выполнение 4 команд с плавающей запятой за такт в смеси из 50% операций загрузки регистров/записи в память, 25% умножений и 25% сложений. Как и MIPS R10000, Alpha 21264 может выдавать 2 результата с пла-

вающей запятой (FLOP) за такт. Лишь микропроцессоры HP PA-8x00 и IBM Power2/P2SC способны выдавать 4 FLOP за такт. Операции деления и извлечения квадратного корня не конвейеризованы и занимают 16 и 33 такта соответственно.

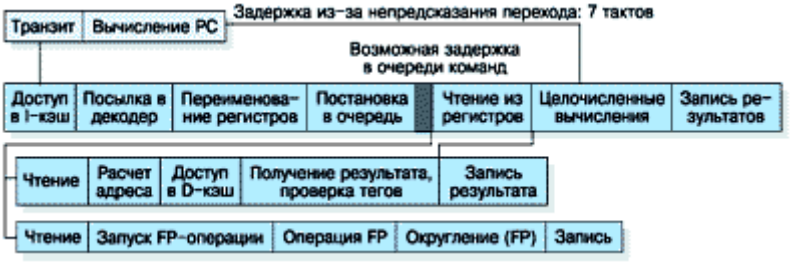


Рис. 21

Важнейшим компонентом современных суперскалярных микропроцессоров является предсказание переходов. Неправильное предсказание заметно ухудшает производительность. Так, средняя задержка при неправильном предсказании в Alpha 21264 составляет свыше 11 тактов. Особенно важным является предсказание переходов для программ, ориентированных на решение коммерческих задач, для которых характерен большой процент команд условного перехода. Алгоритмы предсказания быстро развиваются; одни методы предсказания переходов оказываются точнее других, но разные алгоритмы хорошо работают на разных типах переходов.

Попытка скомбинировать статическое и динамическое предсказания переходов предпринята в HP PA-8500. В Alpha 21264 реализована оригинальная комбинация двух методов динамического предсказания. При этом комбинируются оценки двух таблиц: двухуровневой таблицы "локальных предсказаний" (1024 строки по 10 бит плюс 1024 трехразрядных строки в "сводной" таблице второго уровня), индексом в которой является счетчик команд, и одноуровневой таблицы глобальной истории предсказаний с 2-разрядными полями истории. В третьей таблице ведется "история" обоих предсказаний, на основании которой для каждого

перехода динамически выбирается лучший алгоритм. Эта таблица имеет емкость 4096 двухразрядных полей.

Оценка точности предсказания переходов для тестов SPECint95 составляет порядка 95%. Даже в случае правильно предсказанного перехода в Alpha 21264 из-за двухтактного доступа к I-кэшу приходится предпринимать особые меры по уменьшению задержек, связанных с переходом. Каждая строка в I-кэше содержит 4 команды вместе с дополнительными полями, относящимися к предсказанию, какая строка будет выбираться следующей. Преимуществом 21264 является то, что выборка команд происходит относительно автономно (от работы остальных блоков МП), при этом в декодер направляется по 4 команды за такт.

Подобно многим другим МП, Alpha 21264 имеет стек адресов возврата, применяемый для организации эффективного выхода из вызванных подпрограмм. Этот стек больше, чем в других МП, и позволяет аккуратно отслеживать до 32 уровней вызова подпрограмм.

Минусом микроархитектуры Alpha 21264 является группировка команд в "четверки" и необходимость их соответствующего выравнивания для эффективной обработки переходов. Так, желательно, чтобы адрес перехода указывал на первую команду четверки.

64-разрядная системная шина Alpha 21264 может работать на частотах до 333 МГц, чему соответствует пиковая пропускная способность 2.7 Гбайт/с (поддерживаемое значение - 2.0 Гбайт/с). Это выше, чем в славящихся своей высокой пропускной способностью чипах IBM Power2/P2SC (максимальная пиковая величина - для 77 МГц Power2 - равна 2.5 Гбайт/с). Как и для внешнего кэша, для системной шины могут применяться делители тактовой частоты процессора.

Фактически эта шина осуществляет соединение "точка-точка", что говорит об ориентации на применение коммутаторов вместо традиционной системной шины. Коммутаторы используются в известных многопроцессорных системах: NUMA-системах SGI Origin 2000, серверах серии SPP от Convex/HP и др. Можно предположить, что подобное строение будет применяться и в бу-

дущих мультипроцессорных системах DEC/Compaq с кодовым названием Wildfire, которые должны базироваться на Alpha 21264.

В многопроцессорной системе каждый процессор 21264 имеет свой канал в оперативную память, хотя пропускная способность самой системы памяти разделяется процессорами. Системная шина использует протокол с расщеплением транзакций; до 16 ссылок к памяти могут обрабатываться одновременно. Каждый процессор должен иметь собственное соединение с реализующим интерфейс системной шины набором микросхем "Цунами". Нельзя просто добавить на шину второй процессор - это нарушает связь "точка-точка". Цунами поддерживает соединение с двумя процессорами; возможно создание "4-процессорной" версии.

Набор микросхем Цунами позволяет демультиплексировать 64-разрядные данные шины в 128-, 256- и даже 512-разрядный интерфейс оперативной памяти на SDRAM, а также подсоединять до 2 64-разрядных шин PCI. Ширина тракта к оперативной памяти и число шин PCI определяется количеством используемых микросхем разного типа из набора Цунами.

Alpha 21264 создается на базе 0.35 мкм-технологии и имеет 15.2 млн. Площадь МП - около 302 мм2; рассеяние тепла около 60 ватт при тактовой частоте 500 МГц (около 72 ватт при тактовой частоте 600 МГц

6.5. PA 7100 Hewlett-Packard

Особенностью архитектуры PA 7100 является внекристальная реализация кэш, что позволяет реализовать различные объемы кэш-памяти и оптимизировать конструкцию в зависимости от условий применения (рис. 22). Хранение команд и данных осуществляется в раздельных кэшах, причем процессор соединяется с ними с помощью высокоскоростных 64-битовых шин.

Кэш-память реализуется на высокоскоростных кристаллах статической памяти (SRAM), синхронизация которых осуществляется непосредственно на тактовой частоте процессора. При тактовой частоте 100 МГц каждый кэш имеет полосу пропускания 800 Мбайт/с при выполнении операций считывания и 400 Мбайт/с при выполнении операций записи.

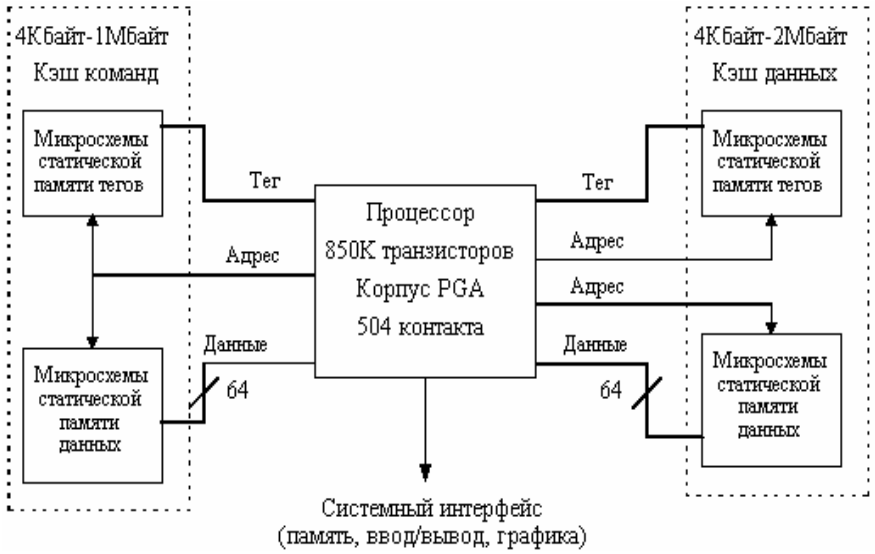


Рис. 22

Микропроцессор аппаратно поддерживает различный объем кэш-памяти: кэш команд может иметь объем от 4 Кбайт до 1 Мбайт, кэш данных - от 4 Кбайт до 2 Мбайт. Чтобы снизить коэффициент промахов применяется механизм хеширования адреса. В обоих кэшах для повышения надежности применяются дополнительные контрольные разряды, причем ошибки кэша команд корректируются аппаратными средствами.

Процессор подсоединяется к памяти и подсистеме ввода/вывода посредством синхронной шины. Процессор может работать с тремя разными отношениями внутренней и внешней тактовой частоты в зависимости от частоты внешней шины: 1:1, 3:2 и 2:1. Это позволяет использовать в системах разные по скорости микросхемы памяти.

Конструктивно на кристалле PA-7100 размещены целочисленный процессор, процессор для обработки чисел с плавающей точкой, устройство управления кэш-памятью, унифицированный буфер TLB, устройство управления, а также ряд интерфейсных схем. Целочисленный процессор включает АЛУ, устройство сдви-

га, сумматор команд перехода, схемы проверки кодов условий, схемы обхода, универсальный регистровый файл, регистры управления и регистры адресного конвейера.

Устройство управления кэш-памятью содержит регистры, обеспечивающие перезагрузку кэш-памяти при возникновении промахов и контроль когерентного состояния памяти. Это устройство содержит также адресные регистры сегментов, буфер преобразования адреса TLB и аппаратуру хеширования, управляющую перезагрузкой TLB.

В состав процессора плавающей точки входят устройство умножения, арифметико-логическое устройство, устройство деления и извлечения квадратного корня, регистровый файл и схемы "закоротки" результата. Интерфейсные устройства включают все необходимые схемы для связи с кэш-памятью команд и данных, а также с шиной данных. Обобщенный буфер TLB содержит 120 строк ассоциативной памяти фиксированного размера и 16 строк переменного размера.

Устройство плавающей точки (рис. 23) реализует арифметику с одинарной и двойной точностью в стандарте IEEE 754.

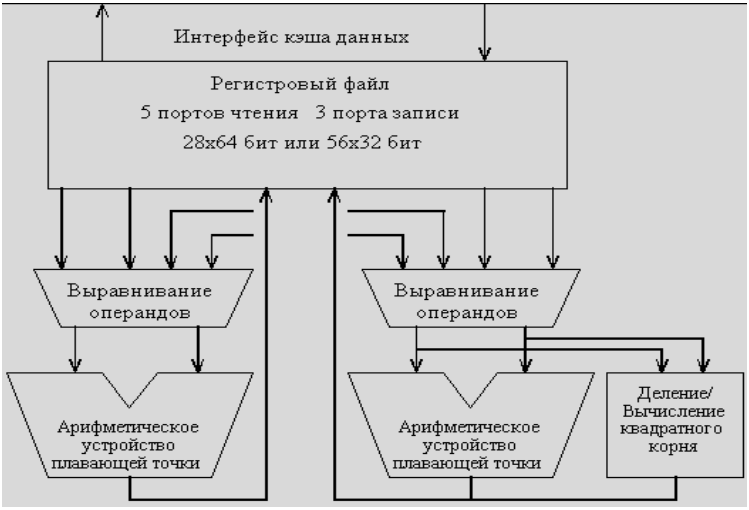


Рис. 23

Его устройство умножения используется также для выполнения операций целочисленного умножения. Устройства деления и вычисления квадратного корня работают с удвоенной частотой процессора. Арифметико-логическое устройство выполняет операции сложения, вычитания и преобразования форматов данных. Регистровый файл состоит из 28 64-битовых регистров, каждый из которых может использоваться как два 32-битовых регистра для выполнения операций с плавающей точкой одинарной точности. Регистровый файл имеет пять портов чтения и три порта записи, которые обеспечивают одновременное выполнение операций умножения, сложения и загрузки/записи.

Большинство улучшений производительности процессора связано с увеличением тактовой частоты до 100 МГц по сравнению с 66 МГц у его предшественника.

Конвейер целочисленного устройства включает шесть ступеней: чтение из кэша команд (IR), чтение операндов (OR), выполнение/чтение из кэша данных (DR), завершение чтения кэша данных (DRC), запись в регистры (RW) и запись в кэш данных (DW). На ступени ID выполняется выборка команд.

Реализация механизма выдачи двух команд требует небольшого буфера предварительной выборки, который обеспечивает предварительную выборку команд за два такта до начала работы ступени IR. Во время выполнения на ступени OR все исполнительные устройства декодируют поля операндов в команде и начинают вычислять результат операции. На ступени DR целочисленное устройство завершает свою работу. Кроме того, кэш-память данных выполняет чтение, но данные не поступают до момента завершения работы ступени DRC.

Результаты операций сложения (ADD) и умножения (MULTIPLY) также становятся достоверными в конце ступени DRC. Запись в универсальные регистры и регистры плавающей точки производится на ступени RW. Запись в кэш данных командами записи (STORE) требует двух тактов. Наиболее раннее двухтактное окно команды STORE возникает на ступенях RW и DW. Однако это окно может сдвигаться, поскольку записи в кэш данных происходят только когда появляется следующая команда записи. Операции деления и вычисления квадратного корня для чи-

сел с плавающей точкой заканчиваются на много тактов позже ступени DW.

Конвейер проектировался с целью максимального увеличения времени, необходимого для выполнения чтения внешних кристаллов SRAM кэш-памяти данных. Это позволяет максимизировать частоту процессора при заданной скорости SRAM. Все команды загрузки (LOAD) выполняются за один такт и требуют только одного такта полосы пропускания кэш-памяти данных. Поскольку кэши команд и данных размещены на разных шинах, в конвейере отсутствуют какие-либо потери, связанные с конфликтами по обращениям в кэш данных и кэш команд.

Процессор может в каждом такте выдавать на выполнение одну целочисленную команду и одну команду плавающей точки. Полоса пропускания кэша команд достаточна для поддержания непрерывной выдачи двух команд в каждом такте. Отсутствуют какие-либо ограничения по выравниванию или порядку следования пары команд, которые выполняются вместе. Кроме того, отсутствуют потери тактов, связанных с переключением с выполнения двух команд на выполнение одной команды. Специальное внимание было уделено тому, чтобы выдача двух команд в одном такте не приводила к ограничению тактовой частоты. Чтобы добиться этого, в кэше команд был реализован специально предназначенный для этого заранее декодируемый бит, чтобы отделить команды целочисленного устройства от команд устройства плавающей точки. Этот бит предварительного декодирования команд минимизирует время, необходимое для правильного разделения команд.

Потери, связанные с зависимостями по данным и управлению, в этом конвейере минимальны. Команды загрузки выполняются за один такт, за исключением случая, когда последующая команда пользуется регистром-приемником команды LOAD. Как правило, компилятор позволяет обойти подобные потери одного такта. Для уменьшения потерь, связанных с командами условного перехода, в процессоре используется алгоритм прогнозирования направления передачи управления. Для оптимизации производительности циклов передачи управления вперед по программе прогнозируются как невыполняемые переходы, а передачи управле-

ния назад по программе - как выполняемые переходы. Правильно спрогнозированные условные переходы выполняются за один такт.

Количество тактов, необходимое для записи слова или двойного слова командой STORE, уменьшено с трех до двух тактов. В более ранних реализациях архитектуры PA-RISC был необходим один дополнительный такт для чтения тега кэш, чтобы гарантировать попадание, а также для того, чтобы объединить старые данные строки кэш-памяти данных с записываемыми данными. PA 7100 использует отдельную шину адресного тега, чтобы совместить по времени чтение тега с записью данных предыдущей команды STORE. Кроме того, наличие отдельных сигналов разрешения записи для каждого слова строки кэш-памяти устраняет необходимость объединения старых данных с новыми, поступающими при выполнении команд записи слова или двойного слова.

Этот алгоритм требует, чтобы запись в микросхемы SRAM происходила только после того, когда будет определено, что данная запись сопровождается попаданием в кэш и не вызывает прерывания. Это требует дополнительной ступени конвейера между чтением тега и записью данных. Такая конвейеризация не приводит к дополнительным потерям тактов, поскольку в процессоре реализованы специальные цепи обхода, позволяющие направить отложенные данные команды записи последующим командам загрузки или командам STORE, записывающим только часть слова. Для данного процессора потери конвейера для команд записи слова или двойного слова сведены к нулю, если непосредственно последующая команда не является командой загрузки или записи. В противном случае потери равны одному такту. Потери на запись части слова могут составлять от нуля до двух тактов.

Моделирование показывает, что подавляющее большинство команд записи в действительности работают с однословным или двухсловным форматом.

Все операции с плавающей точкой, за исключением команд деления и вычисления квадратного корня, полностью конвейеризованы и имеют двухтактную задержку выполнения как в режиме с одинарной, так и с двойной точностью. Процессор может выдавать на выполнение независимые команды с плавающей точкой в

каждом такте при отсутствии каких-либо потерь. Последовательные операции с зависимостями по регистрам приводят к потере одного такта. Команды деления и вычисления квадратного корня выполняются за 8 тактов при одиночной и за 15 тактов при двойной точности. Выполнение команд не останавливается из-за команд деления/вычисления квадратного корня до тех пор, пока не потребуется регистр результата или не будет выдаваться следующая команда деления/вычисления квадратного корня.

Процессор может выполнять параллельно одну целочисленную команду и одну команду с плавающей точкой. При этом "целочисленными командами" считаются и команды загрузки и записи регистров плавающей точки, а "команды плавающей точки" включают команды FMPYADD и FMPYSUB. Эти последние команды объединяют операцию умножения с операциями сложения или вычитания соответственно, которые выполняются параллельно. Пиковая производительность составляет 200 MFLOPS для последовательности команд FMPYADD, в которых смежные команды независимы по регистрам.

Потери для операций плавающей точки, использующих предварительную загрузку операнда командой LOAD, составляют один такт, если команды загрузки и плавающей арифметики являются смежными, и два такта, если они выдаются для выполнения одновременно. Для команды записи, использующей результат операции с плавающей точкой, потери отсутствуют, даже если они выполняются параллельно.

Потери, возникающие при промахах в кэш данных, минимизируются посредством применения четырех разных методов: "попадание при промахе" для команд LOAD и STORE, потоковый режим работы с кэшем данных, специальная кодировка команд записи, позволяющая избежать копирования строки, в которой произошел промах, и семафорные операции в кэш-памяти. Первое свойство позволяет во время обработки промаха в кэш данных выполнять любые типы других команд. Для промахов, возникающих при выполнении команды LOAD, обработка последующих команд может продолжаться до тех пор, пока регистр результата команды LOAD не потребуется в качестве регистра операнда для другой команды.

Компилятор может использовать это свойство для предварительной выборки в кэш необходимых данных задолго до того момента, когда они действительно потребуются. Для промахов, возникающих при выполнении команды STORE, обработка последующих команд загрузки или операций записи в части одного слова продолжается до тех пор, пока не возникает обращений к строке, в которой произошел промах. Компилятор может использовать это свойство для выполнения команд на фоне записи результатов предыдущих вычислений. Во время задержки, связанной с обработкой промаха, другие команды LOAD и STORE, для которых происходит попадание в кэш данных, могут выполняться и другие команды целочисленной арифметики и плавающей точки. В течение всего времени обработки промаха команды STORE, другие команды записи в ту же строку кэш-памяти могут происходить без дополнительных потерь времени. Для каждого слова в строке кэш-памяти процессор имеет специальный индикационный бит, предотвращающий копирование из памяти тех слов строки, которые были записаны командами STORE. Эта возможность применяется к целочисленным и плавающим операциям LOAD и STORE.

Выполнение команд останавливается, когда регистр-приемник команды LOAD, выполняющейся с промахом, требуется в качестве операнда другой команды. Свойство "потоковости" позволяет продолжить выполнение, как только нужное слово или двойное слово возвращается из памяти. Таким образом, выполнение команд может продолжаться как во время задержки, связанной с обработкой промаха, так и во время заполнения соответствующей строки при промахе.

При выполнении блочного копирования данных в ряде случаев компилятор заранее знает, что запись должна осуществляться в полную строку кэш-памяти. Для оптимизации обработки таких ситуаций архитектура определяет специальную кодировку команд записи ("блочное копирование"), которая показывает, что аппаратуре не нужно осуществлять выборку из памяти строки, при обращении к которой может произойти промах кэш-памяти. В этом случае время обращения к кэшу данных складывается из времени, которое требуется для копирования в память старой строки кэш-памяти по тому же адресу в кэш (если он "грязный") и

времени, необходимого для записи нового тега кэш. В процессоре PA 7100 такая возможность реализована как для привилегированных, так и для непривилегированных команд.

Последнее улучшение управления кэш-памятью данных связано с реализацией семафорных операций "загрузки с обнулением" непосредственно в кэш-памяти. Если семафорная операция выполняется в кэш, то потери времени при ее выполнении не превышают потерь обычных операций записи. Это не только сокращает конвейерные потери, но и снижает трафик шины памяти. В архитектуре предусмотрен также другой тип специального кодирования команд, который устраняет требование синхронизации семафорных операций с устройствами ввода/вывода.

Управление кэш-памятью команд позволяет при промахе продолжить выполнение команд сразу же после поступления отсутствующей в кэше команды из памяти. 64-битовая магистраль данных, используемая для заполнения блоков кэша команд, соответствует максимальной полосе пропускания внешней шины памяти 400 Мбайт/с при тактовой частоте 100 МГц.

В процессоре предусмотрен также ряд мер по минимизации потерь, связанных с преобразованиями виртуальных адресов в физические.

Конструкция процессора обеспечивает реализацию двух способов построения многопроцессорных систем. При первом способе каждый процессор подсоединяется к интерфейсному кристаллу, который наблюдает за всеми транзакциями на шине основной памяти. В такой системе все функции по поддержанию когерентного состояния кэш-памяти возложены на интерфейсный кристалл, который посылает процессору соответствующие транзакции. Кэш данных построен на принципах отложенного обратного копирования и для каждого блока кэш-памяти поддерживаются биты состояния "частный" (private), "грязный" (dirty) и "достоверный" (valid), значения которых меняются в соответствии с транзакциями, которые выдает или принимает процессор.

Второй способ организации многопроцессорной системы позволяет объединить два процессора и контроллер памяти и ввода-вывода на одной и той же локальной шине памяти. В такой конфигурации не требуется дополнительных интерфейсных кри-

сталлов и она совместима с существующей системой памяти. Когерентность кэш-памяти обеспечивается наблюдением за локальной шиной памяти. Пересылки строк между кэшами выполняются без участия контроллера памяти и ввода-вывода. Такая конфигурация обеспечивает возможность построения очень дешевых высокопроизводительных многопроцессорных систем.

Процессор поддерживает ряд операций, необходимых для улучшения графической производительности рабочих станций серии 700: блочные пересылки, Z-буфери-зацию, интерполяцию цветов и команды пересылки данных с плавающей точкой для обмена с пространством ввода/вывода.

Процессор построен на базе технологического процесса КМОП с проектными нормами 0.8 микрон, что обеспечивает тактовую частоту 100 МГц.

6.6. R12000 MIPS

Организация памяти микропроцессора R12000. MIPS R12000 использует два уровня кэш-памяти. Первый уровень наборно ассоциативной кэш-памяти, расположенной на кристалле, имеет 32КБ для данных и 32КБ для команд. Второй уровень - внешняя кэш-память объединена для данных и для команд и может иметь размер от 512КБ до 16МБ**.**

При разработке процессора R12000 большое внимание было уделено эффективной реализации иерархии памяти. В данном чипе обеспечиваются раннее обнаружение промахов кэш-памяти и параллельная перезагрузка строк с выполнением другой полезной работой. Реализованные на кристалле кэш-памяти поддерживают одновременную выборку команд, выполнение команд загрузки и записи данных в память, а также операций перезагрузки строк кэш-памяти. Заполнение строк кэш-памяти выполняется по принципу "запрошенное слово первым", что позволяет существенно сократить простои процессора из-за ожидания требуемой информации. Все кэш имеют двухканальную множественно ассоциативную организацию с алгоритмом замещения LRU.

Кэш-память данных первого уровня процессора R12000 имеет емкость 32 Кбайт и организована в виде двух одинаковых

банков емкостью по 16 Кбайт, что обеспечивает двукратное расслоение при выполнении обращений к этой кэш-памяти. Каждый банк представляет собой двухканальную множественно-ассоциативную кэш-память с размером строки (блока) в 32 байта. Кэш данных индексируется с помощью виртуального адреса и хранит теги физических адресов памяти. Такой метод индексации позволяет выбрать подмножество кэш-памяти в том же такте, в котором формируется виртуальный адрес. Однако для того, чтобы поддерживать когерентность с кэш-памятью второго уровня, в L1 хранятся теги физических адресов памяти.

Массивы данных и тегов в каждом банке являются независимыми. Эти четыре массива работают под общим управлением очереди формирования адресов памяти и схем внешнего интерфейса кристалла. В очереди адресов могут одновременно находиться до 16 команд загрузки и записи, которые обрабатываются в четырех отдельных конвейерах. Команды из этой очереди динамически подаются для выполнения в специальный конвейер, который обеспечивает вычисление исполнительного виртуального адреса и преобразование этого адреса в физический. Три других параллельно работающих конвейера могут одновременно выполнять проверку тегов, осуществлять пересылку данных для команд загрузки и завершать выполнение команд записи в память. Хотя команды выполняются в строгом порядке их расположения в памяти, вычисление адресов и пересылка данных для команд загрузки могут происходить неупорядоченно. Схемы внешнего интерфейса кристалла могут производить заполнение или обратное копирование строк кэш-памяти, либо операции просмотра тегов. Такая параллельная работа большинства устройств процессора позволяет процессору R12000 эффективно выполнять реальные многопроцессорные приложения.

Работа конвейеров кэш-памяти данных тесно координирована. Например, команды загрузки могут выполнять проверку тегов и чтение данных в том же такте, что и преобразование адреса. Команды записи сразу же начинают проверку тегов, чтобы, в случае необходимости, как можно раньше инициировать заполнение требуемой строки изL2, но непосредственная запись данных в кэш задерживается до тех пор, пока сама команда записи не станет са-

мой старой командой в общей очереди выполняемых команд и ей не будет позволено зафиксировать свой результат. Промах при обращении к L1 данных инициирует процесс заполнения строки из кэш-памяти второго уровня. При выполнении команд загрузки одновременно с заполнением строки кэш-памяти данные могут поступать по цепям обхода в регистровый файл.

При обнаружении промаха при обращении к кэш-памяти данных ее работа не блокируется, т.е. она может продолжать обслуживание следующих запросов. Это особенно полезно для уменьшения такого важного показателя качества реализованной архитектуры как среднее число тактов на команду (CPI - clock cycles per instruction). Эффект применения неблокируемой кэш-памяти сильно зависит от характеристик самих программ. Для небольших тестов, рабочие наборы которых полностью помещаются в L1, этот эффект не велик. Однако для более реальных программ выигрыш оказывается существенным.

Интерфейс L2 процессора R12000 поддерживает 128-разрядную магистраль данных, которая может работать с тактовой частотой до 300 МГц. Все стандартные синхронные сигналы управления статической памятью вырабатываются внутри процессора. Не требуется никаких внешних интерфейсных схем. Минимальный объем L2 составляет 512 Кбайт, максимальный размер - 16 Мбайт. Размер строки этой кэш-памяти программируется и может составлять 128 или 1024 байт.

Одним из методов улучшения временных показателей работы кэш-памяти является построение псевдо-множественно-ассоциативной кэш-памяти. В такой кэш-памяти частота промахов находится на уровне частоты промахов множественно-ассоциативной памяти, а время выборки при попадании соответствует кэш-памяти с прямым отображением.

Кэш-память R12000 организована именно таким способом, причем для ее реализации используются стандартные синхронные микросхемы памяти SRAM. В одном наборе микросхем памяти находятся оба канала кэша. Информация о частоте использования этих каналов хранится в схемах управления кэшем на процессорном кристалле. Поэтому после обнаружения промаха в первичном

кэше из наиболее часто используемого канала вторичного кэша считываются две четырехсловные строки.

Их теги считываются вместе с первой четырехсловной строкой, а теги альтернативного канала читаются одновременно со второй четырехсловной строкой - это осуществляется простым инвертированием старшего разряда адреса. При этом возможны три случая:

- если происходит попадание по первому каналу, то данные доступны немедленно,
- если попадание происходит по альтернативному каналу, выполняется повторное чтение вторичного КЭШа,
- если отсутствует попадание по обоим каналам, вторичный кэш должен перезаполняться из основной памяти.

Для обеспечения целостности данных в кэш-памяти большой емкости обычной практикой является использование кодов, исправляющих одиночные ошибки (ECC-кодов). В R12000 с каждой четырехсловной строкой хранится 9-разрядный ECC-код и бит четности. Дополнительный бит четности позволяет сократить задержку (поскольку проверка на четность может быть выполнена очень быстро), чтобы предотвратить использование некорректных данных. При этом, если обнаруживается корректируемая ошибка, то чтение повторяется через специальный двухтактный конвейер коррекции ошибок.

Объем внутренней двухканальной множественно-ассоциативной кэш-памяти команд составляет 32 Кбайт. В процессе ее загрузки команды частично декодируются. При этом к каждой команде добавляются 4 дополнительных бита, указывающих на исполнительное устройство, в котором будет выполняться команда. Таким образом, в кэш-памяти команды хранятся в 36-разрядном формате.

6.7. PowerPC 970 IBM

64-битные процессоры семейства PowerPC фирмы IBM находят широкое применение – от устройств для серверов старших моделей (Power4 и Power5) до встраиваемых процессоров (PowerPC 970 с высокопроизводительной векторной Altivec*

SIMD-поддержкой). Это совсем неплохо для архитектуры, совместимой с 32-бит процессорами семейства. Компания IBM пошла по пути "мультипроцессирования на уровне микросхемы" (Chip Level Multiprocessing – CMP). Нестандартная любопытная суперскалярная RISC-архитектура таких процессоров позволяет выдавать на выполнение до 8,5 команд/такт. Не усложняя конструкции процессора, на одном кристалле можно разместить два процессорных ядра с более чем десяток исполнительных устройств. Архитектура процессоров PowerPC оперирует более мощными и гибкими SIMD-командами, чем другие устройства, предназначенные для мультимедийной обработки. Altivec SIMD-поддержка обеспечивает еще большую универсальность, предоставляя возможность использовать 32 специализированных 128-бит векторных регистра, четыре регистровых операнда, 162 векторные команды, а также выполнение параллельных скалярных операций с плавающей точкой.

Процессор серии PowerPC 970, впервые представленный в 2002 году, достаточно дешевое 64-бит воплощение процессора Power4, предназначенное для серверов старших моделей. Хотя уменьшение размеров изделия редко позволяет рассчитывать на получение высоких характеристик, модернизация 64-бит микропроцессора Power4 и добавление векторной Altivec-поддержки позволили создать впечатляющий микропроцессор для серверов, графических рабочих станций, настольных компьютеров.

На кристалле Power4 площадью 415 мм^2, содержащем 170 млн. транзисторов, расположены два процессорных ядра на тактовую частоту 1,3 ГГц, L2 кэш емкостью 1,5 Мбит, контроллер L3 кэша и контроллер межчиповой связи, позволяющий размещать четыре Power4-чипа в одном 5184-выводном многокристальном модуле размером 85x85 мм. Четыре таких модуля могут быть объединены без связующих логических схем. Получаемая в результате 32-канальная микропроцессорная субсистема имеет более 20 тыс. контактных площадок ввода-вывода и рассеивает ~2 кВт.

Разработчики IBM оставили одно процессорное ядро процессора Power4, удалив контроллер L3 кэш, сложный межчиповый контроллер и добавив Altivec-расширение. Процессор PowerPC 970, выполненный по 0,13-мкм КМОП-технологии на КНИ-

подложке с восьмислойной медной металлизацией, содержит 52 млн. транзисторов и размещается на кристалле площадью 118 мм^2. Он имеет два блока L1 с контролем по четности (команд и данных, емкостью 64К и 32 Кбит, соответственно) и L2 с корректировкой ошибок емкостью 512 Кбит. Монтируется он в 576-выводной керамический BGA-корпус размером 25х25 мм.

В результате разработчики получили высокопроизводительный, более традиционный 64-битный микропроцессор с одним ядром и малой потребляемой мощностью (19 Вт при напряжении питания 1,1 В и тактовой частоте 1,2 ГГц и 42 Вт при напряжении 1,3 В и частоте 1,8 ГГц), не требующий сборки в дорогой многокристальный модуль. Наряду с этим, у него более глубокий конвейер (16 ступеней), предусмотрены динамическое предсказание переходов, шина с высокой пропускной способностью (до 7,1 Гбайт/с) и достаточное логическое обрамление, что делает его предпочтительным при выборе процессора SMP-системы.

С 2004 года IBM объявила о массовом производстве процессора PowerPC 970FX по 90-нм КНИ-технологии в сочетании с технологиями напряженного кремния и восьмислойных медных соединений. В микросхеме использован новый метод сложной настройки и управления тактовой частотой и пороговым напряжением транзисторов, позволяющий регулировать эти параметры с шагом 0,5 МГц и 0,5 мВ, соответственно. В результате можно получать четыре различных значения порогового напряжения транзисторов с тонким затворным окислом и еще два значения напряжения для транзисторов с толстым окисным слоем. Это привело к снижению потребляемой мощности процессора до 15 Вт.

PowerPC 970FX находит самое широкое применение – от настольных компьютеров до серверов, накопителей и связных систем. Компания Apple уже сообщила о намерении использовать его в новом стоечном сервере Xserve G5 1U. Микропроцессор PowerPC 970FX уже получил премию аналитиков Microprocessor Report как лучший процессор настольных машин, обойдя Pentium 4 фирмы Intel и Athlon 64 FX-51 компании AMD.

7. Микропроцессоры нетрадиционных архитектур

7.1. Ассоциативные процессоры

Существующие в настоящее время алгоритмы прикладных задач, системное программное обеспечение и аппаратные средства преимущественно ориентированы на традиционную адресную обработку данных. Данные должны быть представлены в виде ограниченного количества форматов (например, массивы, списки, записи), должна быть явно создана структура связей между элементами данных посредством указателей на адреса элементов памяти, при обработке этих данных должна быть выполнена совокупность операций, обеспечивающих доступ к данным по указателям. Такой подход обуславливает громоздкость операционных систем и систем программирования, а также служит препятствием к созданию вычислительных средств с архитектурой, ориентированной на более эффективное использование параллелизма обработки данных.

Ассоциативный способ обработки данных позволяет преодолеть многие ограничения, присущие адресному доступу к памяти, за счет задания некоторого критерия отбора и проведение требуемых преобразований, только над теми данными, которые удовлетворяют этому критерию. Критерием отбора может быть совпадение с любым элементом данных, достаточным для выделения искомых данных из всех данных. Поиск данных может происходить по фрагменту, имеющему большую или меньшую корреляцию с заданным элементом данных.

Исследованы и в разной степени используются несколько подходов, различающихся полнотой реализации модели ассоциативной обработки. Если реализуется только ассоциативная выборка данных с последующим поочередным использованием найденных данных, то говорят об ассоциативной памяти или памяти, адресуемой по содержимому. При достаточно полной реализации всех свойств ассоциативной обработки, используется термин «ассоциативный процессор».

Ассоциативные системы относятся к классу: один поток команд - множество потоков данных (SIMD = Single Instruction Multiple Data). Эти системы включают большое число операцион-

ных устройств, способных одновременно по командам управляющего устройства вести обработку нескольких потоков данных. В ассоциативных вычислительных системах информация на обработку поступает от ассоциативных запоминающих устройств (АЗУ), характеризующиеся тем, что информация в них выбирается не по определенному адресу, а по ее содержанию.

7.2. Матричные процессоры

Наиболее распространенными из систем, класса: один поток команд - множество - потоков данных (SIMD), являются матричные системы, которые лучше всего приспособлены для решения задач, характеризующихся параллелизмом независимых объектов или данных. Организация систем подобного типа на первый взгляд достаточно проста. Они имеют общее управляющее устройство, генерирующее поток команд и большое число процессорных элементов, работающих параллельно и обрабатывающих каждая свой поток данных.

Таким образом, производительность системы оказывается равной сумме производительностей всех процессорных элементов. Однако на практике, чтобы обеспечить достаточную эффективность системы при решении широкого круга задач необходимо организовать связи между процессорными элементами с тем, чтобы наиболее полно загрузить их работой. Именно характер связей между процессорными элементами и определяет разные свойства системы.

Одним из первых матричных процессоров был SOLOMON (60-е годы) (рис. 24). Система SOLOMON содержит 1024 процессорных элемента, соединены в виде матрицы: 32x32. Каждый процессорный элемент матрицы включает в себя процессор, обеспечивающий выполнение последовательных поразрядных арифметических и логических операций, а также оперативное ЗУ, емкостью 16 Кбайт. Длина слова - переменная от 1 до 128 разрядов. Разрядность слов устанавливается программно. По каналам связи от устройства управления передаются команды и общие константы.

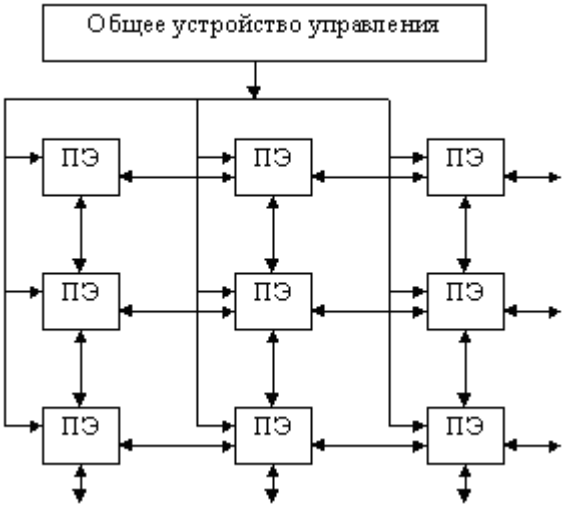


Рис. 24

В процессорном элементе используется, так называемая, многомодальная логика, которая позволяет каждому процессорному элементу выполнять или не выполнять общую операцию в зависимости от значений обрабатываемых данных. В каждый момент все активные процессорные элементы выполняют одну и ту же операцию над данными, хранящимися в собственной памяти и имеющими один и тот же адрес.

Идея многомодальности заключается в том, что в каждом процессорном элементе имеется специальный регистр на 4 состояния - регистр моды. Мода (модальность) заносится в этот регистр от устройства управления. При выполнении последовательности команд модальность передается в коде операции и сравнивается с содержимым регистра моды. Если есть совпадения, то операция выполняется. В других случаях процессорный элемент не выполняет операцию, но может, в зависимости от кода, пересылать свои операнды соседнему процессорному элементу. Такой механизм позволяет выделить строку или столбец процессорных элементов, что очень полезно при операциях над матрицами. Взаимодействуют процессорные элементы с периферийным оборудованием через внешний процессор.

Дальнейшим развитием матричных процессоров стала система ILLIAS-4, разработанная фирмой Barroys. Первоначально система должна была включать в себя 256 процессорных элементов, разбитых на группы, каждый из которых должен управляться специальным процессором. Однако по различным причинам была создана система, содержащая одну группу процессорных элементов и управляющий процессор. Если в начале предполагалось достичь быстродействия порядка 1 млрд. операций в секунду, то реальная система работала с быстродействием около 200 млн. операций в секунду. Эта система в течение ряда лет считалась одной из самых высокопроизводительных в мире.

7.3. ДНК процессоры

В настоящее время в поисках реальной альтернативы полупроводниковым технологиям создания новых вычислительных систем ученые обращают все большее внимание на биотехнологии, или биокомпьютинг, который представляет собой гибрид информационных, молекулярных технологий, также биохимии. Биокомпьютинг позволяет решать сложные вычислительные задачи, пользуясь методами, принятыми в биохимии и молекулярной биологии, организуя вычисления при помощи живых тканей, клеток, вирусов и биомолекул.

Наибольшее распространение получил подход, где в качестве основного элемента (процессора) используются молекулы дезоксирибонуклеиновой кислоты. Центральное место в этом подходе занимает так называемый ДНК - процессор. Кроме ДНК в качестве био-процессора могут быть использованы также белковые молекулы и биологические мембраны.

Так же, как и любой другой процессор, ДНК процессор характеризуется структурой и набором команд. В нашем случае структура процессора - это структура молекулы ДНК. А набор команд - это перечень биохимических операций с молекулами.

Принцип устройства компьютерной ДНК-памяти основан на последовательном соединении четырех нуклеотидов (основных кирпичиков ДНК-цепи). Три нуклеотида, соединяясь в любой последовательности, образуют элементарную ячейку памяти - кодон, которые затем формируют цепь ДНК. Основная трудность в разработке ДНК-компьютеров связана с проведением избирательных однокодонных реакций (взаимодействий) внутри цепи ДНК. Однако прогресс есть уже и в этом направлении. Уже есть экспериментальное оборудование, позволяющее работать с одним из 1020 кодонов или молекул ДНК. Другой проблемой является самосборка ДНК, приводящая к потере информации. Ее преодолевают введением в клетку специальных ингибиторов - веществ, предотвращающих химическую реакцию самосшивки.

Использование молекул DNA для организации вычислений – это не слишком новая идея. Теоретическое обоснование подобной возможности было сделано еще в 50-х годах прошлого века (Р.П. Фейманом). В деталях эта теория была проработана в 70-х годах Ч. Бенеттом и в 80-х М. Конрадом.

Первый компьютер на базе ДНК был создан еще в 1994 г. американским ученым Леонардом Адлеманом. Он смешал в пробирке молекулу ДНК, в которой были закодированы исходные данные, и специальным образом подобранные ферменты. В результате химической реакции структура ДНК изменилась таким образом, что в ней в закодированном виде был представлен ответ задачи. Поскольку вычисления проводились в ходе химической реакции с участием ферментов, на них было затрачено очень мало времени.

Ричард Липтон из Принстона первым показал, как, используя ДНК, кодировать двоичные числа и решать проблему удовлетворения логического выражения. Суть ее в том, что, имея некоторое логическое выражение, включающее n логических переменных, нужно найти все комбинации значений переменных, делающих выражение истинным. Задачу можно решить только перебором $2n$ комбинаций. Все эти комбинации легко закодировать с помощью ДНК, а дальше действовать по методике Адлемана.

Первую модель биокомпьютера, правда, в виде механизма из пластмассы, в 1999 г. создал Ихуд Шапиро из Вейцмановского

института естественных наук. Она имитировала работу “молекулярной машины” в живой клетке, собирающей белковые молекулы по информации с ДНК, используя РНК в качестве посредника между ДНК и белком.

А в 2001 г. Шапиро удалось реализовать вычислительное устройство на основе ДНК, которое может работать почти без вмешательства человека. Система имитирует машину Тьюринга — одну из фундаментальных концепций вычислительной техники. Машина Тьюринга шаг за шагом считывает данные и в зависимости от их значений принимает решения о дальнейших действиях. Теоретически она может решить любую вычислительную задачу. По своей природе молекулы ДНК работают аналогичным образом, распадаясь и рекомбинируя в соответствии с информацией, закодированной в цепочках химических соединений.

Разработанная в Вейцмановском институте установка кодирует входные данные и программы в состоящих из двух цепей молекулах ДНК и смешивает их с двумя ферментами. Молекулы фермента выполняли роль аппаратного, а молекулы ДНК - программного обеспечения. Один фермент расщепляет молекулу ДНК с входными данными на отрезки разной длины в зависимости от содержащегося в ней кода. А другой рекомбинирует эти отрезки в соответствии с их кодом и кодом молекулы ДНК с программой. Процесс продолжается вдоль входной цепи, и, когда доходит до конца, получается выходная молекула, соответствующая конечному состоянию системы.

Этот механизм может использоваться для решения самых разных задач. Хотя на уровне отдельных молекул обработка ДНК происходит медленно - с типичной скоростью от 500 до 1000 бит/с, что во много миллионов раз медленнее современных кремниевых процессоров, по своей природе она допускает массовый параллелизм. По оценкам Шапиро и его коллег, в одной пробирке может одновременно происходить триллион процессов, так что при потребляемой мощности в единицы нановатт может выполняться миллиард операций в секунду.

В 2002 г. фирма Olympus Optical разработала версию ДНК-компьютера, предназначенного для генетического анализа. Он имеет молекулярную и электронную составляющие. Первая осу-

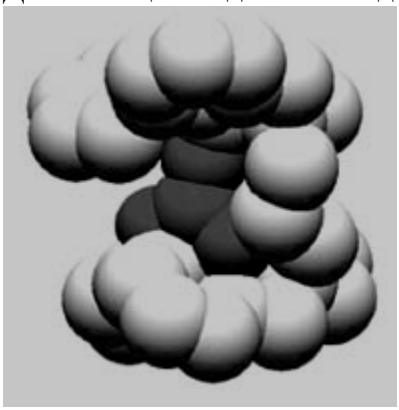
ществляет химические реакции между молекулами ДНК, обеспечивает поиск и выделение результата вычислений. Вторая - обрабатывает информацию и анализирует полученные результаты.

Возможностями биокомпьютеров заинтересовались и военные. Американское агентство по исследованиям в области обороны DARPA выполняет проект, получивший название Bio-Comp (Biological Computations, биологические вычисления). Его цель - создание мощных вычислительных систем на основе ДНК.

Пока до практического применения компьютеров на базе ДНК еще очень далеко. Однако в будущем их смогут использовать не только для вычислений, но и как своеобразные нанофабрики лекарств. Поместив подобное "устройство" в клетку, врачи смогут влиять на ее состояние, исцеляя человека от самых опасных недугов.

7.4. Клеточные процессоры

Клеточные процессоры представляют собойсамоорганизующиеся колонии различных "умных" микроорганизмов, в геном которых удалось включить некую логическую схему, которая могла бы активизироваться в присутствии определенного вещества. Для этой цели идеально подошли бы бактерии, стакан с которыми



и представлял бы собой компьютер. Такие компьютеры очень дешевы в производстве. Им не нужна столь стерильная атмосфера, как при производстве полупроводников.

Главным свойством процессора такого рода является то, что каждая их клетка представляет собой миниатюрную химическую лабораторию. Если биоорганизм запрограммирован, то он просто производит нужные вещества. Достаточно вырастить одну клетку, обладающую заданными качествами, и можно легко и быстро вырастить тысячи клеток с такой же программой.

Основная проблема, с которой сталкиваются создатели клеточных биокомпьютеров, - организация всех клеток в единую работающую систему. На сегодняшний день практические достижения в области клеточных компьютеров напоминают достижения 20-х годов в области ламповых и полупроводниковых компьютеров. В Лаборатории искусственного интеллекта Массачусетского технологического университета создана клетка, способная хранить на генетическом уровне 1 бит информации. Также разрабатываются технологии, позволяющие единичной бактерии отыскивать своих соседей, образовывать с ними упорядоченную структуру и осуществлять массив параллельных операций.

В 2001 г. американские ученые создали трансгенные микроорганизмы (т. е. микроорганизмы с искусственно измененными генами), клетки которых могут выполнять логические операции И и ИЛИ.

Специалисты лаборатории Оук-Ридж, штат Теннесси, использовали способность генов синтезировать тот или иной белок под воздействием определенной группы химических раздражителей. Ученые изменили генетический код бактерий Pseudomonas putida таким образом, что их клетки обрели способность выполнять простые логические операции. Например, при выполнении операции И в клетку подаются два вещества (по сути - входные операнды), под влиянием которых ген вырабатывает определенный белок. Теперь учеными ведутся работы по созданию на базе этих клеток более сложных логических элементов, а также работы по созданию клетки, выполняющей параллельно несколько логических операций.

Потенциал биокомпьютеров очень велик. К достоинствам, выгодно отличающим их от компьютеров, основанных на кремниевых технологиях, относятся:

1) более простая технология изготовления, не требующая для своей реализации столь жестких условий, как при производстве полупроводников

2) использование не бинарного, а тернарного кода (информация кодируется тройками нуклеотидов), что позволит при меньшем количестве шагов перебрать большее число вариантов при анализе сложных систем

3) потенциально исключительно высокая производительность, которая может составлять до 1014 операций в секунду за счет одновременного вступления в реакцию триллионов молекул ДНК

4) возможность хранить данные с плотностью, в триллионы раз превышающей показатели оптических дисков

5) исключительно низкое энергопотребление

Однако, наряду с очевидными достоинствами, биокомпьютеры имеют и существенные недостатки, такие как:

1) сложность со считыванием результатов - современные способы определения кодирующей последовательности не совершенны, сложны, трудоемки и дороги

2) низкая точность вычислений, связанная с возникновением мутаций, прилипанием молекул к стенкам сосудов и т.д.

3) невозможность длительного хранения результатов вычислений в связи с распадом ДНК в течение времени

Хотя до практического использования биокомпьютеров еще очень далеко, но предполагается, что, они найдут достойное применение в медицине и фармакологии, а также с их помощью станет возможным объединение информационных и биотехнологий.

7.5. Коммуникационные процессоры

Коммуникационные процессоры - это микрочипы, являющие собой нечто среднее между жесткими специализированными интегральными микросхемами и гибкими процессорами общего назначения.

Коммуникационные процессоры программируются, как и обычные процессоры, но построены с учетом сетевых задач, оптимизированы для сетевой работы, и на их основе производители - как процессоров, так и оборудования - пишут программное обеспечение для специфических приложений.

Коммуникационный процессор имеет собственную память и оснащен высокоскоростными внешними каналами для соединения с другими процессорными узлами. Его присутствие позволяет в значительной мере освободить вычислительный процессор от нагрузки, связанной с передачей сообщений между процессорны-

ми узлами. Скоростной коммуникационный процессор с RISC-ядром позволяет управлять обменом данными по нескольким независимым каналам, поддерживать практически все распространенные протоколы обмена, гибко и эффективно распределять и обрабатывать последовательные потоки данных с временным разделением каналов.

Сама идея создания процессоров, предназначенных для оптимизации сетевой работы - и при этом достаточно универсальных для программной модификации – родилась в связи с необходимостью устранить различия в подходах к созданию локальных сетей (различные подходы к архитектуре сети, классификации потоков, и т.д

Новая серия коммуникационных процессоров Intel IXP4xx построена на базе распределенной архитектуры XScale и включает мощные мультимедийные возможности, а также развитые сетевые интерфейсы Ethernet. Сочетание высокой производительности и низкого энергопотребления позволяет эффективно применять коммуникационные процессоры Intel не только в классических сетевых приложениях, но и для построения интернет-ориентированных встраиваемых систем промышленного назначения.

Эффективность работы промышленных предприятий сегодня напрямую зависит от гибкости применяемых систем автоматизированного управления. Крупные производственные установки требуют использования нескольких децентрализованных систем управления, связанных друг с другом мощной информационной сетью, способной работать в сложных промышленных условиях. Зачастую эти средства промышленной коммуникации призваны обеспечить возможность гибкого управления, программирования и контроля работы распределенных систем управления из удаленных диспетчерских пунктов. Осуществление этих целей возможно с помощью коммуникационных процессоров, предназначенных для подключения персональных компьютеров к промышленным информационным сетям.

Дополнительные возможности, обеспечиваемые коммуникационными процессорами должны быть интересны, прежде всего, тем пользователям, которым необходимо осуществлять сложные

транзакции или наладить прямую голосовую и видео передачи в рамках сетевой инфраструктуры.

7.6. Процессоры баз данных

Процессорами (машинами) баз данных в настоящее время принято называть программно- аппаратные комплексы, предназначенные для выполнения всех или некоторых функций систем управления базами данных (СУБД). Если в свое время системы управления базами данных предназначались в основном для хранения текстовой и числовой информации, то теперь они рассчитаны на самые различные форматы данных, в том числе графические, звуковые и видео. Процессоры баз данных выполняют функции управления и распространения, обеспечивают дистанционный доступ к информации через шлюзы, а также репликацию обновленных данных с помощью различных механизмов тиражирования.

Современные процессоры баз данных должны обеспечивать естественную связь накапливаемой в базах данных информации со средствами оперативной обработки транзакций и Internet-приложениями. Это должны быть системы, которые дают пользователям возможность в любой момент обратиться к корпоративным данным и проанализировать их, вне зависимости от того, где эти данные размещаются.

Решение таких задач требует существенного увеличения производительности таких систем. Однако традиционная программная реализация многочисленных функций современных СУБД на ЭВМ общего назначения приводит к громоздким и непроизводительным системам с недостаточно высокой надежностью. Необходим поиск новых архитектурных и аппаратных решений. Интенсивные исследования, проводимые в этой области в настоящее время, привели к пониманию необходимости использования в качестве процессоров баз данных специализированных параллельных вычислительных систем. Создание такого рода систем связывается с реализацией параллелизма при выполнении последовательности операций и транзакций, а также конвейерной потоковой обработки данных.

7.7. Потоковые процессоры

Потоковыми называют процессора, в основе работы которых лежит принцип обработки многих данных с помощью одной команды. Согласно классификации Флинна они принадлежат к SIMD архитектуре. Технология SIMD позволяет выполнять одно и то же действие, например вычитание и сложение, над несколькими наборами чисел одновременно. SIMD-операции для чисел двойной точности с плавающей запятой ускоряют работу ресурсоемких приложений для создания контента, трехмерного рендеринга, финансовых расчетов и научных задач. Кроме того, усовершенствованы возможности 64-разрядной технологии MMX (целочисленных SIMD-команд); эта технология распространена на 128-разрядные числа, что позволяет ускорить обработку видео, речи, шифрование, обработку изображений и фотографий. Потоковый процессор повышает общую производительность, что особенно важно при работе с 3D-графическими объектами.

Может быть отдельный потоковый процессор (Single-streaming processor — SSP) и многопотоковый процессор (Multi-Streaming Processor - MSP).

Ярким представителем потоковых процессоров является семейство процессоров Intel, начиная с Pentium III, в основе работы которых лежит технология Streaming SIMD Extensions (SSE, потоковая обработка по принципу "одна команда - много данных"). Эта технология позволяет выполнять такие сложные и необходимые в век Internet задачи, как обработка речи, кодирование и декодирование видео- и аудиоданных, разработка трехмерной графики и обработка изображений.

Бесспорными представителями класса SIMD считаются матрицы процессоров: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 и т.п. В таких системах единое управляющее устройство контролирует множество процессорных элементов. Каждый процессорный элемент получает от устройства управления в каждый фиксированный момент времени одинаковую команду и выполняет ее над своими локальными данными.

Другими представителями SIMD-класса являются векторные процессоры, в основе которых лежит векторная обработка

данных. Векторная обработка увеличивает производительность процессора за счет того, что обработка целого набора данных (вектора) производится одной командой. Векторные компьютеры манипулируют массивами сходных данных подобно тому, как скалярные машины обрабатывают отдельные элементы таких массивов. В этом случае каждый элемент вектора надо рассматривать как отдельный элемент потока данных. При работе в векторном режиме векторные процессоры обрабатывают данные практически параллельно, что делает их в несколько раз более быстрыми, чем при работе в скалярном режиме. Максимальная скорость передача данных в векторном формате может составлять 64 Гб/с, что на 2 порядка быстрее, чем в скалярных машинах. Примерами систем подобного типа является, например, процессоры фирм NEC и Hitachi.

7.8. Процессоры с многозначной (нечеткой) логикой

Идея построения процессоров с нечеткой логикой (fuzzy logic) основывается на нечеткой математике. Математическая теория нечетких множеств, предложенная проф. Л.А. Заде, являясь предметом интенсивных исследований, открывает все большие возможности перед системными аналитиками. Основанные на этой теории различные компьютерные системы, в свою очередь, существенно расширяют область применения нечеткой логики.

Подходы нечёткой математики дают возможность оперировать входными данными, непрерывно меняющимися во времени и значениями, которые невозможно задать однозначно, такими, например, как результаты статистических опросов. В отличие от традиционной формальной логики, известной со времен Аристотеля и оперирующей точными и четкими понятиями типа истина и ложь, да и нет, ноль и единица, нечеткая логика имеет дело со значениями, лежащими в некотором (непрерывном или дискретном) диапазоне.

Функция принадлежности элементов к заданному множеству также представляет собой не жесткий порог "принадлежит - не принадлежит", а плавную сигмоиду, проходящую все значения от нуля до единицы. Теория нечеткой логики позволяет выполнять

над такими величинами весь спектр логических операций - объединение, пересечение, отрицание и др.

Согласно знаменитой теореме FAT (Fuzzy Approximation Theorem), доказанной Коско, любая математическая система может быть аппроксимирована системой, основанной на нечеткой логике. Свое второе рождение теория нечеткой логики пережила в начале восьмидесятых годов, когда сразу несколько групп исследователей (в основном в США и Японии) всерьез занялись созданием электронных систем различного применения, использующих нечеткие управляющие алгоритмы. Используя преимущества нечеткой логики, заключающиеся в простоте содержательного представления, можно упростить проблему, представить ее в более доступном виде и повысить производительность системы.

Задачи с помощью нечёткой логики решаются по следующему принципу:

1) численные данные (показания измерительных приборов, результаты анкетирования) фаззируются (переводятся в нечеткий формат);

2) обрабатываются по определённым правилам;

3) дефаззируются и в виде привычной информации подаются на выход.

Оказалось возможным создание нечеткого процессора, позволяющего выполнять различные нечеткие операции и приближенные рассуждения (нечеткий вывод) в соответствии с правилами логического вывода. В 1986 году в AT&T Bell Labs создавались процессоры с “прошитой” нечеткой логикой обработки информации.

В начале 90-х компания Adaptive Logic из США выпустила кристалл, сделанный по аналогово-цифровой технологии (рис. 25). Он позволит сократить сроки конструирования многих встроенных систем управления реального времени, заменив собой традиционные схемы нечетких микроконтроллеров. Аппаратный процессор нечеткой логики второго поколения принимает аналоговые сигналы, переводит их в нечеткий формат, затем, применяя соответствующие правила, преобразует результаты в формат обычной логики и далее – в аналоговый сигнал.

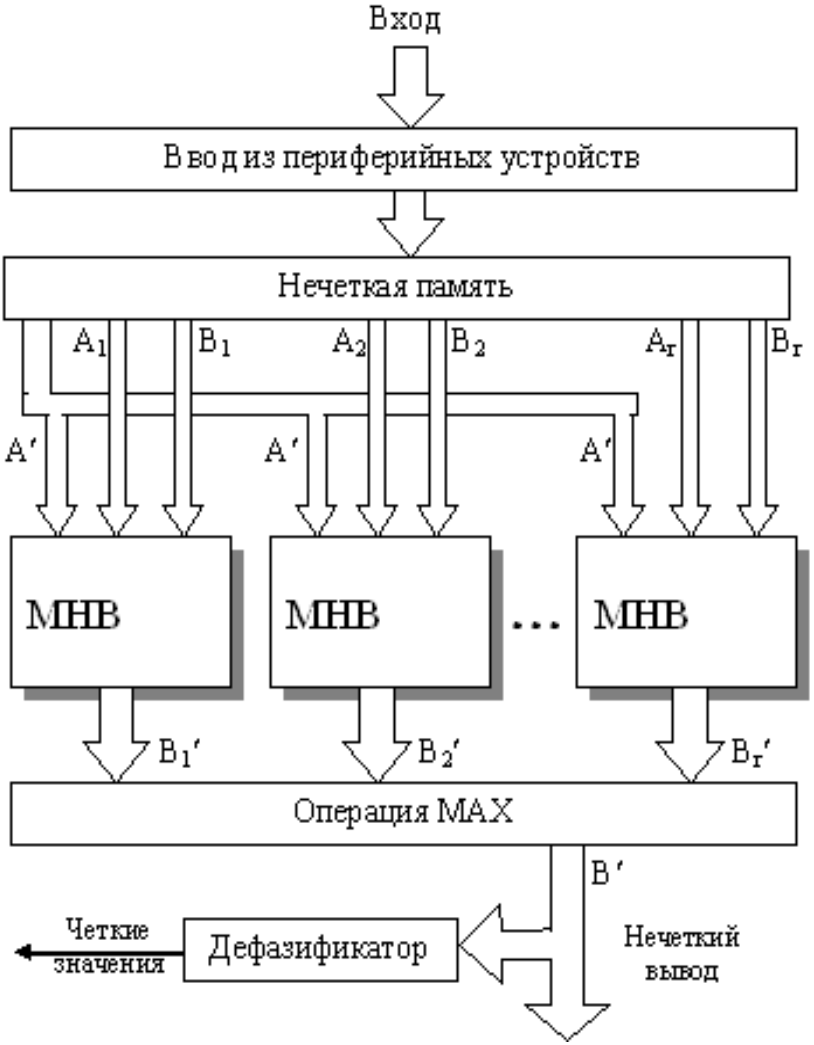


Рис. 25

Все это осуществляется без внешних запоминающих устройств, преобразователей и какого бы ни было программного обеспечения нечеткой логики. Этот микропроцессор относительно прост по сравнению с громоздкими программными обеспечениями. Но так как его основу составляет комбинированный цифровой/аналоговый кристалл, он функционирует на очень высоких скоро-

стях (частота отсчетов входного сигнала – 10 кГц, а скорость расчета – 500 тыс. правил/с), что во многих случаях приводит к лучшим результатам в системах управления по сравнению с более сложными, но медлительными программами.

В Европе и США ведутся интенсивные работы по интеграции fuzzy команд в ассемблеры промышленных контроллеров встроенных устройств (чипы Motorola 68HC11. 12. 21). Такие аппаратные средства позволяют в несколько раз увеличить скорость выполнения приложений и компактность кода по сравнению с реализацией на обычном ядре. Кроме того, разрабатываются различные варианты fuzzy- сопроцессоров, которые контактируют с центральный процессор через общую шину данных, концентрируют свои усилия на размывании/ уплотнении информации и оптимизации использования правил (продукты Siemens Nixdorf).

Идеи нечеткой логики не являются панацеей и не смогут совершить переворот в компьютерном мире. Нечеткая логика не решит тех задач, которые не решаются на основе логики двоичной, но во многих случаях она удобнее, производительнее и дешевле. Разработанные на ее основе специализированные аппаратные решения (fuzzy-вычислители) позволят получить реальные преимущества в быстродействии. Если каскадировать fuzzy-вычислители, мы получим один из вариантов нейропроцессора или нейронной сети. Во многих случаях эти понятия просто объединяют, называя общим термином «neuro-fuzzy logic».

В настоящее время перспективой использовать процессоры, основанные на нечеткой логике всерьез заитересовались военные. Известно, что NASA рассматривает возможность применения (если еще не применяет) нечеткие системы для управления процессами стыковки космических аппаратов.

7.9. Сигнальные процессоры

В ответ на возросшие запросы потребителей фирма Motorola разработала новую архитектуру микросхемы, ориентированную как на выполнение сложных алгоритмов цифровой обработки сигналов, так и на решение задач управления. Семейство микросхем DSP568xx построено на базе ядра 16-разрядного про-

цессора DSP56800 с фиксированной точкой. Это ядро предназначено для эффективного решения задач управления и цифровой обработки сигналов. Реализованный в нем набор команд обеспечивает цифровую обработку сигналов с эффективностью лучших DSP общего назначения и отвечает требованиям простоты создания компактных программ управления.

Ядро DSP56800 является программируемым 16-разрядным КМОП процессором, предназначенным для выполнения цифровой обработки сигналов в реальном масштабе времени и решения вычислительных задач. Ядро DSP56800 (рис. 26) состоит из четырех функциональных устройств: управления программой, генерации адресов, арифметико-логической обработки данных, обработки битов. Для увеличения производительности операции в устройствах выполняются параллельно. Каждое из устройств имеет свой набор регистров и логику управления и организовано таким образом, что может функционировать независимо и одновременно с тремя другими. Внутренние шины адресов и данных связывают между собой память, функциональные и периферийные устройства (регистры периферийных устройств расположены в области памяти). Таким образом, ядро реализует одновременное выполнение нескольких действий: устройство управления выбирает первую команду, устройство генерации адресов формирует до двух адресов второй команды, а АЛУ выполняет умножение третьей команды. Есть альтернативная возможность: в третьей команде операцию может выполнять не АЛУ, а устройство обработки битов. Конвейерная архитектура позволяет реализовать параллельную работу устройств, входящих в состав микросхемы, и существенно сократить время выполнения программы.

Конвейерная архитектура ядра DSP56800 оптимизирована для обеспечения эффективности цифровой обработки сигналов, компактности программ управления и обработки сигналов, и удобства программирования. Ниже приведены некоторые характеристики сигнального процессора:

• производительность 40 MIPS при тактовой частоте 80 МГц и напряжении питания 2.7…3.6 В,

• наличие набора команд совмещенной обработки, имеющих режимы адресации, характерные для программ цифровой обработки сигналов,

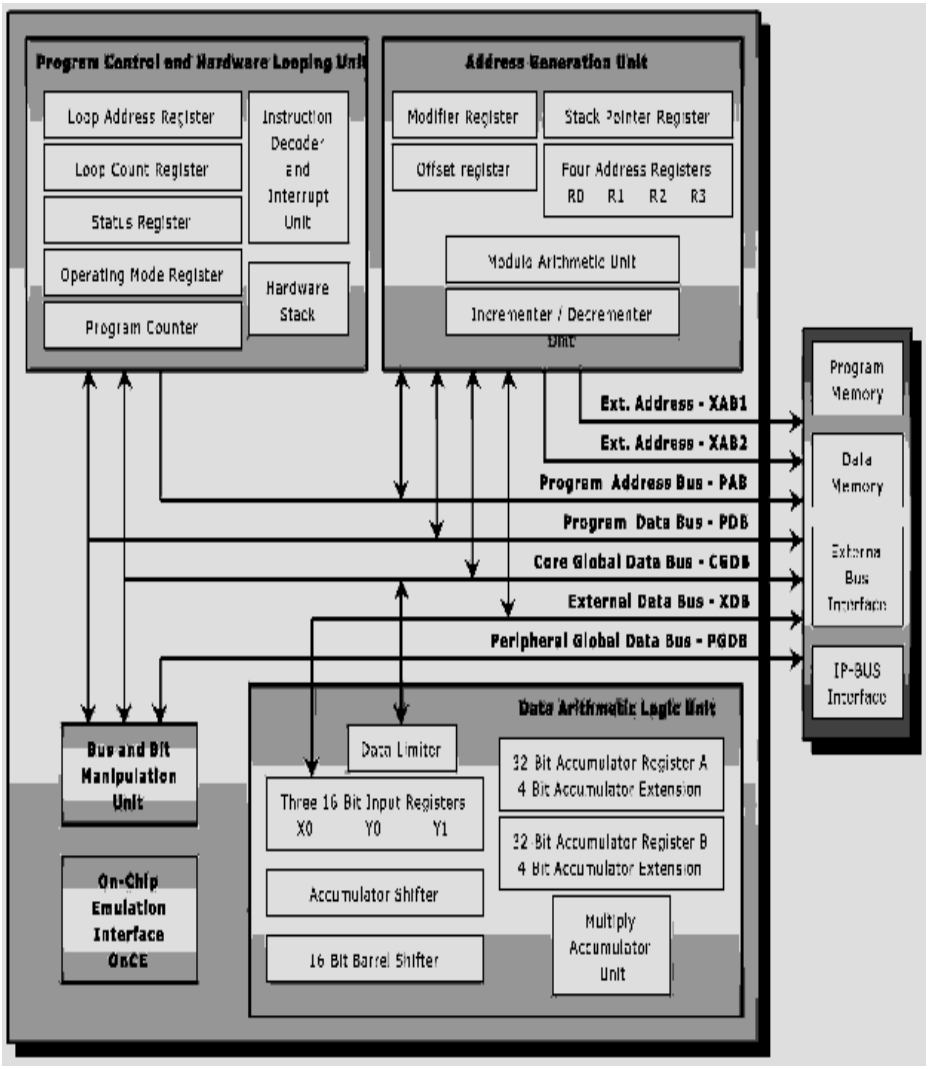


Рис. 26

• однотактный параллельный 16x16 умножитель-сумматор,

• два 36-разрядных аккумулятора, включая биты расширения,

• однотактное 16-разрядное устройство циклического сдвига,

• аппаратная реализация команд DO и REP,

• три внутренние 16-разрядные шины данных и три 16-разрядные шины адреса,

• одна 16-разрядная шина внешнего интерфейса,

• набор команд управления и цифровой обработки,

• режимы адресации такие же, как в сигнальных процессорах, и команды, снижающие объем программы,

• эффективный компилятор языка С и поддержка локальных переменных,

• стек подпрограмм и прерываний, не имеющий ограничения по глубине.

Для любого высокопроизводительного вычислителя, например цифрового сигнального процессора, критичным является процесс ввода/вывода данных с большой скоростью, т. к. при этом замедляется обработка данных. Снижения производительности можно избежать путем использования гибкого набора команд совмещенной с выполнением вычислительных операций передачи данных. Реализованы два типа операций совмещенной передачи - одинарная совмещенная передача и сдвоенное совмещенное чтение. Оба типа операций существенно повышают скорость цифровой обработки сигналов и численных расчетов. Все команды DSP56800 с совмещенной передачей выполняются за один командный цикл и занимают одно слово в памяти программ.

Однократная совмещенная передача позволяет выполнить арифметическое действие и одну передачу данных (чтение или запись) за один командный цикл. Например, можно одной командой выполнить сложение двух чисел и одновременно данные из регистра АЛУ записать в память. Одновременно с этим в устройстве вычисления адресов производятся соответствующие вычисления.

Команда типа двойного совмещенного чтения допускает выполнение арифметической операции и чтения двух величин из X-памяти данных в одной команде за один командный цикл. Например, можно в одной команде выполнить умножение двух чи-

сел, просуммировать с третьим, округлить результат и одновременно выполнить пересылку двух чисел из X-памяти данных в два регистра АЛУ.

Обычные микроконтроллеры, как правило, имеют объем встроенной в микросхему памяти, достаточный для выполнения сложных алгоритмов управления без использования дополнительной внешней памяти. Многие микросхемы DSP содержат встроенную память небольшого объема и, как правило, им требуется внешняя память для хранения программы. Микросхемы же семейства DSP56F8xx имеют встроенную память большого объема. Гарвардская архитектура DSP обеспечивает наличие двух независимых областей памяти - данных и программ. Для хранения в микросхеме данных и программ используется встроенная оперативная память и флэш-память.

Объём памяти каждого типа для микросхем семейства DSP56F8xx приведен в табл. 5. Как память программ, так и память данных могут быть расширены путем подключения внешней памяти. Микросхемы DSP56F803, DSP56F805, DSP56F807 допускают расширение объема внешней памяти до 64 К слов.

Таблица 5

Встроенная память	DSP56F801	DSP56F803	DSP56F805	DSP56F807
Флэш-память программы	8k x 16	32k x 16	32k x 16	60k x 16
Флэш-память данных	2k x 16	4k x 16	4k x 16	8k x 16
ОЗУ программ	1k x 16	512 x 16	512 x 16	2k x 16
ОЗУ данных	1k x 16	2k x 16	2k x 16	4k x 16
Флэш-память программы загрузки	2k x 16	2k x 16	2k x 16	2k x 16

Широкий набор периферийных устройств обычно являлся основной характеристикой микроконтроллеров, встраиваемых в устройства общего назначения. С другой стороны, обычные DSP были ориентированы на численную обработку сигналов и не содержали полного набора встроенных периферийных устройств, необходимых для решения задач управления. Использование внешних периферийных устройств приводит к увеличению числа

микросхем, усложнению платы и существенному возрастанию стоимости изделия.

Микросхемы семейств DSP56F8xx (рис. 27, табл. 6) имеют широкий набор встроенных периферийных устройств, пригодных для использования в системах управления всех типов

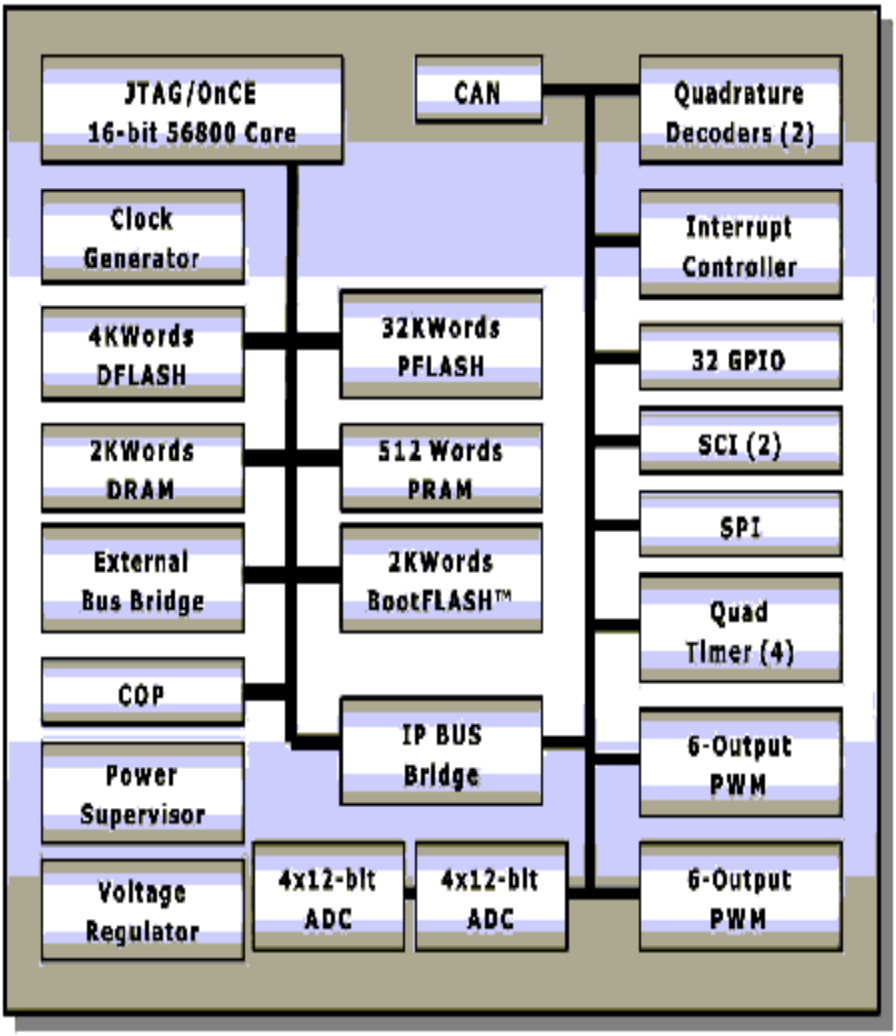


Рис. 27

Таблица 6

	DSP56824	DSP56F801	DSP56F803	DSP56F805	DSP56F80
Тактовая частота, МГц	70	80	80	80	80
Производительность, MIPS	35	40	40	40	40
Интерфейсы	2SPI, SCI, SSI	SCI, SPI	2SCI, SPI, CAN	2SCI, SPI, CAN	2SCI, SPI, CAN
АЦП	-	2 четырехкан. 12-разрядн.	2 четырехкан. 12-разрядн.	2 четырехкан. 12-разрядн.	4 четырехкан. 12-разрядн.
ШИМ-генераторы	-	6-канальн. 15-разр.	6-канальн. 15-разр.	2 6-канальн. 15-разр.	2 6-канальн. 15-разр.
Прочие функциональные особенности	ФАПЧ, 3-кан. таймер	НВИ*, ФАПЧ, 2-кан.таймер	НВИ*, ФАПЧ, 5-кан.таймер	НВИ*, ФАПЧ, 6-кан.таймер	НВИ*, ФАПЧ, 6-кан.таймер

. Этот набор встроенных устройств существенно снижает цену системы по сравнению с реализацией устройств управления на основе традиционных DSP. Более того, так как встроенные устройства имеют заранее определенный интерфейс с ядром DSP (в отличие от внешних периферийных устройств), то упрощаются разработка системы, программирование и управление периферийными устройствами. Таким образом, время разработки программ сокращается. Микросхема DSP56F805 содержит следующие периферийные блоки:

• два шестиканальных ШИМ-генератора (PWMA & PWMB) с привязкой импульсов к центру или краю временного интервала, программированием длительности "мертвого времени" и защитой в случае возникновения аварийных режимов работы; каждый генератор снабжен тремя сенсорами тока и четырьмя входами аварийного отключения,

• два 12-разрядных АЦП с одновременной выборкой, снабженные входными четырехканальными мультиплексорами,

• два квадратурных (синусно-косинусных) декодера (Quad Dec0 & Quad Dec1), каждый с четырьмя входами (или два дополнительных четырехканальных таймера A&B),

• два четырехканальных таймера общего применения с шестью входами: таймер C с двумя входами и таймер D с четырьмя входами,

• контроллер CAN интерфейса A/B с двухвыводными портами приемопередатчиков,

• два двухпроводных последовательных коммуникационных интерфейса (SCI0 & SCI1) или 4 дополнительных линии GPIO,

• последовательный интерфейс периферии (SPI) с настраиваемым четырехпроводным портом или четыре дополнительные линии GPIO,

• сторожевой таймер контроля функционирования процессора,

• два программируемых входа внешних прерываний,

• четырнадцать программируемых и восемнадцать мультиплексированных универсальных портов ввода/вывода (GPIO),

• вход принудительного сброса процессора,

• порт JTAG/OnCE™ (встроенного эмулятора) для отладки, не зависящей от тактовой частоты процессора,

• программируемый генератор с ФАПЧ для формирования тактовой частоты ядра DSP.

В системах управления, как правило, интенсивно используются прерывания от внешних устройств и внутренних периферийных модулей. Обычно микроконтроллеры поддерживают несколько типов внутренних и внешних прерываний и обеспечивают много вариантов маскирования и установки приоритетов.

Обычные DSP обрабатывают только небольшой набор прерываний, которые напрямую взаимодействуют с его ядром. В отличие от них, кристалл DSP56F80x поддерживает большое число прерываний. Хотя число адресуемых прерываний ядра DSP56F8xx мало в сравнении с общим число источников прерываний, многоуровневая встроенная схема мультиплексирования обеспечивает полную и гибкую поддержку 64 источников прерываний, каждый

из которых может маскироваться и имеет программно устанавливаемый приоритет.

Разработка систем на базе микросхем семейства DSP56800 отличается простотой. Внешняя шина обеспечивает выполнение и отладку прикладных программ, размещенных во внешней памяти. Допускается хранение программ и данных во внешней памяти. Чтобы обеспечить функционирование внешней памяти с различным быстродействием программируемые временные задержки для памяти программ и памяти данных могут устанавливаться раздельно.

Набор команд общего назначения, который используется в микропроцессорах с развитыми режимами адресации и командами обработки битов, дает разработчику возможность просто освоить программирование. Сложности, характерные для DSP с предшествующими архитектурами, не доставят ему беспокойства. Программный стек обеспечивает неограниченное число прерываний и вложений подпрограмм, а также поддержку передачи параметров и локальных переменных. Опытный программист найдет широкий набор команд арифметических операций и различные одинарные и двойные обращения к памяти, выполняющиеся совместно с арифметическими операциями. Эффективная работа трансляторов для микросхем с архитектурой DSP56800 обеспечивается использованием в микросхемах команд общего назначения.

Порт отладки JTAG позволяет отлаживать микросхему в составе законченной системы пользователя. Через порт можно задать точки останова программы, проверить и изменить содержимое регистров и ячеек памяти, выполнить другие действия по отладке системы.

Motorola предлагает полный набор программных и аппаратных средств быстрой разработки и отладки систем, реализованных на кристаллах семейства DSP568xx. Средства разработки включают:

- оценочные платы для каждой модификации микросхемы,
- интегрированную среду отладки "Metrowerks Code Warrior" со встроенным кросс-компилятором языка С.

Программная среда разработки предоставляет программисту гибкое модульное окружение, обеспечивая полное использова-

ние возможностей микросхем. Среда допускает различные конфигурации памяти данных и позволяет создавать перемещаемый код, выполнять символьную отладку, гибко компоновать объектные файлы. Реализованы средства создания архива библиотек прикладных программ.

Motorola разработала новый комплект для разработки встроенного программного обеспечения (Embedded Software Development Kit, SDK), дополняющий существующую среду разработки для DSP568xx. Он формирует программную инфраструктуру, обеспечивающую разработку высокоэффективных программ, полностью переносимых и допускающих повторное использование не только в процессорах семейства DSP568xx, но в будущем и в процессорах с другой архитектурой, поддерживаемой SDK. Этот программный продукт, выпускаемый для цифровых сигнальных процессоров фирмы Motorola, предназначен для ускорения разработки и более быстрого выхода изделий на рынок.

Стандартные микроконтроллеры успешно применяются в устройствах управления общего назначения. Однако невысокая производительность не позволяет использовать их в устройствах с повышенными параметрами. Эта ниша заполняется микросхемами семейства DSP568xx, имеющими производительность DSP и снабженными набором периферийных устройств, которые традиционно используют разработчики систем управления.

Архитектура ядра DSP568xx обеспечивает эффективную цифровую обработку данных и решение задач управления. Такие характеристики этой архитектуры, как высокая производительность и набор команд общего назначения, обеспечивают ей лидирующие позиции в тех областях цифровой обработки сигналов, в которых требуется низкая стоимость и малое энергопотребление. Компактность программ и высокая эффективность компилятора позволяет также снизить стоимость системы за счет уменьшения требуемого объема встроенной памяти.

Микросхемы семейства DSP568xx предназначены для применения в недорогих устройствах. Эти микросхемы ориентированы на применение в бытовой технике, для которой необходима низкая стоимость и не требуются высокие параметры. К таким изделиям относятся:

- специализированные и многоцелевые контроллеры,
- проводные и беспроводные модемы,
- системы беспроводной передачи цифровых сообщений,
- цифровые телефонные автоответчики,
- устройства управления серводвигателями и электродвигателями переменного тока,
- цифровые камеры.

Микросхемы этого семейства имеют производительность специализированных DSP. Благодаря наличию набора встроенных периферийных устройств эти микросхемы отвечают требованиям систем управления. Встроенные блоки памяти и периферийные устройства могут существенно снизить стоимость системы, потому что в этом случае уменьшается число внешних компонентов.

8. Архитектуры микропроцессорных систем

8.1. Системы с централизованным, децентрализованным и комбинированным управлением

В МПС с централизованным управлением задача обработки входных сигналов X_1 ,..., X_n с целью формирования управляющих воздействий Y_1 ,...,Y_m решается микропроцессорным устройством (МПУ), включающим МП и элементы памяти, которые соединены каналами связи через цифро-аналоговые преобразователи (ЦАП), исполнительные устройства (ИУ) с объектом (или объектами) управления (ОУ). Обратная связь о состоянии ОУ обеспечивается сигналами от ОУ, поступающими на МПУ через аналого-цифровые преобразователи (АЦП).

Общая структурная схема для этого случая показана на рис. 28. Если осуществляется управление одним, но сложным многомерным объектом (роботом, прокатным станом, доменной печью, самолетом, космическим летательным аппаратом и т. п.), то такая система является связанной. Если же решается задача управления совокупностью независимых по управляемым параметрам одномерных объектов, то система является несвязанной.

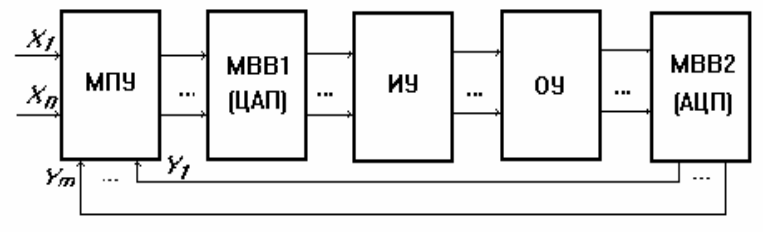


Рис. 28.

В системах с децентрализованным управлением в каждый контур управления включается автономное МПУ. Структурная схема системы с децентрализованным управлением приведена на рис. 29, где МПУ размещены в непосредственной близости от объекта управления ОУ или встроены в него и функционально ориентированы на решение конкретных задач. В качестве МПУ широко применяется программируемые регулирующие микроконтроллеры.

В децентрализованных системах МПУ могут вводиться для передачи ему функций диспетчера либо отсутствовать совсем. В этом случае реализуется комбинированное управление. В комбинированных системах используется обе перечисленные структуры управления.

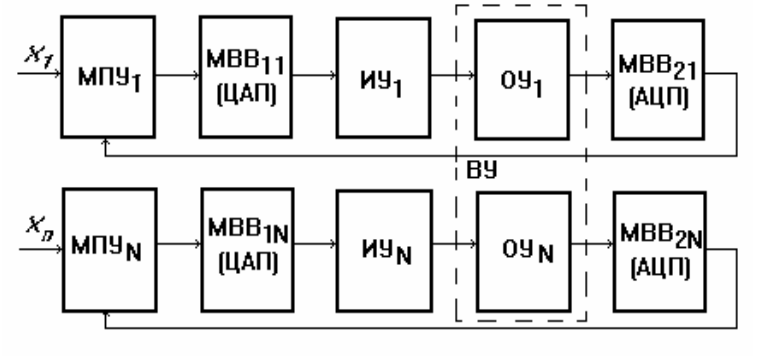


Рис. 29.

Выбор структуры управления в МПС, построенных на базе МПК БИС, зависит от многих взаимосвязанных факторов, важнейшими из которых являются стоимость и надежность систем,

их живучесть, гибкость, способность работать в масштабе реального времени.

Специфика конкретных задач управления показывает, что применение принципа децентрализованного (распределенного) управления при построении МПС в техническом и экономическом плане имеют преимущества по сравнению с другими структурами МПС.

8.2. Системы с перестраиваемой структурой

Задачи, решаемые МПС, могут зависеть от характера входных воздействий, поступающих в систему. Так, например, управление роботом может осуществляться по разным алгоритмам в зависимости от результата решения задачи распознавания представленного роботу объекта. В этом и подобных случаях структура МПС оказывается переменной. В микропроцессорных системах она перестраивается программно.

Пример структурной схемы перестраиваемой МПС управления роботом в общем виде показана на рис. 30.

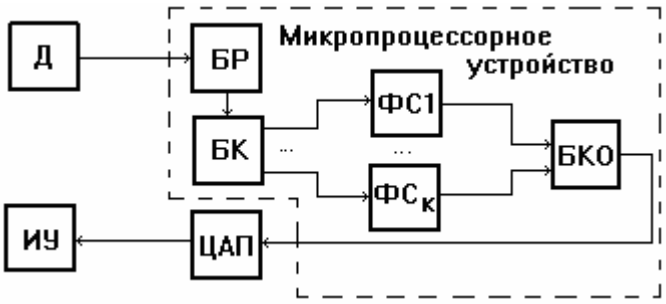


Рис. 30

В состав системы входят: цифровой датчик визуальной информации Д, информация с которого поступает в блок распознавания изображения БР, формирующий электрический логотип изображения. Блок коммутации алгоритмов управления БК производит выбор и обработку программы обслуживания сформированного логотипа. С помощью сигналов формирователей управляю-

щих сигналов $ФС_1$,..., $ФС_к$ блок коммутации выхода БКВ выдает на выход МПУ исполнительный сигнал, который через ЦАП подается на исполнительное устройство ИУ.

Осуществимость перестройки МПС, выполняемой в реальном масштабе времени на программном уровне, является следствием применения в автоматических системах высокопроизводительных МПС, на которые возлагаются задачи обработки больших потоков информации, связанной со статистическим экспресс - анализом случайных сигналов, их идентификацией, классификацией, распознаванием изображений и т. п. Это в конечном счете существенно улучшает показатели качества управления системой.

8.3. Системы с резервированием

Одним из путей увеличения отказоустойчивости МПС является резервирование. Резервирование подразделяется на аппаратурное, программное и информационное.

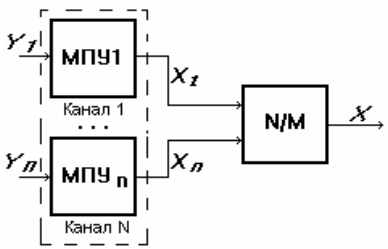


Рис.31

Табл. 7

Y_1	Y_2	Y_3	X
-	-	-	-
-	-	+	-
-	+	-	-
-	+	+	+
+	-	-	-
+	-	+	+
+	+	-	+
+	+	+	+

Распространенными методами аппаратурного резервирования являются методы, основанные на мажоритарной обработке и обработке с переключением каналов. При мажоритарной обработке МПС состоит из *n* независимых каналов обработки информации (рис. 31) и остается в рабочем состоянии до тех пор, пока сохраняют работоспособность *t* из *n* каналов. Например, в системе «2 из 3» работоспособное состояние канала определяется из табл. 7, в

которой значками «+» и «-» определено соответственно работоспособное и неработоспособное состояния канала.

В МПС с переключением каналов избыточные (резервные) каналы обработки информации включаются в работу только после выхода из строя основного или ранее замененного канала. В таких системах имеются дополнительные блоки опознавания (БО) неисправных каналов и их переключения БП (рис.32).

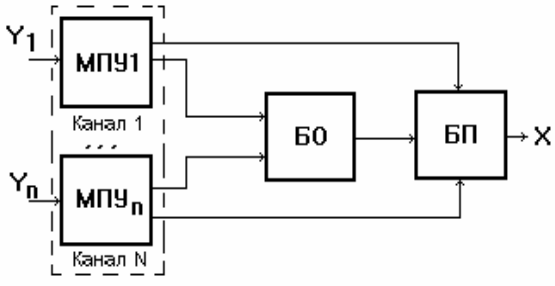


Рис.32

8.4. Иерархические системы

Сложные объекты управления (самолеты, космические аппараты, прокатные станы, роторные конвейерные линии и т. п.) представляют собой совокупность взаимосвязанных многорежимных управляемых систем, объединенных единой системой управления. Основополагающими принципами, определяющими структуру МПС управления подобными объектами, является иерархичность, независимость управления по уровням иерархии и информационная замкнутость. Обобщенная структура иерархической МПС показана на рис.33.

Особенность микроконтроллера проявляется в том, что на его выходе не используется мультиплексирование (число ЦАП равно числу выходных цепей контроллера). Такое построение контроллера связано с необходимостью запоминания каждого значения управляющего сигнала после останова вычислительного процесса. Учитывая, что в распределенной МПС число выходных сигналов невелико, затраты на ЦАП оказываются относительно небольшими.

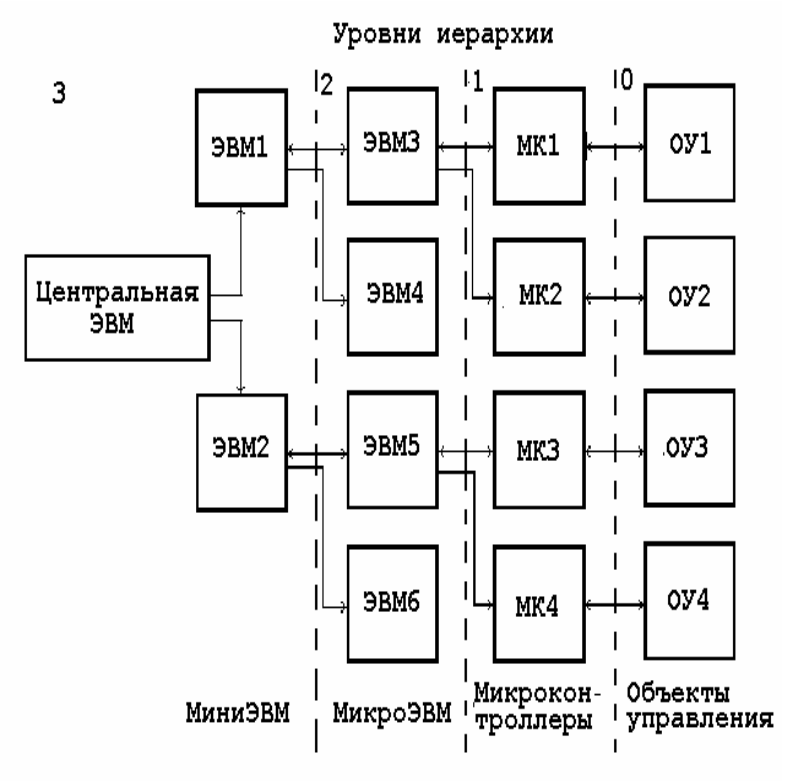


Рис.33

Преобразователи аналоговых сигналов в код позволяют сопрягать микроконтроллер с непрерывными и дискретными датчиками, с исполнительными механизмами пропорционального, позиционного, интегрирующего и другого действия, а также с различными устройствами дискретного и логического управления .

В микроконтроллере может быть применен микропроцессор как с аппаратным, так и с микропрограммным принципом управления. Аппаратное управление основано на внутреннем микропрограммировании. При аппаратном управлении система команд микропроцессора является фиксированной. Она реализована во внутренних жестких электрических связях в кристалле МП и не может быть изменена разработчиком системы. Микропрограммное

управление основано на внешнем микропрограммировании (набор команд может быть нефиксированным и изменяться разработчиком системы).

Особенностью программного обеспечения микроконтроллера является то, что большая часть его памяти программируется на заводе - изготовителе. В нем отсутствуют обычные средства ввода и отладки программ, а также модули сопряжения с ними. Указанные особенности позволяют упростить микроконтроллер и сделать рентабельным его применение для обработки сравнительно небольших массивов информации.

Пульт оператора в микроконтроллере используется для установки требуемой конфигурации регулирующего контура, выбора алгоритма управления, контроля значений технологических переменных, оперативного вмешательства в процесс управления и других целей. Программное обеспечение состоит из программ: диспетчера (координирующего весь вычислительный процесс), рабочих, обслуживания пульта и диагностических. Для программирования используется, как правило, десятичный код, набираемый на панели пульта.

Все алгоритмы микроконтроллера достаточно универсальны и в функциональном отношении эквивалентны типовым звеньям МПС управления объектом или типовой «связке» таких звеньев. Возможности микроконтроллера характеризуют, используя понятие виртуальной (кажущейся) структуры. Виртуальная структура описывает свойства контроллера в традиционных для МПС управления понятиях, основными из которых являются каналы управления, с системной точки зрения эквивалентные отдельному прибору или типовому сочетанию приборов непрерывной системы управления, и конфигурация, определяющая систему связи каналов со входами и выходами контроллера, а также варианты взаимодействия каналов.

По оценкам специалистов, существует ограниченное число (ориентировочно 20 - 25) алгоритмов, комбинация которых позволяет автоматизировать управление процессами и объектами практически любой степени сложности. Эти алгоритмы, оформленные в виде библиотеки программ, хранятся в постоянной памяти и могут быть использованы в любом заданном сочетании. Среди про-

граммируемых регулирующих микроконтроллеров особое место занимают однокристальные микроконтроллеры, выпускаемые серийно. По степени универсальности использования их подразделяют на специализированные, работающие по жесткой программе, и широкого применения, программа действия которых заносится во внешнее запоминающее устройство и может изменяться самим пользователем или по картам-заказам, составленным пользователем .

Примерами однокристальных перепрограммируемых микроконтроллеров являются контроллеры серии К145. Это цифровые структуры последовательного действия, использующие принцип многоуровневого программирования. Однокристальные микроконтроллеры адаптируются к внешним устройствам как по формату управляющих команд, так и по временным характеристикам. Для реализации множества задач управления в таких контроллерах используется специальная система команд, обеспечивающая управление внешними устройствами и выполнение программы. Список команд позволяет организовать как разомкнутую систему управления объектами по жесткой программе, так и замкнутую с большой сетью внутри программных ветвлений в соответствии с условиями, задаваемыми по времени и состоянию датчиков.

На втором уровне иерархии находятся серийные микроЭВМ, которые обеспечивают управление группой функционально связанных объектов. На этом уровне, соответствующем локальному управлению, применяют серийные микроЭВМ многофункционального назначения.

Третий уровень включает управляющие устройства, реализованные на базе мини-ЭВМ, которые координируют работу группы локальных систем.

На четвертом уровне располагается центральная управляющая ЭВМ, которая является высшим координирующим органом в данной структуре.

8.5. Однопроцессорная МПС типа «Общая шина»

Однопроцессорная МПС содержит в своем составе один микропроцессор, один или несколько модулей памяти, систему

связи с объектами управления или измерительными объектами, состоящую из устройств ввода-вывода, которые в общем виде носят название контроллеров ввода - вывода (КВВ), внешних устройств, в качестве которых используются датчики и исполнительные механизмы, и систему ввода - вывода, в которую входят УВВ и ВУ. Перечисленные компоненты МПС, типовая структурная схема которой представлена на рис.34, связаны с системной магистралью посредством соответствующих интерфейсов. В свою очередь магистраль состоит из шины адресов, шины данных и управляющей шины.

Такая система на практике получила название «общая шина».

В большинстве случаев МПС используют магистрально - модульный принцип построения. Суть этого принципа заключается в том, что отдельные блоки являются функционально законченными модулями со своими встроенными схемами управления, выполненными в виде одного или нескольких кристаллов БИС или СБИС.

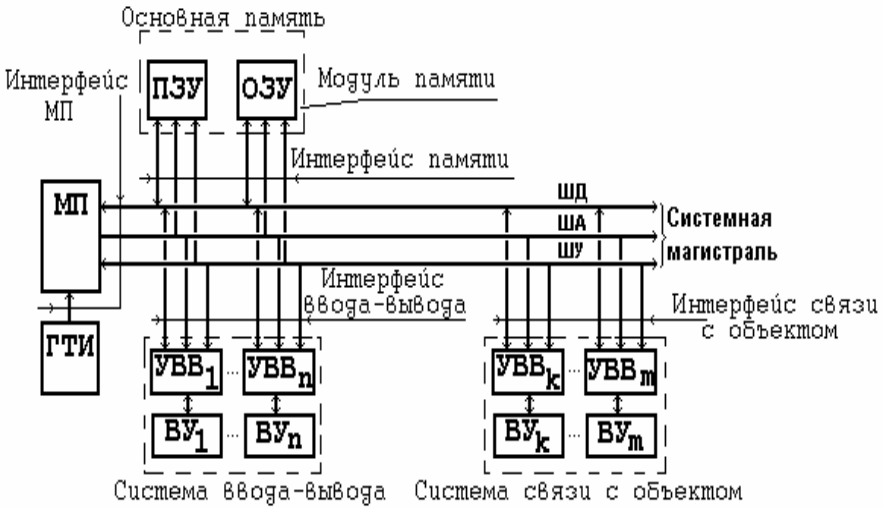


Рис. 34

Межмодульные связи и обмен информацией между модулями осуществляются посредством шин (магистралей), к которым

имеют доступ все основные модули системы. В МПС в каждый данный момент времени возможен обмен информацией только между двумя модулями системы.

Ситуация, при которой три или более модуля требуют одновременного доступа к одной магистрали, является недопустимой, так как она приводит к появлению конфликта. Поэтому обмен информацией в магистрально-модульных системах производится, как правило, путем разделения (арбитража) во времени управления модулями системы магистралей.

Особенностью магистрально-модульного принципа построения МПС является необходимость информационно - логической совместимости модулей. При выполнении этого принципа достигается оптимальная межмодульная передача информации. Он реализуется путем использования единых способов представления информации, алгоритмов управления обменом, форматов команд управления обменом и способов синхронизации, то есть выполнения определенных электрических и конструктивных требований при построении интерфейсов.

При построении МПС наибольшее применение получила трехшинная структура, содержащая шину адреса ША, двунаправленную шину данных ШД и шину управления ШУ. Как видно из рис.34, МПС предполагает наличие общего сопряжения (общего или единого интерфейса) для модулей памяти — постоянных и оперативных запоминающих устройств (ПЗУ и ОЗУ) и периферийных устройств — внешних запоминающих устройств (ВЗУ) и УВВ.

Модуль памяти ОП (или основная память) МПС содержит постоянное запоминающее устройство ПЗУ и оперативное запоминающее устройство ОЗУ. В большинстве системах ОП физически реализуется в виде многоуровневой иерархической системы. Верхние уровни памяти строятся на основе полупроводниковых ПЗУ и ОЗУ, а нижние - на основе магнитных ВЗУ.

Основная память является одним из важнейших компонентов любой МПС и предназначена для хранения программ и данных, используемых или генерируемых выполняющимися программами. Главная характеристика основной памяти - объем или емкость. В

МПС используются две единицы измерения емкости основной памяти - байт и слово.

Байт - это последовательность из 8 бит, рассматриваемая как один элемент данных или памяти. Биты в байте нумеруются от 0 до 7 в порядке справа налево, причем бит 0 считает младшим, а бит 7 - старшим в байте (рис.35, а).

Каждый байт памяти имеет свой адрес. Так, например, в МП с 16 - разрядной ША адреса байтов могут иметь значения в диапазоне целых чисел от 0 до 2^{16} - 1 (до 65535). Следовательно, общее число всех адресов - адресное пространство ОП, составляет 2^{16} = 65 536 = 64К. Эта величина, указываемая в технических характеристиках, называется размером адресного пространства, выраженным числом адресов байт. Иногда ее называют также емкостью ОП.

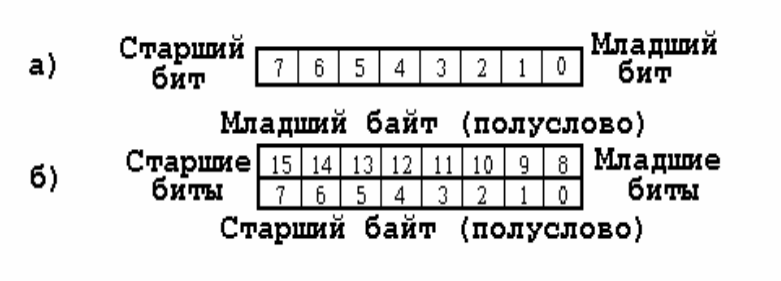


Рис. 35

Другая, более крупная единица измерения емкости ОП - слово, которое состоит из двух байт и рассматривается как один элемент. Из двух байт, составляющих слово (рис.35, б), младший имеет четный адрес, а старший (на единицу больший) - нечетный адрес. Адресом слова считается адрес младшего байта, входящего в слово. Следовательно, слово всегда имеет четный адрес.

Принцип четной адресации слов в МПС должен строго соблюдаться программистом, так как при его нарушении многие команды будут выполняться с ошибкой.

Чтобы дать более полное представление о МПС, помимо высокой производительности необходимо назвать и другие отличительные особенности. Прежде всего это необычные архитектур-

ные решения, направленные на повышение производительности (работа с векторными операциями, организация быстрого обмена сообщениями между процессорами или организация глобальной памяти в многопроцессорных системах и др.).

Понятие архитектуры высокопроизводительной системы является достаточно широким, поскольку под архитектурой можно понимать и способ параллельной обработки данных, используемый в системе, и организацию памяти, и топологию связи между процессорами, и способ исполнения системой арифметических операций. Эти вопросы освещаются ниже.

8.6. Архитектуры с параллельной обработкой данных

В 1966 году М.Флинном был предложен подход к классификации архитектур вычислительных систем. В основу было положено понятие потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. Соответствующая система классификации основана на рассмотрении числа потоков инструкций и потоков данных. Она содержит четыре архитектурных класса:

SISD = Single Instruction Single Data,
MISD = Multiple Instruction Single Data ,
SIMD = Single Instruction Multiple Data,
MIMD = Multiple Instruction Multiple Data.

SISD - одиночный поток команд и одиночный поток данных. К этому классу относятся последовательные МПС, которые имеют один центральный процессор, способный обрабатывать только один поток последовательно исполняемых инструкций (рис. 36). В настоящее время практически все высокопроизводительные системы имеют более одного центрального процессора, однако, каждый из них выполняют несвязанные потоки инструкций, что делает такие системы комплексами SIMD-систем, действующих на разных пространствах данных.

Рис. 36

Для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка. В случае векторных систем векторный поток данных следует рассматривать как поток из одиночных неделимых векторов. Примерами компьютеров с архитектурой SISD являются большинство рабочих станций Compaq, Hewlett-Packard и Sun Microsystems.

MISD - множественный поток команд и одиночный поток данных. Теоретически в этом типе МПС множество инструкций должно выполняться над единственным потоком данных (рис.37).

Рис. 37

Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако развитие эта архитектура МПС до сих пор не получила.

По ряду признаков к этому классу можно отнести конвейерные МПС, однако это не нашло окончательного признания на практике.

SIMD - одиночный поток команд и множественный поток данных (рис.38). Эти системы обычно имеют большое количество процессоров (от 1024 до 16384), которые могут выполнять одну и ту же инструкцию относительно разных данных в жесткой конфигурации. Единственная инструкция параллельно выполняется над многими элементами данных.

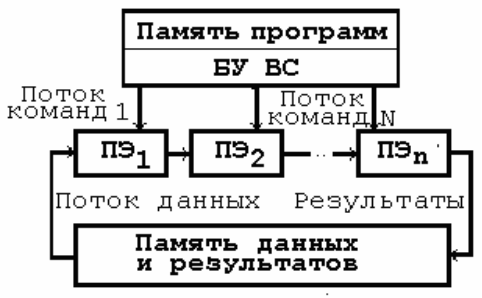


Рис. 38

Примерами SIMD машин являются системы CPP DAP, Gamma II и Quadrics Apemille. Другим подклассом SIMD-систем являются векторные МПС. Они манипулируют массивами сходных данных подобно тому, как скалярные системы обрабатывают отдельные элементы таких массивов. Это делается за счет использования специально сконструированных векторных центральных процессоров. Когда данные обрабатываются посредством векторных модулей, результаты могут быть выданы на один, два или три такта частоты генератора (такт частоты генератора является основным временным параметром системы).

При работе в векторном режиме векторные процессоры обрабатывают данные практически параллельно, что делает их в несколько раз более быстрыми, чем при работе в скалярном режиме. Примерами систем подобного типа является, например, Hitachi S3600.

MIMD - множественный поток команд и множественный поток данных (рис. 39). Эти системы параллельно выполняют несколько потоков инструкций над различными потоками данных. В отличие от многопроцессорных SISD-систем, упомянутых выше,

команды и данные связаны, потому что они представляют различные части одной и той же выполняемой задачи. Например, MIMD-системы могут параллельно выполнять множество подзадач, с целью сокращения времени выполнения основной задачи.

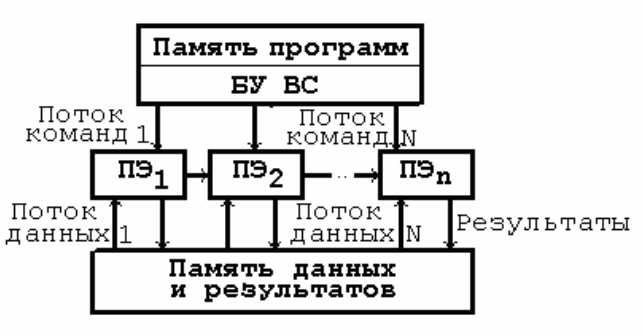


Рис. 39

Наличие большого разнообразия попадающих в данный класс систем, делает классификацию Флинна не полностью адекватной. Действительно и четырехпроцессорный SX-5 компании NEC и тысячепроцессорный Cray T3E оба попадают в этот класс. Это заставляет использовать другой подход к классификации, иначе описывающий классы МПС. Основная идея такого подхода может состоять, например, в следующем. Считаем, что множественный поток команд может быть обработан двумя способами: либо одним конвейерным устройством обработки, работающем в режиме разделения времени для отдельных потоков, либо каждый поток обрабатывается своим собственным устройством. Первая возможность используется в MIMD-системах, которые обычно называют конвейерными или векторными, вторая – в параллельных системах.

В основе векторных МПС лежит концепция конвейеризации, т.е. явного сегментирования арифметического устройства на отдельные части, каждая из которых выполняет свою подзадачу для пары операндов. В основе параллельной системы лежит идея использования для решения одной задачи нескольких процессо-

ров, работающих сообща, причем процессоры могут быть как скалярными, так и векторными.

8.6.1. SMP архитектура

SMP архитектура (symmetric multiprocessing) - симметричная многопроцессорная архитектура (рис.40). Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами.

Память является способом передачи сообщений между процессорами, при этом все вычислительные устройства при обращении к ней имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP архитектура называется симметричной. Последнее обстоятельство позволяет очень эффективно обмениваться данными с другими вычислительными устройствами.

SMP-система строится на основе высокоскоростной системной шины (SGI PowerPath, Sun Gigaplane, DEC TurboLaser), к слотам которой подключаются функциональные блоки трех типов: процессоры (ЦП), операционная система (ОП) и подсистема ввода/вывода (I/O). Для подсоединения к модулям I/O используются уже более медленные шины (PCI, VME64).

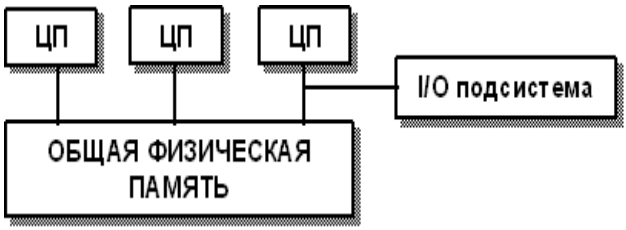


Рис. 40

Наиболее известными SMP-системами являются SMP-сервера и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.) Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически

(в процессе работы) распределяет процессы по процессорам, но иногда возможна и явная привязка.

Основные преимущества SMP-систем:

- простота и универсальность для программирования. Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают абсолютно независимо друг от друга - однако, можно реализовать и модели, использующие межпроцессорный обмен. Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти. Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

- легкость в эксплуатации. Как правило, SMP-системы используют систему охлаждения, основанную на воздушном кондиционировании, что облегчает их техническое обслуживание.

- относительно невысокая цена.

К недостаткам архитектуры можно отнести:

- системы с общей памятью, построенные на системной шине, плохо масштабируемы

Этот важный недостаток SMP-системы не позволяет считать их по-настоящему перспективными. Причины плохой масштабируемости состоят в том, что в данный момент шина способна обрабатывать только одну транзакцию, вследствие чего возникают проблемы разрешения конфликтов при одновременном обращении нескольких процессоров к одним и тем же областям общей физической памяти. Вычислительные элементы начинают друг другу мешать. Когда произойдет такой конфликт, зависит от скорости связи и от количества вычислительных элементов. В настоящее время конфликты могут происходить при наличии 8-24-х процессоров. Кроме того, системная шина имеет ограниченную (хоть и высокую) пропускную способность и ограниченное число слотов. Все это с очевидностью препятствует увеличению производительности при увеличении числа процессоров и числа подключаемых пользователей.

В реальных системах можно использовать не более 32 процессоров. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.

8.6.2. MPP архитектура

MPP архитектура (massive parallel processing) - массивно-параллельная архитектура. Главная особенность такой архитектуры состоит в том, что память физически разделена. В этом случае система строится из отдельных модулей, содержащих процессор, локальный банк операционной памяти (ОП), два коммуникационных процессора (рутера) или сетевой адаптер, иногда - жесткие диски и/или другие устройства ввода/вывода. Один рутер используется для передачи команд, другой - для передачи данных. По сути, такие модули представляют собой полнофункциональные системы (рис.41).

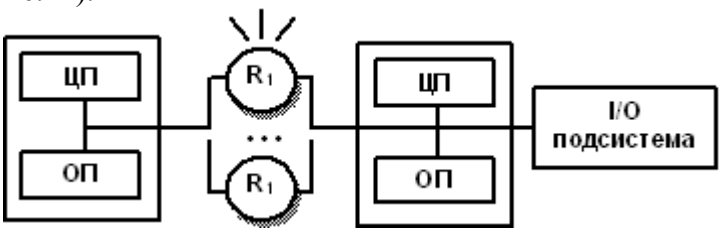


Рис. 41

Доступ к банку ОП из данного модуля имеют только процессоры (ЦП) из этого же модуля. Модули соединяются специальными коммуникационными каналами. Пользователь может определить логический номер процессора, к которому он подключен, и организовать обмен сообщениями с другими процессорами.

Используются два варианта работы операционной системы (ОС) на машинах MPP архитектуры. В одном полноценная операционная система (ОС) работает только на управляющей машине (front-end), на каждом отдельном модуле работает сильно урезанный вариант ОС, обеспечивающий работу только расположенной в нем ветви параллельного приложения. Во втором варианте на каждом модуле работает полноценная UNIX-подобная ОС, устанавливаемая отдельно на каждом модуле.

Главным преимуществом систем с раздельной памятью является хорошая масштабируемость: в отличие от SMP-систем в машинах с раздельной памятью каждый процессор имеет доступ только к своей локальной памяти, в связи с чем не возникает необходимости в потактовой синхронизации процессоров. Практически все рекорды по производительности на сегодняшний день устанавливаются на системах именно такой архитектуры, состоящих из нескольких тысяч процессоров (ASCI Red, ASCI Blue Pacific).

К недостаткам можно отнести:

- отсутствие общей памяти заметно снижает скорость межпроцессорного обмена, поскольку нет общей среды для хранения данных, предназначенных для обмена между процессорами. Требуется специальная техника программирования для реализации обмена сообщениями между процессорами;

- каждый процессор может использовать только ограниченный объем локального банка памяти;

- вследствие указанных архитектурных недостатков требуются значительные усилия для того, чтобы максимально использовать системные ресурсы. Именно этим определяется высокая цена программного обеспечения для массивно-параллельных систем с раздельной памятью.

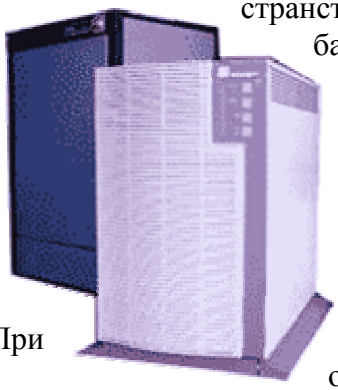
Системами с раздельной памятью являются суперкомпьютеры МВС-1000, IBM RS/6000 SP, SGI/CRAY T3E, системы ASCI, Hitachi SR8000, системы Parsytec.

Системы последней серии CRAY T3E от SGI, основанные на базе процессоров Dec Alpha 21164 с пиковой производительностью 1200 Мфлопс/с (CRAY T3E-1200), способны масштабироваться до 2048 процессоров.

8.6.3. Гибридная архитектура

Гибридная архитектура (NUMA - nonuniform memory access) характеризуется неоднородным доступом к памяти. Она воплощает в себе удобства систем с общей памятью и относительную дешевизну систем с раздельной памятью. Суть этой архитектуры поясним на примере архитектуры сервера NUMA Q-2000, разработанного фирмами IBM и Sequent в 1999 году. Он имеет

особую организацию памяти, а именно: память является физически распределенной по различным частям системы, но логически разделяемой, так что пользователь видит единое адресное пространство. Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти осуществляется в несколько раз быстрее, чем к удаленной. По существу архитектура NUMA является MPP (массивно-параллельная архитектура) архитектурой, где в качестве отдельных вычислительных элементов берутся SMP (симметричная многопроцессорная архитектура) узлы.



Структурная схема системы с гибридной сетью представлена на рис. 42. Она содержит четыре процессора, которые связываются между собой при помощи кроссбара в рамках одного SMP узла. Узлы связаны сетью типа "бабочка" (Butterfly):

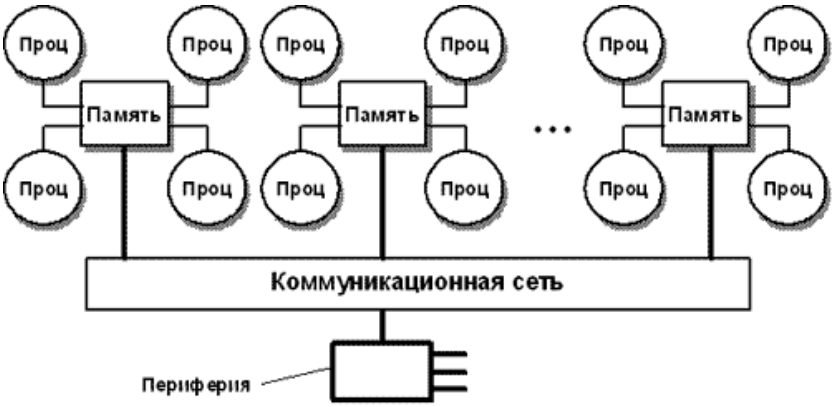


Рис. 42

Впервые идею гибридной архитектуры предложил Стив Воллох и воплотил в системах серии Exemplar. Вариант Воллоха - система, состоящая из 8-ми SMP узлов. Фирма HP купила идею и реализовала на суперкомпьютерах серии SPP. Идею подхватил Сеймур Крей (Seymour R.Cray) и добавил новый элемент - когерентный кэш, создав так называемую архитектуру cc-NUMA (Cache Coherent Non-Uniform Memory Access), которая расшифровывается как "неоднородный доступ к памяти с обеспечением когерентности кэшей". Он ее реализовал на системах типа Origin.

Понятие когерентности кэшей описывает тот факт, что все центральные процессоры получают одинаковые значения одних и тех же переменных в любой момент времени. Действительно, поскольку кэш-память принадлежит отдельной системе, а не всей многопроцессорной системе в целом, данные, попадающие в кэш одной МПС, могут быть недоступны другой. Чтобы избежать этого, следует провести синхронизацию информации, хранящейся в кэш-памяти процессоров.

Для обеспечения подобной когерентности кэшей существуют несколько возможностей:

- использовать механизм отслеживания шинных запросов (snoopy bus protocol), в котором кэши отслеживают переменные, передаваемые к любому из центральных процессоров и, при необходимости, модифицируют собственные копии таких переменных;
- выделять специальную часть памяти, отвечающую за отслеживание достоверности всех используемых копий переменных.

Наиболее известными системами архитектуры cc-NUMA являются: HP 9000 V-class в SCA-конфигурациях, SGI Origin3000, Sun HPC 15000, IBM/Sequent NUMA-Q 2000. На настоящий момент максимальное число процессоров в cc-NUMA-системах может превышать 1000 (серия Origin3000). Обычно вся система работает под управлением единой ОС, как в SMP.

Возможны также варианты динамического "подразделения" системы, когда отдель-

ные "разделы" системы работают под управлением разных ОС.

8.6.4. PVP архитектура

PVP (Parallel Vector Process) - параллельная архитектура с векторными процессорами. Основным признаком PVP-систем является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах. Как правило, несколько таких процессоров (от 1 до16) работают одновременно с общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP). Поскольку передача данных в векторном формате осуществляется намного быстрее, чем в скалярном (максимальная скорость может составлять 64 Гб/с, что на 2 порядка быстрее, чем в скалярных системах), то проблема взаимодействия между потоками данных при распараллеливании становится несущественной. И то, что плохо распараллеливается на скалярных машинах, хорошо распараллеливается на векторных.

Таким образом, системы PVP архитектуры могут являться системами общего назначения (general purpose systems). Однако, поскольку векторные процессоры весьма дороги, эти системы не будут являться общедоступными.

Наиболее популярными системами PVP архитектуры являются:

- CRAY SV-2, SMP архитектура. Пиковая производительность системы в стандартной конфигурации может составлять десятки терафлопс.

- NEC SX-6, NUMA архитектура. Пиковая производительность системы может достигать 8 Тфлопс, производительность 1 процессора составляет 8 Гфлопс. Система масштабируется до 128 узлов.

- Fujitsu-VPP5000 (vector parallel processing)), MPP архитектура. Производительность 1 процессора составляет 9.6 Гфлопс, пиковая производительность системы может достигать 1249 Гфлопс, максимальная емкость памяти - 8 Тб. Система масштабируется до 512 узлов.

8.6.5. Кластерная архитектура

Кластер представляет собой две или больше систем (часто называемых узлами), объединяемых при помощи сетевых технологий на базе шинной архитектуры или коммутатора и предстающих перед пользователями в качестве единого информационно-вычислительного ресурса. В качестве узлов кластера могут быть выбраны серверы, рабочие станции и даже обычные персональные компьютеры.

Преимущество кластеризации для повышения работоспособности становится очевидным в случае сбоя какого-либо узла: при этом другой узел кластера может взять на себя нагрузку неисправного узла, и пользователи не заметят прерывания в доступе. Возможности масштабируемости кластеров позволяют многократно увеличивать производительность приложений для большего числа пользователей. технологий на базе шинной архитектуры или коммутатора. Такие суперкомпьютерные системы являются самыми дешевыми, поскольку собираются на базе стандартных комплектующих элементов ("off the shelf"), процессоров, коммутаторов, дисков и внешних устройств.

Кластеризация может быть осуществлена на разных уровнях компьютерной системы, включая аппаратное обеспечение, операционные системы, программы-утилиты, системы управления и приложения. Чем больше уровней системы объединены кластерной технологией, тем выше надежность, масштабируемость и управляемость кластера.

Кластеры условно делятся на классы:

Класс I. Класс строится целиком из стандартных деталей, которые продают многие продавцы компьютерных компонент (низкие цены, простое обслуживание, аппаратные компоненты доступны из различных источников).

Класс II. Система имеет эксклюзивные или не широко распространенные детали. Этим можно достичь очень хорошей производительности, но при более высокой стоимости.

Как уже указывалось выше, кластеры могут существовать в различных конфигурациях. Наиболее употребляемыми типами кластеров являются:

- системы высокой надежности,
- системы для высокопроизводительных вычислений,
- многопоточные системы.

Заметим, что границы между этими типами кластеров до некоторой степени размыты, и часто существующий кластер может иметь такие свойства или функции, которые выходят за рамки перечисленных типов. Более того, при конфигурировании большого кластера, используемого как система общего назначения, приходится выделять блоки, выполняющие все перечисленные функции.

Кластеры для высокопроизводительных вычислений предназначены для параллельных расчётов. Эти кластеры обычно собраны из большого числа компьютеров. Разработка таких кластеров является сложным процессом, требующим на каждом шаге аккуратных согласований таких вопросов как инсталляция, эксплуатация и одновременное управление большим числом компьютеров, технические требования параллельного и высокопроизводительного доступа к одному и тому же системному файлу (или файлам) и межпроцессорная связь между узлами и координация работы в параллельном режиме. Эти проблемы проще всего решаются при обеспечении единого образа операционной системы для всего кластера. Однако реализовать подобную схему удаётся далеко не всегда и обычно она обычно применяется лишь для не слишком больших систем.

Многопоточные системы используются для обеспечения единого интерфейса к ряду ресурсов, которые могут со временем

произвольно наращиваться (или сокращаться) в размере. Наиболее общий пример этого представляет собой группа Веб-серверов.

В 1994 году Томас Стерлинг (Sterling) и Дон Беккер (Becker) создали 16-и узловой кластер «Beowulf» из процессоров Intel DX4, соединенных сетью 10Мбит/с Ethernet с дублированием каналов Кластер возник в центре NASA Goddard Space Flight Center для поддержки необходимыми вычислительными ресурсами проекта Earth and Space Sciences. Проектно-конструкторские работы над кластером быстро превратились в то, что известно сейчас под названием проект Beowulf.

Проект стал основой общего подхода к построению параллельных кластерных компьютеров и описывает многопроцессорную архитектуру, которая может с успехом использоваться для параллельных вычислений. Beowulf-кластер, как правило, является системой, состоящей из одного серверного узла (который обычно называется головным узлом), а также одного или нескольких подчинённых узлов (вычислительных узлов), соединённых посредством стандартной компьютерной сети. Система строится с использованием стандартных аппаратных компонент, таких как ПК, запускаемых под Linux, стандартных сетевых адаптеров (например, Ethernet) и коммутаторов. Нет особого программного пакета, называемого «Beowulf». Вместо этого имеется несколько кусков программного обеспечения, которые многие пользователи нашли пригодными для построения кластеров Beowulf. Beowulf использует такие программные продукты как операционную систему Linux, системы передачи сообщений PVM, MPI, системы управления очередями заданий и другие стандартные продукты. Серверный узел контролирует весь кластер и обслуживает файлы, направляемые к клиентским узлам.

При разработке кластерных систем существует ряд проблем, среди которых можно выделить следующие.

Архитектура кластерной системы (способ соединения процессоров друг с другом) в большей степени определяет ее производительность, чем тип используемых в ней процессоров. Критическим параметром, влияющим на величину производительности такой системы, является расстояние между процессорами. Так, со-

единив вместе 10 персональных компьютеров, мы получим систему для проведения высокопроизводительных вычислений, проблема, однако, будет состоять в нахождении наиболее эффективного способа соединения стандартных средств друг с другом, поскольку при увеличении производительности каждого процессора в 10 раз производительность системы в целом в 10 раз не увеличится.

Рассмотрим для примера задачу построения симметричной 16-ти процессорной системы, в которой все процессоры были бы равноправны. Наиболее естественным представляется соединение в виде плоской решетки, где внешние концы используются для подсоединения внешних устройств (рис.43).

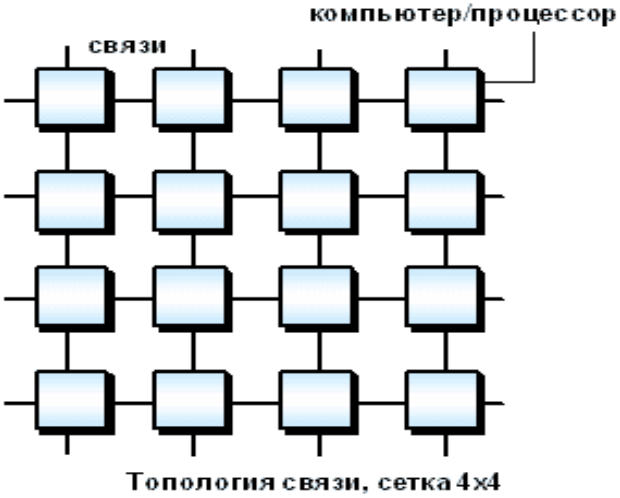


Рис. 43

При таком типе соединения максимальное расстояние между процессорами окажется равным 6 (количество связей между процессорами, отделяющих самый ближний процессор от самого дальнего). Теория же показывает, что если в системе максимальное расстояние между процессорами больше 4, то такая система не может работать эффективно. Поэтому, при соединении 16 процессоров друг с другом плоская схема является не эффективной.

Для получения более компактной конфигурации необходимо решить задачу о нахождении фигуры, имеющей максимальный

объем при минимальной площади поверхности. В трехмерном пространстве таким свойством обладает шар. Но поскольку нам необходимо построить узловую систему, то вместо шара приходится использовать куб (если число процессоров равно 8) или гиперкуб, если число процессоров больше 8. Размерность гиперкуба будет определяться в зависимости от числа процессоров, которые необходимо соединить. Так, для соединения 16 процессоров потребуется 4-х мерный гиперкуб. Для его построения следует взять обычный 3-х мерный куб, сдвинуть в еще одном направлении и, соединив вершины, получить гиперкуб размером 4 (рис. 44). Архитектура гиперкуба является второй по эффективности, но самой наглядной.

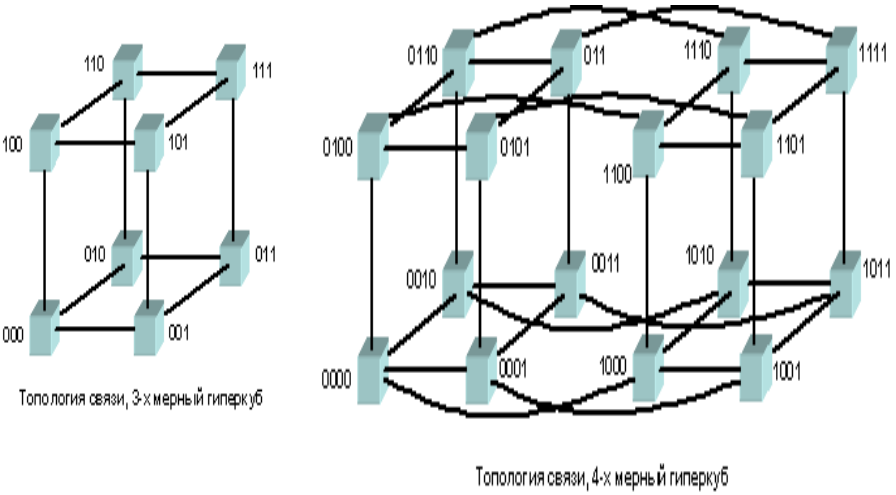


Рис. 44

Используются и другие топологии сетей связи, например,. Наиболее эффективной является архитектура с топологией fat-tree (рис. 45 - архитектура кольца с полной связью по хордам, рис. 46 - кластерная архитектура Fat Free, вид спереди (а) и вид сверху (б)).

. Архитектура fat-tree предложена Лейзерсоном в 1985 году. Процессоры локализованы в листьях дерева, в то время как внутренние узлы дерева скомпонованы во внутреннюю сеть. Поддере-

вья могут общаться между собой, не затрагивая более высоких уровней сети.

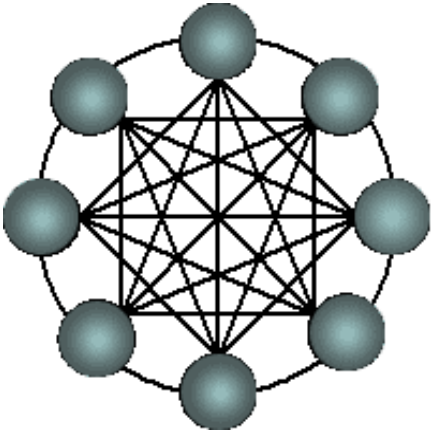
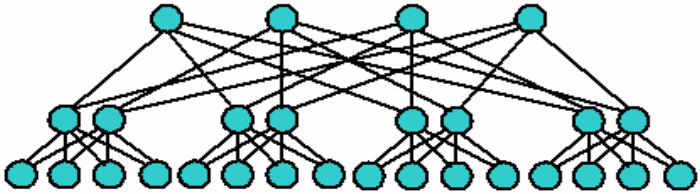


Рис. 45

а)



б)

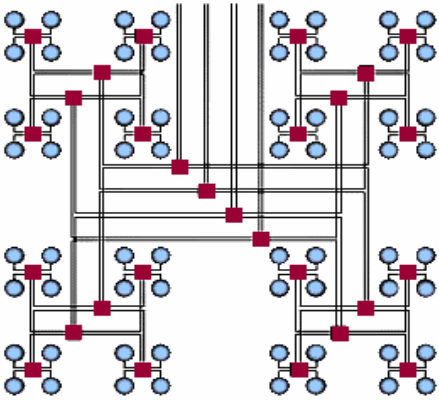


Рис. 46

Поскольку способ соединения процессоров друг с другом больше влияет на производительность кластера, чем тип используемых в ней процессоров, то может оказаться более рентабель-

ным создать систему из большего числа дешевых компьютеров, чем из меньшего числа дорогих.

В кластерах, как правило, используются операционные системы, стандартные для рабочих станций, чаще всего, свободно распространяемые - Linux**,** FreeBSD**,** вместе со специальными средствами поддержки параллельного программирования и балансировки нагрузки.

.

8.6.6. Транспьютеры

Транспьютер – это слово, производное от слов транзистор и компьютер. Транспьютер - это микроэлектронный прибор, объединяющий на одном кристалле микропроцессор, быструю память, интерфейс внешней памяти и каналы ввода-вывода (линки), предназначенные для подключения аналогичных приборов. Прибор спроектирован таким образом, чтобы максимально облегчить построение параллельных вычислительных систем. При соединении транспьютерных элементов между собой требуется минимальное число дополнительных интегральных схем. Связь между транспьютерами осуществляется путем непосредственного соединения линка одного прибора с линком другого. Это позволяет создавать сети с различными топологиями с большим числом элементов.

Производство транспьютеров началось в 1985 году с выпуска 32 – разрядного транспьютера Т414 с фоннеймановской КСНК – архитектурой (рис. 47).

Взаимодействие каждого транспьютера с другими транспьютерами и периферийными устройствами осуществляется посредством 4-х коммуникационных каналов связи, имеющихся в составе БИС. Для передачи сообщений из внутренней и внекристальной локальной памяти по последним каналам применяется механизм блочных ПДП-пересылок. Интерфейсы связи и процессор работают одновременно, что приводит лишь к незначительной потери производительности процессора. Использование прямых последовательных коммуникационных каналов делает ненужным арбитраж приоритетов и исключает проблемы, связанные с пропускной способностью шин и их перегрузкой при введении в систему новых процессоров.

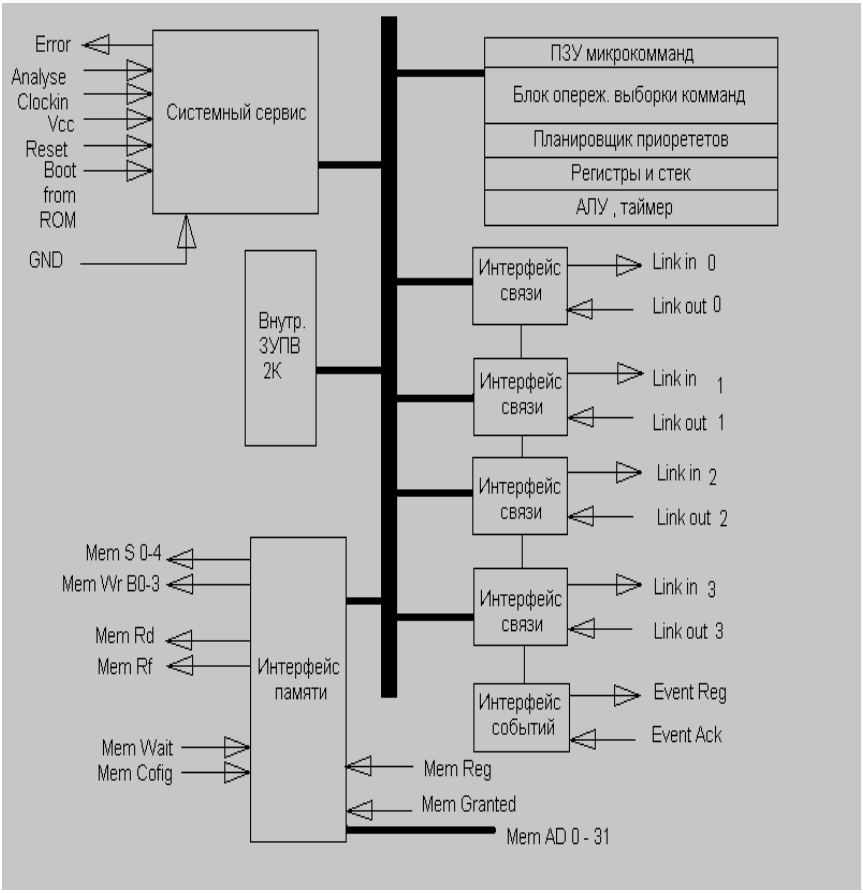


Рис.47.

Каждый исследуемый канал состоит из двух частей, служащих для передачи информации в противоположных направлениях. Пересылка производится в виде последовательностей байтов со скоростью 10-20 Мбит/с, причем каждому байту предшествуют два единичных бита, а завершает передачу один нулевой бит. После передачи байта данных пославший его транспьютер ожидает получения двухбитового подтверждающего сигнала, указывающего на то, что принимающий транспьютер готов к дальнейшему приему информации. Возможен обмен информации между незави-

симотактируемыми системами, если частоты тактирования одинаковы.

Для сопряжения последовательных каналов транспьютера с нетранспьютерными устройствами и интерфейсами связи предусмотрен ряд адаптерных схем. Контроллеры переферийных устройств также могут быть присоединены к имеющейся памяти, посредством которой возможно их обращение ко всему пространству памяти.

Система прерываний в традиционном смысле этого понятия отсутствует, но имеют место аналогичные средства, реализованные в виде двух уровнего приоритета, присваиваемых процессам, ожидающим приема по последовательным каналам и запросному входу события.

Транспьютер может быть использован в качестве отдельного самостоятельного устройства, обеспечивающего производительность 10 млн. оп/с. При этом для программирования используется широкий набор стандартных высокоуровневых языков, т.к. архитектура транспьютера ориентирована на эффективное применение компиляции. Для полной реализации возможностей объединения транспьютеров в сети или матрицы при построении высокопроизводительных систем целесообразно применять язык Окками, позволяющий максимальным образом использовать свойства транспьютеров, ориентированные на распараллеливание обработки.

Аппаратные средства транспьютера прямым образом ориентированы на реализацию параллельной обработки и соответствующих информационных пересылок (рис. 48).

Планировщик поволяет одновременно выполнять любое количество параллельных процессов, между которыми распределяется процессорное время. Время переключения процессов составляет менее 1 мкс, а обмен информации между процессами осуществляется посредством блочных передач ввод-вывод памяти.

Активные процессы, ожидающие выполнения, содержатся в связанной очереди рабочих просранств, которая реализована с помощью двух регистров, один указывает на первый процесс в очереди, а второй указывает на последний процесс. Как только выполнение процесса становится возможным, указатель команды за-

поминается в его рабочем пространстве, и из очереди берется для выполнения следующий процесс.

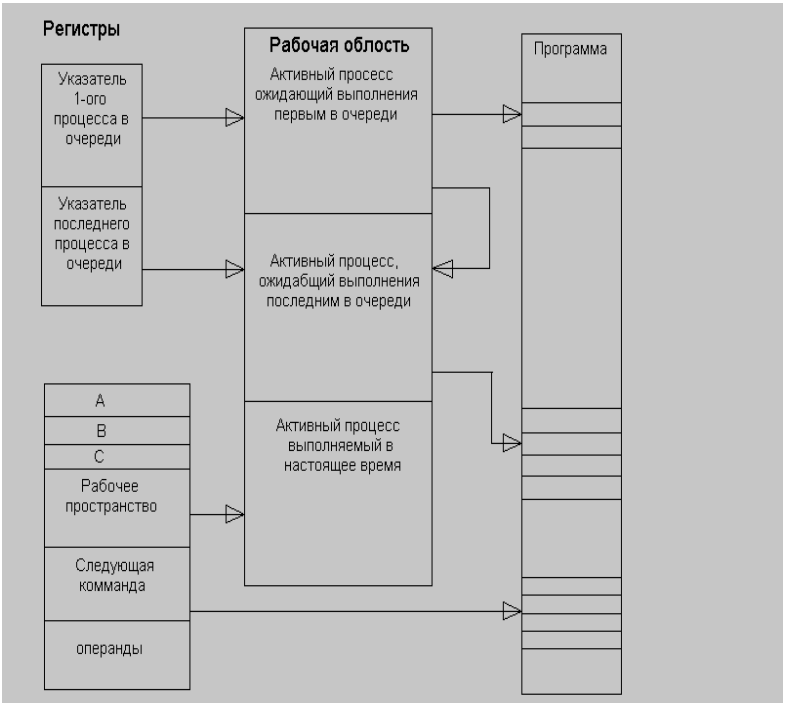


Рис. 48

Транспьютер обеспечивает двухуровневую систему приоритетов. Процессы низкого уровня приоритета должны ожидать выполнения до тех пор, пока в активном состоянии не останется ни одного процесса высокого уровня.

Выполнение процессов низкого уровня разделяется с определенной периодичностью на интервалы времени, чтобы процессорное время равномерно распределялось междузадачами, обработка которых требует больших временных затрат. Продолжительность периода квантования составляет 4096 периодов входного текстового сигнала, имеющего частоту 5 МГц, т.е. примерно 800 мкс. Не должна иметь место ситуация, когда весь период квантов был бы занят выполнением процессов высокого уровня.

Подготовка процесса к вводу или выводу заключается в загрузке в вычислительный стек:

- индентификатора канала;
- количества подлежащих пересылке байтов;
- указателя буфера.

Передача сообщения по внутренему каналу осуществляется путем занесения идентификатора первого процесса, оказавшегося готовым в слово канала и занесения указателя в рабочее пространство. После этого процессор выполняет очередной процесс из планирующей очереди.

Когда оказывается готовым второй процесс, использующий этот же канал, происходит перепись сообщения, ожидающий процесс добавляется к очереди активных процессов и производится начальная установка канала в пустое состояние.

Когда сообщение пересылается по внешнему каналу, процессор возлагает эту передачу на автономный интерфей канала и исключает процесс из плана передач. Есле интерфейс канала реализует передачу сообщения посредством прямого доступа к памяти, процессор вновь вводит ожидающий процесс в расписание пересылок. В то время когда имет место внешняя пересылка сообщения, процессор может выполнять другие процессы.

Интерфейс включает три регистра, которые содержат:

- указатель рабочего пространства процесса;
- указатель сообщений;
- количество байтов подлежащих пересылке.

При необходимости ввода или вывода сообщения производится инициализация этих трех регистров и занесение указателя команды в рабочеепространство процесса. После того как оба процесса инициализируют свои канальные интерфейсы, происходит перепись сообщений и оба интерфейса ставят процесс в конец локальной очереди активных процессов.

Способ, последством которого транспьютерное семейство реализует параллельную обработку при использованиинескольких транспьютерных СБИС, определяется организацией последовательных каналов и Оккам-каналов. Каждый последовательный канал позволяет реализовать два Оккам-канала, по одному для каж-

дого из двух направлений передачи. Пока ЦПУ занят обработкой, обмен информацией может происходить по всем четырем каналам.

При времени цикла транспьютера Т414 50 нс на 4-е канала одновременно работающих в обоих направлениях со скоростью 100Мбит/с с использованием внутренней памяти затрагивается 8% производительности ЦПУ.

Сообщения пересылаются в виде последовательностей байтов. Каждый 8-битовый байт предваряется двумя единичными битами и сопровождается одним нулевым битом. В передающий транспьютер передается сигнал подтверждения приема состоящий из одного единичного и одного нулевого битов. Для обеспечения непрерывности передачи информации сигнал подтверждения начинает передаваться после получения двух заголовочных байтов.

Необходимо отметить, что архитектура транспьютера включает решение проблемы соединений. Для решения этой проблемы INMOS мог бы добавить несколько дополнительных коммукационных каналов в чип, но общее число каналов оставалось бы все-таки небольшим ввиду ограничений технологии VLSI. Вместо этого INMOS избрал радикальное решение - добавление аппаратного мультиплексора, который позволяет четко делить физический коммуникационный канал.

В пару к чипу INMOS также разработал соответствующий высокопроизводительный чип коммутации, так что транспьютеры могут соединяться в полнодоступную сеть коммутации пакетов. Каналы связи между транспьютерами станут виртуальными каналами и их число определяется решаемой программой. С помощью чипов коммуникации, отвечающих за эффективную доставку сообщений по указанному назначению, виртуальные каналы могут запускаться между транспьютерами, которые не соединены напрямую физическими коммуникационными каналами.

Когда осуществляется программирование какой-либо задачи, то необходимо однозначно определить, какой физический коммуникационный канал закрепляется за каждым каналом передачи сообщений и может потребоваться несколько таких однозначных определений для приема сообщения от отдаленного транспьютера. Это делает программу зависимой от точной тополо-

гии конкретной сети, следовательно, она становится неприложимой к сети с любой другой топологией.

В чипе два процесса, расположенные в произвольных местах сети. Они могут использовать один и тот же программный канал для связи. Аппаратура же сама упорядочит маршрут передачи сообщения. С этим улучшением, программы могут быть независимыми от топологии сети, и поэтому более переносимыми.

Архитектура транспьютера включает четыре физических коммуникационных канала. Но она содержит встроенный в чип контроллер коммуникаций, который поддерживает множество виртуальных каналов путем разделения сообщений в пакеты и поочередной передачи пакетов от нескольких сообщений через один физический коммуникационный канал.

Физические коммуникационные каналы чипа почти впятеро быстрее (100 Мбит/сек полный дуплекс), чем транспьютерные физические коммуникационные каналы. Кроме того, системный пакет обеспечивает более эффективное использование канала, поэтому производительность коммуникационного канала будет улучшена немотря на сложность мультиплексирования.

Чип разделяет сообщения произвольной длины на последовательность 32-байтовых пакетов, в отличие от существующего транспьютера, который посылает сообщения байт за байтом. Смешанные пакеты, принадлежащие к различным сообщениям (т.е. к различным виртуальным каналам) требуют, чтобы каждый пакет имел заголовок, который бы говорил, какой виртуальный канал он использует.

Каждый пакет заканчивается специальным меркетом EOP (End-Of-Packet - конец пакета), за исключением последнего пакета, который имеет маркер EOM (End-Of-Message - конец сообщения). Таким образом, нет нужды использовать дополнительную логику для счета пакетов, аппаратура знает, когда сообщение закончилось и позволяет передавать сообщения короче 32 байт (и последние части длинных сообщений, которые могут не занимать целый пакет).

Для сохранения синхронизации передачи сообщений, приемник должен подтверждать получение каждого пакета; подтверждение пакета - это пакет, не содержащий данных. Каждое вирту-

альное соединение содержит два виртуальных канала, один для передачи пакетов данных вовне, другой для приема подтверждений. Процесс посылки пакетов сообщений не может продолжаться до тех пор, пока передддача последнего пакета не будет подтверждена. Подтверждение посылается тогда, когда принимается первый байт пакета, оно разрешает непрерывную передачу сообщений, если приемник готов к вводу информации.

Запросы на передачу ставятся в очередь по каждому физическому коммуникационному каналу связи, так что от МП не требуется ожидания окончания передачи пакета.

В случае, если приемный процесс не готов к приему пакета, чип обеспечивает буферизацию ровно одного пакета на каждый физический коммуникационрый канал. По этой же причине существующие транспьютеры буферизуют 1 байт в аппаратном регистре. В чипе процесс буферизации будет осуществляться в память, что предпочтительнее буферизации в регистр, поэтому в случае достаточного объема и производительности памяти на одном процессоре может быть образовано любое число виртуальных каналов.

Для полной реализации концепции виртуальных каналов, требуется дополнительный чип переключения маршрутов С104 (коммутатор) (рис. 49).

Чип С104 является полным чипом обмена коммутируемыми пакетами, аналогичным по принципу работы телефонной сети органического доступа для обмена деловой информацией. Чип С104 содержит 32 физических коммуникационных транспьютерных канала и может коммутировать сообщение от одного из 32 чипов к любым другим чипам.

Чип С104 содержит коммутатор 32×32 для соединения любого из его физических коммуникационных каналов с любым другим, и чудесно простую логику для подключения адресата для каждого принятого сообщения. Чипу не требуется внутреннее CPU или значительное количество памяти благодаря схеме коммутации, примененной разработчиками INMOSa.

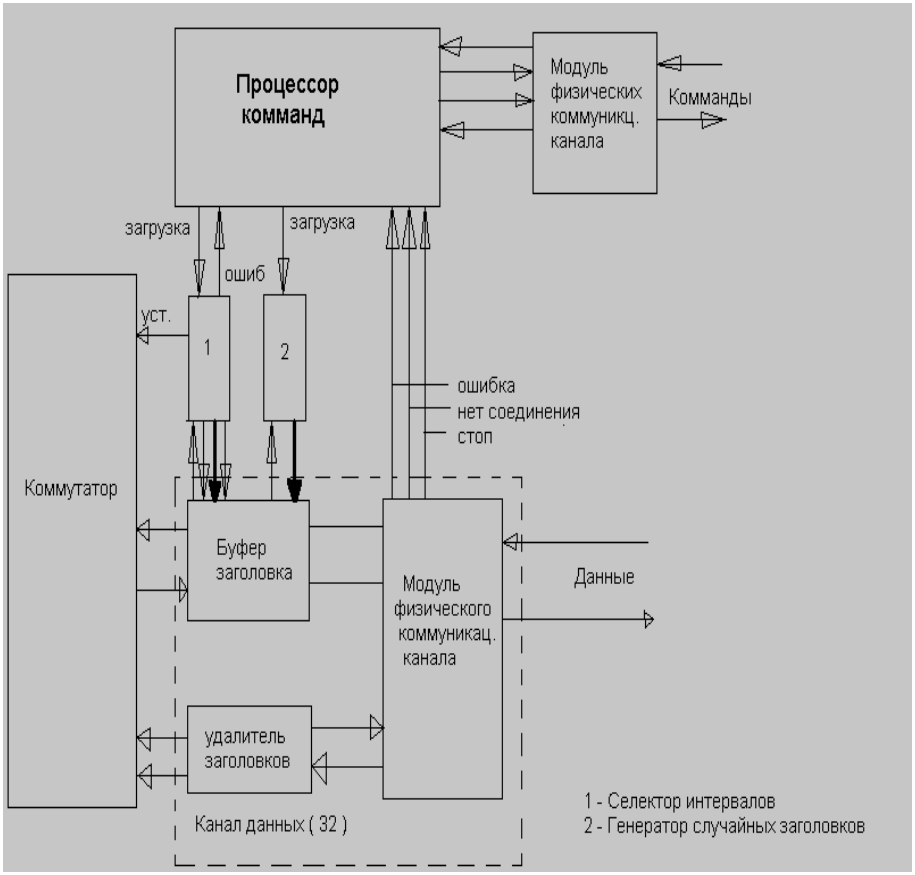


Рис.49

В большинстве сетей коммутации пакетов каждый переключатель маршрута принимает полный пакет и сохраняет его в буфере, декодирует адресную информацию пакета, и пересылает пакет к следующему узлу назначения. Эта схема не очень подходит для высокопроизводительных транспьютерных сетей, поскольку она вносит относительно большую задержку между передачей пакета и его прибыванием в пункт назначения (т.е.имеет большое время ожидания), к тому же требует от чипа коммутации C104 наличия промежуточной буферной памяти.

INMOS выбрал вместо этой схемы решение, называемое червячным каналом маршрутизации, в котором читается только

заголовок пакета и дешифруется его адрес. Затем, если требуемый физический коммуникационный канал свободен, остальная часть пакета посылается напрямую от входа к выходу потоком, без всякой буферизации.

Это означает, что заголовок пакета должен попасть в новый узел коммутации в то время, как тело пакета еще проходит через предыдущие узлы коммутации. В действительности, в сетях среднего размера, заголовок пакета может вообще не появиться от передающего абонента полностью, он достигнет пункта назначения еще до того, как абонент завершит его передачу. В результате, открывается сквозной канал от передающего абонента к принимающему, пока данные передаются непрерывно, и который закрывается, как только конец пакета "протянется" через канал. Аналогия с червячным каналом была выбрана из наблюдения за червяком, прокладывающим канал в песке, который закрывается снова за его хвостом.

В каждый момент времени, через промежуточный узел маршрутизации червячного канала может проходить только один пакет, поэтому любые другие пакеты, которым требуется этот узел, будут ожидать, пока узел не очистится. Итак, давая хорошую стратегию маршрутизации, червячный канал относится к каналам с очень малым временем ожидания, и он подразумевает, что чипу C104 нужен только 1-2 байтовый буфер для заголовков вместо буфера под целый пакет.

Червячный канал маршрутизации в общем невидим для сообщений посылающих и принимающих абонентов, поскольку он создается на аппаратном уровне ниже механизма синхронизации посылки/подтверждения.

Полная маршрутная стратегия требует решения алгоритма маршрутизации пакета, опираясь на адрес, содержащийся в его заголовке, т.е. какой канал будет открыт для передачи пакета. Алгоритмы маршрутизации являются хорошо изученной областью благодаря их колоссальной важности для индустрии передачи сообщений. Хороший алгоритм маршрутизации должен быть завершенным (т.е. он должен гарантировать, что каждый переданный пакет в результате достигнет абонента), не содержать в себе взаимоблокировок (dead-lock), должен быть оптимальным (т.е. пакеты

должны следовать по кратчайшим маршрутам) и малоизбыточным (т.е. пакетыдолжны иметь короткие заголовки).

Он также должен быть масштабируемым (т.е. хорошим для сетей любого размера) и многосторонним (т.е. хорошим для любой топологии сети). Вдобавок, если он предназначен для работы на недорогом, быстром оборудовании, хороший алгоритм маршрутизации должен быть простым. Простота вдвойне важна, когда вы используете быстрый маршрутизатор червячного канала, во избежание ухудшения времени ожидания сети. Не существует алгоритмов, полностью удовлетворяющих всем этим требованиям.

Одним стандартным решением является таблица маршрутизации, которая включает в себя в виде списка все узлы сети и все их соединения, подобно телефонному справочнику; этот алгоритм является завершенным, оптимальным и многосторонним, но не дешевым, быстрым или масштабируемым. Другим общим решением является побитное разрушение, в котором каждый узел сети, которого достигает сообщение, "глотает" один бит из заголовка сообщения; этот алгоритм является завершенным, дешевым, быстрым и оптимальным, но не многосторонним или масштабируемым, поэтому он предназначается для структуры типа двоичное дерево фиксированного размера.

INMOS избрал сравнительно новый алгоритм, называемый интервальной маршрутизацией (рис.50), который является завершенным, свободным от взаимоблокировок, недорогим, быстрым и масштабируемым, могущим быть близким к оптимальному или многосторонним, в зависимости от того, как он используется. Каждый транспьютер (или другой узел назначения, такой, как межсетевой шлюз или переферийный чип) маркируется номером, так сеть из N транспьютеров будет маркирована 0,1,2... N-1. Этот номер используется как адрес назначения в заголовках пакетов.

В каждом переключателе маршрута каждый физический коммуникационный канал маркируется интервалом возможных величин заголовка, и пакеты передаются через тот канал, интервал которого попадают величины их заголовков.

Описание интервала (0,3) должно интерпретироваться как указание на то, что величина заголовка пакета должна больше или равна 0 и меньше 3, для того, чтобы лежать в этом интервале. Ин-

тервалы не перекрываются и пронумерованы так, что каждый заголовок попадает только в один интервал.

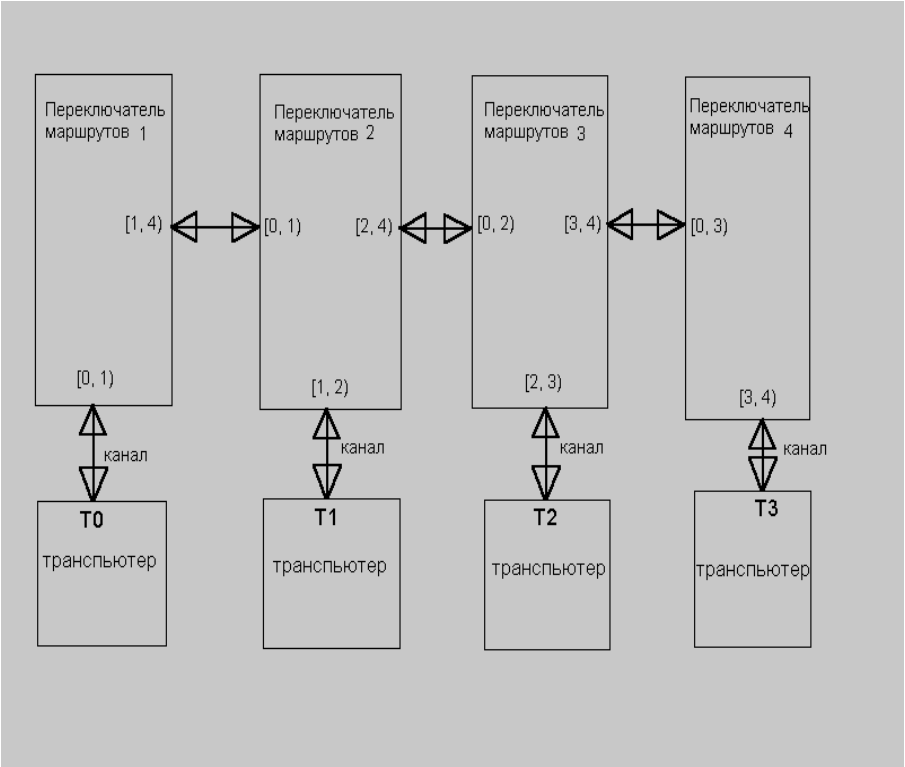


Рис. 50

В результате каждый интервал говорит: "В этом направлении лежит часть сети, содержащая все указанные в интервале номера процессоров". Для разработчика аппаратуры привлекательность метода интервальной маршрутизации заключается в том, что он может быть применен для маршрутизации пакета с использованием одной единственной операции сравнения. Так логика чипа C104 состоит чуть больше, чем из пары компараторов и интегральных регистров на каждый физический коммуникационный канал. Чип C104 может использовать одноили двухбайтовые заголовки, поэтому в одиночной сети может быть маршрутизировано до 65536 транспьютеров.

Хотя схема интервальной маршрутизации может быть гарантирована от взаимоблокировок, она может стать жертвой горячих точек. Когда слишком много сообщений маршрутизировано через один узел сети, большинство доставляемых пакетов остановится на непредсказуемый интервал времени, решающим образом снижая скорость прохождения через узел, такой узел сети называется горячей точкой. Эта проблема сильно отличается от взаимоблокировки, которая является логической ошибкой и результат которой бесконечное ожидание сообщения, это скорее свойство топологии сети и алгоритма маршрутизации, чем применения. Для любой топологии сети, как вы можете догадаться, вероятно найдется алгоритм применения, который может порождать горячие точки.

Можно избежать горячих точек равномерным распределением сетевого трафика, и можно запрограмировать чип C104 для выполнения этого распределения. Маршрутизация становится двухфазным процессом, в первой фазе которого пакет посылается в случайно выбранный узел сети, а из этого узла сообщение посылается к его пункту конечного назначения - схема называется универсальная маршрутизация.

Очевидно, что универсальная маршрутизация увеличит время ожидания в сети и ограничит ее максимальную пропускную способность, но использование этого алгоритма гарантирует, что наихудшая производительность сети будет не сильно отличаться от максимальной. С горячими точками наихудшая производительность сети может ухудшиться на порядки, и фактор дешевизны сети перестает играть роль вследствие непригодности сети для работы. Физически схема универсальной маршрутизации добавляет дополнительный случайный заголовок в начале каждого пакета, который удаляется снова внутри случайного промежуточного узла сети.

Несмотря на то, что механизм виртуальных каналов вызывает наибольший интерес, процессор содержит ряд других улучшений для облегчения жизни разработчиков операционной системы и для улучшения поддержки языков, таких как C и АДА.

Подобно существующему T800, чип будет иметь встроенный сопроцессор плавающей точки (FPU) и четыре физических

коммуникационных канала. Ядро процессора будет иметь встроенное кеширование и контроллер DRAM, который будет поддерживать адресные режимы статической памяти. Эти возможности в счетании сулучшеной технологией изготовления дают перспективу достижения пиковой производительности в 100 млн. операций в секунду и 20 млн. операций с плавающей точкой в секунду. INMOS планирует объединить схему защиты памяти, что позволит каждому процессору иметь доступ к четырём областям памяти. Эти защищённые области могут быть использованы для программ, данных, стека и динамической области памяти при применении операционных систем типа UNIX.

8.6.7. МВС – 1000

МВС-1000 — система 3-го поколения, разработанная в России фирмой «Квант». Она основана на использовании микропроцессоров Alpha 21164 (разработка фирмы DEC-Compaq; выпускается также заводами фирм Intel и Samsung) с производительностью до 1-2 млрд. операций в секунду и присоединенной оперативной памятью объемом 0,1-2 Гбайт.

Мультипроцессорный массив системы с блоками вторичного электросилового питания и вентиляцией располагается в стойках размером 550x650x2200 мм3 промышленного стандарта; вес заполненной стойки — 220 кг, потребляемая мощность до 4 кВт. Система МВС-1000 с производительностью до 1 TFLOPS состоит из 8 стоек (512 узлов).

Система реализуется в двух модификациях: на базе «транспьютероподобного» связного микропроцессора TMS320C44 (фирма Texas Instruments), имеющего 4 канала с пропускной способностью каждого — 20 Мбайт/с, либо на базе связного микропроцессора SHARC ADSP 21060 (фирма Analog Devices), имеющего 6 внешних каналов с пропускной способностью каждого — 40 Мбайт/с.

Исполнение МВС-1000К отличается использованием для межпроцессорного обмена коммутационной сети MYRINET (фирма Myricom, США) с пропускной способностью канала в дуплексном режиме 2x160 Мбайт/с. Кроме того, предусмотрено подклю-

чение к каждому процессору памяти на жестком диске с объемом 2-9 Гбайт. В стандартной стойке располагается до 64 процессоров системы МВС-1000 или 24 процессоров системы МВС-1000К. Предусмотрены средства системного объединения стоек для установок с большим числом процессоров.

Процессорные узлы связаны между собой по оригинальной схеме, сходной с топологией двухмерного тора (для 4-линковых узлов). Структурный модуль (рис.51) состоит из 16 вычислительных модулей (ВМ), образующих матрицу 4х4. При этом четыре угловых элемента матрицы соединяются через транспьютерные линки по диагонали попарно. Оставшиеся 12 линков преднзаначаются для подсоединения внешних устройств (4 линка угловых ВМ) и соединений с подобными модулями. Максимальная длина пути в таком структурном модуле равна трем (против шести в исходной матрице 4х4).

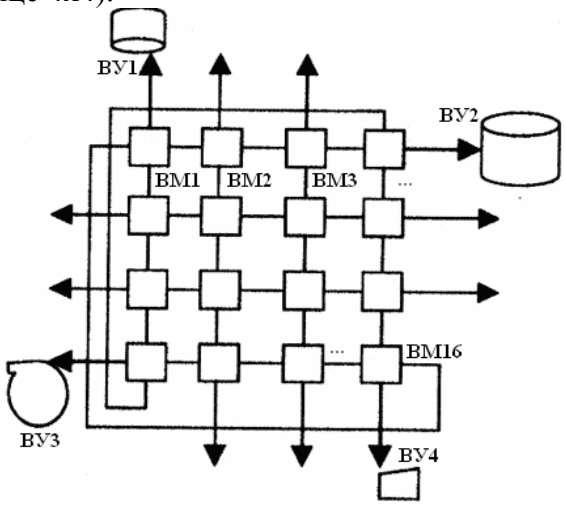


Рис.51

Конструктивным образованием МВС-1000 является базовый вычислительный блок, содержащий 32 вычислительных модуля (рис.52). Максимальная длина пути между любыми между любыми из 32 вычислительных модулей равна пяти, как в булевском гиперкубе. При этом число свободных линков после комплектации блока составляет 16, что позволяет продолжить проце-

дуру объединения. Возможна схема объединения двух базовых блоков в 64-процессорную систему, которая приведена на рис. 53.

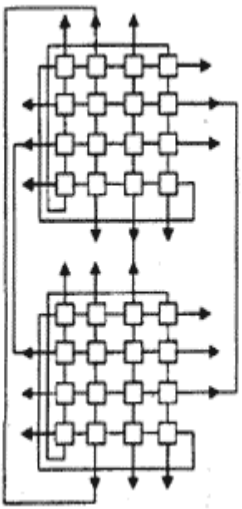


Рис. 52

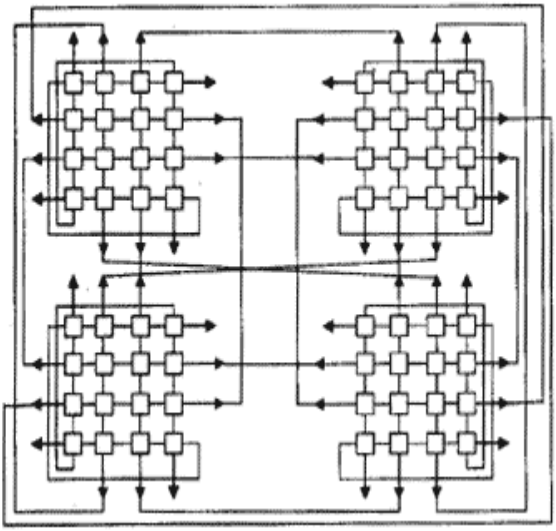


Рис.53

Для управления массивом процессоров и внешними устройствами, а также для доступа к системе извне используется так называемый хост-компьютер (управляющая машина). Обычно это рабочая станция AlphaStation с процессором Alpha и операционной системой Digital Unix (Tru64 Unix) или ПК на базе Intel с операционной системой Linux.

Начиная с 1999 года, все вновь выпускаемые МВС-1000 строятся как кластеры выделенных рабочих станций. Это означает, что, в отличие от ранних версий МВС-1000, в качестве вычислительного модуля используются не специализированные ЭВМ, предназначенные только для применения в качестве деталей суперкомпьютерной установки, а обычные, универсальные персональные компьютеры. Соответственно, и в качестве коммуникационной аппаратуры используются не специализированные «транспьютероподобные» процессоры, а обычные сетевые платы и коммутаторы, применяемые для построения офисных локальных сетей. Такой подход стал не только возможным, но и единственно

оправданным, по мере совершенствования коммуникационной аппаратуры общего назначения, в первую очередь, с появлением современных сетевых коммутаторов.

В качестве базовой ОС узла используется Linux, что является, фактически, общепринятым мировым стандартом для построения систем такого класса. Это позволило многократно расширить и упростить, по сравнению с ранними версиями МВС-1000, адаптацию самого разнообразного программного обеспечения, как свободно распространяемого, так и коммерческого.

8.6.8. Молекулярные компьютеры.

Ученые из Калифорнийского университета в Лос-Анджелесе, работающие в составе группы Hewlett-Packard, заявили о создании молекулярных переключателей, которые способны функционировать как элементы памяти.

Над созданием систем нового поколения исследовательские группы таких крупных корпораций, как IBM и Hewlett-Packard, работают уже давно. Hewlett-Packard специализируется на создании молекулярных, а IBM - квантовых компьютеров.

Преимущества молекулярных компьютеров состоят в чрезвычайно малых размерах и в мизерном количестве потребляемой энергии. В отличие от обычных компьютеров, в молекулярных, вместо кремниевых транзисторов, используются молекулы. Ранее та же группа исследователей смогла заставить молекулы ротаксана переходить из одного состояния в другое. Таким образом, можно было реализовать постоянное запоминающее устройство, вроде CD-ROM диска.

Использование молекулярных элементов для записи информации открывает новые физические возможности повышения эффективности вычислительных систем. Это объясняется существенным уменьшением потерь энергии на выполнение одной логической операции, массовым параллелизмом, высокой плотностью записи информации вплоть до 1 бита на несколько атомов.

Существует огромное число молекулярных систем, имеющих несколько устойчивых состояний (электрических, магнитных, оптических, химических), в которых можно осуществлять перехо-

ды между этими состояниями под действием внешних факторов (электрическое и магнитное поля, электромагнитное излучение, химические реакции). Из таких молекул или групп молекул могут быть сконструированы различные элементарные вычислительные устройства. Размеры этих элементов составляют 10-50 нм и, следовательно, позволяют создавать устройства с плотностью записи до 10^{12} бит/см^2. При этом, быстродействие молекулярных элементов определяется временами электронных переходов и может достигать величин порядка 10^9- 10^{12} оп/сек. Энергопотребление внутримолекулярных процессов очень мало и составляет 10^{-8} - 10^{-10} Вт.

Перечисленные теоретически возможные преимущества молекулярных систем известны давно, однако не было методов создания упорядоченных молекулярных ансамблей. Одним из первых методов решения этой задачи была технология пленок Ленгмюра-Блоджетт.

Первые двоичные молекулярные элементы были основаны на изменении состояния молекулярного ансамбля при воздействии на него электрического сигнала. Одним из таких элементов, является молекулярная цепочка, представляющая собой чередование потенциальных ям и барьеров. В такой цепочке возможен эффект резонансного туннелирования. При совпадении энергии входящего в цепочку свободного электрона с одним из уровней в потенциальной яме вероятность прохождения электрона через цепочку равна 1. При подаче потенциала на одну из молекул, соответствующих квантовой яме уровни смещаются, условия резонанса нарушаются и цепочка становится непроводящей. На основе эффекта резонансного туннелирования можно реализовать различные логические элементы. Для создания таких элементов используются молекулы порфиринов, соединенные углеводородными мостиками.

Для управления состоянием отдельной молекулы применяют зонд туннельного микроскопа, который можно использовать для исследований при разработке и создании молекулярного компьютера.

Архитектура параллельных вычислительных систем, традиционно используемая в разрабатываемой вычислительной технике, реализуется на основе твердотельной микроэлектронной

элементной базы, но совершенно очевидно, что такая архитектура вполне подходит для молекулярных систем. Идея параллельных вычислений открывает путь к решению двух принципиальных проблем создания молекулярных вычислительных устройств. С одной стороны решается проблема адресации, т.к. отпадает необходимость контакта к каждому логическому молекулярному элементу. С другой стороны, многие типы параллельных вычислительных систем устойчивы к выходу из строя определенного числа логических элементов. Это происходит в результате технологических дефектов молекулярных слоев, которые не приводят к выходу из строя всей вычислительной системы, а лишь ослабляют интенсивность выходного сигнала.

В качестве вычислительных систем параллельного действия представляется перспективным использование таких природных биологических объектов, как бактериородопсин, родопсин, фитохром, ряд ферментов и белков, а также природные нейроструктуры. При использовании молекулярных структур необходимо решить такую очень важную проблему, как стыковка биологических молекулярных объектов с макроскопическими опто- и микроэлектронными устройствами ввода-вывода информации. В качестве основного микроэлектронного объекта стыковки элементов памяти с устройствами управления можно рассматривать полевой МДП-транзистор, затвор которого связан с молекулярным фрагментом памяти. Очевидно, что при площади затвора транзистора 1 мкм^2 будет восприниматься суммарный отклик от большого числа молекул (10^3-10^4).

Проблему по снятию сигнала с такого количества молекул можно решать различными методами. Одним из таких методов является стыковка обычных МДП полевых транзисторов с фрагментами пурпурных мембран, которые представляют собой белковые молекулы бактериородопсина, упакованные в двухслойную липидную мембрану. Основным свойством молекулы бактериородопсина является перенос протона с одной стороны мембраны на другую при поглощении кванта света с длиной волны 570 нм.Акт переноса протона может быть зарегистрирован с помощью полевого транзистора, в котором на месте затвора расположена пурпурная мембрана.

Поскольку линейные размеры пурпурных мембран сравнимы с размерами транзисторов (до 1,0 мкм), то такая стыковка оказывается вполне реальной. Слои бактериородопсина, состоящие из пурпурных мембран могут быть нанесены на подложку методом Ленгмюра-Блоджетт. Таким образом, можно выполнять стыковку микроэлектронных систем с молекулярными, а также производить преобразование оптических сигналов в электрические.

Основной особенностью перечисленных выше молекулярных носителей, используемой разработчиками запоминающих структур, является наличие двух состояний молекулярной структуры. Указанная способность позволяет определять текущее состояние молекулы с помощью лазера, настроенного на соответствующую частоту.

Рассмотрим систему памяти, в которой молекулярный носитель запоминает данные в трехмерной матрице в виде прозрачной капсулы, заполненной полиакридным гелем. Один из вариантов такой капсулы может иметь размеры 20 х 20 х 40 мм. Для функционирования памяти капсулу помещают в запоминающую структуру, состоящую из нескольких лазеров и детекторной матрицы, реализованной на базе прибора, использующего принцип зарядовой инжекции, и позволяющей производить запись и чтение данных.

Перед записью данных с помощью желтого страничного лазера на 590 нм производится активизация возбужденной плоскости в материале внутри капсулы. Получение энергоактивной плоскости в виде страницы данных, требует наличия пространственного светового модулятора. Таким модулятором может быть жидкокристаллическая матрица, создающая маску на пути лазерного луча. Таким образом, при активизации записывающей плоскости происходит перевод молекул в О - состояние, оответствующее значению бита "0".

Для записи информации зажигается красный лазер, перед которым также устанавливается пространственный световой модулятор, отображающий двоичные данные. При этом последний создает на пути луча необходимую маску, способствующую облучению только определенных точек страницы. Молекулы в таких

местах будут соответствовать двоичной единице в отличие от оставшейся части страницы, представляющей двоичные нули.

Для чтения информации необходимо включить считывающий лазер, при этом, из-за различия в спектрах поглощения, можно идентифицировать двоичные нули и единицы. Молекулы, представляющие двоичный нуль, поглощают красный свет, а представляющие двоичную единицу пропускают луч мимо себя. В результате получается рисунок из светлых и темных участков на фоторегистрирующей матрице. Последняя позволяет хранить страницу цифровой информации.

Для стирания данных достаточно короткого импульса синего лазера, чтобы вернуть молекулы из состояния двоичной единицы в двоичный нуль. Страница данных может быть прочитана без разрушения до 5000 раз, после чего освежается с помощью новой записи.

С помощью предложенной запоминающей структуры можно получить доступ к данным со скоростью до 90 МВ/с при условии об]единения по восемь запоминающих битовых ячеек в байт с параллельным доступом и соответствующей схемотехнической реализации системы памяти. Емкость данных в рассмотренной кювете может достигать до 1,4 GB, поскольку дальнейшее увеличение емкости данных связано с конструктивными особенностями линзовой системы, а также с качеством непосредственно самого молекулярного носителя.

При обработке молекулярного носителя лазерным лучом во время записи-стирания с энергией выше 8- 10 мДж/см^2 не достигается существенного увеличения уровня сигнала считывания информации. Сигнал с фоторегистрирующей матрицы обрабатывается и регистрируется на ЭВМ.

Проведенные исследования позволяют говорить о хранении и обработке информации на молекулярном уровне. Идентификация двоичных нулей и единиц производится за счет разности спектров поглощений лазерного из- лучения различными состояниями исследуемой молекулярной структуры. При этом молекулы, поглощающие свет, отвечают двоичному нулю, а молекулы, пропускающие лазерный луч мимо себя, отвечают двоичной единице.

Исследуемые молекулярные запоминающие структуры,

кроме сверхвысокой плотности записи информации, почти не имеют потерь энергии на выполнение одной логической операции, а также позволяют достигать массового параллелизма. Гибридные системы хранения и обработки информации, которые совмещают свойства твердотельных ЭВМ и молекулярных вычисли- тельных систем, могут быть эффективно использованы в различных отраслях науки и техники.

Значительные перспективы, которые открывает технология записи и считывания информации на светочувствительном материале, позволяют обеспечить максимальную скорость доступа к данным. Это достигается за счет того, что обрабатываемый массив данных кодируется в один большой блок данных, который, в свою очередь, записывается всего за одно обращение. Если учесть, что такая запоминающая система для обработки информации не содержит движущихся частей и доступ к страницам осуществляется параллельно, то запоминающее устройство в целом может позволить хранить информацию объемом в десятки гигабайт.

Молекулярные компьютеры, по мнению специалистов, должны быть реализованы путем объединения кремниевого чипа с молекулярными компонентами в один гибридный чип. Такой чип будет обладать в тысячи раз большим быстродействием, а память построенная на таком принципе может быть в миллионы раз объемнее. И главное, что подобные технологии будут крайне дешевы, в десятки раз дешевле сегодняшних технологий.

Сама идея этих логических элементов не является революционной: кремниевые микросхемы содержат миллиарды точно таких же. Но преимущества в потребляемой энергии и размерах способны сделать компьютеры вездесущими. Молекулярный компьютер размером с песчинку может содержать миллиарды молекул. А если научиться делать компьютеры не трехслойными, а трехмерными, преодолев ограничения процесса плоской литографии, применяемого для изготовления микропроцессоров сегодня, преимущества станут еще больше.

Кроме того, молекулярные технологии сулят появление микромашин, способных перемещаться и прилагать усилие. Причем для создания таких устройств можно применять даже традиционные технологии травления. Когда-нибудь эти микромашины

будут самостоятельно заниматься сборкой компонентов молекулярного или атомного размера.

Первые опыты с молекулярными устройствами еще не гарантируют появления таких компьютеров, однако это именно тот путь, который предначертан всей историей предыдущих достижений. Массовое производство действующего молекулярного компьютера вполне может начаться где-нибудь между 2005 и 2015 годами.

8.6.9. Оптические МПС

Оптические системы в качестве носителя информации используют свет. Простейшим способом передачи данных при помощи луча света является его модуляция по принципу "есть свет - нет света". У света есть одна особенность: распространяясь в пространстве, он всегда заполняет собой какую-то его часть. Следовательно, существует возможность кодирования информации путем изменения амплитуды и фазы луча не только во времени, но и в пространстве.

Формой реализации этой идеи является оптический элемент, называемый транспарантом. Попросту говоря, это кусочек прозрачного материала, на который каким-либо способом нанесено изображение, представляющее собой пространственное (в координатах, связанных с транспарантом) распределение коэффициента поглощения, коэффициента преломления (или толщины) или же того и другого одновременно. В первом случае получим транспарант, модулирующий пространственное распределение амплитуды проходящего через него света. Примером такого транспаранта может являться кадрик обычной фотопленки.

Во втором - получаем так называемый фазовый транспарант. Как правило, он совершенно прозрачен, и невооруженным глазом невозможно разглядеть на нем никакого изображения. Материал пластинки, будучи неоднороден по величине коэффициента преломления или, чаще всего, по толщине, изменяет (модулирует) распределение фаз проходящей сквозь него световой волны - по той простой причине, что различные участки такого транспаранта пропускают свет за разное время. В конечном итоге это приводит к

возникновению за пластинкой транспаранта заданного пространственного распределения фаз. Еще говорят о формировании определенного фазового профиля волнового фронта (рис. 54).

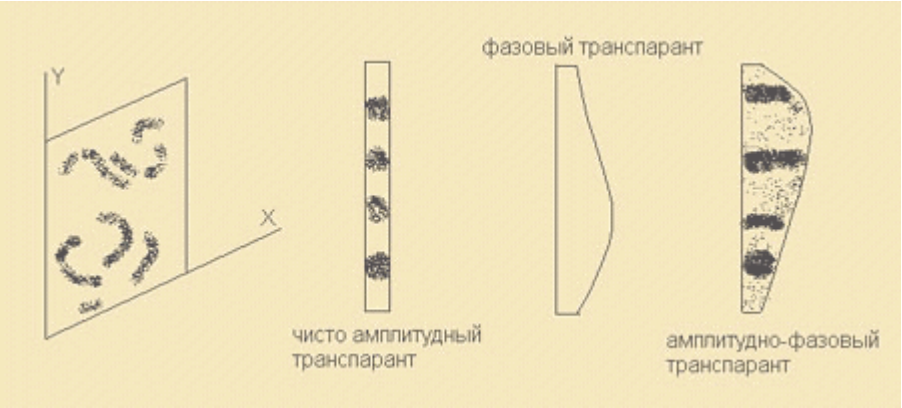


рис. 54

Из сказанного выше становится очевидным, что любой самый привычный оптический элемент - будь то линза или целый объектив - представляет собой амплитудно-фазовый транспарант (иногда используют термин "фильтр").

Транспаранты можно изготавливать на базе технологий, позволяющих управлять оптическими свойствами вещества (жидкие кристаллы или электрооптические материалы), что дает возможность помимо пространственной модуляции осуществлять еще и временную, а также строить адаптивные, в том числе с обратными связями оптические системы.

В технологии оптической обработки данных информацию удобно представлять пространственно-временным распределением амплитуды и фазы светового луча.

Когда мы говорим об амплитудно-фазовой модуляции света, мы, по существу, уже касаемся вопросов не только кодирования или представления информации, но и ее обработки.

Само понятие "модуляция" ничего не говорит о том, что происходит с модулированным сигналом дальше. Привычно думать, что ничего особенного с ним не происходит. Распространяется себе и все.

В данном случае в пространстве за транспарантом световая волна не "просто распространяется". Там, в этой области, интерферируют части волны, прошедшие различные участки транспаранта (в общем случае по принципу "каждый - со всеми"), и формируется новая структура волны, порой совсем не похожая на первоначальную. И этот процесс является управляемым.

Именно поэтому здесь уместно говорить не столько о модуляции, сколько о преобразовании структуры волны, то есть, по сути, об обработке информации, записанной в этой структуре.

Сейчас мы затронули чрезвычайно важный, я бы сказал, принципиально важный момент. Из математики мы знаем, что любые процессы или объекты могут быть представлены как сумма (ряд) некоторых элементарных периодических функций, например, гармонических колебаний. Этот ряд называют еще спектром или спектральным представлением объекта.

Любые изменения некоторого объекта, допустим, оптического изображения, связаны с изменениями его спектрального состава. Поэтому транспаранты в технике оптических вычислений называют еще пространственными амплитудно-фазовыми фильтрами, подчеркивая, что они осуществляют некоторый процесс фильтрации, то есть изменения состава пространственных частот исходного изображения. Именно поэтому и говорят, что комбинация пространственного фильтра и некоторого объема пространства за ним - уже представляет собой оптический процессор.

Как это все эти принципы работают "на практике". Создадим экспериментальную установку: ширма с тремя источниками света A, B и C, экран (см. рис. 55).

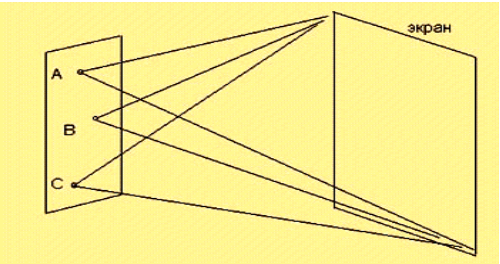


рис. 55

Как сделать, чтобы на экран попадал свет только от источников A и C?

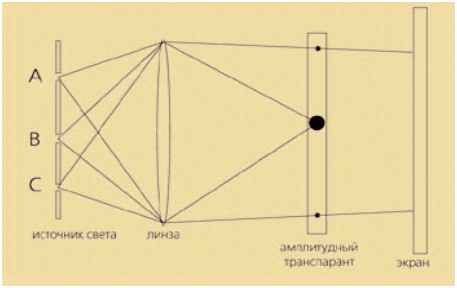


рис. 56

Поставим на пути света линзу (фазовый транспарант). В пространстве за собой она построит изображения источников так, как это показано на рис. 56. Если теперь дополнить нашу установку амплитудным транспарантом-фильтром (попросту говоря, стеклянной пластинкой с непрозрачным участком напротив изображения источника B), то в пространстве за фильтром будет попрежнему присутствовать свет от источников A и C, но света от источника B там не будет.

В случае использования в системе когерентного света при помощи описанного устройства можно фильтровать изображения источников света и световое поле объектов произвольной конфигурации, то есть даже в тех случаях, когда ими являются просто физические объекты.

Три объекта A, B и C освещены когерентным светом. На экран падает свет, отраженный всеми предметами. Вместо экрана, кстати, можно взять фотоаппарат. Тогда на пленке будут сфотографированы все три объекта.

Можно ли сделать так, чтобы фотоаппарат, снимая сцену, не зафиксировал объект B? Как "вычесть" отраженное от него световое поле из общего поля сцены? В рамках методов обычной оптики эта задача неразрешима.

Для решения этой задачи применяют специальный вид амплитудно-фазовых транспарантов, которые называются комплексно-сопряженными фильтрами. Математические подробности авто-

ры не приводят, но отмечают, что эти оптические элементы обладают свойством собирать в одну точку световое поле, обладающее некоторой заранее заданной структурой. Если нам известна конфигурация (амплитудно-фазовое распределение) поля, рассеянного объектом В, то, изготовив соответствующий фильтр, можно решить поставленную задачу, например, так (рис. 57):

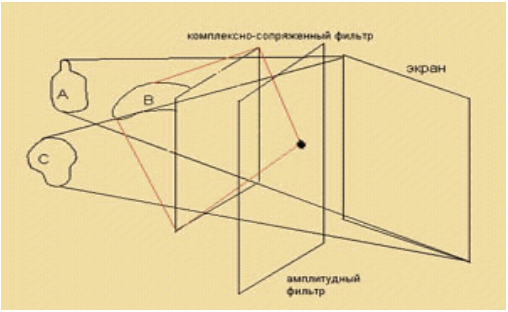


Рис.57

Свет, рассеянный объектом В, собирается в точку, где перекрывается амплитудным фильтром. Теперь на пленке в фотоаппарате будет формироваться все та же сцена, что и раньше, но объекта В на ней не будет.

Описанные процессы используются в качестве основы для проектирования оптических устройств распознавания образов или быстрого поиска необходимой информации в больших массивах данных, сформированных на оптических носителях - фото- или голографических транспарантах.

Теперь самое время рассмотреть случаи, когда требуется не просто выделение или распознавание требуемого информационного объекта, но именно преобразование его.

Из схемы записи (съемки) голограммы (рис. 58), понятно, что, во-первых, голограмма в сути своей представляет не что иное, как амплитудно-фазовый транспарант, на котором зафиксировано поле, получившееся в результате интерференции опорного пучка и отраженного от объекта света. А во-вторых, что опорный источник сам может являться некоторым объектом.

Тогда становится очевидным, что если обычная голограмма, освещаемая опорным лучом, восстанавливает изображение объекта съемки, то голограмма, полученная путем записи интерференции света от двух объектов, будет в свете одного из них восстанавливать изображение другого и наоборот (это свойство носит название "принцип обратимости голограмм").

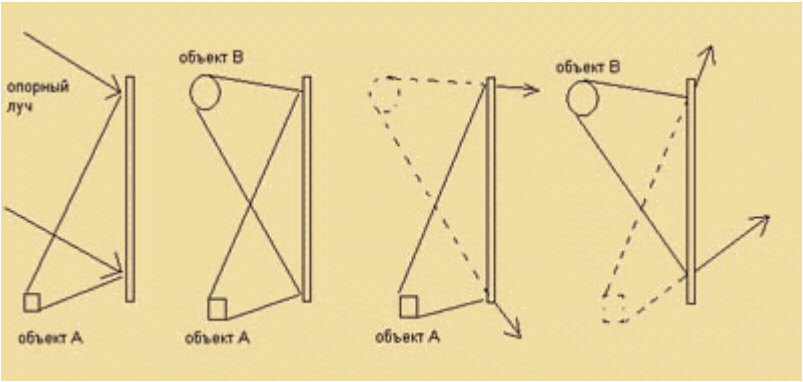


рис. 58

Практическая польза здесь очевидна: получается возможность осуществить "преобразование" любого изображения в заранее заданное другое (строго говоря, не преобразование, а как бы "замену" или "подстановку").

Много возможностей таит включение в схемы оптических вычислителей контуров обратных связей и нелинейных элементов!

Так же много перспектив сулит интенсивно развивающийся в последние годы подход к созданию оптических процессоров как квантовых устройств с распределенными параметрами. В них уже не найти привычных линз, зеркал или пластинок-фильтров. Их функции распределены по объему рабочего пространства процессора. Массовое производство такого рода устройств по своему революционному воздействию на отрасль сравнимо с началом выпуска интегральных микросхем.

Обработка информации в оптической системе может осуществляться как в процессе переноса изображения (представляющего собой специальным образом подготовленный входной сиг-

нал) через оптическую систему, реализующую вычислительную среду, так и путем осуществления переключений в так называемом оптическом транзисторе. Можно легко показать, что при линейных размерах изображения 1 см, разрешении 3 мкм и длине оптической системы порядка 30 см (давно доступные оптикам технологические нормы) можно получить пиковую производительность порядка 10^{16} элементарных операций в секунду!

Коммутация информационных каналов в оптическом компьютере осуществляется с большой скоростью и отличается простотой реализации за счет того, что лучи света в пустом пространстве распространяются, не взаимодействуя друг с другом. По сравнению с обычной электроникой выигрыш очень быстро растет с ростом числа коммутируемых каналов. Использование третьего измерения для ввода/вывода информации в оптоэлектронных чипах создает дополнительные возможности, которым у электронных соединений нет никаких аналогов.

Еще одно уникальное свойство оптических систем: в прозрачной среде информация, закодированная оптическим лучом, может обрабатываться без затрат энергии. Естественно, закон сохранения энергии при этом не нарушается. Наконец, отметим, что оптическая система ничего не излучает во внешнюю среду, обеспечивая защиту компьютера от перехвата информации. И обратно: оптическая система надежно защищена от сторонних электромагнитных наводок.

Основные элементы оптических компьютеров с переносом изображения давно известны. Это - линза, зеркало, оптический транспарант и слой пространства. В настоящее время к ним добавились волноводные элементы, а также лазеры, полупроводниковые многоэлементные фотоприемники, нелинейные оптические среды, разного рода дефлекторы и светоклапанные устройства.

Базисная логическая функция, элементарный кирпичик, с помощью которого можно построить любой, сколь угодно сложный цифровой компьютер, имеет множество оптических реализаций. На рис. 59 дан простой пример построения многовходовой функции ИЛИ-НЕ/И-НЕ с помощью линзы L и порогового устройства-инвертора N.

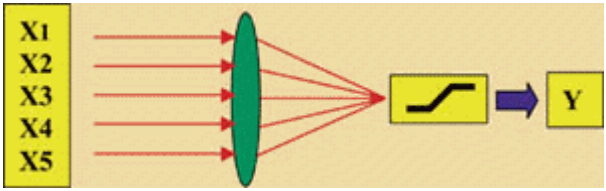


Рис. 59

Здесь в качестве порогового элемента можно использовать как оптическое светоклапанное устройство (переключающаяся, бистабильная оптическая среда), так и простой фотоэлектронный приемник с нелинейной передаточной характеристикой (то есть нелинейной зависимостью интенсивности выходного светового потока от входного).

На рис. 60 показан оптический процессор, реализующий произвольное матричное преобразование входного вектора-строки в выходной вектор-столбец. Здесь LED - линейка светоизлучающих диодов. Они расположены на фокальной линии цилиндрической линзы L1. T - оптический транспарант с записанной на нем матрицей пропускания T(i, j).

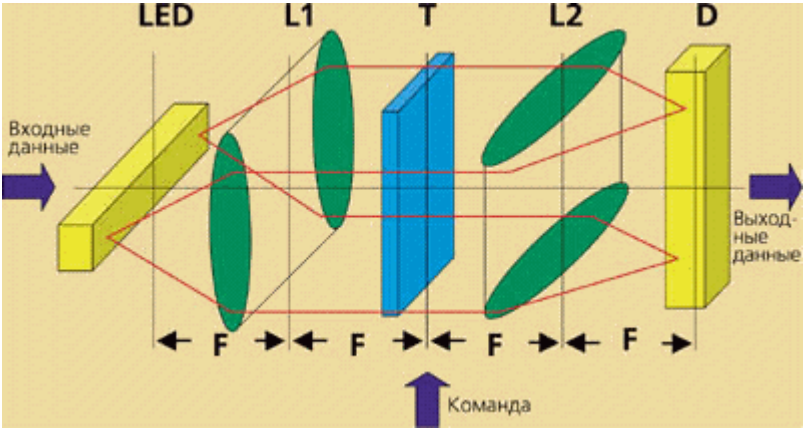


Рис. 60

Строки матрицы параллельны образующей первой линзы. L2 - цилиндрическая линза, образующая которой параллельна столбцам матрицы транспаранта. Она собирает лучи, прошедшие

через элементы одной строки, на одном пикселе многоэлементного линейного фотоприемника D. Нетрудно видеть, что входной X и выходной Y вектора связаны линейным преобразованием Y=TX

В оптической системе возможна также обработка двумерных структур. На рис. 61 представлена схема оптического процессора, реализующего операцию свертки двух изображений, которая лежит в основе работы многих устройств ассоциативной памяти и распознавания образов.

Здесь S - плоский однородный источник света, L1 и L2 - сферические линзы, D - матричный фотоприемник, T1 и T2 - транспаранты, пропускание которых соответствует двум обрабатываемым изображениям. Распределение интенсивности излучения на матричном фотоприемнике пропорционально интегралу J(x, y)=тт(T1(x-u, y-v) T2(u, v) du dv.

В предыдущих примерах свет выполнял ту же роль, что и электроны в проводниках обычных микросхем. При этом в качестве "проводов" выступали геометрические лучи. Понятно, что с таким же успехом свет можно загнать в волновод и организовать вычислительную среду по принципам, близким к идеологии электронной полупроводниковой микросхемотехники. Этим занимается интегральная и волоконная оптика.

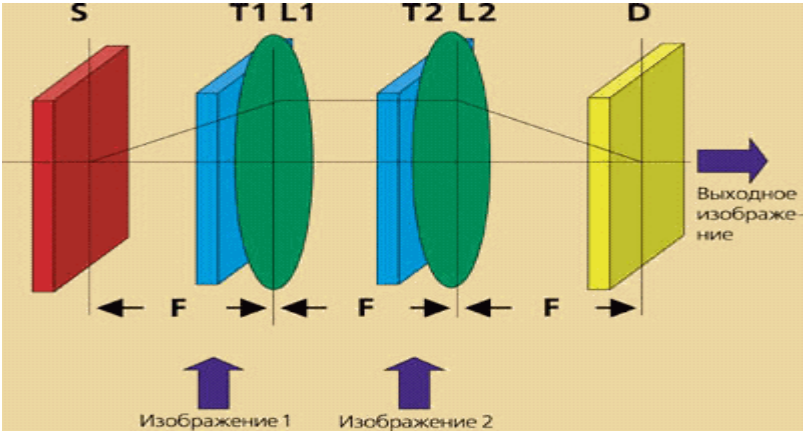


Рис. 61

Принципиально новые возможности дает использование свойств пространственной когерентности излучения. Структура когерентного оптического процессора, так называемая 4F-схема, приведена на рис. 62. Здесь LS - лазерная осветительная система, формирующая широкий пучок когерентного излучения. T1 и T2 - амплитудно-фазовые транспаранты, модулирующие фазу и амплитуду проходящей световой волны. L1 и L2 - сферические линзы с фокусным расстоянием F. Результирующий сигнал считывается матричным фотоприемником D.

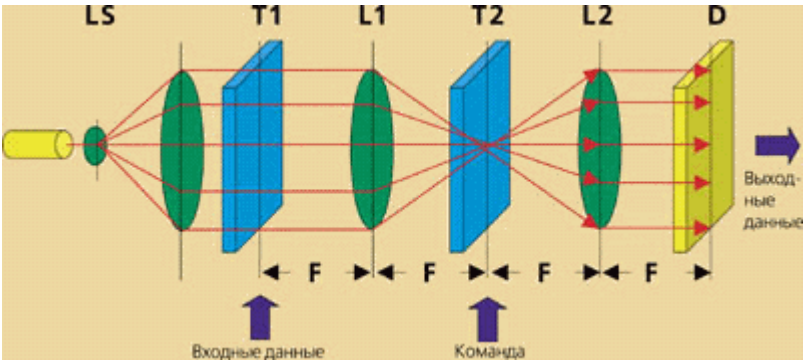


Рис. 62

Распределение амплитуды светового поля в плоскости фотодетектора пропорционально свертке амплитудного пропускания первого транспаранта с Фурье-образом амплитудного пропускания второго транспаранта. Процессоры такого типа используются в качестве комплексных пространственных фильтров в системах улучшения качества изображения, а также в системах распознавания образов.

Если же нас интересует Фурье-спектр двумерного сигнала, то он вычисляется с помощью линзы L и слоя пространства длиной F так, как показано на рис. 63. Остальные элементы предназначены для ввода-вывода данных и для освещения системы.

Отметим, что для обычного компьютера, использующего быстрый алгоритм Кули-Тьюки, длительность выполнения Фурье-преобразования растет с ростом числа точек дискретизации n пропорционально n log(n). В оптическом компьютере эта процедура даже в двумерном случае выполняется всего за один машинный

такт, что делает оптический компьютер незаменимым для военных целей, а также для решения задач, требующих быстрой оценки ситуации и управления в реальном времени.

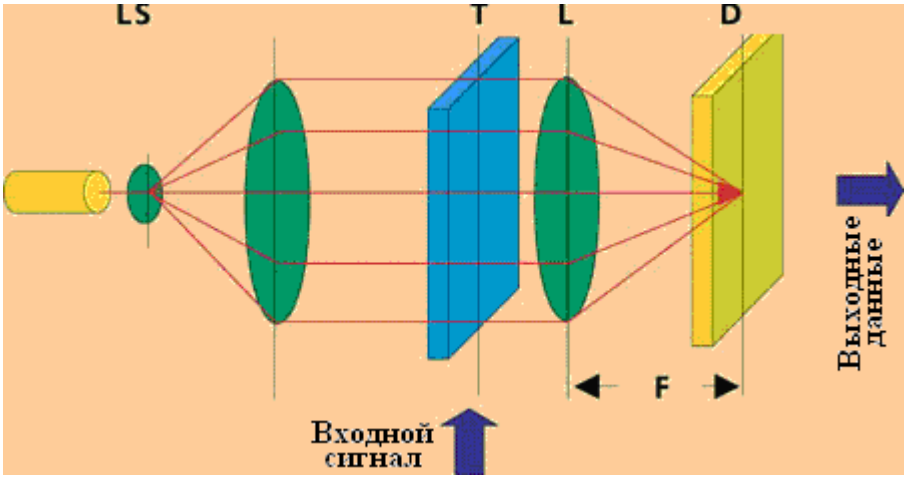


Рис. 63

Замечательным свойством оптического компьютера с передачей изображения является его способность за один такт обрабатывать двумерные картинки, причем машинная команда сама может представляться картинкой. Откуда же эти картинки брать? Их источником могут быть внешняя среда, результат предыдущего вычисления, оптическая память. Если из внешней среды изображение можно ввести с помощью объектива, а результат предыдущего вычисления возвратить на вход процессора с помощью системы зеркал, то в качестве памяти можно использовать различные оптические и оптоэлектронные устройства записи, хранения и извлечения изображений. Например, кассету со слайдами.

Особый интерес представляют голографические устройства памяти. Такая память обладает рядом достоинств. Голограмма сохраняет информацию не только об интенсивности, но и о фазе световой волны, что в оптике принципиально важно, а с утилитарной точки зрения - позволяет повысить объем записываемой информации. Кроме того, различные картинки можно записывать в одно и то же место, используя весь объем носителя, а не тонкий слой по-

верхности (как в случае обычной оптической или магнитной памяти). По оценкам специалистов, объемная плотность записи информации может превышать величину 10^{11} бит/см^3, а скорость ввода информации с голограмм - несколько гигабит в секунду.

Кроме того, голограмма сама может использоваться в качестве принципиального узла оптического процессора, выступая одновременно в роли буферной памяти и обрабатывающего элемента. Информация в таких голограммах записывается путем изменения показателя преломления по всему объему носителя. Обычно для этого используются прозрачные материалы с ярко выраженными нелинейными оптическими характеристиками, например, кристаллы ниобата лития.

На рис. 64 показано примерное устройство оптической памяти с объемной голографической средой. Ввод информации осуществляется с помощью управляемого оптического транспаранта. Адресацией при записи-считывании управляет опорный луч. Считываемая информация фокусируется в плоскости многоэлементного матричного фотоприемника D.

В силу того, что емкость голографической памяти огромна, а время выборки мало, кардинально меняется весь подход к организации вычислительного процесса. Например, можно вернуться к идее широкого использования поиска по справочникам и таблицам функций, схем принятия решений, таблиц умножения, наконец. Разумеется, алгоритмический подход к обработке информации в оптическом компьютере сохранится, но его основой будет язык более высокого уровня организации, ориентированный на параллельную обработку сложных структур данных.

Речь пойдет об интерфейсах. Может показаться, что оптическая вычислительная среда, обладая уникальной способностью одновременно обрабатывать большие массивы информации, как бы оторвана от хорошо разработанной и всенародно любимой электронной вычислительной инфраструктуры, в которой обработка и передача информации осуществляется последовательно. Однако на самом деле любой оптический компьютер должен быть обрамлен обычной электроникой.

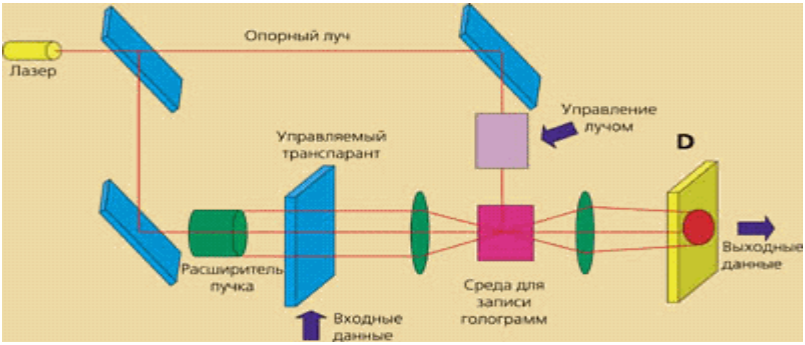


Рис. 64

Сегодня при вводе оптического изображения в ЭВМ вакуумные передающие телевизионные трубки полностью вытеснены приборами с зарядовой связью ПЗС (CCD - charge coupled device). Устройства на основе ПЗС воспринимают изображение параллельно по оптическому каналу, а передают последовательно по электронному. В ряде случаев последовательная перекачка содержимого ПЗС в оперативную память ЭВМ ограничивает быстродействие всего оптоэлектронного вычислительного комплекса. В настоящее время имеются российские и зарубежные разработки оптоэлектронных интерфейсов, в которых матричный фотоприемник конструктивно совмещен с элементами RAM в оперативной памяти ЭВМ. Отсюда - один шаг до использования оптического канала доступа к электронным чипам через третье измерение. Такие работы сегодня ведутся в зарубежных лабораториях. Оказывается, даже в стандартной персоналке длина соединений между вентилями микросхем, расположенных на разных печатных платах, порой измеряется метрами. Использование оптического канала для организации "междучипового" обмена позволяет на порядок сократить задержку распространения сигнала.

От ЭВМ к оптической системе информация передается с помощью управляемых транспарантов, или пространственных модуляторов света ПМС (SLM - spatial light modulator). Однако жидкокристаллические панели, чаще всего использующиеся для этих целей, обладают существенным недостатком - сравнительно низким быстродействием. Существуют также оптические транспаран-

ты с электронным управлением на цилиндрических магнитных доменах и термопластиках.

К числу перспективных направлений относится разработка ПМС с так называемыми умными ячейками (SP, smart pixels). Они могут модулировать амплитуду падающей световой волны как с использованием сигналов электронного интерфейса, так и с помощью второй световой волны. По сути - это прозрачная интегральная схема, выполненная по оптоэлектронной технологии и использующая параллельный оптический вход и выход информации с помощью третьего измерения. Интересно, что для первых экспериментов с устройствами типа SLM-SP обычные чипы кремниевой памяти покрывали тонким слоем жидкого кристалла. Электрическое поле вблизи заряженных ячеек памяти ориентирует молекулы жидкого кристалла, обнаруживая электрооптический эффект в отраженном свете. Впоследствии для этой цели стали разрабатывать специальные чипы, а также применять электро-магнитооптические жидкости и использовать эффект Фарадея. Такие транспаранты оказались более быстродействующими.

Еще один класс интерфейсных устройств образуют многоэлементные лазеры. Последнее достижение в этой области - матрицы полупроводниковых лазеров размером 1000x1000 светоизлучающих элементов. Излучение генерируется перпендикулярно плоскости чипа. В таких оптоэлектронных БИС используется технология изготовления лазеров с вертикальным расположением резонаторов (VCSEL, vertical cavity surface-emitting lasers). Недостаток подобных устройств состоит в том, что в каждый момент времени свет может излучаться только лазерами, расположенными либо в одной строке, либо в одном столбце матрицы.

По-видимому, в дальнейших разработках следует ожидать объединения технологий SLM-SP и VCSEL.

Правда жизни на стороне "аналоговой" природы: в некоторых наиболее эффектных своих применениях оптический процессор является аналоговой машиной. С другой стороны, способность оптической вычислительной среды образовывать параллельные связи между большим числом элементов и одновременно выполнять операции типа взвешивания и суммирования аналоговых сигналов является идеальной для построения нейросетей. Если в сис-

тему ввести транспарант, пропускание которого задано значениями весовых коэффициентов W(i), получим простейшую модель нейрона - персептрон, показанный на рис. 65.

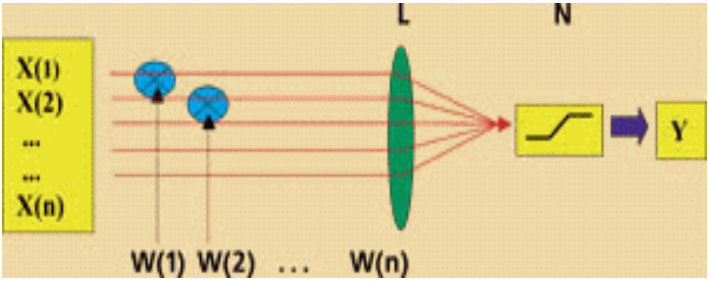


Рис. 65

Аналогично схему, представленную на рис. 65, можно рассматривать как слой нейронов. Для этого элементы фотоприемника должны иметь пороговую передаточную характеристику, а транспарант должен задавать таблицу весовых коэффициентов. Причем матрица преобразования совершенно не обязана быть квадратной. Более того, оптическими средствами можно создать связи между нейронами, организованными в двумерные массивы. Матрица связей при этом (формально) будет четырехмерной.

Для работы нейросети низкая точность вычислений, присущая аналоговым компьютерам, не имеет большого значения.

Объединяя рассмотренные оптические схемы в многослойные структуры и вводя в них обратные связи, можно создавать сложные быстродействующие нейросети. А на их основе - нейрокомпьютеры широкого применения. Это направление считается одним из самых перспективных в области оптического компьютеростроения.

Успехи оптоэлектроники последнего десятилетия позволили достаточно хорошо отработать технологию производства всех основных компонентов оптического компьютера. Уже сейчас его можно было бы эффективно использовать в качестве сателлитного устройства к обычной персональной ЭВМ. Однако массового производства оптических компьютеров для широкого использования не наблюдается. Причина первая: долгое время работы в этой об-

ласти были закрытыми. Вторая причина проистекает из первой: никто не занимался маркетинговой проработкой оптического компьютера для народа.

Стоимость оптического компьютера довольно велика. Главные области его использования - военная техника, криминология, защита информации, банковская сфера, а также научные исследования. Основные работы за рубежом ведутся в США, Японии, Западной Европе. В США одним из ведущих координаторов работ является известное Агентство по передовым оборонным исследовательским проектам (DARPA, Defence Advanced Research Projects Agency). То самое Агентство, которое стояло у истоков Bitnet, прототипа Интернета. Исследования и разработки ведутся также в интересах ВВС, ВМС, НАСА, Организации СОИ и др. Решаемые задачи: машинное зрение, искусственный интеллект, распознавание и одновременное сопровождение сотен и тысяч целей, навигация, контроль земной поверхности, связь.

Из последних разработок в гражданском секторе отметим оптический процессор для проверки подлинности кредитных карт, паспортов и других важных документов. Дело в том, что жулики научились читать и подделывать обычные отражательные голограммы.

Сотрудники Коннектикутского университета и Оптического научного центра Аризонского университета предложили преобразовывать изображение отпечатков пальцев, лица, подписи владельца документа в специальный фазовый профиль, который одновременно записывается в прозрачной голографической среде на карте и в базе данных. Проверить подлинность карты можно лишь с помощью оптического процессора или коррелятора. Подделать фазовый профиль на прозрачной пластинке практически невозможно.

Уже в 1996 году фирмой IBM была разработана голографическая память. Для ввода картинок в память использовался управляемый транспарант размером 1024x1024 пикселей, способный работать с частотой 1 тыс. изображений в секунду. Для вывода информации применялась камера ПЗС размером 2048x2048 пикселей.

Фирмой OptiComp (США) разработан оптоэлектронный 32-битный RISC-процессор, способный обеспечить независимые соединения между 8192 оптическими каналами. Его производительность составляет около 10^{12} двоичных операций в секунду. Работая в режиме поиска текста, процессор может просматривать большие базы данных со скоростью 80 тыс. страниц (на каждой странице по 5 тыс. знаков) в секунду.

В совместной разработке Колорадского университета и исследовательского института JILA (США) оптический нейрокомпьютер используется для распознавания человеческой речи. Фурье-спектры отрезков человеческой речи преобразовывались в визуальную картину, которая далее обрабатывалась оптическими методами. Система работала гораздо быстрее и надежнее, чем специально обученная нейросеть на электронных чипах.

Несмотря на определенные сложности в разработке и изготовлении оптических компьютерных систем, идея его создания не умерла. Более того, сейчас она снова стала актуальной. Дело в том, что, увеличивая и увеличивая быстродействие процессора, производители с той же неизбежностью вынуждены работать со все более и более мелкими объектами. Сейчас отметка находится на рубеже 0,06 мкм, но это, разумеется, не предел. Нанотехнология, которая вначале тоже рассматривалась как некая полуфантастическая технология, постепенно превратилась в обычную задачу для современных технологов. Вот тут и выяснилось, что в нанотехнологии оптические подходы очень даже хороши.

Это можно объяснить следующим. С быстродействием у оптических компонентов все отлично. Если тактовая частота современных компьютерных систем только сейчас подошла к 3-4 ГГц, то оптические переключатели преодолели этот рубеж еще лет двадцать назад. Эффективность работы оптических переключателей можно видеть на примере волоконно-оптических линий связи. Здесь оптическая технология окончательно победила технологию электрическую.

Наибольшее распространение получили переключатели на основе $LiNbO_3$. Ниобат лития - уникальный кристалл. У него отличные электрооптические характеристики, он обладает замечательными акустооптическими свойствами и вдобавок ко всему еще

и нелинейными характеристиками. Поэтому в системах обработки оптической информации устройства на основе ниобата лития встречаются чаще всего.

В области создания элементов памяти для оптических систем тоже достигнуты значительные успехи, более того, в области оптической памяти в последние годы получены особенно интересные результаты. Активизация этих работ была стимулирована развитием носителей информации, а именно лазерных компакт-дисков и систем магнитооптической записи. Поскольку CD и магнитооптические диски уже стали коммерческими продуктами, то крупные фирмы имеют возможность наряду с финансированием чисто прикладных задач поддерживать и другие исследования в области оптической памяти.

Магнитооптическая память уже сейчас является вполне доступным продуктом. При этом стандартная плотность записи коммерческого диска составляет около 1 бит на 3 мкм2 (что соответствует 128 Мбайт для 3,5-дюймового диска).

Исследования в этой области продвинулись сейчас очень далеко. В Information Storage Technology Group of MESA+ Institute (университет Твенте, г. Энсхеде, Нидерланды) успешно проведена работа по детектированию бита информации в области 5 Ангстрем. Счет идет просто на отдельные атомы. Полученные результаты действительно впечатляют: если эти разработки дойдут до промышленного производства, то 3,5-дюймовые диски емкостью в десятки гигабайт станут вполне доступны.

Есть еще одно направление оптической памяти - голографическая память. И здесь надо снова вспомнить о $LiNbO_3$. Мы уже говорили о нем, когда речь шла о системах оптической обработки информации. Разумеется, оставить такой выдающийся кристалл без внимания было нельзя. В последние годы на нем научились получать лазерную генерацию (легировав кристалл эрбием), а сейчас создают целые системы голографической памяти. Число голограмм на одном кристалле достигло десяти тысяч, и это не предел.

В оптических системах проблема интеграции является наиболее важной и в конечном итоге решающей. Когда все компоненты разработаны и испытаны, возникает очередная проблема - собрать все воедино. Для оптического компьютера эта проблема зна-

чительно сложнее, чем для обычного. Электрические связи реализовать гораздо проще. Можно, конечно, все элементы объединить с помощью оптического волокна (и такие технологии тоже развиваются), но в таких системах существуют довольно значительные потери информации. Поэтому не прекращаются попытки сделать все на одной "платформе".

Конечно, первое, на что ориентируются разработчики, так это кремний. Технология изготовления, обработки и производства разработаны до последних мелочей, существуют огромные производственные мощности. Нужна лишь база для производства. Поэтому любые новые оптические разработки на основе кремния принимаются обычно на ура. Но у кремния есть один хронический недостаток: он плохо совместим с активными оптическими элементами. Оптические усилители и переключатели очень плохо уживаются с кремниевой технологией.

Следующее направление - создание базы на основе альтернативных полупроводниковых структур. Здесь есть свои очень важные плюсы: можно формировать структуры с совершенно разными свойствами, встраивать их в оптические интегральные схемы (ОИС), нет проблем с источником света, поскольку ОИС сами могут генерировать лазерное излучение. Более того, не так давно произошел прорыв в области полупроводниковых лазеров, и они сейчас безусловные лидеры продаж. Казалось бы, вот готовая платформа для оптических компьютеров. Но есть две проблемы.

Первая - технологическая. Она связана со сложностью переориентации разработчиков систем на кремниевой основе на совершенно новое направление. Вторая - экономическая. Главной причиной убедительной победы кремниевой технологии была дешевизна. Кремния на земном шаре много и он достаточно дешев для использования в массовом производстве. Для новых структур требуются материалы гораздо более редкие и, как следствие, более дорогие, например галлий. Да и технология гораздо дороже.

Возникает законный вопрос: так будет ли в ближайшем будущем оптический компьютер, или нет. У оптических компьютеров есть свой "троянский конь", и имя ему - Internet. То, что в ближайшие годы именно Internet будет определять положение на компьютерном рынке, сейчас понимают почти все. При передаче

информации главной проблемой является трафик. Число пользователей растет, объем информации тоже. И здесь оптика уже полностью и окончательно одержала верх над "классической" электроникой. И сейчас не только вся информация перегоняется по оптическому кабелю, но и системы сопровождения тоже стали оптическими: электроника просто не тянет таких скоростей передачи и обработки информации. Пропускная же способность оптических систем растет, и возможности роста колоссальны.

Один из основных способов увеличения пропускной способности - системы спектрального уплотнения. И не зря было упомянуто, что на структурах из ниобата лития делают не только переключатели, лазерные усилители и голограммы. На них создают системы спектрального уплотнения. Почему бы не предположить, что вслед за чисто оптическими системами передачи Internet-информации могут прийти и чисто оптические системы обработки информации.

8.6.10. Нейронная архитектура

Нейрокомпьютер - это вычислительная система с архитектурой MSIMD, в которой реализованы два принципиальных технических решения: упрощен до уровня нейрона процессорный элемент однородной структуры и резко усложнены связи между элементами; программирование вычислительной структуры перенесено на изменение весовых связей между процессорными элементами.

Общее определение нейрокомпьютера может быть представлено в следующем виде. Нейрокомпьютер - это вычислительная система с архитектурой аппаратного и программного обеспечения, адекватной выполнению алгоритмов, представленных в нейросетевом логическом базисе

Каждый нейрон получает сигналы от соседних нейронов по специальным нервным волокнам. Эти сигналы могут быть возбуждающими или тормозящими. Их сумма составляет электрический потенциал внутри тела нейрона. Когда потенциал превышает некоторый порог, нейрон переходит в возбужденное состояние и посылает сигнал по выходному нервному волокну. Отдельные искус-

ственные нейроны соединяются друг с другом различными методами. Это позволяет создавать разнообразные нейронные сети с различной архитектурой, правилами обучения и возможностями.

Термин “искусственные нейронные сети” у многих ассоциируется с фантазиями об андроидах и бунте роботов, о машинах, заменяющих и имитирующих человека. Это впечатление усиливают многие разработчики нейросистем, рассуждая о том, как в недалеком будущем, роботы начнут осваивать различные виды деятельности, просто наблюдая за человеком. Если переключиться на уровень повседневной работы, то нейронные сети это всего-навсего сети, состоящие из связанных между собой простых элементов формальных нейронов. Большая часть работ по нейроинформатике посвящена переносу различных алгоритмов решения задач на такие сети.

В основу концепции положена идея о том, что нейроны можно моделировать довольно простыми автоматами, а вся сложность мозга, гибкость его функционирования и другие важнейшие качества определяются связями между нейронами. Каждая связь представляется как совсем простой элемент, служащий для передачи сигнала. Коротко эту мысль можно выразить так: “структура связей все, свойства элементов ничто”.

Совокупность идей и научно-техническое направление, определяемое описанным представлением о мозге, называется коннекционизмом (connection связь). С реальным мозгом все это соотносится примерно так же, как карикатура или шарж со своим прототипом. Важно не буквальное соответствие оригиналу, а продуктивность технической идеи.

С коннекционизмом тесно связан следующий блок идей:

- однородность системы (элементы одинаковы и чрезвычайно просты, все определяется структурой связей);
- надежные системы из ненадежных элементов и “аналоговый ренессанс” использование простых аналоговых элементов;
- “голографические” системы при разрушении случайно выбранной части система сохраняет свои свойства.

Предполагается, что широкие возможности систем связей компенсируют бедность выбора элементов, их ненадежность и возможные разрушения части связей.

Для описания алгоритмов и устройств в нейроинформатике выработана специальная “схемотехника”, в которой элементарные устройства (сумматоры, синапсы, нейроны и т.п.) объединяются в сети, предназначенные для решения задач. Для многих начинающих кажется неожиданным, что ни в аппаратной реализации нейронных сетей, ни в профессиональном программном обеспечении эти элементы вовсе не обязательно реализуются как отдельные части или блоки. Используемая в нейроинформатике идеальная схемотехника представляет собой особый язык описания нейронных сетей и их обучения. При программной и аппаратной реализации выполненные на этом языке описания переводятся на более подходящие языки другого уровня.

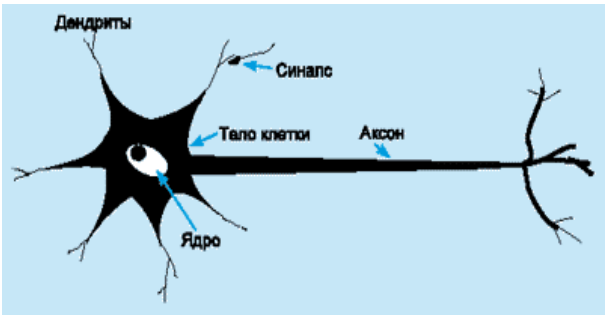


Рис. 66

Нейрон (нервная клетка) является особой биологической клеткой, которая обрабатывает информацию (рис. 66). Она состоит из тела клетки (cell body), или сомы (soma), и двух типов внешних древоподобных ветвей: аксона (axon) и дендритов (dendrites). Тело клетки включает ядро (nucleus), которое содержит информацию о наследственных свойствах, и плазму, обладающую молекулярными средствами для производства необходимых нейрону материалов. Нейрон получает сигналы (импульсы) от других нейронов через дендриты (приемники) и передает сигналы, сгенерированные телом клетки, вдоль аксона (передатчик), который в конце разветвляется на волокна (strands). На окончаниях этих волокон находятся синапсы (synapses).

Синапс является элементарной структурой и функциональным узлом между двумя нейронами (волокно аксона одного нейрона и дендрит другого). Когда импульс достигает синаптического окончания, высвобождаются определенные химические вещества, называемые нейротрансмиттерами. Нейротрансмиттеры диффундируют через синаптическую щель, возбуждая или затормаживая, в зависимости от типа синапса, способность нейрона-приемника генерировать электрические импульсы. Результативность синапса может настраиваться проходящими через него сигналами, так что синапсы могут обучаться в зависимости от активности процессов, в которых они участвуют. Эта зависимость от предыстории действует как память, которая, возможно, ответственна за память человека.

Кора головного мозга человека является протяженной, образованной нейронами поверхностью толщиной от 2 до 3 мм с площадью около 2200 $см^2$, что вдвое превышает площадь поверхности стандартной клавиатуры. Кора головного мозга содержит около 1011 нейронов, что приблизительно равно числу звезд Млечного пути. Каждый нейрон связан с 103 - 104 другими нейронами. В целом мозг человека содержит приблизительно от 1014 до 1015 взаимосвязей.

Нейроны взаимодействуют посредством короткой серии импульсов, как правило, продолжительностью несколько мсек. Сообщение передается посредством частотно-импульсной модуляции. Частота может изменяться от нескольких единиц до сотен герц, что в миллион раз медленнее, чем самые быстродействующие переключательные электронные схемы. Тем не менее, сложные решения по восприятию информации, как, например, распознавание лица, человек принимает за несколько сотен мс. Эти решения контролируются сетью нейронов, которые имеют скорость выполнения операций всего несколько мс. Это означает, что вычисления требуют не более 100 последовательных стадий. Другими словами, для таких сложных задач мозг "запускает" параллельные программы, содержащие около 100 шагов. Это известно как правило ста шагов. Рассуждая аналогичным образом, можно обнаружить, что количество информации, посылаемое от одного нейрона другому, должно быть очень маленьким (несколько бит). От-

сюда следует, что основная информация не передается непосредственно, а захватывается и распределяется в связях между нейронами.

История создания искусственных нейронов уходит своими корнями в 1943 год, когда шотландец МакКаллок и англичанин Питтс создали теорию формальных нейросетей, а через пятнадцать лет Розенблатт изобрел искусственный нейрон (персептрон), который, впоследствии, и лег в основу нейрокомпьютера.

Искусственный нейрон имитирует в первом приближении свойства биологического нейрона. На вход искусственного нейрона поступает некоторое множество сигналов, каждый из которых является выходом другого нейрона. Каждый вход умножается на соответствующий вес, аналогичный синаптической силе, и все произведения суммируются, определяя уровень активации нейрона.

На рис. 67 представлена модель реализующая эту идею. Хотя сетевые парадигмы весьма разнообразны, в основе почти всех их лежит эта конфигурация. Здесь множество входных сигналов, обозначенных x_1, x_2, $x_3...x_n$, поступает на искусственный нейрон. Эти входные сигналы, в совокупности обозначаемые вектором **X**, соответствуют сигналам, приходящим в синапсы биологического нейрона. Каждый сигнал умножается на соответствующий вес w_1, w_2, $w_3...w_n$, и поступает на суммирующий блок (**Σ**) (адаптивный сумматор). Каждый вес соответствует "силе" одной биологической синаптической связи (множество весов в совокупности обозначается вектором **W**).

Суммирующий блок, соответствующий телу биологического элемента, складывает взвешенные входы алгебраически, создавая на выходе сигнал **XW**. Он преобразуется активационной функцией **F** и дает выходной нейронный сигнал **OUT** .

Активационная функция может быть обычной линейной функцией:

OUT=K(XW),

где **K** - постоянная, пороговой функцией,

OUT=1, если **XW>T,**

OUT=0 в остальных случаях,

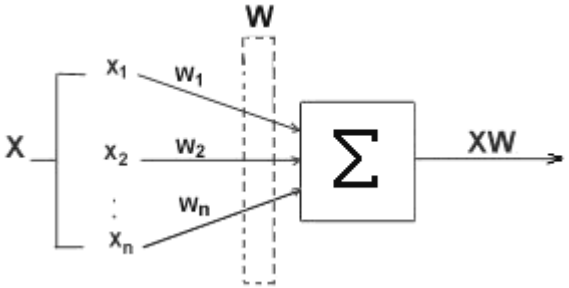


Рис. 67.

где **T** - некоторая постоянная пороговая величина, или же функцией более точно моделирующей нелинейную передаточную характеристику биологического нейрона и представляющей нейронной сети большие возможности.

На рис. 68 блок обозначенный **F** , принимает сигнал **XW** и выдает сигнал **OUT**. Если блок **F** сужает диапазон изменения величины **XW** так, что при любых значениях **XW** значения **OUT** принадлежат некоторому конечному интервалу, то **F** называется сжимающей функцией.

В качестве сжимающей функции часто используется логистическая или сигмоидальная (S- образная) функция, показанная на рис. 69. Эта функция математически выражается как

$$F(x)=1/(1+e^{-x}) .$$

Таким образом, $OUT=1/(1+e^{-XW})$.

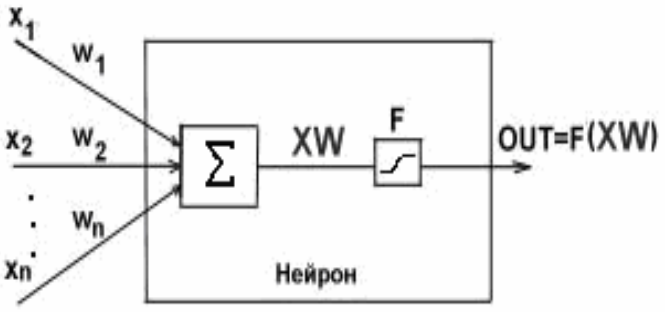


Рис. 68

По аналогии с электронными системами активационную функцию можно считать нелинейной усилительной характеристикой искусственного нейрона. Коэффициент усиления вычисляется как отношение приращения величины OUT к вызвавшему его небольшому приращению величины XW. Он выражается наклоном кривой при определенном уровне возбуждения и изменяется от малых значений при больших отрицательных возбуждениях (кривая почти горизонтальна) до максимального значения при нулевом возбуждении и снова уменьшается, когда возбуждение становится большим положительным.

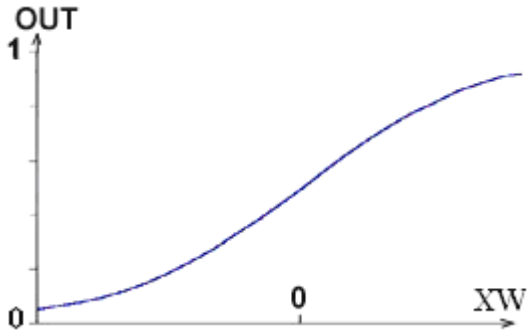


Рис. 69.

Гроссберг (1973) обнаружил, что подобная нелинейная характеристика решает поставленную им дилемму шумового насыщения. Слабые сигналы нуждаются в большом сетевом усилении, чтобы дать пригодный к использованию выходной сигнал. Однако усилительные каскады с большими коэффициентами усиления могут привести к насыщению выхода шумами усилителей (случайными флуктуациями), которые присутствуют в любой физически реализованной сети. Сильные входные сигналы в свою очередь также будут приводить к насыщению усилительных каскадов, исключая возможность полезного использования выхода. Центральная область логистической функции, имеющая большой коэффициент усиления, решает проблему обработки слабых сигналов, в то время как в области с падающим усилением на положительном и

отрицательном концах подходят для больших возбуждений. Таким образом, нейрон функционирует с большим усилением в широком диапазоне уровня входного сигнала.

Рассмотренная простая модель искусственного нейрона игнорирует многие свойства своего биологического двойника. Например, она не принимает во внимание задержки во времени, которые воздействуют на динамику системы. Входные сигналы сразу же порождают выходной сигнал. И что более важно, она не учитывает воздействий функции частотной модуляции или синхронизирующей функции биологического нейрона, которые ряд исследователей считают решающими. Несмотря на эти ограничения, сети, построенные из этих нейронов, обнаруживают свойства, сильно напоминающие биологическую систему. Только время и исследования смогут ответить на вопрос, являются ли подобные совпадения случайными или следствием того, что в модели верно схвачены важнейшие черты биологического нейрона

Каждый нейрон получает сигналы от соседних нейронов по специальным нервным волокнам. Эти сигналы могут быть возбуждающими или тормозящими. Их сумма составляет электрический потенциал внутри тела нейрона. Когда потенциал превышает некоторый порог, нейрон переходит в возбужденное состояние и посылает сигнал по выходному нервному волокну. Отдельные искусственные нейроны соединяются друг с другом различными методами. Это позволяет создавать разнообразные нейронные сети с различной архитектурой, правилами обучения и возможностями.

Искусственная Нейронная Сеть (ИНС) может рассматриваться как направленный граф со взвешенными связями, в котором искусственные нейроны являются узлами. По архитектуре связей ИНС могут быть сгруппированы в два класса: сети прямого распространения, в которых графы не имеют петель, и рекуррентные сети, или сети с обратными связями.

В наиболее распространенном семействе сетей первого класса, называемых многослойным перцептроном, нейроны расположены слоями и имеют однонаправленные связи между слоями. На рисунке представлены типовые сети каждого класса. Сети прямого распространения являются статическими в том смысле, что на заданный вход они вырабатывают одну совокупность вы-

ходных значений, не зависящих от предыдущего состояния сети. Рекуррентные сети являются динамическими, так как в силу обратных связей в них модифицируются входы нейронов, что приводит к изменению состояния сети.

Здесь каждый нейрон передает свой выходной сигнал остальным нейронам, включая самого себя. Выходными сигналами сети могут быть все или некоторые выходные сигналы нейронов после нескольких тактов функционирования сети. Все входные сигналы подаются всем нейронам. Элементы слоистых и полносвязных сетей могут выбираться по-разному. Существует, впрочем, стандартный выбор: нейрон с адаптивным неоднородным линейным сумматором на входе. Для полносвязной сети входной сумматор нейрона фактически распадается на два: первый вычисляет линейную функцию от входных сигналов сети, второй линейную функцию от выходных сигналов других нейронов, полученных на предыдущем шаге. Функция активации нейронов (характеристическая функция) это нелинейный преобразователь выходного сигнала сумматора. Если функция одна для всех нейронов сети, то сеть называют однородной (гомогенной). Если же характеристическая функция зависит еще от одного или нескольких параметров, значения которых меняются от нейрона к нейрону, то сеть называют неоднородной (гетерогенной).

Составлять сеть из нейронов стандартного вида не обязательно. Слоистая или полносвязная архитектуры не налагают существенных ограничений на участвующие в них элементы. Единственное жесткое требование, предъявляемое архитектурой к элементам сети, это соответствие размерности вектора входных сигналов элемента (она определяется архитектурой) числу его входов. Если полносвязная сеть функционирует до получения ответа заданное число тактов k, то ее можно представить как частный случай k-слойной сети, все слои которой одинаковы и каждый из них соответствует такту функционирования полносвязной сети.

Существенное различие между полносвязной и слоистой сетями становится очевидным, когда число тактов функционирования заранее не ограничено и слоистая сеть так работать не может. Доказаны теоремы о полноте: для любой непрерывной функции нескольких переменных можно построить нейронную сеть,

которая вычисляет эту функцию с любой заданной точностью. Так что нейронные сети в каком-то смысле могут все.

Способность к обучению является фундаментальным свойством мозга. В контексте ИНС процесс обучения может рассматриваться как настройка архитектуры сети и весов связей для эффективного выполнения специальной задачи. Обычно нейронная сеть должна настроить веса связей по имеющейся обучающей выборке. Функционирование сети улучшается по мере итеративной настройки весовых коэффициентов. Свойство сети обучаться на примерах делает их более привлекательными по сравнению с системами, которые следуют определенной системе правил функционирования, сформулированной экспертами.

Для конструирования процесса обучения, прежде всего, необходимо иметь модель внешней среды, в которой функционирует нейронная сеть - знать доступную для сети информацию. Эта модель определяет парадигму обучения. Во-вторых, необходимо понять, как модифицировать весовые параметры сети - какие правила обучения управляют процессом настройки. Алгоритм обучения означает процедуру, в которой используются правила обучения для настройки весов.

Существуют три парадигмы обучения: "с учителем", "без учителя" (самообучение) и смешанная. В первом случае нейронная сеть располагает правильными ответами (выходами сети) на каждый входной пример. Веса настраиваются так, чтобы сеть производила ответы как можно более близкие к известным правильным ответам. Усиленный вариант обучения с учителем предполагает, что известна только критическая оценка правильности выхода нейронной сети, но не сами правильные значения выхода. Обучение без учителя не требует знания правильных ответов на каждый пример обучающей выборки. В этом случае раскрывается внутренняя структура данных или корреляции между образцами в системе данных, что позволяет распределить образцы по категориям. При смешанном обучении часть весов определяется посредством обучения с учителем, в то время как остальная получается с помощью самообучения.

Теория обучения рассматривает три фундаментальных свойства, связанных с обучением по примерам: емкость, слож-

ность образцов и вычислительная сложность. Под емкостью понимается, сколько образцов может запомнить сеть, и какие функции и границы принятия решений могут быть на ней сформированы. Сложность образцов определяет число обучающих примеров, необходимых для достижения способности сети к обобщению. Слишком малое число примеров может вызвать "переобученность" сети, когда она хорошо функционирует на примерах обучающей выборки, но плохо - на тестовых примерах, подчиненных тому же статистическому распределению.

Известны 4 основных типа правил обучения: коррекция по ошибке, машина Больцмана, правило Хебба и обучение методом соревнования.

- правило коррекции по ошибке. При обучении с учителем для каждого входного примера задан желаемый выход d. Реальный выход сети y может не совпадать с желаемым. Принцип коррекции по ошибке при обучении состоит в использовании сигнала (d-y) для модификации весов, обеспечивающей постепенное уменьшение ошибки. Обучение имеет место только в случае, когда перцептрон ошибается. Известны различные модификации этого алгоритма обучения.

- обучение Больцмана представляет собой стохастическое правило обучения, которое следует из информационных теоретических и термодинамических принципов. Целью обучения Больцмана является такая настройка весовых коэффициентов, при которой состояния видимых нейронов удовлетворяют желаемому распределению вероятностей. Обучение Больцмана может рассматриваться как специальный случай коррекции по ошибке, в котором под ошибкой понимается расхождение корреляций состояний в двух режимах .

- правило Хебба: если нейроны с обеих сторон синапса активизируются одновременно и регулярно, то сила синаптической связи возрастает. Важной особенностью этого правила является то, что изменение синаптического веса зависит только от активности нейронов, которые связаны данным синапсом. Это существенно упрощает цепи обучения в реализации VLSI.

- обучение методом соревнования. В отличие от обучения Хебба, в котором множество выходных нейронов могут возбуж-

даться одновременно, при соревновательном обучении выходные нейроны соревнуются между собой за активизацию. Это явление известно как правило "победитель берет все". Подобное обучение имеет место в биологических нейронных сетях. Обучение посредством соревнования позволяет кластеризовать входные данные: подобные примеры группируются сетью в соответствии с корреляциями и представляются одним элементом. При обучении модифицируются только веса "победившего" нейрона. Эффект этого правила достигается за счет такого изменения сохраненного в сети образца (вектора весов связей победившего нейрона), при котором он становится чуть ближе ко входному примеру.

Одной из особенностей нейросетевых методов обработки информации является высокая параллельность вычислений и, следовательно, целесообразность использования специальных средств аппаратной поддержки. В значительной мере успех в решении рассмотренных задач обусловлен использованием оригинальных ускорительных плат. Такие платы работают параллельно с процессором обыкновенного ПК и несут на себе основную вычислительную нагрузку, превращая основной процессор компьютера в устройство управления и обслуживания мощных вычислительных средств, расположенных на ускорительной плате.

Например в НТЦ "Модуль" разработаны многопроцессорные ускорительные платы МЦ5.001 и МЦ5.002. Первая из них имеет в своем составе 4 микропроцессора TMS320C40 с тактовой частотой 50 МГц и пиковой производительностью 275 MIPS. Каждый процессор имеет свою локальную статическую память объемом 1 Мбайт. К 2 процессорам дополнительно подключены 2 блока динамической памяти объемом 16 Мбайт каждый. К одному из процессоров подключена также статическая память объемом 1 Мбайт, используемая для обмена данными с ПК. Процессоры соединены друг с другом специальными высокоскоростными каналами с пропускной способностью 20 Мбайт/с каждый. Наращивание и комплексирование плат осуществляется на материнской плате ПК с помощью шины ISA.

Ускорительная плата МЦ5.002 содержит 6 процессоров TMS320C40 и выполнена в конструктиве VME, что позволяет ис-

пользовать ее в бортовых системах, расположенных на летательном аппарате.

Нейропроцессор обычно состоит из двух основных блоков: скалярного, выполняющего роль универсального вычислительного устройства, и векторного, ориентированного на выполнение векторно-матричных операций. Скалярное устройство обеспечивает интерфейсы с памятью и коммуникационными портами, позволяющими объединять процессоры в вычислительные сети различной конфигурации. Основное назначение скалярного устройства - подготовка данных для векторной части процессора. Для этого существует несколько режимов адресации, интерфейс с памятью, наборы арифметических и логических операций, возможность работы с регистровыми парами.

Центральным звеном нейропроцессора является целочисленное векторное устройство, обладающее возможностями обработки данных различной разрядности. Оно оперирует n-разрядными словами. Таким образом, процессор рассчитан на высокопроизводительную обработку больших массивов целочисленных данных. К примеру отечественный нейропроцессор, разработанный в НТЦ "Модуль". Скалярное устройство обеспечивает интерфейсы с памятью и 2 коммуникационными портами. Скалярное устройство имеет адресных регистров и такое же количество регистров общего назначения разрядностью 32 бита каждый.

Центральным звеном нейропроцессора является целочисленное векторное устройство, обладающее возможностями обработки данных различной разрядности. Оно оперирует 64-разрядными словами, которые могут быть разбиты на целочисленные составляющие практически произвольной разрядности в пределах от 1 до 64 бит. На каждую инструкцию векторного процессора затрачивается от 1 до 32 тактов. При этом одновременно обрабатывается до 32 64-разрядных слов. Для организации непрерывной подачи данных в операционное устройство (ОУ) векторного процессора используются внутренние блоки памяти, называемые векторными регистрами. Они выполняют роль буфера операндов, буфера для хранения матрицы весов, очереди результатов. При выполнении команды в операционном устройстве операнды по очереди извлекаются из внутреннего буфера и подаются на

один из входов ОУ. Внутри ОУ производятся вычисления, а их результат заносится в буфер результатов. Векторные инструкции, хотя и занимают несколько тактов процессорного времени, могут выполняться параллельно с инструкциями скалярного процессора. Таким образом, процессор рассчитан на высокопроизводительную обработку больших массивов целочисленных данных.

Нейропроцессор выполнен по технологии 0,5 мкм. Его тактовая частота 33 МГц. На специальных векторно-матричных операциях он дает увеличение производительности в десятки раз по сравнению с процессором TMS320C40. Благодаря наличию коммуникационных портов с интерфейсом, идентичным портам TMS320C40, нейропроцессор может быть интегрирован в гетерогенную многопроцессорную систему.

Для нейропроцессора разработан полный пакет системного программного обеспечения, включая символьный отладчик, и ряд прикладных библиотек, в частности библиотеку векторно-матричных вычислений.

Специфика рассматриваемых вычислительных средств и решаемых задач обуславливает новые требования к технике программирования. Программисту приходится оперировать другими категориями, по-другому строить логику программы, решать задачи, которые не могли возникнуть при традиционном программировании. Перед ним стоит задача - максимально эффективно использовать ресурсы вычислительной системы, правильно распределить нагрузку между процессорами, задействовать их специфичные возможности.

Здесь на первый план выходят методы параллельной обработки данных. Причем слова "параллельная обработка" можно понимать как обработку на параллельно работающих процессорах, так и одновременную обработку нескольких элементов данных на одном процессоре. Современный процессор позволяет выполнять несколько инструкций за один такт, что заставляет программиста продумывать как способы организации самих вычислений, так и способы подготовки данных, для того чтобы параллельно выполняемые процессы не блокировали друг друга.

Трудности, возникающие при программировании многопроцессорных систем, хорошо известны: синхронизация парал-

лельных процессов, механизмы обмена данными, проблемы "критических участков", когда несколько процессов задействуют одни и те же ресурсы. Еще одной важной особенностью современных процессоров является высокая разрядность операндов, например 64 бита, что позволяет размещать в них по несколько малоразрядных элементов данных и обрабатывать их параллельно. Примером эффективного использования отмеченной особенности современных процессоров является технология MMX, где 64-разрядный регистр разбивается на 8 независимых байтов или на 4 16-битных слова, которые обрабатываются параллельно. Независимость элементов состоит в том, что при смещениях или вычитании не происходит заимствования битов у соседних элементов.

По сравнению с обычными компьютерами, нейрокомпьютеры обладают рядом преимуществ. Во первых — высокое быстродействие, связанное с тем, что алгоритмы нейроинформатики обладают высокой степенью параллельности. Во вторых — нейросистемы делаются очень устойчивыми к помехам и разрушениям. В третьих — устойчивые и надежные нейросистемы могут создаваться из ненадежных элементов, имеющих значительный разброс параметров.

Несмотря на перечисленные выше преимущества, эти устройства имеют ряд недостатков:

1. Они создаются специально для решения конкретных задач, связанных с нелинейной логикой и теорией самоорганизации. Решение таких задач на обычных компьютерах возможно только численными методами.

2. В силу своей уникальности эти устройства достаточно дорогостоящи.

Иллюстрацией преимуществ нейросистем по сравнению с другими их типами может быть диаграмма, представленная на рис.70.

Большинство современных нейросистем представляют собой просто персональный компьютер или рабочую станцию, в состав которых входит дополнительная нейроплата. К их числу относятся, например, компьютеры серии FMR фирмы Fujitsu. Такие системы имеют бесспорное право на существование, поскольку их возможностей вполне достаточно для разработки новых алгорит-

мов и решения большого числа прикладных задач методами нейроматематики.

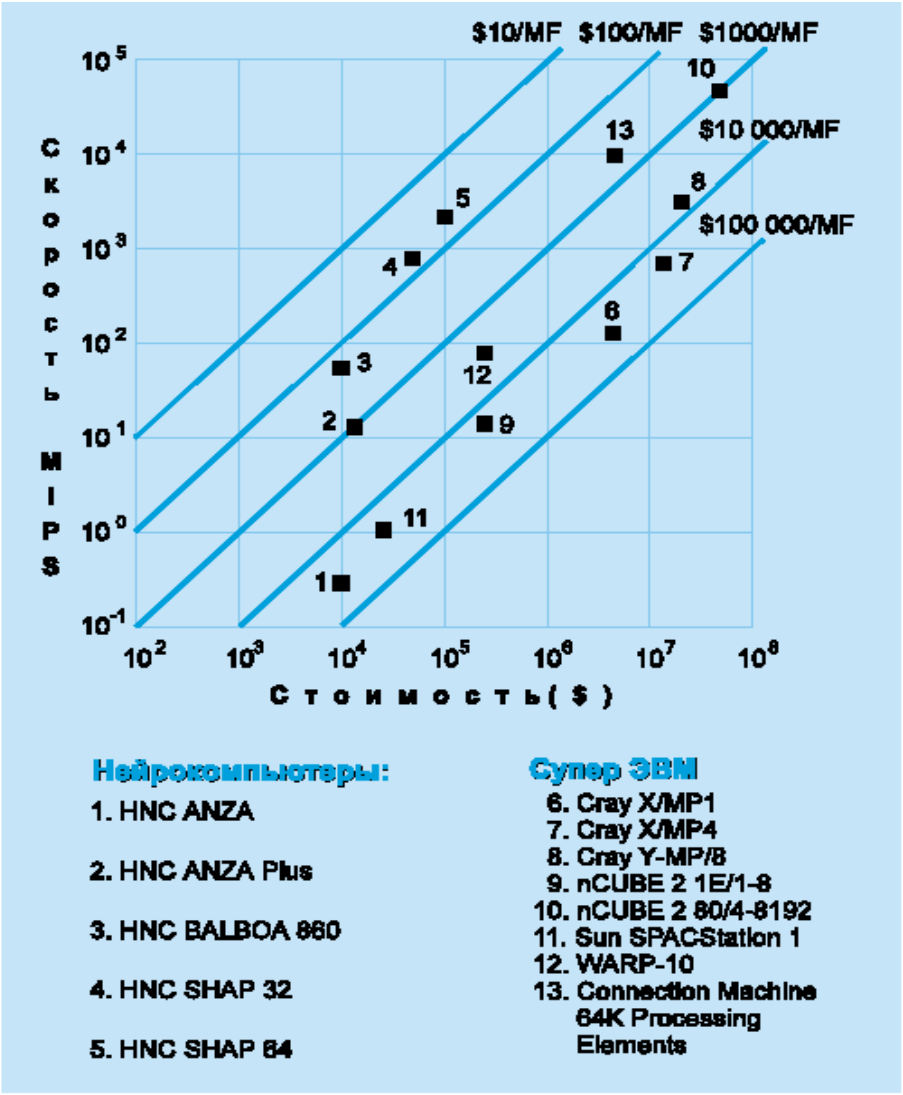


Рис. 70

Однако наибольший интерес представляют специализированные нейрокомпьютеры, непосредственно реализующие принципы НС. Типичными представителями таких систем являются

компьютеры семейства Mark фирмы TRW (первая реализация перцептрона, разработанная Розенблатом, называлась Mark I).

Модель Mark III фирмы TRW представляют собой рабочую станцию, содержащую до 15 процессоров семейства Motorola 68000 с математическими сопроцессорами. Все процессоры объединены шиной VME. Архитектура системы, поддерживающая до 65 000 виртуальных процессорных элементов с более чем 1 млн. настраиваемых соединений, позволяет обрабатывать до 450 тыс. межсоединений/с.

Mark IV - это однопроцессорный суперкомпьютер с конвейерной архитектурой. Он поддерживает до 236 тыс. виртуальных процессорных элементов, что позволяет обрабатывать до 5 млн. межсоединений/с.

Компьютеры семейства Mark имеют общую программную оболочку ANSE (Artificial Neural System Environment), обеспечивающую программную совместимость моделей.

Помимо указанных моделей фирмы TRW предлагает также пакет Mark II - программный эмулятор НС.

Другой интересной моделью является нейрокомпьютер NETSIM, созданный фирмой Texas Instruments на базе разработок Кембриджского университета. Его топология представляет собой трехмерную решетку стандартных вычислительных узлов на базе процессоров 80188. Компьютер NETSIM используется для моделирования таких моделей НС, как сеть Хопфилда - Кохонена и НС с обратным распространением. Его производительность достигает 450 млн. межсоединений/с.

Фирма Computer Recognitiion Systems (CRS) продает серию нейрокомпьютеров WIZARD/CRS 1000, предназначенных для обработки видеоизображений. Размер входной изображения 512 x 512 пиксел. Модель CRS 1000 уже нашла применение в промышленных системах автоматического контроля.

8.6.11. Масштабируемая архитектура

Ярким примером масштабируемой архитектуры является система Cray T3E, которая использует микропроцессоры DEC 21164 (DEC Alpha EV5), RISK-процессоры с пиковой производи-

тельностью 600 Мфлоп и 21164A для машин Cray T3E-900 и Cray T3E-1200. Каждый процессорный элемент (ПЭ) Cray T3E имеет свою собственную DRAM-память объёмом от 64 Мбайт до 2 Гбайт. В отличие от системы CRAY T3D, в которой исполняемая задача запрашивает фиксированное количество процессоров на все время выполнения, в CRAY T3E неиспользуемые процессоры могут использоваться другими задачами. Модели T3E, T3E-900, T3E-1200, T3E-1350.

Каждый узел в системе содержит один процессорный элемент (ПЭ), включающий процессор, память и средство коммутации, которое осуществляет связь между ПЭ. Система конфигурируется до 2048 процессоров. Пиковая производительность составляет 2,4 Тфлоп.

Разделяемая, высокопроизводительная, глобально адресуемая подсистема памяти делает возможным обращение к локальной памяти каждого ПЭ в Cray T3E. Процессорные элементы в системе Cray T3E связаны в трехмерный тор двунаправленной высокоскоростной сетью с малым временем задержки, которая в шесть раз превосходит по скорости аналогичную сеть в Cray T3D. Также добавлена адаптивная маршрутизация, при которой возможен обход участков с высокой эффективностью передачи.

Cray T3E выполняют операции ввода/вывода через многочисленные порты на один или более каналы GigaRing. Каналы ввода/вывода интегрированы в 3-х мерную межузловую сеть и пропорциональны размеру системы. При этом при добавлении ПЭ пропускная способность каналов ввода/вывода увеличивается и масштабируемые приложения могут выполнятся на системах с большим числом процессоров также эффективно, как на системах с меньшим числом процессоров.

Для Cray T3E была создана масштабируемая версия операционной системы ОС UNICOS — ОС UNICOS/mk. Операционная система UNICOS/mk разделена на программы-серверы, распределенные среди процессоров Cray T3E. Это позволяет управлять набором ресурсов системы как единым целым. Локальные серверы обрабатывают запросы ОС, специфичные для каждого ПЭ. Глобальные серверы обеспечивают общесистемные возможности такие, как управление процессами и файловые операции.

В добавлении к пользовательским ПЭ, которые выполняют приложения и команды, системы Cray T3E включает специальные системные ПЭ, которые выполняют глобальные сервера UNICOS/mk. Так как глобальные сервера расположены на системных ПЭ и не дублируются по всей системе, UNICOS/mk эффективно масштабируема, полно функциональна и обслуживает от десятков до тысячи ПЭ с минимальным перегрузкой.

UNICOS/mk обеспечивает следующие программные функции:

- Распределение серверов управления файлами. Функции файлового сервера распределены, используя локальные файловые программы-сервера, для обеспечения максимальной производительности и эффективности.
- ПЭ может генерировать не только последовательную, но и параллельную передачу данных, используя некоторые или даже все ПЭ данной программы.
- Множество глобальных файловых серверов: Система управления файлами распределена на множество системных ПЭ, которые позволяют полностью использовать параллельные дисковые каналы, поддерживаемые на Cray T3E.

Cray T3E-1200 в два раза превышает производительность систем Cray T3E при уменьшенной вдвое стоимости за Мфлоп. Конфигурации в воздушно-жидкостном охлаждении имеют от 6-и процессоров, а в жидкостном — от 32 процессоров. Каждый процессор имеет производительность в 1,2 Тфлоп, для всей системы пиковая производительность меняется от 7,2 Гфлоп до 2,5 Тфлоп. Масштабируется до тысяч процессоров. Серия выпущена в 1997 году.

Система предназначается для наиболее важных научных и технических задач в аэрокосмической, автомобильной, финансовой, химико-фармацевтической, нефтяной и т.д. промышленностях, также в широких областях прикладных исследований, включая химию, гидродинамику, предсказание погоды и сейсмические процессы.

Для поддержки масштабируемости используется оперативная система UNICOS/mk — масштабируемая версия UNICOSR. T3E-1200 поддерживает как явное распараллеливание распределённой памяти посредством CF90 и C/C++ с передачей собщений (MPI, MPI-2 и PVM) и передачу данных, так и неявное распараллеливание посредством возможностей HPF и Cray CRAFT.

T3E выполняет операции ввода/вывода через многочисленные порты на один и более каналов посредством интерфейса GigaRing. Каждый канал сдвоенного кольца ввода/вывода, содержащий в двух кольцах данные, которые перемещаются в противоположных направлениях, передают данные ввода/вывода с высокой пропускной способностью и повышенной надёжностью. Все каналы ввода/вывода доступны и управляемы всеми процессорными элементами. В T3E каждый интерфейс GigaRing имеет максимальную пропускную способность в 500 Мбайт/с.

В дополнение к высокой производительности и пропускной способности процессорных элементов и высокой масштабируемости, Cray T3E-1200 имеет две уникальные особенности: STREAMS и E-Регистры. STREAMS доводят до максимума пропускную способность локальной памяти, позволяя микропроцессору запускать при полной скорости для ссылки для векторoподобных данных. E-Регистры предоставляют операции gather/scatter (соединение/вразброс) для ссылок на локальную и удалённую память, и используют полную пропускную способность внутреннего соединения для удалённого чтения и записи отдельного слова.

Оценка производительности системы производилась при решении плотной линейной системы уравнений порядка 148800 на машине T3E-1200 с 1200 процессорами. Была достигнута скорость в 1,127 Тфлоп/с, что составляет 63% эффективности.

9. Многопроцессорные системы

9.1. Общие требования, предъявляемые к МПС

Отношение стоимость / производительность. Появление любого нового направления в вычислительной технике определяется требованиями компьютерного рынка. Поэтому у разработчиков компьютеров нет одной единственной цели. Большая универсальная вычислительная машина (мейнфрейм) или суперкомпьютер стоят дорого. Для достижения поставленных целей при проектировании высокопроизводительных конструкций приходится игнорировать стоимостные характеристики.

Суперкомпьютеры фирмы Cray Research и высокопроизводительные мейнфреймы компании IBM относятся именно к этой категории компьютеров. Другим крайним примером может служить низкостоимостная конструкция, где производительность принесена в жертву для достижения низкой стоимости. К этому направлению относятся персональные компьютеры различных клонов IBM PC. Между этими двумя крайними направлениями находятся конструкции, основанные на отношении стоимость/ производительность, в которых разработчики находят баланс между стоимостными параметрами и производительностью. Типичными примерами такого рода компьютеров являются миникомпьютеры и рабочие станции.

Для сравнения различных компьютеров между собой обычно используются стандартные методики измерения производительности. Эти методики позволяют разработчикам и пользователям использовать полученные в результате испытаний количественные показатели для оценки тех или иных технических реше-

ний, и в конце концов именно производительность и стоимость дают пользователю рациональную основу для решения вопроса, какой компьютер выбрать.

Надежность и отказоустойчивость. Важнейшей характеристикой вычислительных систем является надежность. Повышение надежности основано на принципе предотвращения неисправностей путем снижения интенсивности отказов и сбоев за счет применения электронных схем и компонентов с высокой и сверхвысокой степенью интеграции, снижения уровня помех, облегченных режимов работы схем, обеспечение тепловых режимов их работы, а также за счет совершенствования методов сборки аппаратуры.

Отказоустойчивость - это такое свойство вычислительной системы, которое обеспечивает ей, как логической машине, возможность продолжения действий, заданных программой, после возникновения неисправностей. Введение отказоустойчивости требует избыточного аппаратного и программного обеспечения. Направления, связанные с предотвращением неисправностей и с отказоустойчивостью, - основные в проблеме надежности. Концепции параллельности и отказоустойчивости вычислительных систем естественным образом связаны между собой, поскольку в обоих случаях требуются дополнительные функциональные компоненты. Поэтому, собственно, на параллельных вычислительных системах достигается как наиболее высокая производительность, так и, во многих случаях, очень высокая надежность. Имеющиеся ресурсы избыточности в параллельных системах могут гибко использоваться как для повышения производительности, так и для повышения надежности. Структура многопроцессорных и многомашинных систем приспособлена к автоматической реконфигурации и обеспечивает возможность продолжения работы системы после возникновения неисправностей.

Следует помнить, что понятие надежности включает не только аппаратные средства, но и программное обеспечение. Главной целью повышения надежности систем является целостность хранимых в них данных.

Масштабируемость. Масштабируемость представляет собой возможность наращивания числа и мощности процессоров,

объемов оперативной и внешней памяти и других ресурсов вычислительной системы. Масштабируемость должна обеспечиваться архитектурой и конструкцией компьютера, а также соответствующими средствами программного обеспечения.

Добавление каждого нового процессора в действительно масштабируемой системе должно давать прогнозируемое увеличение производительности и пропускной способности при приемлемых затратах. Одной из основных задач при построении масштабируемых систем является минимизация стоимости расширения компьютера и упрощение планирования. В идеале добавление процессоров к системе должно приводить к линейному росту ее производительности. Однако это не всегда так. Потери производительности могут возникать, например, при недостаточной пропускной способности шин из-за возрастания трафика между процессорами и основной памятью, а также между памятью и устройствами ввода/вывода. В действительности реальное увеличение производительности трудно оценить заранее, поскольку оно в значительной степени зависит от динамики поведения прикладных задач.

Возможность масштабирования системы определяется не только архитектурой аппаратных средств, но зависит от заложенных свойств программного обеспечения. Масштабируемость программного обеспечения затрагивает все его уровни от простых механизмов передачи сообщений до работы с такими сложными объектами как мониторы транзакций и вся среда прикладной системы. В частности, программное обеспечение должно минимизировать трафик межпроцессорного обмена, который может препятствовать линейному росту производительности системы. Аппаратные средства (процессоры, шины и устройства ввода/вывода) являются только частью масштабируемой архитектуры, на которой программное обеспечение может обеспечить предсказуемый рост производительности. Важно понимать, что простой переход, например, на более мощный процессор может привести к перегрузке других компонентов системы. Это означает, что действительно масштабируемая система должна быть сбалансирована по всем параметрам.

Совместимость и мобильность программного обеспечения. Концепция программной совместимости впервые в широких масштабах была применена разработчиками системы IBM/360. Основная задача при проектировании всего ряда моделей этой системы заключалась в создании такой архитектуры, которая была бы одинаковой с точки зрения пользователя для всех моделей системы независимо от цены и производительности каждой из них. Огромные преимущества такого подхода, позволяющего сохранять существующий задел программного обеспечения при переходе на новые (как правило, более производительные) модели были быстро оценены как производителями компьютеров, так и пользователями и начиная с этого времени практически все фирмы-поставщики компьютерного оборудования взяли на вооружение эти принципы, поставляя серии совместимых компьютеров. Следует заметить однако, что со временем даже самая передовая архитектура неизбежно устаревает и возникает потребность внесения радикальных изменений архитектуру и способы организации вычислительных систем.

В настоящее время одним из наиболее важных факторов, определяющих современные тенденции в развитии информационных технологий, является ориентация компаний-поставщиков компьютерного оборудования на рынок прикладных программных средств. Это объясняется прежде всего тем, что для конечного пользователя в конце концов важно программное обеспечение, позволяющее решить его задачи, а не выбор той или иной аппарат-

ной платформы. Переход от однородных сетей программно совместимых компьютеров к построению неоднородных сетей, включающих компьютеры разных фирм-производителей, в корне изменил и точку зрения на саму сеть: из сравнительно простого средства обмена информацией она превратилась в средство интеграции отдельных ресурсов - мощную распределенную вычислительную систему, каждый элемент которой (сервер или рабочая станция) лучше всего соответствует требованиям конкретной прикладной задачи.

Этот переход выдвинул ряд новых требований. Прежде всего такая вычислительная среда должна позволять гибко менять количество и состав аппаратных средств и программного обеспечения в соответствии с меняющимися требованиями решаемых задач. Во-вторых, она должна обеспечивать возможность запуска одних и тех же программных систем на различных аппаратных платформах, т.е. обеспечивать мобильность программного обеспечения. В третьих, эта среда должна гарантировать возможность применения одних и тех же человеко-машинных интерфейсов на всех компьютерах, входящих в неоднородную сеть. В условиях жесткой конкуренции производителей аппаратных платформ и программного обеспечения сформировалась концепция открытых систем, представляющая собой совокупность стандартов на различные компоненты вычислительной среды, предназначенных для обеспечения мобильности программных средств в рамках неоднородной, распределенной вычислительной системы.

Одним из вариантов моделей открытой среды является модель OSE (Open System Environment), предложенная комитетом IEEE POSIX. На основе этой модели национальный институт стандартов и технологии США выпустил документ "Application Portability Profile (APP). The U.S. Government's Open System Environment Profile OSE/1 Version 2.0", который определяет рекомендуемые для федеральных учреждений США спецификации в области информационных технологий, обеспечивающие мобильность системного и прикладного программного обеспечения. Все ведущие производители компьютеров и программного обеспечения в США в настоящее время придерживаются требований этого документа.

9.2. Классификация систем параллельной обработки данных

На протяжении всей истории развития вычислительной техники делались попытки найти какую-то общую классификацию, под которую подпадали бы все возможные направления развития компьютерных архитектур. Ни одна из таких классификаций не могла охватить все разнообразие разрабатываемых архитектурных решений и не выдерживала испытания временем. Тем не менее в научный оборот попали и широко используются ряд терминов, которые полезно знать не только разработчикам, но и пользователям компьютеров.

Любая вычислительная система (будь то супер-ЭВМ или персональный компьютер) достигает своей наивысшей производительности благодаря использованию высокоскоростных элементов и параллельному выполнению большого числа операций. Именно возможность параллельной работы различных устройств системы (работы с перекрытием) является основой ускорения основных операций.

Параллельные ЭВМ часто подразделяются по классификации Флинна на машины типа SIMD и MIMD. Как и любая другая, приведенная выше классификация несовершенна: существуют машины прямо в нее не попадающие, имеются также важные признаки, которые в этой классификации не учтены. В частности, к машинам типа SIMD часто относят векторные процессоры, хотя их высокая производительность зависит от другой формы параллелизма - конвейерной организации машины. Многопроцессорные векторные системы, типа Cray Y-MP, состоят из нескольких векторных процессоров и поэтому могут быть названы MSIMD (Multiple SIMD).

Классификация Флинна не делает различия по другим важным для вычислительных моделей характеристикам, например, по уровню "зернистости" параллельных вычислений и методам синхронизации.

Можно выделить четыре основных типа архитектуры систем параллельной обработки:

- **конвейерная и векторная обработка.** Основу конвейерной обработки составляет раздельное выполнение некоторой операции в несколько этапов (за несколько ступеней) с передачей данных одного этапа следующему. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько операций.

Конвейеризация эффективна только тогда, когда загрузка конвейера близка к полной, а скорость подачи новых операндов соответствует максимальной производительности конвейера. Если происходит задержка, то параллельно будет выполняться меньше операций и суммарная производительность снизится. Векторные операции обеспечивают идеальную возможность полной загрузки вычислительного конвейера.

При выполнении векторной команды одна и та же операция применяется ко всем элементам вектора (или чаще всего к соответствующим элементам пары векторов). Для настройки конвейера на выполнение конкретной операции может потребоваться некоторое установочное время, однако затем операнды могут поступать в конвейер с максимальной скоростью, допускаемой возможностями памяти. При этом не возникает пауз ни в связи с выборкой новой команды, ни в связи с определением ветви вычислений при условном переходе. Таким образом, главный принцип вычислений на векторной машине состоит в выполнении некоторой элементарной операции или комбинации из нескольких элементарных операций, которые должны повторно применяться к некоторому блоку данных. Таким операциям в исходной программе соответствуют небольшие компактные циклы.

- **системы типа SIMD**. Машины типа SIMD состоят из большого числа идентичных процессорных элементов, имеющих собственную память. Все процессорные элементы в такой машине выполняют одну и ту же программу. Очевидно, что такая машина, составленная из большого числа процессоров, может обеспечить очень высокую производительность только на тех задачах, при решении которых все процессоры могут делать одну и ту же работу. Модель вычислений для машины SIMD очень похожа на модель вычислений для векторного процессора: одиночная операция выполняется над большим блоком данных.

В отличие от ограниченного конвейерного функционирования векторного процессора, матричный процессор (синоним для большинства SIMD-машин) может быть значительно более гибким. Обрабатывающие элементы таких процессоров - это универсальные программируемые ЭВМ, так что задача, решаемая параллельно, может быть достаточно сложной и содержать ветвления. Обычное проявление этой вычислительной модели в исходной программе примерно такое же, как и в случае векторных операций: циклы на элементах массива, в которых значения, вырабатываемые на одной итерации цикла, не используются на другой итерации цикла.

Модели вычислений на векторных и матричных ЭВМ настолько схожи, что эти ЭВМ часто обсуждаются как эквивалентные.

- **системы типа MIMD**. Термин "мультипроцессор" покрывает большинство машин типа MIMD и (подобно тому, как термин "матричный процессор" применяется к машинам типа SIMD) часто используется в качестве синонима для машин типа MIMD. В мультипроцессорной системе каждый процессорный элемент (ПЭ) выполняет свою программу достаточно независимо от других процессорных элементов.

Процессорные элементы, конечно, должны как-то связываться друг с другом, что делает необходимым более подробную классификацию машин типа MIMD. В мультипроцессорах с общей памятью (сильносвязанных мультипроцессорах) имеется память данных и команд, доступная всем ПЭ. С общей памятью ПЭ связываются с помощью общей шины или сети обмена. В противоположность этому варианту в слабосвязанных многопроцессорных системах (машинах с локальной памятью) вся память делится между процессорными элементами и каждый блок памяти доступен только связанному с ним процессору. Сеть обмена связывает процессорные элементы друг с другом.

Базовой моделью вычислений на MIMD-мультипроцессоре является совокупность независимых процессов, эпизодически обращающихся к разделяемым данным. Существует большое количество вариантов этой модели. На одном конце спектра - модель распределенных вычислений, в которой программа делится на до-

вольно большое число параллельных задач, состоящих из множества подпрограмм. На другом конце спектра - модель потоковых вычислений, в которых каждая операция в программе может рассматриваться как отдельный процесс. Такая операция ждет своих входных данных (операндов), которые должны быть переданы ей другими процессами. По их получении операция выполняется, и полученное значение передается тем процессам, которые в нем нуждаются. В потоковых моделях вычислений с большим и средним уровнем гранулярности, процессы содержат большое число операций и выполняются в потоковой манере.

- **многопроцессорные системы с SIMD-процессорами**. Многие современные супер-ЭВМ представляют собой многопроцессорные системы, в которых в качестве процессоров используются векторные процессоры или процессоры типа SIMD. Такие системы относятся к машинам класса MSIMD.

Языки программирования и соответствующие компиляторы для машин типа MSIMD обычно обеспечивают языковые конструкции, которые позволяют программисту описывать "крупнозернистый" параллелизм. В пределах каждой задачи компилятор автоматически векторизует подходящие циклы. Машины типа MSIMD, как можно себе представить, дают возможность использовать лучший из этих двух принципов декомпозиции: векторные операции ("мелкозернистый" параллелизм) для тех частей программы, которые подходят для этого, и гибкие возможности MIMD-архитектуры для других частей программы.

Многопроцессорные системы за годы развития вычислительной техники претерпели ряд этапов своего развития. Исторически первой стала осваиваться технология SIMD. Однако в настоящее время наметился устойчивый интерес к архитектурам MIMD. Этот интерес главным образом определяется двумя факторами:

- архитектура MIMD дает большую гибкость: при наличии адекватной поддержки со стороны аппаратных средств и программного обеспечения MIMD может работать как однопользовательская система, обеспечивая высокопроизводительную обработку данных для одной прикладной задачи, как многопрограммная

машина, выполняющая множество задач параллельно, и как некоторая комбинация этих возможностей.

- архитектура MIMD может использовать все преимущества современной микропроцессорной технологии на основе строгого учета соотношения стоимость/производительность. В действительности практически все современные многопроцессорные системы строятся на тех же микропроцессорах, которые можно найти в персональных компьютерах, рабочих станциях и небольших однопроцессорных серверах.

Одной из отличительных особенностей многопроцессорной вычислительной системы является сеть обмена, с помощью которой процессоры соединяются друг с другом или с памятью. Модель обмена настолько важна для многопроцессорной системы, что многие характеристики производительности и другие оценки выражаются отношением времени обработки к времени обмена, соответствующим решаемым задачам. Существуют две основные модели межпроцессорного обмена: одна основана на передаче сообщений, другая - на использовании общей памяти.

В многопроцессорной системе с общей памятью один процессор осуществляет запись в конкретную ячейку, а другой процессор производит считывание из этой ячейки памяти. Чтобы обеспечить согласованность данных и синхронизацию процессов, обмен часто реализуется по принципу взаимно исключающего доступа к общей памяти методом "почтового ящика".

В архитектурах с локальной памятью непосредственное разделение памяти невозможно. Вместо этого процессоры получают доступ к совместно используемым данным посредством передачи сообщений по сети обмена. Эффективность схемы коммуникаций зависит от протоколов обмена, основных сетей обмена и пропускной способности памяти и каналов обмена.

Часто, и притом необоснованно, в машинах с общей памятью и векторных машинах затраты на обмен не учитываются, так как проблемы обмена в значительной степени скрыты от программиста. Однако накладные расходы на обмен в этих машинах имеются и определяются конфликтами шин, памяти и процессоров. Чем больше процессоров добавляется в систему, тем больше процессов соперничают при использовании одних и тех же данных и шины,

что приводит к состоянию насыщения. Модель системы с общей памятью очень удобна для программирования и иногда рассматривается как высокоуровневое средство оценки влияния обмена на работу системы, даже если основная система в действительности реализована с применением локальной памяти и принципа передачи сообщений.

В сетях с коммутацией каналов и в сетях с коммутацией пакетов по мере возрастания требований к обмену следует учитывать возможность перегрузки сети. Здесь межпроцессорный обмен связывает сетевые ресурсы: каналы, процессоры, буферы сообщений. Объем передаваемой информации может быть сокращен за счет тщательной функциональной декомпозиции задачи и тщательного диспетчирования выполняемых функций.

Таким образом, существующие MIMD-машины распадаются на два основных класса в зависимости от количества объединяемых процессоров, которое определяет и способ организации памяти и методику их межсоединений.

К первой группе относятся машины с общей (разделяемой) основной памятью, объединяющие до нескольких десятков (обычно менее 32) процессоров. Сравнительно небольшое количество процессоров в таких машинах позволяет иметь одну централизованную общую память и объединить процессоры и память с помощью одной шины. При наличии у процессоров кэш-памяти достаточного объема высокопроизводительная шина и общая память могут удовлетворить обращения к памяти, поступающие от нескольких процессоров. Поскольку имеется единственная память с одним и тем же временем доступа, эти машины иногда называются UMA (Uniform Memory Access). Такой способ организации со сравнительно небольшой разделяемой памятью в настоящее время является наиболее популярным. Структура подобной системы представлена на рис. 66.

Вторую группу машин составляют крупномасштабные системы с распределенной памятью. Для того чтобы поддерживать большое количество процессоров приходится распределять основную память между ними, в противном случае полосы пропускания памяти просто может не хватить для удовлетворения запросов, поступающих от очень большого числа процессоров. Естественно

при таком подходе также требуется реализовать связь процессоров между собой. На рис. 67 показана структура такой системы.

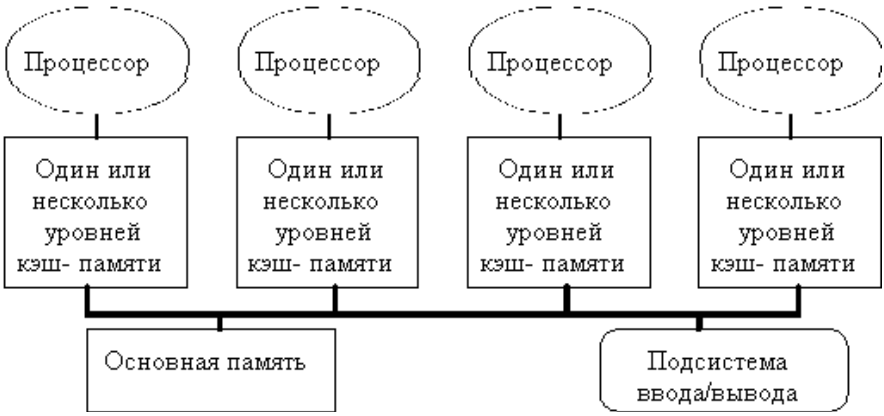


Рис. 66

С ростом числа процессоров просто невозможно обойти необходимость реализации модели распределенной памяти с высокоскоростной сетью для связи процессоров. С быстрым ростом производительности процессоров и связанным с этим ужесточением требования увеличения полосы пропускания памяти, масштаб систем (т.е. число процессоров в системе), для которых требуется организация распределенной памяти, уменьшается, также как и уменьшается число процессоров, которые удается поддерживать на одной разделяемой шине и общей памяти.

Распределение памяти между отдельными узлами системы имеет два главных преимущества. Во-первых, это эффективный с точки зрения стоимости способ увеличения полосы пропускания памяти, поскольку большинство обращений могут выполняться параллельно к локальной памяти в каждом узле. Во-вторых, это уменьшает задержку обращения (время доступа) к локальной памяти. Эти два преимущества еще больше сокращают количество процессоров, для которых архитектура с распределенной памятью имеет смысл.

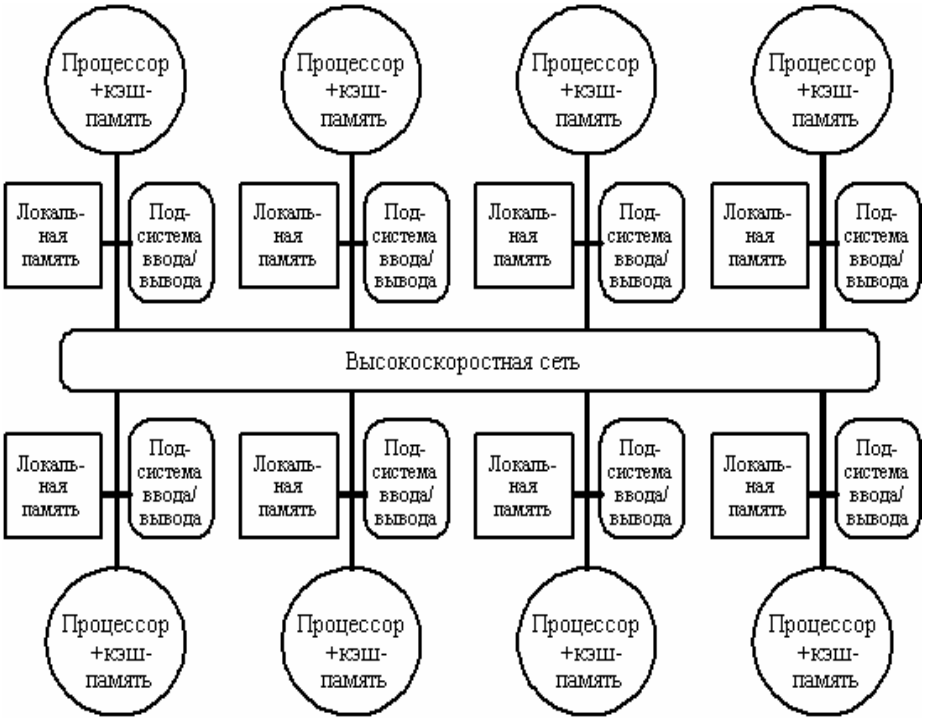


Рис. 67

Обычно устройства ввода/вывода, также как и память, распределяются по узлам и в действительности узлы могут состоять из небольшого числа (2-8) процессоров, соединенных между собой другим способом. Хотя такая кластеризация нескольких процессоров с памятью и сетевой интерфейс могут быть достаточно полезными с точки зрения эффективности в стоимостном выражении, это не очень существенно для понимания того, как такая машина работает, поэтому мы пока остановимся на системах с одним процессором на узел. Основная разница в архитектуре, которую следует выделить в машинах с распределенной памятью заключается в том, как осуществляется связь и какова логическая модель памяти.

9.3. Модели связи и архитектуры памяти

Как уже было отмечено, любая крупномасштабная многопроцессорная система должна использовать множество устройств памяти, которые физически распределяются вместе с процессорами. Имеется две альтернативных организации адресации этих устройств памяти и связанных с этим два альтернативных метода для передачи данных между процессорами. Физически отдельные устройства памяти могут адресоваться как логически единое адресное пространство, что означает, что любой процессор может выполнять обращения к любым ячейкам памяти, предполагая, что он имеет соответствующие права доступа. Такие машины называются машинами с распределенной разделяемой (общей) памятью (DSM - distributed shared memory), масштабируемые архитектуры с разделяемой памятью, а иногда NUMA's - Non-Uniform Memory Access, поскольку время доступа зависит от расположения ячейки в памяти.

В альтернативном случае, адресное пространство состоит из отдельных адресных пространств, которые логически не связаны и доступ к которым не может быть осуществлен аппаратно другим процессором. В таком примере каждый модуль процессор-память представляет собой отдельный компьютер, поэтому такие системы называются многомашинными (multicomputers).

С каждой из этих организаций адресного пространства связан свой механизм обмена. Для машины с единым адресным пространством это адресное пространство может быть использовано для обмена данными посредством операций загрузки и записи. Поэтому эти машины и получили название машин с разделяемой (общей) памятью. Для машин с множеством адресных пространств обмен данными должен использовать другой механизм: передачу сообщений между процессорами; поэтому эти машины часто называют машинами с передачей сообщений.

Каждый из этих механизмов обмена имеет свои преимущества. Для обмена в общей памяти это включает:

- совместимость с хорошо понятными используемыми как в однопроцессорных, так и маломасштабных многопроцессорных

системах, механизмами, которые используют для обмена общую память.

- простота программирования, когда модели обмена между процессорами сложные или динамически меняются во время выполнения. Подобные преимущества упрощают конструирование компилятора.

- более низкая задержка обмена и лучшее использование полосы пропускания при обмене малыми порциями данных.

- возможность использования аппаратно управляемого кэширования для снижения частоты удаленного обмена, допускающая кэширование всех данных как разделяемых, так и неразделяемых.

Основные преимущества обмена с помощью передачи сообщений являются:

- аппаратура может быть более простой, особенно по сравнению с моделью разделяемой памяти, которая поддерживает масштабируемую когерентность кэш-памяти.

- модели обмена понятны, принуждают программистов (или компиляторы) уделять внимание обмену, который обычно имеет высокую, связанную с ним стоимость.

Конечно, требуемая модель обмена может быть надстроена над аппаратной моделью, которая использует любой из этих механизмов. Поддержка передачи сообщений над разделяемой памятью, естественно, намного проще, если предположить, что машины имеют адекватные полосы пропускания. Основные трудности возникают при работе с сообщениями, которые могут быть неправильно выровнены и сообщениями произвольной длины в системе памяти, которая обычно ориентирована на передачу выровненных блоков данных, организованных как блоки кэш-памяти. Эти трудности можно преодолеть либо с небольшими потерями производительности программным способом, либо существенно без потерь при использовании небольшой аппаратной поддержки.

Построение механизмов реализации разделяемой памяти над механизмом передачи сообщений намного сложнее. Без предполагаемой поддержки со стороны аппаратуры все обращения к разделяемой памяти потребуют привлечения операционной системы как для обеспечения преобразования адресов и защиты памяти,

так и для преобразования обращений к памяти в посылку и прием сообщений. Поскольку операции загрузки и записи обычно работают с небольшим объемом данных, то большие накладные расходы по поддержанию такого обмена делают невозможной чисто программную реализацию.

При оценке любого механизма обмена критичными являются три характеристики производительности:

1. *Полоса пропускания:* в идеале полоса пропускания механизма обмена будет ограничена полосами пропускания процессора, памяти и системы межсоединений, а не какими-либо аспектами механизма обмена. Связанные с механизмом обмена накладные расходы (например, длина межпроцессорной связи) прямо воздействуют на полосу пропускания.

2. *Задержка:* в идеале задержка должна быть настолько мала, насколько это возможно. Для ее определения критичны накладные расходы аппаратуры и программного обеспечения, связанные с инициированием и завершением обмена.

3. *Упрятывание задержки:* насколько хорошо механизм скрывает задержку путем перекрытия обмена с вычислениями или с другими обменами.

Каждый из этих параметров производительности воздействует на характеристики обмена. В частности, задержка и полоса пропускания могут меняться в зависимости от размера элемента данных. В общем случае, механизм, который одинаково хорошо работает как с небольшими, так и с большими объемами данных будет более гибким и эффективным.

Таким образом, отличия разных машин с распределенной памятью определяются моделью памяти и механизмом обмена. Исторически машины с распределенной памятью первоначально были построены с использованием механизма передачи сообщений, поскольку это было очевидно проще и многие разработчики и исследователи не верили, что единое адресное пространство можно построить и в машинах с распределенной памятью. С недавнего времени модели обмена с общей памятью действительно начали поддерживаться практически в каждой разработанной машине (характерным примером могут служить системы с симметричной мультипроцессорной обработкой).

Хотя машины с централизованной общей памятью, построенные на базе общей шины все еще доминируют в терминах размера компьютерного рынка, долговременные технические тенденции направлены на использование преимуществ распределенной памяти даже в машинах умеренного размера. Как мы увидим, возможно наиболее важным вопросом, который встает при создании машин с распределенной памятью, является вопрос о кэшировании и когерентности кэш-памяти.

9.4. Многопроцессорные системы с общей памятью

Требования, предъявляемые современными процессорами к полосе пропускания памяти можно существенно сократить путем применения больших многоуровневых кэшей. Тогда, если эти требования снижаются, то несколько процессоров смогут разделять доступ к одной и той же памяти. Начиная с 1980 года эта идея, подкрепленная широким распространением микропроцессоров, стимулировала многих разработчиков на создание небольших мультипроцессоров, в которых несколько процессоров разделяют одну физическую память, соединенную с ними с помощью разделяемой шины.

Из-за малого размера процессоров и заметного сокращения требуемой полосы пропускания шины, достигнутого за счет возможности реализации достаточно большой кэш-памяти, такие машины стали исключительно эффективными по стоимости. В первых разработках подобного рода машин удавалось разместить весь процессор и кэш на одной плате, которая затем вставлялась в заднюю панель, с помощью которой реализовывалась шинная архитектура. Современные конструкции позволяют разместить до четырех процессоров на одной плате. В такой машине кэши могут содержать как разделяемые, так и частные данные. Частные данные - это данные, которые используются одним процессором, в то время как разделяемые данные используются многими процессорами, по существу обеспечивая обмен между ними. Когда кэшируется элемент частных данных, их значение переносится в кэш для сокращения среднего времени доступа, а также требуемой полосы пропускания.

Поскольку никакой другой процессор не использует эти данные, этот процесс идентичен процессу для однопроцессорной машины с кэш-памятью. Если кэшируются разделяемые данные, то разделяемое значение реплицируется и может содержаться в нескольких кэшах. Кроме сокращения задержки доступа и требуемой полосы пропускания такая репликация данных способствует также общему сокращению количества обменов. Однако кэширование разделяемых данных вызывает новую проблему: когерентность кэш-памяти.

Мультипроцессорная когерентность кэш-памяти. Проблема, о которой идет речь, возникает из-за того, что значение элемента данных в памяти, хранящееся в двух разных процессорах, доступно этим процессорам только через их индивидуальные кэши. На рис. 68 показан простой пример, иллюстрирующий эту проблему.

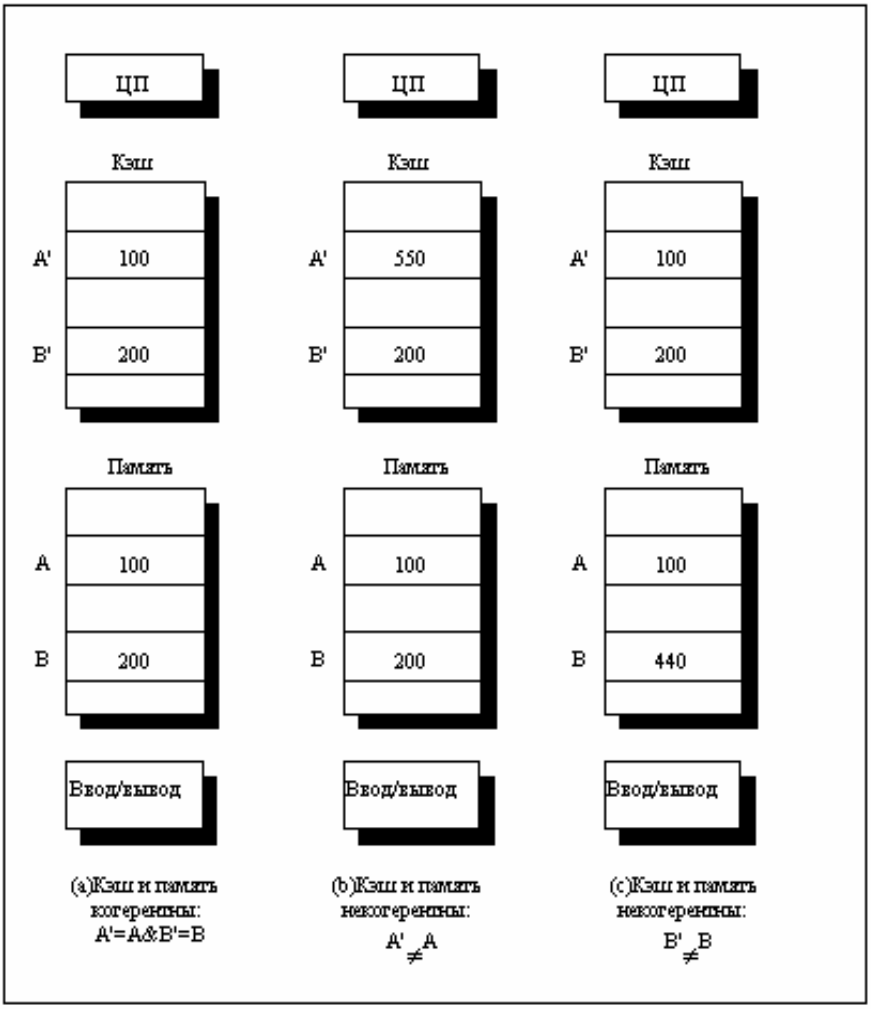
Проблема когерентности памяти для мультипроцессоров и устройств ввода/вывода имеет много аспектов. Обычно в малых мультипроцессорах используется аппаратный механизм, называемый протоколом, позволяющий решить эту проблему. Такие протоколы называются протоколами когерентности кэш-памяти.

Существуют два класса таких протоколов:

1. Протоколы на основе справочника (directory based). Информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы). Этот подход будет рассмотрен в разд. 10.3.

2. Протоколы наблюдения (snooping). Каждый кэш, который содержит копию данных некоторого блока физической памяти, имеет также соответствующую копию служебной информации о его состоянии. Централизованная система записей отсутствует.

Обычно кэши расположены на общей (разделяемой) шине и контроллеры всех кэшей наблюдают за шиной (просматривают ее) для определения того, не содержат ли они копию соответствующего блока.



A' и B' - кэшированные копии элементов A и B в основной памяти

a) Когерентное состояние кэша и основной памяти.

b) Предполагается использование кэш-памяти с отложенным обратным копированием, когда ЦП записывает значение 550 в ячейку A. В результате A' содержит новое значение, а в основной памяти осталось старое значение 100. При попытке вывода A из памяти будет получено старое значение.

c) Подсистема ввода/вывода вводит в ячейку памяти B новое значение 440, а в кэш-памяти осталось старое значение B.

Рис. 68

В мультипроцессорных системах, использующих микропроцессоры с кэш-памятью, подсоединенные к централизованной общей памяти, протоколы наблюдения приобрели популярность, поскольку для опроса состояния кэшей они могут использовать заранее существующее физическое соединение - шину памяти.

Неформально, проблема когерентности памяти состоит в необходимости гарантировать, что любое считывание элемента данных возвращает последнее по времени записанное в него значение. Это определение не совсем корректно, поскольку невозможно требовать, чтобы операция считывания мгновенно видела значение, записанное в этот элемент данных некоторым другим процессором. Если, например, операция записи на одном процессоре предшествует операции чтения той же ячейки на другом процессоре в пределах очень короткого интервала времени, то невозможно гарантировать, что чтение вернет записанное значение данных, поскольку в этот момент времени записываемые данные могут даже не покинуть процессор.

Вопрос о том, когда точно записываемое значение должно быть доступно процессору, выполняющему чтение, определяется выбранной моделью согласованного (непротиворечивого) состояния памяти и связан с реализацией синхронизации параллельных вычислений. Поэтому с целью упрощения предположим, что мы требуем только, чтобы записанное операцией записи значение было доступно операции чтения, возникшей немного позже записи и что операции записи данного процессора всегда видны в порядке их выполнения.

С этим простым определением согласованного состояния памяти мы можем гарантировать когерентность путем обеспечения двух свойств:

1. Операция чтения ячейки памяти одним процессором, которая следует за операцией записи в ту же ячейку памяти другим процессором, получит записанное значение, если операции чтения и записи достаточно отделены друг от друга по времени.

2. Операции записи в одну и ту же ячейку памяти выполняются строго последовательно (иногда говорят, что они сериализованы): это означает, что две подряд идущие операции записи в од-

ну и ту же ячейку памяти будут наблюдаться другими процессорами именно в том порядке, в котором они появляются в программе процессора, выполняющего эти операции записи.

Первое свойство очевидно связано с определением когерентного (согласованного) состояния памяти: если бы процессор всегда бы считывал только старое значение данных, мы сказали бы, что память некогерентна.

Необходимость строго последовательного выполнения операций записи является более тонким, но также очень важным свойством. Представим себе, что строго последовательное выполнение операций записи не соблюдается. Тогда процессор P1 может записать данные в ячейку, а затем в эту ячейку выполнит запись процессор P2. Строго последовательное выполнение операций записи гарантирует два важных следствия для этой последовательности операций записи. Во-первых, оно гарантирует, что каждый процессор в машине в некоторый момент времени будет наблюдать запись, выполняемую процессором P2. Если последовательность операций записи не соблюдается, то может возникнуть ситуация, когда какой-нибудь процессор будет наблюдать сначала операцию записи процессора P2, а затем операцию записи процессора P1, и будет хранить это записанное P1 значение неограниченно долго.

Более тонкая проблема возникает с поддержанием разумной модели порядка выполнения программ и когерентности памяти для пользователя: представьте, что третий процессор постоянно читает ту же самую ячейку памяти, в которую записывают процессоры P1 и P2; он должен наблюдать сначала значение, записанное P1, а затем значение, записанное P2. Возможно, он никогда не сможет увидеть значения, записанного P1, поскольку запись от P2 возникла раньше чтения. Если он даже видит значение, записанное P1, он должен видеть значение, записанное P2, при последующем чтении.

Подобным образом любой другой процессор, который может наблюдать за значениями, записываемыми как P1, так и P2, должен наблюдать идентичное поведение. Простейший способ добиться таких свойств заключается в строгом соблюдении порядка операций записи, чтобы все записи в одну и ту же ячейку могли

наблюдаться в том же самом порядке. Это свойство называется последовательным выполнением (сериализацией) операций записи (write serialization). Вопрос о том, когда процессор должен увидеть значение, записанное другим процессором достаточно сложен и имеет заметное воздействие на производительность, особенно в больших машинах.

Альтернативные протоколы. Имеются две методики поддержания описанной выше когерентности. Один из методов заключается в том, чтобы гарантировать, что процессор должен получить исключительные права доступа к элементу данных перед выполнением записи в этот элемент данных. Этот тип протоколов называется протоколом записи с аннулированием (write ivalidate protocol), поскольку при выполнении записи он аннулирует другие копии. Это наиболее часто используемый протокол как в схемах на основе справочников, так и в схемах наблюдения. Исключительное право доступа гарантирует, что во время выполнения записи не существует никаких других копий элемента данных, в которые можно писать или из которых можно читать: все другие кэшированные копии элемента данных аннулированы.

Чтобы увидеть, как такой протокол обеспечивает когерентность, рассмотрим операцию записи, вслед за которой следует операция чтения другим процессором. Поскольку запись требует исключительного права доступа, любая копия, поддерживаемая читающим процессором, должна быть аннулирована (в соответствии с названием протокола).

Таким образом, когда возникает операция чтения, произойдет промах кэш-памяти, который вынуждает выполнить выборку новой копии данных. Для выполнения операции записи мы можем потребовать, чтобы процессор имел достоверную (valid) копию данных в своей кэш-памяти прежде, чем выполнять в нее запись.

Таким образом, если оба процессора попытаются записать в один и тот же элемент данных одновременно, один из них выиграет состязание у второго (мы вскоре увидим, как принять решение, кто из них выиграет) и вызывает аннулирование его копии. Другой процессор для завершения своей операции записи должен сначала получить новую копию данных, которая теперь уже должна содержать обновленное значение.

Альтернативой протоколу записи с аннулированием является обновление всех копий элемента данных в случае записи в этот элемент данных. Этот тип протокола называется протоколом записи с обновлением (write update protocol) или протоколом записи с трансляцией (write broadcast protocol). Обычно в этом протоколе для снижения требований к полосе пропускания полезно отслеживать, является ли слово в кэш-памяти разделяемым объектом, или нет, а именно, содержится ли оно в других кэшах. Если нет, то нет никакой необходимости обновлять другой кэш или транслировать в него обновленные данные.

Разница в производительности между протоколами записи с обновлением и с аннулированием определяется тремя характеристиками:

1. Несколько последовательных операций записи в одно и то же слово, не перемежающихся операциями чтения, требуют нескольких операций трансляции при использовании протокола записи с обновлением, но только одной начальной операции аннулирования при использовании протокола записи с аннулированием.

2. При наличии многословных блоков в кэш-памяти каждое слово, записываемое в блок кэша, требует трансляции при использовании протокола записи с обновлением, в то время как только первая запись в любое слово блока нуждается в генерации операции аннулирования при использовании протокола записи с аннулированием. Протокол записи с аннулированием работает на уровне блоков кэш-памяти, в то время как протокол записи с обновлением должен работать на уровне отдельных слов (или байтов, если выполняется запись байта).

3. Задержка между записью слова в одном процессоре и чтением записанного значения другим процессором обычно меньше при использовании схемы записи с обновлением, поскольку записанные данные немедленно транслируются в процессор, выполняющий чтение (предполагается, что этот процессор имеет копию данных). Для сравнения, при использовании протокола записи с аннулированием в процессоре, выполняющим чтение, сначала произойдет аннулирование его копии, затем будет производиться

чтение данных и его приостановка до тех пор, пока обновленная копия блока не станет доступной и не вернется в процессор.

Эти две схемы во многом похожи на схемы работы кэш-памяти со сквозной записью и с записью с обратным копированием. Также как и схема задержанной записи с обратным копированием требует меньшей полосы пропускания памяти, так как она использует преимущества операций над целым блоком, протокол записи с аннулированием обычно требует менее тяжелого трафика, чем протокол записи с обновлением, поскольку несколько записей в один и тот же блок кэш-памяти не требуют трансляции каждой записи. При сквозной записи память обновляется почти мгновенно после записи (возможно с некоторой задержкой в буфере записи). Подобным образом при использовании протокола записи с обновлением другие копии обновляются так быстро, насколько это возможно. Наиболее важное отличие в производительности протоколов записи с аннулированием и с обновлением связано с характеристиками прикладных программ и с выбором размера блока.

Основы реализации. Ключевым моментом реализации в многопроцессорных системах с небольшим числом процессоров как схемы записи с аннулированием, так и схемы записи с обновлением данных, является использование для выполнения этих операций механизма шины. Для выполнения операции обновления или аннулирования процессор просто захватывает шину и транслирует по ней адрес, по которому должно производиться обновление или аннулирование данных.

Все процессоры непрерывно наблюдают за шиной, контролируя появляющиеся на ней адреса. Процессоры проверяют не находится ли в их кэш-памяти адрес, появившийся на шине. Если это так, то соответствующие данные в кэше либо аннулируются, либо обновляются в зависимости от используемого протокола. Последовательный порядок обращений, присущий шине, обеспечивает также строго последовательное выполнение операций записи, поскольку когда два процессора конкурируют за выполнение записи в одну и ту же ячейку, один из них должен получить доступ к шине раньше другого.

Один процессор, получив доступ к шине, вызовет необходимость обновления или аннулирования копий в других процессорах. В любом случае, все записи будут выполняться строго последовательно. Один из выводов, который следует сделать из анализа этой схемы, заключается в том, что запись в разделяемый элемент данных не может закончиться до тех пор, пока она не захватит доступ к шине.

В дополнение к аннулированию или обновлению соответствующих копий блока кэш-памяти, в который производилась запись, мы должны также разместить элемент данных, если при записи происходит промах кэш-памяти. В кэш-памяти со сквозной записью последнее значение элемента данных найти легко, поскольку все записываемые данные всегда посылаются также и в память, из которой последнее записанное значение элемента данных может быть выбрано (наличие буферов записи может привести к некоторому усложнению).

Однако для кэш-памяти с обратным копированием задача нахождения последнего значения элемента данных сложнее, поскольку это значение скорее всего находится в кэш, а не в памяти. В этом случае используется та же самая схема наблюдения, что и при записи: каждый процессор наблюдает и контролирует адреса, помещаемые на шину. Если процессор обнаруживает, что он имеет модифицированную ("грязную") копию блока кэш-памяти, то именно он должен обеспечить пересылку этого блока в ответ на запрос чтения и вызвать отмену обращения к основной памяти. Поскольку кэш с обратным копированием предъявляют меньшие требования к полосе пропускания памяти, они намного предпочтительнее в мультипроцессорах, несмотря на некоторое увеличение сложности. Поэтому далее мы рассмотрим вопросы реализации кэш-памяти с обратным копированием.

Для реализации процесса наблюдения могут быть использованы обычные теги кэш. Более того, упоминавшийся ранее бит достоверности (valid bit), позволяет легко реализовать аннулирование. Промахи операций чтения, вызванные либо аннулированием, либо каким-нибудь другим событием, также не сложны для понимания, поскольку они просто основаны на возможности наблюдения. Для операций записи мы хотели бы также знать, име-

ются ли другие кэшированные копии блока, поскольку в случае отсутствия таких копий, запись можно не посылать на шину, что сокращает время на выполнение записи, а также требуемую полосу пропускания.

Чтобы отследить, является ли блок разделяемым, мы можем ввести дополнительный бит состояния (shared), связанный с каждым блоком, точно также как это делалось для битов достоверности (valid) и модификации (modified или dirty) блока. Добавив бит состояния, определяющий является ли блок разделяемым, мы можем решить вопрос о том, должна ли запись генерировать операцию аннулирования в протоколе с аннулированием, или операцию трансляции при использовании протокола с обновлением. Если происходит запись в блок, находящийся в состоянии "разделяемый" при использовании протокола записи с аннулированием, кэш формирует на шине операцию аннулирования и помечает блок как частный (private). Никаких последующих операций аннулирования этого блока данный процессор посылать больше не будет. Процессор с исключительной (exclusive) копией блока кэш-памяти обычно называется "владельцем" (owner) блока кэш-памяти.

При использовании протокола записи с обновлением, если блок находится в состоянии "разделяемый", то каждая запись в этот блок должна транслироваться. В случае протокола с аннулированием, когда посылается операция аннулирования, состояние блока меняется с "разделяемый" на "неразделяемый" (или "частный"). Позже, если другой процессор запросит этот блок, состояние снова должно измениться на "разделяемый". Поскольку наш наблюдающий кэш видит также все промахи, он знает, когда этот блок кэша запрашивается другим процессором, и его состояние должно стать "разделяемый".

Поскольку любая транзакция на шине контролирует адресные теги кэша, потенциально это может приводить к конфликтам с обращениями к кэшу со стороны процессора. Число таких потенциальных конфликтов можно снизить применением одного из двух методов: дублированием тегов, или использованием многоуровневых кэшей с "охватом" (inclusion), в которых уровни, находящиеся ближе к процессору являются поднабором уровней, находящихся дальше от него. Если теги дублируются, то обращения

процессора и наблюдение за шиной могут выполняться параллельно. Конечно, если при обращении процессора происходит промах, он должен будет выполнять арбитраж с механизмом наблюдения для обновления обоих наборов тегов.

Точно также, если механизм наблюдения за шиной находит совпадающий тег, ему будет нужно проводить арбитраж и обращаться к обоим наборам тегов кэш (для выполнения аннулирования или обновления бита "разделяемый"), возможно также и к массиву данных в кэше, для нахождения копии блока. Таким образом, при использовании схемы дублирования тегов процессор должен приостановиться только в том случае, если он выполняет обращение к кэш в тот же самый момент времени, когда механизм наблюдения обнаружил копию в кэш. Более того, активность механизма наблюдения задерживается только тогда, когда кэш имеет дело с промахом.

Если процессор использует многоуровневый кэш со свойствами охвата, тогда каждая строка в основном кэш имеется и во вторичном кэш. Таким образом, активность по наблюдению может быть связана с кэш второго уровня, в то время как большинство активностей процессора могут быть связаны с первичным кэш. Если механизм наблюдения получает попадание во вторичный кэш, тогда он должен выполнять арбитраж за первичный кэш, чтобы обновить состояние и возможно найти данные, что обычно будет приводить к приостановке процессора. Такое решение было принято во многих современных системах, поскольку многоуровневый кэш позволяет существенно снизить требований к полосе пропускания. Иногда может быть даже полезно дублировать теги во вторичном кэш, чтобы еще больше сократить количество конфликтов между активностями процессора и механизма наблюдения.

В реальных системах существует много вариаций схем когерентности кэш, в зависимости от того используется ли схема на основе аннулирования или обновления, построена ли кэш-память на принципах сквозной или обратной записи, когда происходит обновление, а также имеет ли место состояние "владения" и как оно реализуется. В таблице 8 представлены несколько протоколов

с наблюдением и некоторые машины, которые используют эти протоколы.

Таблица 8

Наименование	Тип протокола	Стратегия записи в память	Уникальные свойства Применение
Одиночная запись	Запись с аннулированием	Обратное копирование при первой записи	Первый описанный в литературе протокол наблюдения -
Synapse N+1	Запись с аннулированием	Обратное копирование	Точное состояние, где "владельцем является память" Машины Synapse. Первые машины с когерентной кэш-памятью
Berkely	Запись с аннулированием	Обратное копирование	Состояние "разделяемый" Машина SPUR университета Berkely
Illinois	Запись с аннулированием	Обратное копирование	Состояние "приватный"; может передавать данные из любого кэша Серии Power и Challenge компании Silicon Graphics
"Firefly"	Запись с транс-ляцией	Обратное копирование для "приватных" блоков и сквозная запись для "разделяемых"	Обновление памяти во время трансляции SPARCcenter 2000

9.5. Многопроцессорные системы с локальной памятью

Существуют два различных способа построения крупномасштабных систем с распределенной (локальной) памятью. Простейший способ заключается в том, чтобы исключить аппаратные механизмы, обеспечивающие когерентность кэш-памяти, и сосредоточить внимание на создании масштабируемой системы памяти.

Наиболее известным примером такой системы является компьютер T3D компании Cray Research. В этих машинах память распределяется между узлами (процессорными элементами) и все узлы соединяются между собой посредством того или иного типа

сети. Доступ к памяти может быть локальным или удаленным. Специальные контроллеры, размещаемые в узлах сети, могут на основе анализа адреса обращения принять решение о том, находятся ли требуемые данные в локальной памяти данного узла, или размещаются в памяти удаленного узла. В последнем случае контроллеру удаленной памяти посылается сообщение для обращения к требуемым данным.

Чтобы обойти проблемы когерентности, разделяемые (общие) данные не кэшируются. Конечно, с помощью программного обеспечения можно реализовать некоторую схему кэширования разделяемых данных путем их копирования из общего адресного пространства в локальную память конкретного узла. В этом случае когерентностью памяти также будет управлять программное обеспечение. Преимуществом такого подхода является практически минимальная необходимая поддержка со стороны аппаратуры, хотя наличие, например, таких возможностей как блочное (групповое) копирование данных было бы весьма полезным. Недостатком такой организации является то, что механизмы программной поддержки когерентности подобного рода кэш-памяти компилятором весьма ограничены. Существующая в настоящее время методика в основном подходит для программ с хорошо структурированным параллелизмом на уровне программного цикла.

Машины с архитектурой, подобной Cray T3D, называют процессорами (машинами) с массовым параллелизмом (MPP - Massively Parallel Processor). К машинам с массовым параллелизмом предъявляются взаимно исключающие требования. Чем больше объем устройства, тем большее число процессоров можно расположить в нем, тем длиннее каналы передачи управления и данных, а значит и меньше тактовая частота. Происшедшее возрастание нормы массивности для больших машин до 512 и даже 64К процессоров обусловлено не ростом размеров машины, а повышением степени интеграции схем, позволившей за последние годы резко повысить плотность размещения элементов в устройствах. Топология сети обмена между процессорами в такого рода системах может быть различной. В таблице 9 приведены характеристики сети обмена для некоторых коммерческих MPP.

Таблица 9

Фирма	Название	Количество узлов	Базовая топология
Thinking Machines	CM-2	1024-4096	12-мерный куб
nCube	nCube/ten	1-1024	10-мерный куб
Intel	iPSC/2	16-128	7-мерный куб
Maspar	MP-1216	32-512	2-мерная сеть+ступенчатая Omega
Intel	Delta	540	2-мерная сеть
Thinking Machines	CM-5	32-2048	многоступенчатое толстое дерево
Meiko	CS-2	2-1024	многоступенчатое толстое дерево
Intel	Paragon	4-1024	2-мерная сеть
Cray Research	T3D	16-1024	3-мерный тор

Для построения крупномасштабных систем альтернативой рассмотренному в предыдущем разделе протоколу наблюдения может служить протокол на основе справочника, который отслеживает состояние кэшей. Такой подход предполагает, что логически единый справочник хранит состояние каждого блока памяти, который может кэшироваться. В справочнике обычно содержится информация о том, в каких кэш имеются копии данного блока, модифицировался ли данный блок и т.д. В существующих реализациях этого направления справочник размещается рядом с памятью.

Имеются также протоколы, в которых часть информации размещается в кэш-памяти. Положительной стороной хранения всей информации в едином справочнике является простота протокола, связанная с тем, что вся необходимая информация сосредоточена в одном месте. Недостатком такого рода справочников является его размер, который пропорционален общему объему памяти, а не размеру кэш-памяти. Это не составляет проблемы для машин, состоящих, например, из нескольких сотен процессоров, поскольку связанные с реализацией такого справочника накладные расходы можно преодолеть. Но для машин большего размера необходима методика, позволяющая эффективно масштабировать структуру справочника.

В частности, чтобы предотвратить появление узкого горла в системе, связанного с единым справочником, можно распределить части этого справочника вместе с устройствами распределенной локальной памяти. Таким образом можно добиться того, что обращения к разным справочникам (частям единого справочника) могут выполняться параллельно, точно также как обращения к локальной памяти в распределенной памяти могут выполняться параллельно, существенно увеличивая общую полосу пропускания памяти. В распределенном справочнике сохраняется главное свойство подобных схем, заключающееся в том, что состояние любого разделяемого блока данных всегда находится во вполне определенном известном месте.

На рис. 69 показан общий вид подобного рода машины с распределенной памятью. Вопросы детальной реализации протоколов когерентности памяти для таких машин выходят за рамки настоящего обзора.

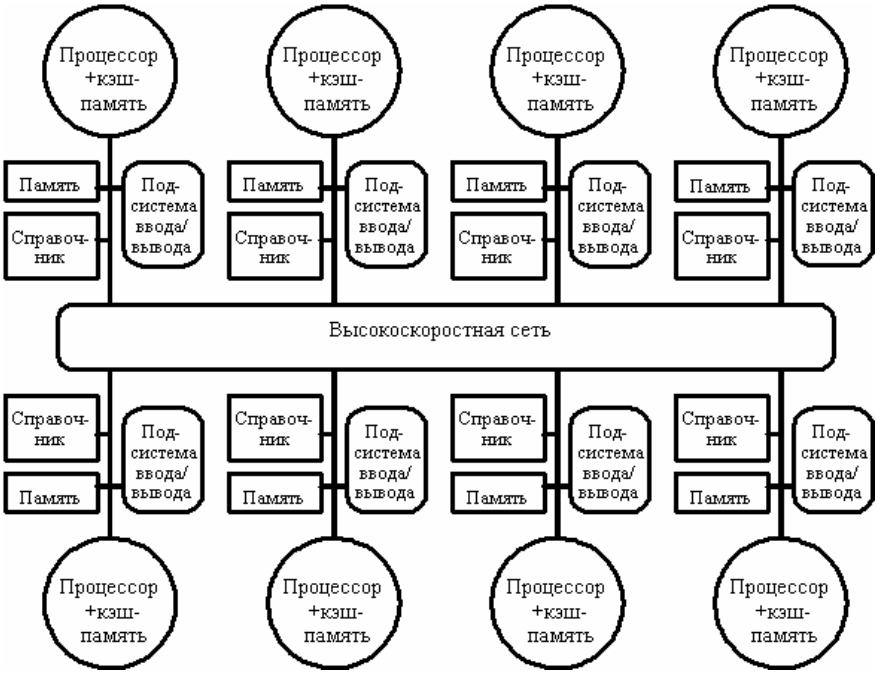


Рис. 69

10. Режимы обмена в МПС

Организация обмена в МПС - важная функция микропроцессора. В обмене принимают участие активное и пассивные устройства. Активным устройством в большинстве случаев является МП, пассивным - основная память или ВУ.

Основными режимами обмена являются программно - управляемый обмен, обмен в режиме прерывания и обмен в режиме прямого доступа к памяти.

Программно-управляемый обмен. Обмен осуществляется по инициативе МП и предназначен для обмена данными между МП и ВУ (или ОП) и их программной обработки. Алгоритм работы МП приведен на рис.70.

Программно - управляемый обмен осуществляется по инициативе обрабатываемой команды и включает чтение информации в микропроцессор из ОП, запись информации в ОП из МП, ввод информации в МП из ВУ и вывод информации из МП во ВУ. Рассмотрим перечисленные виды обмена.

Чтение информации в микропроцессор из основной памяти (рис. 71, а) начинается с момента выдачи из МП на ША значения адреса ячейки ОП, из которой должно быть произведено чтение информации. По синхронизирующему импульсу «чтение» (RD), поступающему из МП на ШУ, активизируются искомые ячейки ОП. Информация из ОП поступает на ШД, передается в МП и записывается в соответствующий регистр МП.

Запись информации в основную память из МП (рис. 71,б) начинается так же, как и в первом случае: из МП на ША поступает значение адреса ячейки ОП, в которую должна быть произведена запись, а МП вырабатывает на линии ШУ сигнал «запись» (WR). Одновременно информация из МП поступает на ШД, передается в ОП и записывается в соответствующую ячейку памяти.

Ввод информации в МП из внешнего устройства начинается по сигналу синхронизации от управляющего устройства или МП, но на ША поступает адрес конкретного канала КВВ, который соединен с требуемым ВУ и через который будет происходить ввод (чтение) информации в МП. Такой канал называется портом. Через некоторое время на линии ШУ МП формирует управляющий

сигнал RD «чтение» (или «ввод»). Запрошенный по указанному адресу порт активизируется, и по сигналу RD информация из ВУ поступает через порт на ШД. По ней информация передается в МП.

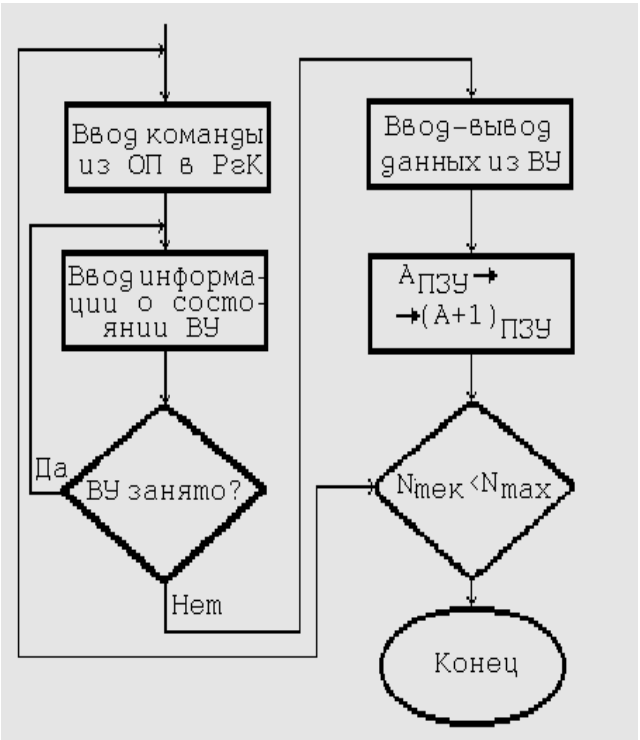


Рис. 70.

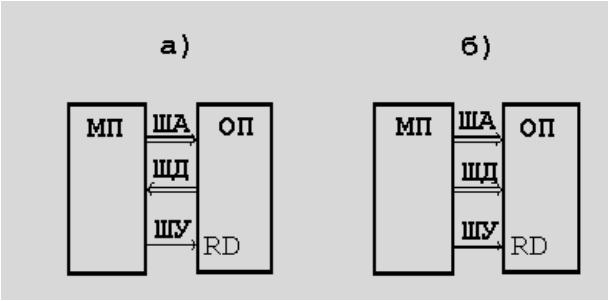


Рис. 71.

Вывод информации из МП во внешнее устройство осуществляется путем формирования МП на линиях ША адреса канала (порта) КВВ, который соединен с требуемым ВУ. Через некоторый промежуток времени МП формирует на линии ШУ сигнал WR «запись» (или «вывод») и выдает на ШД информацию, которая должна быть записана (выведена) в требуемое ВУ. Запрошенный по указанному адресу порт вывода активизируется и информация с ШД поступает в заданное ВУ.

Необходимо отметить, что в рассмотренных режимах обмена всегда участвует специальный регистр МП, называемый аккумулятором: из него информация передается в ШД при выводе ее из МП, и в него она поступает при вводе информации в МП.

При рассмотрении описанных выше режимов обмена с участием МП не ставился вопрос о готовности пассивных ВУ к проведению обмена. Между тем их быстродействие существенно различается в зависимости от вида ВУ. Если в МПС используются ВУ, имеющие быстродействие, сравнимое с быстродействием МП, то МП как правило не проводит анализ готовности ВУ к обмену. Обмен в этом случае носит название синхронным обменом. В тех случаях, когда быстродействие ВУ ниже быстродействия МП (например, клавиатура пульта оператора), то синхронный способ обмена неприменим и используется так называемый асинхронный обмен.

Асинхронный обмен происходит также под управлением программы, но лишь в том случае, когда ВУ подготовлено к обмену. Об этом сообщает сигнал готовности, формируемый ВУ через КВВ (или самим КВВ) на соответствующей линии ШУ. Процесс обмена, инициированный программой, начинается с анализа процессором готовности ВУ к обмену. При отсутствии сигнала готовности МП переходит в состояние ожидания, о чем извещает остальные функциональные модули специальным сигналом (WA*I*T) на одной из линий ШУ. После прихода сигнала готовности происходит непосредственная процедура обмена.

Основным недостатком асинхронного обмена являются потери времени процессора на ожидание того момента, когда устройство будет готово к обмену, и такие потери для некоторых устройств могут оказаться значительными. Так, например, при вводе

информации с пульта оператора среднее время между нажатиями клавиши составляет не менее 0,1 с. Время же самой операции ввода информации с клавиши в МП обычно не превышает 10 мкс. Очевидно, что полезное время работы МП в этом случае не превышает 0,01% общего времени обмена.

Из приведенного примера видно, что за время ожидания очередного сигнала готовности МП способен выполнить достаточно большое число операций в соответствии с командами программы, если бы сигнал готовности ВУ мог останавливать (прерывать) выполнение основной программы и переводить МП в режим выполнения процедуры обмена. Сигнал готовности такого вида называется сигналом прерывания, а способ обмена с использованием сигналов прерывания получил название обмена в режиме прерывания.

Обмен в режиме прерывания. Обмен в режиме прерывания предназначен для обработки программ обслуживания запросов прерывания, сформированных ВУ в процессе накопления ими информации за время работы. Время формирования запросов прерывания ВУ - явление случайное и, в большинстве случаев, не может быть запрограммировано.

Обмен в режиме прерывания производится по инициативе ВУ или КВВ, обслуживающего данное ВУ, и осуществляется именно в те моменты времени, когда соответствующее ВУ готово к передаче данных в МП. По мере готовности к передаче данных контроллер прерываний, обслуживающий данное ВУ, вырабатывает сигнал запроса прерывания, который МП анализирует и, при необходимости, прерывает обрабатываемую программу и переходит к операции обмена - вводу и обработке программы обслуживания прерывания.

Различают прерывания аппаратные, программные и специальные.

Аппаратные прерывания, на практике еще называемые внешними прерываниями, имеют место при воздействии сигналов, которые вырабатываются ВУ, требующими обслуживания. Аппаратные прерывания используются, как правило, для обслуживания ВУ по запросу этих устройств. Они могут быть немаскируемые и маскируемые.

Немаскируемые прерывания - прерывания, которые вызываются внешними, аппаратными средствами и не могут быть запрещены выполняемой программой. Запросы на такие прерывания подаются на специальный вход микропроцессора - вход немаскируемых прерываний. Они обслуживаются обязательно и немедленно вне зависимости от важности выполняемой в данный момент времени программы.

Маскируемые прерывания - прерывания, которые могут быть разрешены или запрещены программным путем - включением в программу специальных команд, разрешающих или запрещающих прерывания на данном участке программы. Для реализации таких прерываний в микропроцессоре имеется один или несколько входов для запросов на обслуживание маскируемых прерываний.

Программные прерывания происходят под воздействием команд прерывания, включенных в основную программу. Здесь инициатива программного прерывания исходит от самой программы. Программные прерывания используются:

— для обслуживания устройств ввода - вывода по опросу,

— для вызова вспомогательных программ операционной системы - так называемых "утилит".

В последнем случае механизм прерываний оказывается более эффективным, чем механизм перехода на программу обслуживания прерывания, хотя в принципе они очень близки.

Специальные прерывания возникают в системе в ходе выполнения основной программы под воздействием сигналов, вырабатываемых внутренними аппаратными средствами. Специальные прерывания можно назвать также внутренними прерываниями. Причинами их появления могут быть:

а) программные сбои или ошибки, являющиеся следствием попыток выполнить неразрешенную (неверную) команду или обратиться к запрещенной области памяти (при ошибке адресации), и др.;

б) аппаратные сбои, являющиеся следствием критического изменения или внезапного отключения питания, неисправностей отдельных узлов аппаратуры, вызывающих неправильное функционирование аппаратуры, и др.;

в) переполнение разрядной сетки, т. е. получение в ходе вычислений чисел, выходящих за пределы диапазона допустимых значений;

г) трассировка - выполнение программы в пошаговом режиме при ее отладке и некоторые другие.

В МПС используются одноуровневые и многоуровневые системы обработки прерывания.

При одноуровневой системе обработки прерываний все контроллеры прерывания имеют одинаковый приоритет по отношению к приоритету системы и подключаются последовательно к линии, по которой передается сигнал разрешения прерывания (рис. 72). При этом ближайшему к МП контроллеру прерываний ($КПр_1$) присваивается наименьший адрес, а самому удаленному ($КПр_N$) - наибольший.

Запросы от ВУ через соответствующие контроллеры прерываний КПр поступают на вход «1» запроса прерывания МП по единственной линии, к которой подключаются все КПр системы. При этом запросы на прерывание от одного или нескольких КПр могут маскироваться программно. Если на линии появляется запрос от одного какого либо КПр, то МП воспринимает его, анализирует на предмет приоритетности ВУ, пославшего запрос, и, в случае более высокого приоритета устройства по сравнению с приоритетом системы, проводит дополнительные операции по сохранению обрабатываемой программы и, после этого, выдает на выходе «2» сигнал разрешения прерывания.

На практике возможно одновременное поступление запросов на линию «1» от двух или более КПр. Если приоритет МП, ведущего обработку текущей программы, выше приоритета устройств, то любой запрос прерывания им игнорируется. Если же приоритет МП ниже приоритета всех ВУ, то в ответ на любой запрос прерывания МП подготавливается к переходу обработки программы обслуживания прерываний и посылает сигнал на линию «2» разрешения прерывания. Этот сигнал последовательно проходит через все контроллеры прерываний, начиная с самого ближнего по линии связи к МП, и кончая тем, который послал сигнал прерывания. Этот КПр запрещает, или блокирует, дальнейшее его распространение и выдает на ШД начальный адрес программы об-

служивания прерывания, уникальный для данного ВУ. МП воспринимает этот адрес, транслирует его передачу на ША, вводит первую команду программы обслуживания прерывания и начинает ее выполнение.

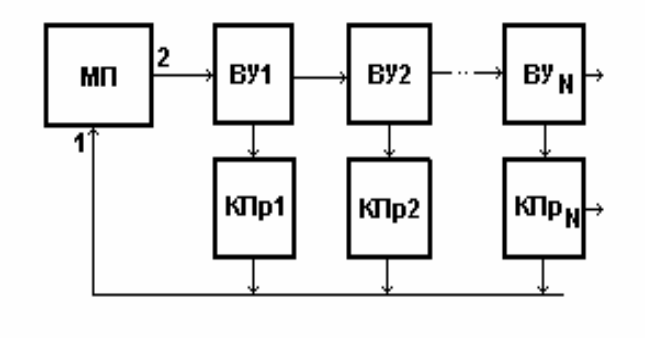


Рис.72.

Последовательность действий, выполняемая МП при обработке прерывания от ВУ, которая справедлива также при обработке и других классов прерываний, состоит в следующем.

1. Сигнал прерывания поступает в МП во время выполнения (i - 1) - й команды обрабатываемой программы. Микропроцессор полностью завершает эту команду, при этом в счетчике команд уже подготовлен адрес следующей i - й команды. По специальной команде, с помощью указателя стека, МП запоминает в стеке сначала содержимое счетчика команд (т.е. адрес i - й команды обрабатываемой программы), а затем текущее содержимое регистра состояния МП, которое считается «старым» словом состояния (ССП) МП, и, при необходимости, содержимое программно - доступных регистров (например, аккумулятора, РОН), участвовавших в формировании результатов обработки прерванной программы, а также дополнительно проводит необходимые операции по обслуживанию прерывания, зависящий от источника прерывания.

2. По адресу «вектора прерывания», сформированного контроллером прерываний, МП загружает из вектора прерывания в счетчик команд начальный адрес программы обслуживания прерывания, а в регистр состояний - «новое» слово состояния МП.

Вектор прерывания - это адрес первой команды программы обслуживания определенного типа прерываний. В некоторых микропроцессорах, например, в МП Z80000 фирмы Zilog (США), К1801ВМ3, вектор включает в себя не только адрес первой команды программы обслуживания прерывания, но и слово состояния процессора, в котором отведено 2, 3 или 4 разряда для указания уровня приоритета данного прерывания.

3. МП переходит к запуску и обработке программы обслуживания прерывания.

В «новом» ССП как правило устанавливается новое значение приоритета МП по отношению ко всем или определенному ВУ системы. Оно определяет возможность повторного прерывания МП от того же или другого источника во время выполнения программы обслуживания прерывания. Если есть необходимость в запрете прерывания до полного завершения программы обслуживания прерывания, то в «новом» слове состояния МП на время обработки программы обслуживания прерывания должен быть установлен высший приоритет МП.

После своего завершения программа обслуживания прерывания специальной командой восстанавливает из стека записанное туда ранее содержимое регистров, соответствующее прерванной программе. Затем эта же программа выполняет команду выхода из прерывания, в результате которой адрес i - й команды, «старое» слово состояния МП и, если это необходимо, содержимое программно - доступных регистров МП будут в него загружены. С этого момента прерванная программа продолжит свою работу с i - й команды прерванной программы.

Многоуровневая система прерывания характеризуется тем, что в системе может быть несколько КПр, которые могут одновременно послать запросы от ВУ на обслуживание прерывания. Очевидно, в МП должен быть предусмотрен механизм для приема в каждый момент времени единственного запроса для обслуживания прерывания. Этот механизм реализуется, как правило, аппаратными средствами и позволяет в первую очередь выбирать порядок обслуживания ВУ: либо первым удовлетворяется запрос прерывания от того контроллера, который включен ближе к МП в цепочке передачи сигналов «запрос прерывания - разрешение пре-

рывания», либо первым удовлетворяется запрос прерывания с КПр, программно имеющего наивысший приоритет.

Разделение всех ВУ в системе по уровням приоритетности обуславливается важностью формируемой ими информации. ВУ, в которых накопленная информация требует незамедлительной ее обработки, имеют наивысший или более высокий приоритет по отношению к МП. Те ВУ, в которых информация может храниться некоторое время без обработки, имеют средний или более низкий приоритет.

Максимальное число программ, обслуживающих ВУ, которые могут прервать друг друга, характеризуют «уровень прерывания». При этом порядок обработки программ при одновременном их запросе определяется уровнем приоритета обслуживаемых ВУ.

В многоуровневых системах прерывания задание уровня приоритета осуществляется тремя способами:

а) использованием в самом МП (или в контроллере прерываний) нескольких входов запросов на прерывание, каждый из которых имеет свой уровень приоритета;

б) с помощью 2 - 4-разрядной шины для подачи кода, несущего информацию об уровне приоритета того или иного КПр, запросившего прерывание;

в) использованием внешних аппаратных средств.

В многоуровневых системах прерываний программа обслуживания ВУ низкого уровня может быть прервана запросом на прерывание от Кпр, обслуживающего ВУ высокого уровня. Если же запрос на прерывание от КПр имеет тот ж уровень, что и обслуживаемое на момент запроса прерывание, или более низкий, то МП не реагирует на этот запрос до тех пор, пока не закончит обслуживание устройства Здесь условно показано, что после поступления сигнала запроса прерывания управление запоминанием состояния и возвратом в прерываемую программу возложено на саму прерывающую программу. В этом случае прерывающая программа состоит из трех частей:

— подготовительной, осуществляющей запоминание состояния прерванной программы,

— заключительной, обеспечивающей восстановление состояния прерванной программы,

— прерывающей программы, выполняющей затребованный запросом режим. с более высоким уровнем приоритета.

Для оценки эффективности системы прерывания, реализующей обмен, используются следующие характеристики системы прерывания.

Время реакции - время между появлением запроса прерывания и началом выполнения прерывающей программы.

На рис. 73 приведена упрощенная временная диаграмма процесса прерывания.

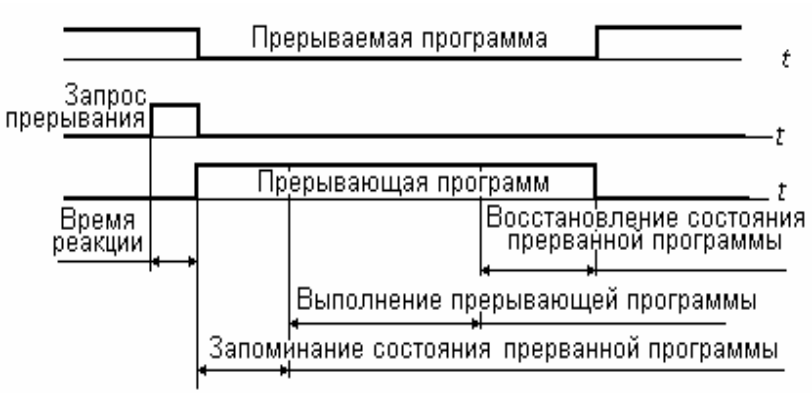


Рис. 73

Для одного и того же запроса прерывания задержки в исполнении прерывающей программы зависят от того, сколько программ с более высоким приоритетом ВУ ждут обслуживания. Поэтому время реакции определяется для запроса с наивысшим приоритетом ВУ.

Время реакции зависит от того, в какой момент допустимо прерывание. Большей частью прерывание допускается после окончания текущей команды. В этом случае время реакции определяется в основном длительностью выполнения команды. Оно может оказаться недопустимо большим для систем, работающих в реальном масштабе времени. В таких системах часто допускается прерывание обрабатываемой программы после любого такта выполнения команды. Однако при этом возрастает количество информации, подлежащей запоминанию и восстановлению при пере-

ключении программ, так как в этом случае необходимо сохранять также и состояния в момент прерывания счетчика тактов, регистра кода операции и некоторых других. Поэтому такая организация прерывания возможна в МПС с быстродействующей сверхоперативной памятью.

Имеются ситуации, в которых желательно немедленное прерывание. Если аппаратура контроля обнаружила ошибку, то целесообразно сразу же прервать операцию, пока ошибка не оказала влияния на следующие такты работы системы.

Затраты времени на переключение программ равны суммарному расходу времени на запоминание и восстановление состояния программы:

Глубина прерывания - максимальное число программ, которые могут прерывать друг друга. Если после перехода к прерывающей программе и вплоть до ее окончания прием других запросов запрещается, то система имеет глубину прерывания, равную «1». Если допускается последовательное прерывание до N программ, то глубина прерывания будет равна N.

Глубина прерывания обычно совпадает с числом уровней приоритета в системе прерываний. На рис. 74 показан обмен в режиме прерывания в системах с единичной глубиной прерывания (а) и с различной глубиной прерывания (б) в предположении, что приоритет каждого следующего запроса выше предыдущего.

Рис. 74

В первом случае появление каждого следующего сигнала «2», «3» и т. д. запроса прерывания не влияет на алгоритм работы системы - МП продолжает выполнение программы «1», «2» обработки предыдущего прерывания до ее завершения. Только после этого он приступает к обработке следующей программы обработки прерывания.

Во втором случае обработка текущей программы прерывается в случае, если на вход системы поступает сигнал запроса прерывания от устройства с более высоким приоритетом, чем приоритет текущей программы. Так, например, сигнал запроса прерывания «2» (рис. 74,б) ВУ более высокого приоритета, чем приоритет МПС, ведет к прерыванию текущей программы и началу обработки программы обслуживания прерывания ВУ, выставившего запрос «2». Аналогично обслуживается сигнал «3» от ВУ, имеющего более высокий приоритет, чем ВУ, пославшего сигнал «2». После обработки программ «3» и «2» МП возвращается в ранее прерванную программу «1».

Необходимо отметить, что, чем большее значение глубины прерывания имеет система, тем более быстрой реакцией на срочные запросы она обладает. Однако, если запрос окажется необслуженным к моменту прихода нового запроса от того же источника, то возникает так называемое насыщение системы прерывания. В этом случае предыдущий запрос прерывания от данного источника будет системой утрачен, что является недопустимым. Для устранения этого явления быстродействие МПС, характеристики системы прерывания, число источников прерывания и частота возникновения запросов должны быть согласованы таким образом, чтобы насыщение было невозможным.

Приоритетное обслуживание запросов прерывания. Вектор прерывания содержит всю необходимую информацию для перехода к прерывающей программе, в том числе ее начальный адрес. Каждому запросу (уровню) прерывания соответствует свой вектор прерывания, способный инициировать выполнение соответствующей прерывающей программы. Векторы прерывания обычно находятся в специально выделенных фиксированных ячейках памяти. Главное место в процедуре перехода к прерывающей программе занимают передача из соответствующего регистра (регистров)

процессора в память (в частности, в стек) на сохранение текущего вектора состояния прерываемой программы (чтобы можно было вернуться к ее исполнению) и загрузка в регистр (регистры) процессора вектора прерывания прерывающей программы, к которой при этом переходит управление процессом.

Процедура организации перехода к прерывающей программе включает в себя выделение из выставленных запросов такого, который имеет наибольший приоритет.

Различают абсолютный и относительный приоритеты. Запрос, имеющий абсолютный приоритет, прерывает выполняемую программу и инициирует выполнение соответствующей прерывающей программы. Запрос с относительным приоритетом является первым кандидатом на обслуживание после завершения выполнения текущей программы. Если наиболее приоритетный из выставленных запросов прерывания не превосходит по уровню приоритета выполняемую процессором программу, то запрос прерывания игнорируется или его обслуживание откладывается до завершения выполнения текущей программы.

Простейший способ установления приоритетных соотношений между запросами (уровнями) прерывания состоит в том, что приоритет определяется порядком присоединения линии сигналов запросов ко входам системы прерывания. При появлении нескольких запросов прерывания первым воспринимается запрос, поступивший на вход с меньшим номером. В этом случае приоритет является жестко фиксированным. Изменить приоритетные соотношения в этом случае можно лишь пересоединением линий сигналов запросов на входах системы прерывания.

Процедура прерывания с опросом источников (флажков) прерывания. При указанном способе задания приоритета между запросами каждому источнику запросов соответствует разряд (флажок) в регистре запросов прерывания (регистре флажков). При наличии запроса или нескольких запросов прерывания формируется сигнал подтверждения прерывания, инициирующий выполняемую специальной программой или аппаратурой процедуру опроса регистра запросов прерывания для установления источника, выставившего запрос прерывания наивысшего приоритета.

Более гибким и динамичным является векторное прерывание, при котором исключается опрос источников прерывания.

Программно - управляемый приоритет прерывающих программ. Относительная степень важности программ, их частота повторения, относительная степень срочности в ходе вычислительного процесса могут меняться, требуя установления новых приоритетных отношений. Поэтому во многих случаях приоритет между прерывающими программами не может быть зафиксирован раз и навсегда. Необходимо иметь возможность изменять по мере надобности приоритетные соотношения программным путем, то есть приоритет между прерывающими программами должен быть динамичным или, другими словами, программно -управляемым.

В МПС программно - управляемый приоритет прерывающих программ может быть реализован по порогу прерывания и по маске прерывания.

Установка приоритета по порогу прерывания предполагает в ходе вычислительного процесса программным путем изменять уровень приоритета (порог прерывания) процессора (а следовательно, и обрабатываемой в данный момент процессором программы) относительно приоритетов запросов источников прерывания (в основном внешних устройств). Порог прерывания задается командой программы специальным кодом порога прерывания, который служит для выделения наиболее приоритетного запроса прерывания, сравнения его приоритета с порогом прерывания и, если он оказывается выше порога, выработки общего сигнала прерывания, по которому начинается процедура прерывания.

Маска прерывания представляет собой двоичный код, разряды которого поставлены в соответствие запросам или классам прерывания. Маска загружается командой текущей программы в регистр маски. Состояние «1» в данном разряде регистра маски разрешает, а состояние «0» запрещает (маскирует) прерывание текущей программы от соответствующего запроса. Таким образом, программа, изменяя маску в регистре маски, может устанавливать произвольные приоритетные соотношения между программами без перекоммутации линий, по которым поступают запросы прерывания. Каждая прерывающая программа может установить свою маску. При формировании маски сигналы логической «1» уста-

навливаются в разряды, соответствующие запросам (прерывающим программам) с более высоким, чем у данной программы, приоритетом.

Обмен в режиме прямого доступа к памяти. В рассмотренных ранее режимах обмен информацией осуществляется между МП и ОП или между МП и ВУ. Однако на практике часто возникает необходимость оперативного обмена информацией между ВУ и ОП без ее обработки. В этом случае при использовании ранее описанных режимов процедура обмена должна содержать два цикла. В первом цикле информация сначала должна быть передана из ВУ (или ОП) в аккумулятор МП, во втором цикле - информация из аккумулятора должна быть занесена в ОП(или ВУ).

При обмене с медленнодействующими ВУ и передачах больших массивов информации такая двухступенчатая процедура существенно снижает скорость обмена, то есть ведет к снижению быстродействия МПС в целом. В связи с этим используется метод обмена, при котором запись информации в ОП из ВУ или считывание информации из памяти во внешнее устройство происходит непосредственно без участия МП. Такой вид обмена получил название обмена в режиме прямого доступа к памяти (ПДП).

Для организации обмена применяется специальное управляющее устройство - контроллер ПДП, который при обмене выполняет функции активного устройства, то есть устанавливает адрес ячейки ОП или порта ВУ, участвующих в обмене, на линиях ША, формирует необходимые управляющие сигналы на линиях ШУ, определяет начало передачи информации по линиям ШД.

При программно - управляемом обмене и при обмене в режиме прерывания магистралью (шинами адреса, данных и управления) распоряжается МП. При обмене в режиме ПДП магистраль должна быть передана в распоряжение контроллера ПДП, а МП необходимо отключить от шин. Для этой цели в МП предусмотрен вход специального управляющего сигнала «запрос захвата шин», при поступлении которого после окончания текущего цикла выполнения команды управляющее устройство МП переводит буферные устройства его шин в режим с высоким выходным сопротивлением (высокоимпедансное состояние). МП при этом отключается от магистралей и его управляющее устройство на специ-

альной линии управления формирует сигнал разрешения захвата шин контроллеру ПДП и представляет магистраль в его распоряжение для передачи либо одного слова, либо целого массива информации.

При передаче одного слова контроллер ПДП занимает магистраль для обмена только на один цикл работы управляющего устройства. На следующем цикле ею снова распоряжается МП. В очередной цикл магистраль снова предоставляется контроллеру ПДП и т. д., пока не будут переданы все слова массива. Все время, пока осуществляется обмен в режиме ПДП, на входе МП присутствует сигнал «запрос захвата шин». Это так называемый мультиплексный обмен в режиме ПДП, при котором МП может продолжать выполнение основной программы в режиме разделения времени с процедурой обмена в режиме ПДП. При этом нет необходимости запоминать в стековой памяти ОЗУ содержимое счетчика команд и аккумулятора, так как МП в обмене не участвует.

При передаче массива информации магистраль занимается контроллером ПДП на все время передачи информации и МП останавливает свою работу на время обмена. Такой режим обмена с ПДП называется монопольным и он обладает максимальной скоростью передачи информации.

Для формирования значений адресов ячеек и необходимых управляющих сигналов в составе контроллера ПДП предусмотрено несколько регистров: регистры адреса, в которых находятся начальное и текущее значения адресов, регистр - счетчик передаваемых слов, регистр управления, регистр состояния и т. п. Обычно в составе контроллера ПДП предусматривается несколько каналов для подключения внешних устройств. В этом случае контроллер помимо прочего осуществляет арбитраж запросов на захват шин магистрали с учетом приоритета подключенных к нему ВУ.

Необходимо отметить, что контроллеры ПДП выполняются в виде конструктивно законченных функциональных модулей или в виде БИС, входящих в комплект МП БИС.

11. Каналы передачи информации в МПС

Работа МПС характеризуется интенсивным обменом информацией между их основными частями: МП, ОП, УВВ и мультисистемными средствами. Связь МП с ОП, УВВ требует нескольких каналов передачи информации - интерфейсов. В зависимости от функционального назначения интерфейсные схемы делятся на несколько уровней (рис. 75).

Шинные интерфейсы 1 уровня обеспечивают обмен информацией между всеми (или основными) модулями микропроцессорной системы. Выбор большинства шин этого уровня определяется архитектурой микропроцессора, например интерфейс И-41(Multibus) используется для МПК серий К580, К1810, межмодульный параллельный интерфейс (МПИ) - для МПК серий К1801/1809, К1811, К581, К5S8 и др.

Наибольшего применения среди шин этого уровня нашли шинные интерфейсы PCI, МПИ, Unibus (Общая шина), И-41 (Multibus), Multibus 11, Futurebus, Fastbus. Они обеспечивают взаимодействие периферийного оборудования, подключенного через соответствующие контроллеры, с вычислительным ядром МПС. В связи с увеличением скоростей работы внешних устройств, постоянным ростом их числа и необходимостью одновременного использования ВУ требования к пропускной способности системной магистрали постоянно растут. Поэтому их число велико и продолжает расти, что отражает, с одной стороны, постоянные рост числа и совершенствование микропроцессоров, а с другой - возникновение все более сложных задач, решаемых ими.

Очевидно, что при сопряжении МП с ОП практически не требуется никаких дополнительных средств, то для сопряжения МП с УВВ требуются специальные устройства, обеспечивающие передачу определенных наборов сигналов. Поэтому шины обмена информацией подключаются не непосредственно к УВВ, а через интерфейсные устройства, структура, принцип работы и технические характеристики которых в сильной степени зависят от совместимости сопрягаемых компонентов.

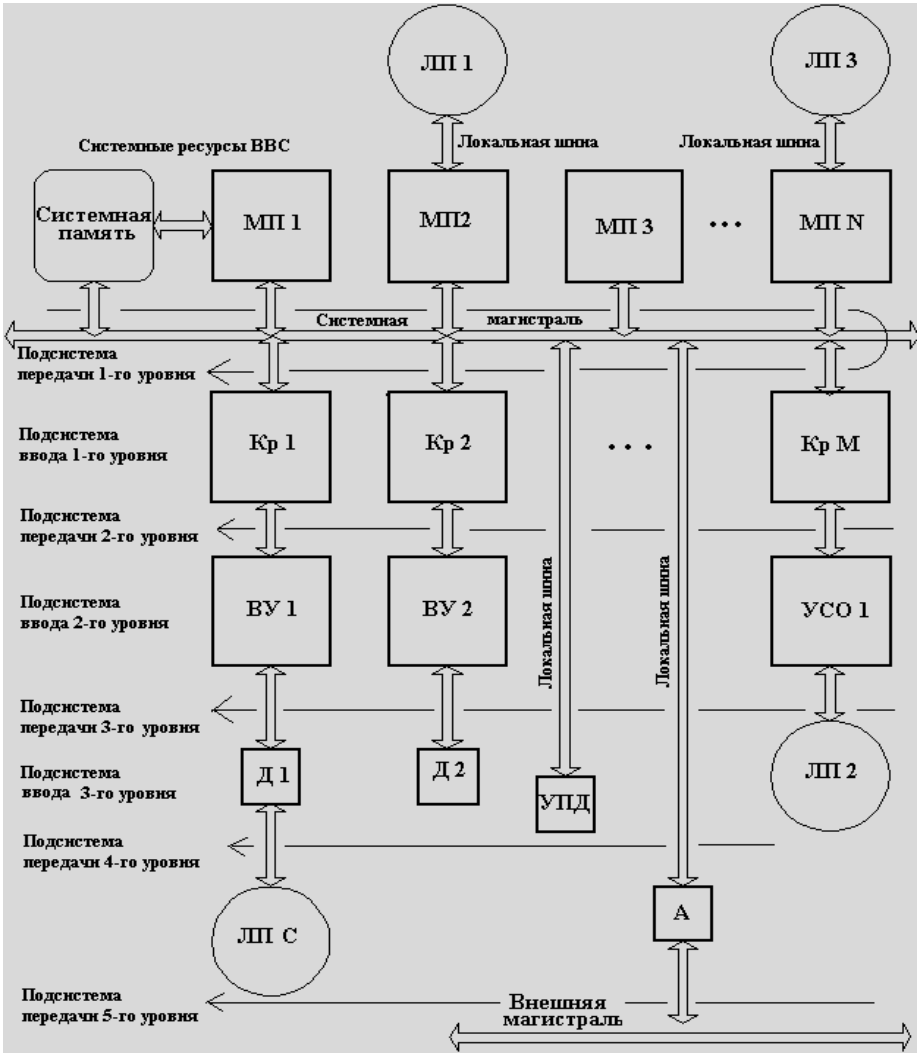


Рис. 75

Совместимость определяется следующими основными признаками: быстродействием, кодами, используемыми для обмена, архитектурой процессора, электрическими характеристиками. Если объединяемые компоненты не соответствуют друг другу по одному или нескольким признакам, то для взаимного подключения используют специальные электронные схемы, называемые интер-

фейсными модулями. Необходимость использования интерфейсных модулей объясняется также и тем, что архитектура процессора с точки зрения набора и организации ШД, ША, набора управляющих сигналов определяет протокол или метод синхронизации МП и УВВ, и все передачи данных, кодов, признаков состояния, управляющих сигналов должны подчинятся этому протоколу обмена. К тому же электрические характеристики МП должны быть совместимы с характеристиками логических схем интерфейса, которые в свою очередь согласуются с ВУ с помощью контроллеров.

Способы структурной и функциональной организации контроллеров ВУ определяются в основном двумя факторами:

- форматами данных и режимами работы конкретных ВУ;
- типом системного интерфейса МПС.

Как показывает практика, создание для конкретного типа ВУ уникального контроллера, обеспечивающего полную электрическую, информационную и конструктивную совместимость данного ВУ с системной магистралью, является сложной с технической и экономической точки зрения задачей. Поэтому наиболее рациональным является стандартизация информационных и управляющих сигналов, которыми обмениваются МП с контроллером и ВУ.

С развитием микроэлектроники появилась возможность реализовать стандартные интерфейсные функции в виде БИС. Для различных МПК БИС разработаны контроллеры, обеспечивающие связь ВУ по стандартному последовательному или по стандартному параллельному каналу передачи данных.

Прежде, чем начать обзор шин, необходимо сказать несколько слов о том, что представляет собой системная шина, и для чего она нужна в компьютере. Шина, в самом простом случае, есть множество проводников для соединения различных компонентов системы в единую систему таким образом, чтобы можно было согласовать их работу. Основной обязанностью системной шины является передача информации между базовым микропроцессором и остальными электронными компонентами системы. По этой шине осуществляется не только передача информации, но и адресация устройств, а также обмен специальными служебными сигналами.

Таким образом, системную шину можно представить как совокупность сигнальных линий, объединенных по их назначению:

- Control lines (управление)
- Address lines (адреса)
- Data lines (данные)

Для того, чтобы описать примерную работу шины, возьмем шину обычного PC, состоящую минимум из линий адреса, данных и линий управления/строба. Самое простое решение, которое здесь можно использовать - это программируемый ввод-вывод. Линии управления используются для синхронизации передачи данных, путем генерирования последовательности импульсов. Возможны две схемы управления, например, раздельные линии управления чтением и записью, либо линия стробирования STROBE и линия чтения - записи в соответствующем состоянии (высокий уровень - для одного сигнала, низкий - для другого).

Шины для PC имеют тенденцию, когда используются раздельные линии управления чтением и записью (фактически две такие линии используются для доступа к памяти, а две дополнительных линии - для осуществления ввода- вывода). В этом случае центральный процессор посылает данные на периферийные устройства, подключенные к шине. ЦП устанавливает стробирующий сигнал по линии ввода - вывода. Этот импульс показывает, что предшествующий адрес на линии адреса правильный, а периферия может начать чтение с шины данных. Кроме перечисленных выше сигналов имеются также и другие сигналы управления, присутствующие на реальной системной шине.

Существует множество системных шин, в том числе и локальных, для PC и других типов компьютеров. Наиболее известными являются S-100, S-100 / IEEE696, ISA, EISA, Nubus, Multibus-II, MCA, Sbus, Mbus, SCSI, VL-Bus, Futurebus+, VME, PCI.

Шина S-100 была создана для 8-разрядных микропроцессоров и различных промышленных приложений. Типичные ее характеристики были такие:

- размеры: 134 мм x 254 мм, 100 выводов
- разъем: 50 выводов на каждой стороне платы
- нерегулируемое напряжение питания: +8В, +16В.

Шина S-100 нашла широкое применение в периферийных платах, она входила в состав плат памяти, устройств последовательного и параллельного интерфейсов, плат контроллеров гибких магнитных дисков, видео-плат, плат музыкальных синтезаторов и т.д. S-100 обеспечивала 16 линий данных, 16 линий адреса (при этом максимальное адресное пространство составляло 64Кбайт), 3 линии питания, 8 линий для прерываний и 39 управляющих линий. Эта шина использовалась для микропроцессоров Intel 8080, Zilog Z-80 и Motorola 6500 и 6800. Некоторые фирмы создали на базе S-100 свои стандарты подобной шины.

Одним из таких примеров может служить стандарт шины S-100/IEEE696, которой разрабатывался в 1983 году. Полученная шина имела следующие характеристики:

- дополнительные 8 разрядов адреса позволили адресовать до 16 Мбайтов памяти (таким образом, всего получилось 24 линии адреса).

-поддержка 16 - разрядных микропроцессоров путем добавления еще двух сигналов sixteen request (SXTRO, 58 линия) и sixteen acknowledge (SIXTN, 60 линия).

-линия 12 была зарезервирована для сигнала немаскируемого прерывания (NMI).

Полная спецификация этой шины включает до 100 сигналов. Рабочая частота при этом достигает 10 МГц. Шина S-100 и ее модификации нашли применение при разработках небольших промышленных приложений. Основными достоинствами этой шины являются низкая цена и поддержка шины большим числом промышленных разработчиков.

У компьютеров IBM PC AT и IBM PC XT системная шина была предназначена для одновременной передачи только 8 разрядов данных, так как используемый в компьютерах микропроцессор i8088 имел 8 линий данных. Кроме этого, системная шина включала 20 адресных линий, которые ограничивали адресное пространство пределом в 1 Мбайт.

Для работы с внешними устройствами в этой шине были предусмотрены также 4 линии аппаратных прерываний и 4 линии для требования внешними устройствами прямого доступа в память (DMA - Direct Memory Access). Для подключения плат расширения

использовались специальные 62-контактные разъемы. Заметим, что системная шина и микропроцессор синхронизировались от одного тактового генератора с частотой 4,77 МГц. Таким образом, теоретически скорость передачи данных могла достигать более 4.5 Мбайт/с.

Шина ISA. В компьютерах PC AT, использующих микропроцессор i80286, впервые стала применяться новая системная шина ISA (Industry Standard Architecture), полностью реализующая возможности упомянутого микропроцессора. Количество адресных линий было увеличено на четыре, а данных - на восемь. Таким образом, можно было передавать параллельно уже 16 разрядов данных, а благодаря 24 адресным линиям напрямую обращаться к 16 Мбайтам системной памяти. Количество линий аппаратных прерываний в этой шине было увеличено с 7 до 15, а каналов DMA - с 4 до 7.

Надо отметить, что новая системная шина ISA полностью включала в себя возможности старой 8-разрядной шины, то есть все устройства, используемые в PC XT, могли без проблем применяться и в PC AT 286. Системные платы с шиной ISA позволили выполнять синхронизацию работы самой шины и микропроцессора разными тактовыми частота ми, за счет чего устройства, выполненные на платах расширения, могли работать медленнее, чем базовый микропроцессор. Это стало особенно актуальным, когда тактовая частота процессоров превысила 10-12 МГц. Теперь системная шина ISA работает асинхронно на частоте 8 МГц; таким образом, теоретически максимальная скорость передачи может достигать 16 Мбайт/с.

Шина ISA имеет следующие параметры:

а) для IBM PC XT:

- 20 адресных линий (A0 - A19),
- 8 линий данных (двунаправленных),
- максимальная пропускная способность 1.2 Мбайт/сек,
- 6 линий запроса прерывания (IRQ2 - IRQ7),
- 3 линии DMA,
- рабочая частота шины 4.77 МГц.

б) для IBM PC AT:

- 16 линий данных,

- максимально адресуемая память - до 16 Мбайт (224),
- добавлены дополнительные 5 линий IRQ (тактируемые по фронту),
- частичная поддержка множества мастеров шины путем введения дополнительных сигналов,
- пропускная способность 5.3 Мбайт/сек,
- рабочая частота шины 8 МГц.

С появлением новых микропроцессоров, таких как i80386 и i486, стало очевидно, что одним из вполне преодолимых препятствий на пути повышения производительности компьютеров с этими микропроцессорами является системная шина типа ISA. Дело в том, что возможности этой шины для построения высокопроизводительных систем следующего поколения были практически исчерпаны.

Шина EISA обеспечивает больший возможный объем адресуемой памяти, 32-разрядную передачу данных, в том числе и в режиме DMA, улучшенную систему прерываний и арбитраж DMA, автоматическую конфигурацию системы и плат расширения.

Шина EISA (Extended Industry Slandard Architecture) первоначально была ориентирована на вполне конкретную область применения - на компьютеры, оснащенные высокоскоростными подсистемами внешней памяти на жестких магнитных дисках с буферной кэш-памятью. Такие компьютеры до сих пор используются в основном в качестве мощных файл-серверов или рабочих станций.

В EISA-разъем на системной плате компьютера, помимо, разумеется, специальных EISA-плат, может вставляться либо 8-, либо 16-разрядная плата расширения, предназначенная для обыкновенной PC AT с шиной ISA. Это обеспечивается поистине гениальным, но простым конструктивным решением. EISA-разъемы имеют два ряда контактов, один из которых (верхний) использует сигналы шины ISA, а второй (нижний) - соответственно EISA.

Контакты в соединителях EISA расположены так, что рядом с каждым сигнальным контактом находится контакт "земля". Благодаря этому сводится к минимуму вероятность генерации электромагнитных помех, а также уменьшается восприимчивость к та-

ким помехам. Шина EISA позволяет адресовать 4-Гбайтное адресное пространство, доступное микропроцессорам i80386/486.

Однако доступ к этому пространству могут иметь не только центральный процессор, но и платы управляющих устройств типа bus master - главного абонента (то есть устройства, способные управлять передачей данных по шине), а также устройства, организующие режим DMA.

Стандарт EISA поддерживает многопроцессорную архитектуру для "интеллектуальных" устройств (плат), оснащенных собственными микропроцессорами. Поэтому данные, например, от контроллеров жестких дисков, графических контроллеров и контроллеров сети могут обрабатываться независимо, не загружая при этом основной процессор. Теоретически максимальная скорость передачи по шине в так называемом пакетном режиме (burst mode) может достигать 33 Мбайт/с, В обычном (стандартном) режиме скорость передачи по шине EISA не превосходит, разумеется, известных значений для ISA.

На шине EISA предусматривается метод централизованного управления, организованный через специальное устройство - системный арбитр. Таким образом поддерживается использование ведущих устройств на шине, однако предусматривается также предоставление шины запрашивающим устройствам по циклическому принципу.

Как и для шины ISA, в системе EISA имеется 7 каналов DMA. Выполнение DMA-функций полностью совместимо с аналогичными операциями на ISA- шине, хотя они могут происходить и несколько быстрее. Контроллеры DMA имеют возможность поддерживать 8-, 16- и 32-разрядные режимы передачи данных. В общем случае возможно выполнение одного из четырех циклов обмена между устройством DMA и памятью системы. Это - ISA - совместимые циклы, использующие для передачи данных 8 тактов шины; циклы типа А, исполняемые за 6 тактов шины; цикпы типа В, исполняемые за 4 такта шины, и циклы типа С (или burst), в которых передача данных происходит за один такт шины. Типы циклов А, В и С поддерживаются 8-, 16- и 32-разрядными устройствами, причем возможно автоматическое изменение размера (ширины) данных при передаче в не соответствующую размеру память.

Большинство ISA-совместимых устройств, использующих DMA, могут работать почти в 2 раза быстрее, если они будут запрограммированы на применение циклов A или B, а не стандартных (и сравнительно медленных) ISA-циклов. Такая производительность достигается только путем улучшения арбитража шины, а не в ущерб совместимости с ISA.

Приоритеты DMA в системе могут быть либо "вращающимися" (переменными), либо жестко установленными. Линии прерывания шины ISA, по которым запросы прерывания передаются в виде перепадов уровней напряжения (фронтов сигналов), сильно подвержены импульсным помехам. Поэтому в дополнение к привычным сигналам прерываний на шине ISA, активным только по своему фронту, в системе EISA предусмотрены также сигналы прерываний, активные по уровню. Причем для каждого прерывания выбор той или иной схемы активности может быть запрограммирован заранее. Собственно прерывания, активные по фронту, сохранены в EISA только для совместимости со "старыми" адаптерами ISA, обслуживание запросов на прерывание которых производит схема, чувствительная к фронту сигнала.

Понятно, что прерывания, активные по уровню, менее подвержены шумам и помехам, нежели обычные. К тому же (теоретически) по одной и той же физической линии можно передавать бесконечно большое число уровней прерывания. Таким образом, одна линия прерывания может использоваться для нескольких запросов.

Для компьютеров с шиной EISA предусмотрено автоматическое конфигурирование системы. Каждый изготовитель плат расширения для компьютеров с шиной EISA поставляет вместе с этими платами и специальные файлы конфигурации. Информация из этих файлов используется на этапе подготовки системы к работе, которая заключается в разделении ресурсов компьютера между отдельными платами.

Для "старых" плат адаптеров пользователь должен сам подобрать правильное положение DIP-переключателей и перемычек, однако сервисная программа на EISA-компьютерах позволяет отображать установленные положения соответствующих переключателей на экране монитора и дает некоторые рекомендации по пра-

вильной их установке. Помимо этого, в архитектуре EISA предусматривается выделение определенных групп адресов ввода - вывода для конкретных слотов шины - каждому разъему расширения отводится адресный диапазон 4Кбайта. Это также позволяет избежать конфликтов между отдельными платами EISA. Кроме того, шина по-прежнему тактируется частотой около 8 МГц, а скорость передачи увеличивается в основном благодаря увеличению разрядности шины данных. Отметим, что шина EISA имеет следующие параметры:

- 32 - разрядный режим передачи
- максимальная пропускная способность - до 33 Мбайт/сек
- 32 - разрядная адресация памяти, что обеспечивает до 4 Гбайт адресуемого пространства памяти
- множество мастеров шин
- программируемые прерывания по уровню или по фронту синхросигнала
- автоматическая конфигурация плат

Шина Nubus обладает примерно теми же характеристиками, что и ISA.

Шина Multibus-II была разработана в 1985 г. как развитие широко применяемого в промышленной автоматике стандарта Multibus. Multibus-II является 32-разрядной шиной и может работать со скоростью управляющего процессора - вплоть до достижения пропускной способности 80 Мбайт/с. В отличие от других рассматриваемых здесь шин, Multibus обладает возможностью высокоскоростной передачи сообщений между различными управляющими устройствами. При этом механизм передачи позволяет организовывать "интеллектуальное" взаимодействие между процессорами и контроллерами. Это особенно важно при создании многопроцессорных систем и построении сложных комплексов промышленной электроники.

Шина содержит пять магистралей (рис. 76), логическая организация которых приспособлена для решения задач определенного класса:

— i PSB - магистраль параллельная системная межмашинная;

— i LBX - магистраль параллельная локального расширения;

— i SSB - магистраль последовательная системная;

— i SBX - магистраль параллельная расширения ввода - вывода;

— MDMA - параллельная магистраль каналов прямого доступа к памяти.

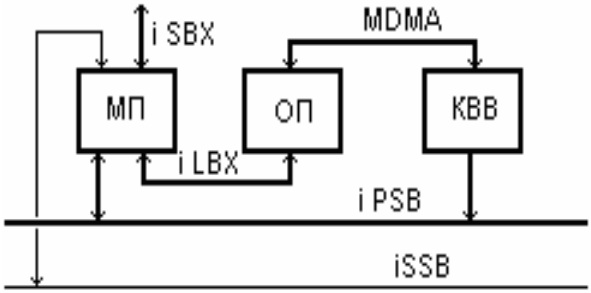


Рис. 76

Одновременное использование магистралей iPSB, iLBX, MDMA обеспечивает суммарную пропускную способность 96 Мбайт/с без учета скорости передачи по магистрали iSBX, предназначенной специально для организации ввода - вывода. Параллельные магистрали используют синхронное стробирование, повышающее помехоустойчивость. Магистрали iPSB и iLBX содержат контрольные разряды по четности, а магистраль iSSB использует 16 - разрядный код циклического избыточного кодирования.

Основная системная магистраль iPSB объединяет все модули системы и через нее осуществляется управление и контроль за ее работой. В ее состав входят 32 совмещенные линии адреса/данных, пять линий параллельного арбитража, с помощью которых осуществляется процедура приоритетов при прерываниях.

Шина MC (MicroChannel) появилась в 1987 г. в компьютерах PS/2. Достаточно быстрая (до 20 МГц, до 76 Мбайт/с) и широкая (32 бита), шина содержала рад удачных архитектурных решений и вполне могла бы бороться за лидерство среди системных шин. Она обладает следующими особенностями:

- 8/16/32 - разрядные линии передачи данных,

- прерывания по уровню сигнала (в отличие от ISA, где прерывания - по фронту синхросигнала),

- 24 или 32 адресных линии (адресация до 4 Гбайт памяти),

- автоматическая конфигурация плат (на основе информации в ROM этих плат),

- асинхронный протокол передачи данных.

Шина Sbus разработана в 1989 г. для работы с частотой до 25 МГц. Она предназначена для передачи 32-разрядных данных. Ее особенностью являются возможность автоматически транслировать виртуальные адреса в физические, распознавать ошибки при передаче данных и инициировать повторы.

Шина Mbus создана в 1990 г. и предназначена для передачи 64-разрядных данных. Mbus допускает совместное использование с другими шинами, имеет портативные варианты исполнения и предусматривает возможности передачи сообщений.

Шина SCSI (Small Computer System Interface) регламентирован стандартом IEC 9316, который унифицирует основные уровни для базовых типов периферийных устройств, главным образом накопителей магнитных дисков, АЦПУ, а также возможности расширения функций посредством специальных кодов и полей. В интерфейсе используется логическая адресация всех блоков данных и возможность считывания с устройств прямого доступа информации о числе имеющихся блоков.

Максимальная скорость передачи данных составляет до 4 Мбайт/сек, длина кабеля до 6 м при использовании обычных приемопередатчиков и до 25 м дифференциальных приемопередатчиков. Архитектура интерфейса предусматривает несколько видов организации взаимодействия эадатчиков (инициаторов) и исполнителей (приемников) с использованием необязательного распределенного арбитража. Время арбитража не превышает 10 мкс.

Дополнительные возможности такие: два варианта физической реализации, использование четности, синхронная передача данных и др. Команды разделены на обязательные, расширенные, необязательные и уникальные. Устройства выполняют все обязательные команды для данного типа устройств команды, а также

ряд других команд. Кроме того. в стандарте определены расширенные команды дли устройств прямого доступа, постоянные команды для всех типов устройств, уникальные команды для жестких дисков, ленточных накопителей, принтеров, оптических дисков, процессоров, байты состояния всех типов устройств.

Максимальное число подключенных устройств - 8. Каждое устройство идентифицируется соответствующим разрядом, размещаемым на линии данных. SCSI-2 является одной из "старых" периферийных шин, используемых, с доработками, и поныне. Спецификация SCSI разрабатывалась американским институтом национальных стандартов ANSI. Чуть позже она расширилась до SCSI-2 и SCSI-3.

Типичная SCSI обладает следующими характеристиками:

- 8 - разрядная параллельная шина ввода-вывода,
- каждый адаптер может поддерживать до 7 устройств,
- поддерживаются различные устройства (CD-ROM, ленточные накопители, сканеры, магнитооптические устройства и т. д.),
- пропускная способность 4 Мбайт/сек,
- поддержка синхронной и асинхронной схем передачи данных.

SCSI-2 расширяет возможности основного стандарта. Она имеет максимальную пропускную способность до 10 Мбайт/сек при 8 - разрядной шине и до 40 Мбайт/сек - при 32-разрядной шине. Существует несколько спецификаций приложений для SCSI:

- Narrow SCSI 8-разрядная версия SCSI,
- Wide SCSI 16- и 32-разрядные версии SCSI-2,
- Fast SCSI SCSI-2, которая поддерживает скорость передачи до 10 Мбайт/сек

Разработчики компьютеров, системные платы которых основывались на микропроцессорах i80386/486, стали использовать раздельные шины для памяти и устройств ввода-вывода. Это позволило максимально задействовать возможности оперативной памяти, так как именно в этом случае память может работать с наивысшей для нее скоростью. Тем не менее при таком подходе вся система не может обеспечить достаточной производительности, так как устройства, подключенные через разъемы расшире-

ния, не могут достичь скорости обмена, сравнимой с процессором. В основном это касается работы с контроллерами накопителей и видеоадаптерами.

Для решения данной проблемы стали использовать так называемые локальные (local или mezzanine) шины, которые непосредственно связывают процессор с контроллерами периферийных устройств. Известны две стандартные локальные шины: **VL-bus** (или VLB), предложенная ассоциацией VESA (Video Electronics Standards Association), и **PCI** (Peripheral Component Interconnect), разработанная фирмой Intel. Обе эти шины, предназначенные, вообще говоря, для одного и того же - для увеличения быстродействия компьютера, позволяют таким периферийным устройствам, как видеоадаптеры и контроллеры накопителей, работать с тактовой частотой 33 МГц и выше. Обе эти шины используют разъемы типа MCA.

Шина VL-Bus является расширением шины процессора 486. Выводы процессора подключаются непосредственно к контактам разъема шины. В некоторых платах адаптеров VL-Bus имеются буферы для хранения данных на время ожидания готовности периферийного устройства. Таким образом, схемная реализация VL-bus оказывается более, дешевой и простой, чем, например, PCI. Спецификация VESA, в частности, предусматривает, что к шине, которая является локальной 32-разрядной шиной системного микропроцессора, может подключаться до трех периферийных устройств. В качестве таких устройств в настоящее время выступают контроллеры накопителей, видеоадаптеры и сетевые платы.

Конструктивно VL-bus выглядит как короткий соединитель типа MCA (112 контактов), установленный, например, рядом с разъемами расширения ISA или EISA. При этом 32 линии используются для передачи данных и 30 - для передачи адреса. Максимальная скорость передачи по шине VL-bus теоретически может составлять около 130 Мбайт/с.

Заметим, что в настоящее время шина VL-bus представляет из себя сравнительно недорогое дополнение для компьютеров с шиной ISA, причем с обеспечением обратной совместимости. Появилась версия 2.0 шинной архитектуры VL-Bus, в которую введены такие новшества, как мультиплексированный 64-

разрядный канал данных, буферизация сигналов для работы с быстродействующими системными платами и более высокая максимальная тактовая частота - 50 МГц. Количество разъемов расширения увеличится до трех разъемов на 40 МГц и до двух на 50 МГц. Ожидаемая скорость передачи теоретически должна возрасти до 400 Мбайт/с.

Стандарт IEEE 896.1-1988, названный Futurebus+, претендует на роль шины завтрашнего дня для систем массового применения. Стандарт Futurebus+ был разработан ассоциацией VITA (VFEA International Trade Association) в 1988 г. специально для высокоскоростных систем передачи информации. Требования к Futurebus+ были составлены таким образом, чтобы преодолеть все ограничения, присущие VME в телекоммуникационных системах. Ширина Futurebus+ - до 256 бит, максимальная скорость - 3,2 Гбайт/с, рабочая частота ограничивается лишь возможностями управляющего процессора.

Для сложных высокоскоростных шин, помимо упомянутых выше "мостов", применяются так называемые mezzanine-bus - более простые и "узкие" шины, сопрягаемые с основной без использования дополнительной управляющей электроники. Для Futurebus+ такими mezzanine-bus являются Sbus и PCI.

Шина PCI обладает несколькими преимуществами перед основной версией VL-Bus. В соответствии со спецификацией PCI к шине могут подключаться до 10 устройств. Это, однако, не означает использования такого же числа разъемов расширения - ограничение относится к общему числу компонентов, в том числе расположенных и на системной плате. Поскольку каждая плата расширения PCI может разделяться между двумя периферийными устройствами, то уменьшается общее число устанавливаемых разъемов.

Шина PCI может использовать 124-контактный разъем (32-разрядная) или 188-контактный разъем (64-разрядная передача данных), при этом теоретически возможная скорость обмена составляет соответственно 132 и 264 Мбайт/с. На системных платах устанавливаются обычно не более трех разъемов.

Предполагается, что стандарт PCI лучше соответствует растущим потребностям в скоростной обработке данных на настоль-

ных машинах, поскольку превосходит стандарт VL-Bus по сложности, гибкости и функциональной насыщенности. Windows принесла в мир ПК полноцветную графику. Процессор 486 выполняет пересылки данных по 32-разрядной шине, тактируемой частотой 33 МГц. Как только выдаваемый им мощный поток графических данных попадает на шину ISA, он упирается в "узкое горло". Эта шина работает на частоте всего лишь 8 МГц, а ее разрядность равна 16. По мере того как в прикладных программах начинают все шире использоваться многоцветная графика, "живое" видео и рендеринг трехмерных изображений, разработчикам систем и периферийных устройств пришлось предусмотреть другой способ связи с узлами машины, требующими наиболее интенсивного обмена данными.

Стандартная локальная шина обеспечивает единообразный способ подключения устройств к быстродействующей шине процессора и тем самым позволяет устранить "узкие места" во всех новых ПК. Шина PCI поддерживает 32-разрядный канал передачи данных между процессором и периферийными устройствами, работает на высокой тактовой частоте (33 МГц) и имеет максимальную пропускную способность 120 Мбайт/с. Кроме того, шина PCI в некоторой степени обеспечивает обратную совместимость с существующими периферийными устройствами, рассчитанными на шину ISA.

В стандарте PCI предусмотрены контроллер и акселератор, образующие локальную шину, не связанную с шиной процессора. В ней используется несколько способов повышения пропускной способности. Один из них - блочная передача последовательных данных. Если данные не являются последовательными, требуется дополнительное время на установку адреса каждого их элемента. Шина PCI создает между ЦП и периферийными устройствами некоторый промежуточный уровень. В результате получается процессорно-независимая шина, как ее называет Intel. Ее легко подключить к самым различным процессорам, в их числе Pentium (Intel), Alpha (DEC), MIPS R4400 и PowerPC (Motorola, Apple и IBM).

Для производителей систем это означает снижение затрат на разработку, так как с процессорами разного типа можно ис-

пользовать одни и те же элементы и устройства. Стандарт PC1 предусматривает обширный список дополнительных функций. К ним относится автоматическая конфигурация периферийных устройств, позволяющая пользователю устанавливать новые устройства без особых проблем.

PCI поддерживает целый спектр периферийных устройств и обладает средствами управления передачей данных (что освобождает процессор от рутинной возни с трафиком). Нет нужды говорить, что все обмены по шине буферизованы. PCI легко совместима с большинством известных шин. Разработаны и реализованы в виде стандартных микросхем многочисленные "мосты"; PCI/ISA, PCI/EISA, PPC/PCI и другие. Многие производители ПК практикуют также слоты двойного назначения - например, PCI/ISA, позволяющие на одно и то же место устанавливать устройства ввода-вывода в различных стандартах.

Интерфейс МПИ с мультиплексированными линиями адреса и данных предназначен для обеспечения информационной и электрической совместимости устройств системы. Он реализуется на основе магистрали и логических узлов, входящих в каждое подключаемое к ней устройство. Устройства в совокупности составляют единое адресное пространство магистрали.

В интерфейсе коды адреса и данных передаются по одной и той же группе сигнальных линий мультиплексированной шине обмена информацией) с разделением во времени. Принцип работы интерфейса при передаче данных — асинхронный, а при передаче адреса — синхронный.

В каждый момент времени на магистрали может выполняться один из трех видов взаимодействий подключенных к ней устройств: передача управления магистралью, адресный обмен (одиночный или блочный), прерывание.

Передача управления магистралью осуществляется в соответствии со схемой приоритета. Приоритет устройства определяется его положением на линии «разрешение на захват магистрали» (РЗМ) относительно других устройств. Приоритет устройства убывает по мере удаления устройства от микропроцессора, управляющего захватом магистрали, в направлении распространения сигнала РЗМ. При процедуре передачи управления магистралью

активное устройство, готовое к выполнению функции ведущего, асинхронно выставляет запрос на захват магистрали. МП выдает разрешение на захват магистрали после завершения текущего цикла обмена информацией или другого взаимодействия.

Адресный обмен строится по принципу ведущий — ведомый. В любой момент времени на магистрали взаимодействуют только одно ведущее и одно ведомое устройство. Ведущее устройство инициирует обмен информацией и задает его режим. При этом интерфейс может обеспечить режимы одиночного (обязательного) и блочного (необязательного) обменов данными.

Прерывание выполняемой программы МП осуществляет по запросам ВУ. При обработке запроса на прерывание процессор запоминает состояние прерванной программы и продолжает ее после завершения прерывающей программы. Контроллер ВУ, запросившего прерывание, по разрешению процессора выдает вектор прерывания, определяющий вход в процедуру обработки программы данного прерывания. Разрешение на выдачу вектора прерывания МП выдает в соответствии с многоуровневой системой приоритетов.

В зависимости от формата адреса процессора и диспетчера памяти пространство магистрали может составлять 64, 128, 256, 512, 1024, 2048, 4996, 8192 или 16 384 Кбайт. Во всех случаях 8 Кбайт адресного пространства магистрали используются для адресации ВУ, остальной для ячеек внутренних запоминающих устройств.

По магистрали информация передается в двоичном позиционном коде. Длина слова данных составляет 8 или 16 бит, а формат передаваемого адреса - 16 - 24 бит.

Интерфейс Unibus содержит магистраль из 56 сигнальных линий. Все устройства подсоединяются к этим линиям параллельно. Пять симплексных сигнальных линий используются для управления шиной приоритета, остальные 51 линий являются дуплексными; 18 адресных линий используются ведущим устройством для выборки ведомого устройства, с которым предстоит установить связь. Одна из линий адреса задает байт, к которому при операциях с байтами происходит обращение; 16 линий данных используются для передачи информации между ведущим и ведомым

устройствами. Две линии управления задают одну из четырех возможных операций обмена (два режима ввода и два - вывода).

Все передачи по общей шине осуществляются по методу «запрос - ответ». Такая организация взаимодействия позволяет объединить на магистрали устройства различного быстродействия. Для взаимной синхронизации ведущего и ведомого устройств используются две линии синхронизации. Для передачи управления магистралью ведущему устройству используется 11 линий приоритета (линии запроса, разрешения и подтверждения выбора). Для осуществления ввода-вывода данных без участия программы предусмотрен режим прямого доступа к памяти.

Интерфейс И-41 является одним из вариантов интерфейса Multibus, объединяющего стандартизованные интерфейсы IEEE, VME - bus, AMS - bus и др., с сохранением состава линий и их функций.

Интерфейсы 2 уровня обеспечивают объединение внешних устройств и устройств связи с объектами (УСО), которые используются в тех случаях, когда ВУ и УСО не имеют встроенного системного интерфейса и не могут подключаться непосредственно к системной магистрали. Наибольшее распространение здесь получили интерфейс ИРПС для радиального подключения устройств с последовательной передачей информации и интерфейс ИРПР для подключения устройств с параллельной передачей информации. С их помощью подключаются практически все периферийные устройства (дисплеи, принтеры, клавиатура, графопостроители и т. д.), за исключением внешних запоминающих устройств, предъявляющих более высокие требования к пропускной способности интерфейса.

В качестве интерфейса УСО могут быть использованы магистраль КАМАК или специальные интерфейсные платы - контроллеры, обеспечивающие подключение модулей УСО к системному интерфейсу. Сопряжение малого интерфейса с системной магистралью осуществляется при помощи контроллера К (рис. 3.18).

Интерфейсы 3 уровня предназначены для объединения датчиков и исполнительных устройств. Большое разнообразие датчиков и исполнительных устройств на сегодняшний день привело к разработке огромного числа этих интерфейсов. Интерфейсы

4 уровня представляют собой интерфейсы устройств передачи данных (УПД). К ним относятся интерфейсы телеграфных, телефонных, высокочастотных, оптоволоконных и других каналов для передачи данных на большие расстояния. Сюда же относятся интерфейсы распределенных систем управления общего и специального назначения (КАМАК МЭК - 640, МЭК - 625 - 1 последовательный, ИЛПС - 2 и др.) и интерфейсы локальных сетей общего назначения (Р - 802 и др.)

Интерфейсы 5 уровня включают внешние относительно микропроцессорной системы интерфейсы. Соединение внешнего интерфейса с системным осуществляется при помощи специального адаптера интерфейсов.

В большинстве из рассмотренных интерфейсов применяют три режима передачи данных (и соответственно три типа каналов связи): симплексный, полудуплексный и дуплексный.

Симплексный режим обеспечивает одностороннюю связь между передатчиком и приемником, территориально разнесенных между собой.

Полудуплексный режим обеспечивает двусторонний обмен данными между двумя точками, в каждой из которых имеется передатчик и приемник, но одновременная передача в двух направлениях невозможна. Для изменения направления передачи требуется некоторое время переключения (коммутации).

При передаче больших объемов информации применяются дуплексный режим, обеспечивающий одновременную передачу информации в обоих направлениях.

Обмен информацией в интерфейсах может производиться с использованием синхронного (обмен со стробированием) и асинхронного (обмен с квитированием) принципов обмена. В первом случае устройство - источник (контроллер) определяет темп выдачи и приема информации и синхронизирует все процессы, связанные с трансляцией данных. Обычно синхронизируется прохождение в линии каждого бита, группы битов (символа) и сообщения.

Асинхронный принцип передачи в интерфейсах, как правило, основан на режиме запроса - ответа. В этом случае устройство - источник по одной из линии интерфейса вырабатывает сигнал о выдаче данных на ШД и направляет его в устройство - приемник.

Приемник фиксирует поступление сигнала готовности источника, принимает данные и извещает об этом источник сигналом, появляющимся на другой линии (строб готовности приемника). Источник, восприняв ответ, снимает передаваемые данные. Таким образом, интервал времени, в течение которого источник выводит данные на шину интерфейса, является переменным и зависит от характеристик как самого источника, так и приемника сигналов, а также характеристик линий связи.

Хотя при синхронной передаче данных по сравнению с асинхронной более эффективно используется канал связи и достигается лучшая помехозащищенность передаваемых данных, в интерфейсах автоматизированных систем научных исследований применяют, как правило, асинхронный способ передачи. Это обусловлено возможностью передавать в асинхронном режиме данные со скоростью, соответствующей быстродействию того устройства, с которым в данный момент времени происходит обмен информацией (автоматическая подстройка скорости передачи данных).

Интерфейс AGP предназначен для вывода информации на внешние устройства, в том числе отображения данных. Она содержит шину и устройство передачи информации (видеоускоритель), образующие интерфейсную схему. В настоящее время наибольшее применение получил интерфейс AGP

В начале 1997 г. фирмой Intel был разработан новый стандарт для вывода графики, получивший название AGP (Accelerated Grafics Port). Здесь видеопамять располагается не на графическом адаптере, а в ОЗУ компьютера. В процессе обработки информации процессор автоматически выделяет необходимый объем памяти для вывода графики. Физически это будет реализовано в виде добавки для шины PCI и полностью прозрачно для нее.

AGP работает на частоте основной памяти (66 МГц) и в обычном режиме (x1), при котором данные передаются только по переднему фронту тактового сигнала, дает возможность достичь пиковой пропускной способности 266 Мбайт/с, а в режиме (x2), при котором данные передаются и по переднему, и по заднему фронту тактового сигнала, при этом пропускная способность достигает значения в 532 Мбайт/с.

AGP работает в двух режимах. Первый из них основан на традиционной модели DMA, а второй - на новой модели DIME. В зависимости от выбранного режима данные по-разному распределяются между основной и локальной памятью, что, в свою очередь, влияет на качество отображаемой картинки и частоту смены кадров.

В режиме DMA для графики используется только локальная память видеоускорителя, а данные, расположенные вне ее, предварительно загружаются в локальную память и лишь затем обрабатываются видеопроцессором. При этом AGP выполняет роль быстрой шины.

В режиме DIME для построения изображения видеоускоритель использует локальную и системную память. При этом любая структура данных может располагаться как в локальной, так и в системной памяти. Данные не копируются предварительно из системной памяти в локальную, а интерпретируются «на месте».

Необходимо отметить одну важную особенностью AGP, которая состоит в том, что память под текстуры выделяется операционной системой по требованию исполняемой программы и остается доступной для него. Поскольку текстура может занимать более одной страницы оперативной памяти (более 4 Кбайт), то в спецификацию AGP включена таблица переадресации графики, содержимое которой должно быть согласовано с таблицами переадресации операционной системы. Таким образом, поддержка AGP не может ограничиваться драйвером производителя, здесь требуется поддержка на уровне операционной системы. Такая поддержка реализована, например, в версии Windows 98.

Преимуществом при использовании AGP является следующее:

- интерфейс реализован по принципу «соединение точка-точка», при котором отсутствуют проблемы с арбитражем шин,
- в интерфейсе предусмотрены раздельные шины для передачи команд и данных, повышающие пропускную способность интерфейса,
- применение конвейеризации и технологии отложенного выполнения команд, позволяющие вплотную приблизиться к теоретическому скоростному пределу интерфейса.

Принимая во внимание перечисленные преимущества интерфейса AGP разработано большое количество видеоускорителей, совместимых со спецификацией AGP, среди которых можно отметить Asus 3DexPlorer 3000, ATI Xpert@Work, Diamond Viper330, Matrox Millenium II, STB Velocity 128 и др.

В заключение необходимо отметить, что единого подхода по оценке быстродействия пока нет. Связано это с тем, что, во-первых, разные видеоускорители на аппаратном уровне реализуют очень разные наборы функций, а, во-вторых, различные видеопрограммы используют существенно разные функции. Большое разнообразие функций является единственной главной причиной, не позволяющей однозначно определить быстродействие. Однако проведенные приближенные оценки показывают, что для большинства программ трехмерной графики быстродействие микропроцессорных систем вывода видеоинформации с использованием видеоускорителей, использующих шину PCI, и систем с использование интерфейса AGP приблизительно одинаково.

12. Организация памяти МПС

Память современных вычислительных систем имеет иерархическую многоуровневую структуру. Чем выше уровень, тем выше требуемое быстродействие соответствующей памяти. Основная или оперативная память относится к верхнему уровню памяти.

Сравнительно небольшая емкость ОП (до нескольких десятков Мбайт) компенсируется практически неограниченной емкостью внешний запоминающих устройств на магнитных дисках (до нескольких Гбайт). Однако эти устройства сравнительно медленные, и время обращения за данными для дисков составляет десятки миллисекунд. Поэтому вычислительный процесс должен протекать с возможно меньшим числом обращений к внешним запоминающим устройствам и максимально возможным использованием ОП.

Быстродействие ОП часто оказывается недостаточным для обеспечения требований, предъявляемых к скорости работы МПС. Это проявляется в несоответствии пропускных способностей про-

цессора и ОП. Возникающая проблема выравнивания их пропускных способностей решается путем использования буферных памятей небольшой емкости и повышенного быстродействия. Они используются для хранения команд и данных, относящихся к обрабатываемому участку программы.

Оперативная память является наиболее дефицитным ресурсом в вычислительных системах, которым надо пользоваться экономно и эффективно. Проблема усложняется при переходе к мультипрограммным системам, так как в них ОП одновременно использует несколько программ (заданий). В таких системах важным стоит вопрос исключения несанкционированного воздействия одних программ на другие. Это достигается с помощью механизма защиты памяти.

Защита памяти. Если в памяти одновременно могут находиться несколько независимых программ, необходимы специальные меры по предотвращению или ограничению обращений одной программы к областям памяти, используемым другими программами. Программы могут содержать такие ошибки, которые, если этому не воспрепятствовать, приводят к искажению информации, принадлежащей другим программам. Последствия таких ошибок особенно опасны, если разрушению подвергнутся программы операционной системы. Другими словами, надо исключить воздействие программы пользователя на работу программ других пользователей и программ операционной системы.

Чтобы воспрепятствовать разрушению одних программ другими, достаточно защитить область памяти данной программы от попыток записи в нее со стороны других программ, а в некоторых случаях и своей программы (защита от записи), при этом допускается обращение других программ к этой области памяти для считывания данных.

В других случаях, например при ограничениях на доступ к информации, хранящейся в системе, необходимо иметь возможность запрещать другим программам производить как запись, так и считывание в данной области памяти. Такая защита от записи и считывания помогает отладке программы, при этом осуществляется контроль каждого случая выхода за область памяти своей программы.

Для облегчения отладки программ желательно выявлять и такие характерные ошибки в программах, как попытки использования данных вместо команд или команд вместо данных в собственной программе, хотя эти ошибки могут и не разрушать информацию.

Необходимо отметить следующие варианты дифференцированной защиты при различных операциях с памятью:

— задается отношение к области памяти чужой программы, определяющее, относится защита памяти только к операции записи или к любому обращению в память;

— задается одно из следующих отношений к области памяти собственной программы;

а) разрешается доступ к данному блоку как для записи, так и для считывания;

в) разрешается обращение любого вида, но по адресу, взятому только из счетчика команд;

г) разрешается обращение по адресу из любого регистра, кроме счетчика команд.

Если нарушается защита памяти, исполнение программы приостанавливается и вырабатывается запрос прерывания по нарушению защиты памяти.

Защита от вторжения программ в чужие области памяти может быть организована различным образом, при этом реализация защиты не должна заметно снижать производительность МПС и требовать слишком больших аппаратурных затрат.

Защита отдельных ячеек памяти. В управляющих вычислительных комплексах необходимо обеспечить возможность отладки новых программ параллельно с функционированием находящихся в памяти рабочих программ, управляющих технологическим процессом. Это может быть достигнуто выделением в каждой ячейке памяти специального «разряда защиты». Установка «1» в этот разряд запрещает производить запись в данную ячейку.

В системах с мультипрограммной обработкой большого числа программ защищаются не отдельные ячейки, а области памяти или блоки, на которые делится память, при этом часто предусматривается возможность указывать для разных программ различные

допустимые режимы обращения к отдельным областям или блокам памяти.

Существует несколько методов защиты памяти, среди которых можно выделить метод граничных регистров и метод ключей защиты.

Метод граничных регистров состоит во введении двух граничных регистров, указывающих верхнюю и нижнюю границы области памяти, куда программа имеет право доступа. При каждом обращении к памяти проверяется, находится ли используемый адрес в установленных границах; при выходе за границы обращение к памяти подавляется и формируется запрос прерывания, передающий управление операционной системе.

Содержание граничных регистров устанавливаются операционной системой перед тем, как для очередной исполняемой программы начнется активный цикл. Если для динамичного распределения памяти используется базовый регистр, то он одновременно определяет и нижнюю границу. Верхняя граница подсчитывается операционной системой в соответствии с длиной программы оперативной памяти.

Метод ключей защиты. По сравнению с предыдущим данный метод является более гибким: он позволяет организовать доступ программы к областям памяти, расположенным не подряд.

Память в логическом отношении делится на одинаковые блоки. Каждому блоку памяти ставится в соответствие код, называемый ключом защиты памяти, а каждой программе, принимающей участие в мультипрограммной обработке, присваивается код ключа программы. Доступ программы к данному блоку памяти для чтения и записи разрешен, если ключи совпадают или один из них имеет код «0». Коды ключей защиты памяти хранятся в специальной памяти защиты ключей, более быстродействующей, чем ОП.

Пропускная способность процессора и МПС. Кэш – память. Непрерывный рост производительности (скорости работы) МПС проявляется, в первую очередь, в повышении скорости работы электронных схем, а также специальных архитектурных решений — конвейерная и векторная обработка данных, кэширование памяти и др.

Быстродействие оперативной памяти также растет, но все время отстает от быстродействия аппаратурных средств процессора, в значительной степени потому, что одновременно происходит опережающий рост ее емкости. Это делает более трудным уменьшение времени цикла работы памяти.

Без согласования пропускных способностей процессора и памяти невозможно в системе реализовать производительность, соответствующую быстродействию процессора. Преодолеть указанное противоречие и согласовать пропускные способности памяти и процессора помогают специальные структурные решения.

Конвейеризация процедур цикла выполнения команды (рабочего цикла машины) в простейшем случае предполагает выполнение параллельно во времени операции в АЛУ с выборкой из памяти следующей команды.

Буферизация — использование включенных между процессором и ОП существенно более, чем ОП, быстродействующих буферных памятей сравнительно небольшой емкости. На рис. 76 показана структура процессора, содержащая буфер команд и буфер операндов.

Представленные на рис. 76 схемы буферной памяти скрыты от программиста в том смысле, что он не может их адресовать, может даже не знать об их существовании. Поэтому они получили название кэш - памятей. Структура высокопроизводительных 32- и 64-разрядных МП содержит объединенную кэш - память для фрагментов программ и групп данных, при этом в ряде случаев наряду с кэш - памятью сохраняется небольшой буфер на несколько команд. Обмен информацией между кэш-памятью и микропроцессором осуществляется высокоскоростной локальной шиной, при этом кэш-команд имеет свою шину, а кэш-данных свою. В последних разработках (Pentium III и IV) работа шин осуществляется независимо друг от друга, за что они получили название USB - двойной независимой шины.

Кэш - память чаще всего располагается на одном кристалле с процессором, но может располагаться и вне кристалла, но при этом она находится на той же плате вычислителя и служит высокоскоростным буфером между процессором и относительно медленной основной памятью. При некоторых обращениях к памяти

соответствующие значения заносятся в кэш, и в ходе следующих операций чтения по тем же адресам обращения происходят только к кэш - памяти.

Рис. 76

Для обращения к кэш - памяти, размещенной вне кристалла процессора, но на одной плате с процессорной БИС, может потребоваться несколько циклов памяти, тогда как при обращении к кэш - памяти, находящейся в процессорном кристалле, может оказаться достаточно одного такого цикла; однако даже размещение кэш - памяти на одной плате с процессором позволяет избежать большого числа циклов ожидания, которые неизбежны при работе с памятью, расположенной на отдельной плате и взаимодействующей с процессором через системную шину, а также снизить нагрузку системной шины.

Кэш - память может использоваться для хранения либо команд, либо данных, либо информации обоих этих видов. Выборка ее содержимого может производиться произвольным образом, и, следовательно, в адресном диапазоне кэш-памяти можно разместить командные циклы с входящими в них командами переходов.

Производительность кэш - памяти определяется временем доступа и вероятностью удачных обращений. Она зависит от объема кэш-памяти и количества битов, записываемых в него при каждом обращении к основной памяти, называемого длиной строки. С увеличением длины строки повышается вероятность того, что следующее обращение будет удачным, т. е. необходимая

информация окажется в кэш-памяти. Если при объеме кэш 4 Кбайт и длине строки 4 байт вероятность удачных обращений составляет 80%, то при увеличении длины строки до 8 байт эта величина может достигнуть 85%. Однако дальнейшее увеличение длины строки не приводит к заметному росту вероятности удачных обращений. Так, например, увеличение длины строки еще в два раза (до 16 байт) позволяет довести вероятность удачных обращения примерно до 87%.

Полная производительность памяти является функцией времени доступа к кэш-памяти, вероятности удачных обращений к кэш и временем обращения к основной памяти, которое происходит при неудачном обращении к кэш. Возможна, например, ситуация, когда в системе памяти вероятность удачных обращений к кэш, время доступа к которому равно 120 нс, составляет 80%, а вероятность обращений к основной памяти с временем доступа 600 нс равна 20%. При этом среднее время доступ к памяти составляет $(0{,}8 \times 120) + 0{,}2 \times (600 + 120) = 240$ нс.

При обращении процессора к ОП для считывания в кэш передается блок информации, содержащий нужное слово. При этом происходит опережающая выборка, так как высока вероятность того, что ближайшие обращения будут происходить к словам этого же блока, уже находящимся в кэш-памяти. Это приводит к значительному уменьшению среднего времени, затрачиваемого на выборку данных.

Эффективность кэш-памяти, зависящая от ее емкости, размера слова, соотношения времен считывания слова из кэш-памяти и блока из ОП проявляется в уменьшении среднего времени, затрачиваемого на выборку слова данных.

Можно выделить два типа кэш - памяти:

— с запоминанием новой информации одновременно в кэш и в ОП («сквозное запоминание»), при этом в ОП есть всегда последняя копия хранящейся в кэш информации. Однако в этом случае длинный цикл ОП снижает производительность процессора;

— с запоминанием новой информации только в кэш и копированием ее в ОП только при передаче в другие устройства или при вытеснении из кэш-памяти.

Динамическое распределение памяти. Виртуальная страничная и сегментная память. Эффективное распределение ресурсов памяти между программами не может быть статическим, т. е. не может производиться предварительно до пуска программы. В процессе обработки программ потребности в ресурсах памяти отдельных программ изменяются, что заранее не может быть учтено. Зачастую целесообразнее распределять память между программами динамически непосредственно в ходе вычислительного процесса, т. е. осуществлять динамическое распределение памяти. При этом должна обеспечиваться возможность независимой работы программистов над своими программами, подлежащими мультипрограммной обработке.

Необходимо отметить, что динамическое распределение памяти не должно приводить к дроблению ее свободного пространства — фрагментации памяти, затрудняющему ее использование. Это достигается организацией одноуровневой виртуальной памяти, допускающей адресацию на все адресное пространство. Размер его определяется количеством разрядов, которые могут быть использованы для представления адреса. К тому же в мультипрограммных системах размещение всех исполняемых программ полностью в ОП во многих случаях невыполнимо: программы часто имеют большую длину, а емкости существующих ОП ограничены. Однако нет принципиальной необходимости в том, чтобы вся программа находилась в ОП, так как в любой момент времени работа программы концентрируется на определенных сравнительно небольших участках. Таким образом, в ОП следует хранить только используемые в данный период времени части программ, а неиспользуемые части могут располагаться во ВЗУ. Программируя свою программу, программист не знает, в комбинации с какими программами будет выполняться его программа, какое место в памяти отведет ей операционная система.

При подготовке программ используются условные адреса. Позднее в процессе выполнения программы операционная система выделяет активным частям программы место в памяти и условные адреса переводятся в исполнительные. Эта процедура получила название динамического распределения памяти.

Осуществление динамического распределения чисто программным путем привело бы к значительным потерям машинного времени. Целесообразнее пользоваться для этой цели аппаратурными средствами.

Один из способов динамического распределения памяти основан на использовании базовых регистров. Операционная система каждой исполняемой программе ставит в соответствие свой базовый адрес. Базовые адреса программ находятся в общих регистрах. При выполнении программы реальный или физический адрес образуется суммированием базового и относительного адресов. При динамическом распределении памяти с помощью базовых регистров программа (или, по крайней мере, та ее часть, адрес которой преобразуется с помощью одного и того же базового адреса) должна располагаться в последовательных ячейках и вводиться в ОП целиком, хотя в ближайшем цикле активности, возможно, может потребоваться лишь небольшой фрагмент программы.

При рассматриваемом способе динамического распределения памяти свободная память может состоять из несвязанных областей (фрагментация памяти) и для ввода нужной программы может понадобиться сдвиг содержимого памяти. На рис. 77, а показано распределение памяти между программами A, B, C, D, из которых две, например, A и D,. являются в данный момент наименее активными и следовательно, могут рассматриваться как кандидаты на удаление во внешнюю память

Если вновь вводимая программа E (рис. 77, б) больше любой из программ A и D, то для ее размещения в памяти необходимо, как показано на рис. 77, в, сдвигать программы B и C. Это перемещение связано с потерей времени. Более того, в ряде систем подобное перемещение требует выполнения заново операции редактирования связей в программе и новой заг-рузки программы, что в ряде случаев ведет к существенному снижению быстродействия системы в целом. Эти и некоторые другие недостатки в распределении памяти отсутствуют в виртуальной памяти со страничной организацией.

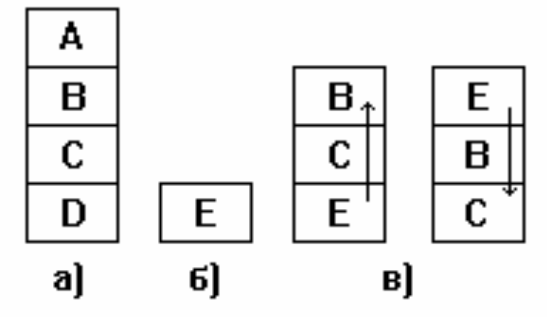


Рис.77

Виртуальная память есть способ, организации памяти мультипрограммной вычислительной системы, при котором достигается гибкое динамическое распределение памяти, устраняется ее фрагментация и создаются значительные удобства для работы программистов. Это удается достигнуть без заметного снижения производительности МП ценой усложнения аппаратуры и операционной системы и процессов их функционирования.

Принцип виртуальной памяти предполагает, что пользователь при подготовке своей программы имеет дело не с физической ОП, действительно работающей в составе вычислительной системы и имеющей некоторую фиксированную емкость, а с виртуальной (т. е. кажущейся) одноуровневой памятью, емкость которой равна всему адресному пространству, определяемому размером адресных полей в форматах команд и базовых регистров.

Пользователь имеет в своем распоряжении все адресное пространство системы независимо от объема ее физической памяти и объемов памятей, необходимых для других программ, участвующих в мультипрограммной обработке. На всех этапах подготовки программ, включая загрузку в ОП, программа представляется в виртуальных адресах, и лишь при самом исполнении машинной команды производится преобразование виртуальных адресов в реальные адреса действующей памяти (в так называемые физические адреса.). Преобразование виртуальных адресов в физические упрощается и устраняется фрагментация памяти, если физическую и виртуальную память разбить на блоки двух типов. К первому

типу относятся блоки, образующие страницы, ко второму типу - сегменты.

При страничной организации память разбивается на страницы фиксированного размера, например, по 512 байт. В них осуществляется загрузка программ под управлением операционной системы. Страницам виртуальной и физической памяти присваивают номера, называемые номерами соответственно виртуальных и физических страниц. Каждая физическая страница способна хранить одну из виртуальных страниц. Порядок расположения (нумерация) байт в виртуальной и физической страницах сохраняется одним и тем же.

В мультипрограммной системе страничная организация памяти дает определенные преимущества. Когда новая программа загружается в ОП, она может быть направлена в любые свободные в данный момент физические страницы независимо от того, расположены они подряд или нет. Не требуется перемещения информации в остальной части памяти. Страничная организация позволяет сократить объем передачи информации между внешней памятью и ОП, так как страница программы не должна нагружаться до тех пор, пока она действительно не понадобится. Сначала в ОП загружается начальная страница программы и ей передается управление. Если по ходу работы делается попытка выборки слов из другой страницы, то производится автоматическое обращение к операционной системе, которая осуществляет загрузку из ВЗУ требуемой страницы.

На рис. 78 показано соответствие между виртуальной и физической памятями, устанавливаемое страничной таблицей. Очевидно, что физические страницы могут содержаться в текущий момент времени как в оперативной, так и во внешней памяти.

Страничная таблица для каждой программы формируется операционной системой в процессе распределения памяти и перерабатывается ею каждый раз, когда в распределении памяти производятся изменения. Процедура обращения к памяти состоит в том, что номер виртуальной страницы извлекается из адреса и используется для входа в страничную таблицу, которая указывает номер соответствующей физической страницы. Этот номер вместе с номером байта, взятым непосредственно из виртуального адреса,

представляет собой физический адрес, по которому происходит обращение к ОП.

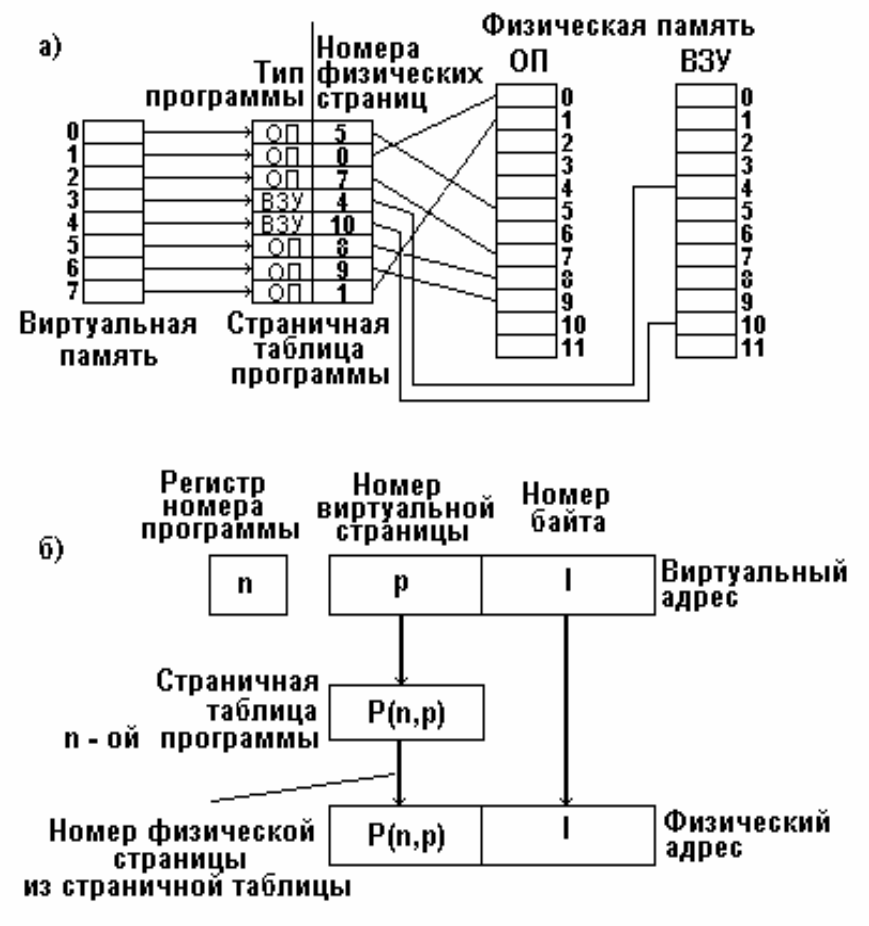


Рис.78

Если страничная таблица указывает на размещение требуемой информации во внешней памяти, то обращение к ОП не может состояться немедленно: операционная система должна организовать передачу из внешней памяти в ОП нужной страницы.

Для каждой из программ, обрабатываемых в мультипрограммном режиме, организуется своя область виртуальной памяти

и создается своя страничная таблица, при этом все программы делят между собой одну общую физическую память.

Страничные таблицы программ хранятся в ОП, и обращение к нужной строке активной страничной таблицы в ОП происходит по адресу, который определяется номером активной программы и номером виртуальной страницы.

Для ускорения преобразования адресов используется небольшая сверхоперативная память, куда передается из ОП страничная таблица активной программы. В другом варианте в сверхоперативной памяти могут находиться сведения о номерах виртуальных и соответствующих физических страниц для нескольких недавно использовавшихся страниц, в том числе принадлежащим разным программам. В этом варианте сверхоперативная память, используемая при преобразовании адресов, строится как ассоциативная с обращением не по адресу, а по содержанию хранимой в ячейке информации — в данном случае по хранимому в ячейке номеру программы и номеру виртуальной страницы.

Сегментная организация находит широкое применение в модульном программировании, при котором с целью упрощения понимания, написания и контроля программы и/или отдельных ее частей для реализации каждой функции используется отдельный программный модуль. Модуль определяется функцией, которую он выполняет, а не размерами, которые могут быть разными для различных модулей. Сегментная организация виртуальной памяти позволяет каждому модулю занимать свою собственную сплошную область памяти, тогда как при страничной организации модуль разбивается на страницы.

Логический адрес, вырабатываемый процессором для системы сегментной виртуальной памяти, состоит из номера сегмента и смещения. Таблица сегментов содержит значения базовых адресов (представляющих собой начальные адреса сегментов в физической памяти) и границ (определяющих объемы сегментов). Величина смещения не должна превышать значение границы. Защита сегмента, содержащего законченный программный модуль, обеспечивается путем однократного задания прав на использование и доступ для записи и чтения.

В процессе работы системы с сегментной виртуальной памятью в ней могут появляться так называемые «дырки», представляющие собой неиспользуемые области, которые требуют объединения. В системах реального времени это действие может служить причиной определенных неудобств. Сегментная виртуальная память должна быть снабжена алгоритмом размещения, с помощью которого осуществляется поиск неиспользуемых зон памяти для размещения каждого сегмента. В системах страничной виртуальной памяти необходимости в этих сложных алгоритмах нет; кроме того, при страничной организации виртуальной памяти отсутствует опасность появления «дырок» между программами (внешней фрагментации), а величина «дырок» внутри программы (внутренней фрагментации) не может превышать объема одного блока для каждой программы.

Сегментно - страничная организация памяти. До сих предполагалось, что виртуальная память, которой располагает программист, представляет собой непрерывный массив с единой нумерацией байт. Однако программа обычно состоит из нескольких массивов — подпрограмм, одной или нескольких секций данных. Так как заранее длины этих массивов неизвестны, то удобно, чтобы при программировании каждый массив имел свою собственную нумерацию байт, начинающуюся с нуля и продолжающуюся в возрастающем порядке. Желательно также, чтобы составленная таким образом программа могла работать при динамическом распределении памяти, не требуя от программиста усилий по объединению различных ее частей в единый массив. Эта задача решается в некоторых вычислительных системах путем использования особого метода преобразования виртуальных адресов в физические, называемого сегментно - страничной организацией памяти.

Виртуальная память каждой программы делится на части, именуемые сегментами, с независимой адресацией байт внутри каждой части. К виртуальному адресу следует добавить дополнительные разряды левее номере страницы; эти разряды определяют номер сегмента.

Возникает определенная иерархия в организации программ, состоящая из четырех ступеней: программа → сегмент → страница

→ байт. Этой иерархии программ соответствует иерархия таблиц, служащих для перевода виртуальных адресов в физические. Программная таблица для каждой программы, загруженной в систему, указывает начальный адрес соответствующей сегментной таблицы. Сегментная таблица перечисляет сегменты данной программы с указанием начального адреса страничной таблицы, относящейся к данному сегменту. Страничная таблица определяет расположение каждой из страниц сегмента в памяти. Страницы сегмента могут располагаться не подряд, часть страниц данного сегмента может находиться в оперативной памяти, остальные — во внешней.

Рассмотрим для примера организацию сегментно-страничной виртуальной памяти некоторой вычислительной системы. Пусть сегмент представляет собой блок последовательных адресов размером 64 Кбайт или 1 Мбайт, размер страницы - 2 или 4 Кбайт. При этом начальные адреса сегментов и страниц кратны их размерам. Размеры сегментов и страниц виртуальной памяти активной в данный момент программы задаются значениями «0» соответствующих разрядов управляющего регистра. Тогда виртуальный адрес (как и физический) будет иметь длину 24 разряда, причем поле номера сегмента будет занимать 8 или 4 старших разряда соответственно для сегментов размером в 64 Кбайт и 1 Мбайт, поле номера байта - 11 или 12 младших разрядов для страниц размером 2048 и 4046 байт. Промежуточные разряды адреса занимает поле номера страниц, которое может иметь 4, 5, 8 или 9 разрядов в зависимости от размеров сегмента и страницы.

Сегментные и страничные таблицы находятся в ОП, а в программной таблице нет необходимости, так как для каждой активной в данный момент программы управляющий регистр хранит начальный адрес и длину соответствующей сегментной таблицы. Хранит он также и номер программы.

Процесс преобразования адресов представлен на рис. 79. В общем случае преобразование адреса происходит в два этапа и требует двух дополнительных обращений к ОП (рис. 79, а). На первом этапе начальный адрес сегментной таблицы, установленный в управляющем регистре 1, суммируется с номером сегмента из виртуального адреса. В результате образуется адрес, по которо-

му из ОП считывается строка сегментной таблицы, содержащая адрес начала и длину страничной таблицы для данного сегмента. На втором этапе полученный адрес начала страничной таблицы суммируется с номером страницы из виртуального адреса, при этом образуется адрес, по которому из ОП считывается строка страничной таблицы. Если эта страница оказывается в ОП, то в старшие разряды регистра физического адреса передается ее номер, а в младшие заносится номер байта из регистра виртуального адреса. Формирование физического адреса на этом завершается.

Если нужная физическая страница оказывается во внешней памяти, то формируется сигнал прерывания, осуществляющий одноименную процедуру, называемую прерыванием по страничному сбою.

Операционная система инициирует передачу этой страницы из внешней памяти в ОП (при этом меняется номер физической страницы) и корректирует соответствующим образом страничную таблицу, находящуюся в оперативной памяти. В старшие разряды регистра физического адреса передается новый номер физической страницы, а в младшие — номер байта. Таким образом формируется физический адрес.

Далее выполняется запрошенное программой обращение к ОП. Одновременно информация о текущей странице (номерах программы, сегмента, виртуальной и соответствующей физической страницы) помещается в сверхоперативную ассоциативную память или в блок быстрой переадресации небольшой емкости.

Ассоциативная память. Ассоциативная память хранит указанные данные для небольшого числа недавно использовавшихся страниц. При наличии ассоциативной памяти значительно ускоряется процесс преобразования адресов, так как на каждом участке вычислительного процесса обращения к ОП сосредотачиваются на небольшом числе страниц, и поэтому всегда существует большая вероятность того, что текущее обращение произойдет к странице, информация о которой уже имеется в ассоциативной памяти, а следовательно, возможно быстрое преобразование адресов без дополнительных обращений к ОП.

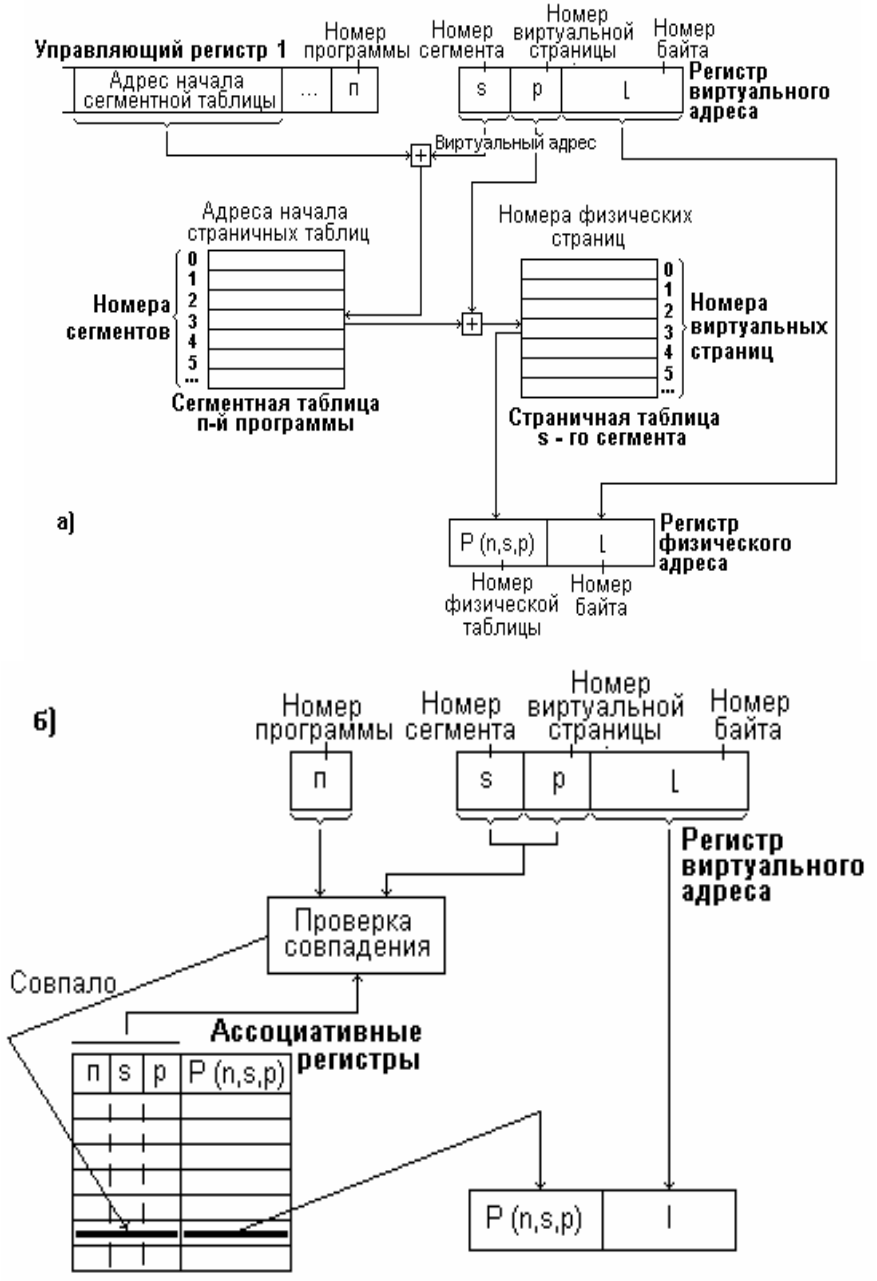

Управляющий регистр 1
Номер программы
Номер сегмента
Номер виртуальной страницы
Номер байта
Адрес начала сегментной таблицы
n
s
p
l
Регистр виртуального адреса
Виртуальный адрес
Адреса начала страничных таблиц
Номера физических страниц
Номера сегментов
0
1
2
3
4
5
Номера виртуальных страниц
Сегментная таблица n-й программы
Страничная таблица s - го сегмента
а)
P (n,s,p)
l
Регистр физического адреса
Номер физической таблицы
Номер байта
б)
Номер программы
Номер сегмента
Номер виртуальной страницы
Номер байта
n
s
p
l
Регистр виртуального адреса
Проверка совпадения
Совпало
Ассоциативные регистры
n
s
p
P (n,s,p)
P (n,s,p)
l

Рис.79.

Преобразование адресов всегда начинается с просмотра ассоциативной памяти. Если оказывается, что в одной из ее строк (ассоциативном регистре) хранится информация о странице, к которой должно произойти обращение, то из этой строки непосредственно выбирается номер физической страницы и дополнительные обращения к ОП (к сегментной и страничной таблицам) не производятся (рис.79,6). Если нужной информации нет в ассоциативной памяти, то делается попытка сократить время преобразования путем исключения одного дополнительного обращения к ОП (первый этап на рис. 79, а). Может оказаться так, что страница, к которой происходит обращение, принадлежит сегменту предыдущего обращения к ОП.

В аппаратуре преобразования адресов сохраняются номер сегмента и адрес начала его страничной таблицы для предыдущего обращения. Если совпадают номера сегментов текущего и предыдущего обращений, первый этап преобразования исключается, используется сохраненный адрес начала сегментной таблицы и выполняется только второй этап преобразования, т. е. производится только одно дополнительное обращение к ОП. Если номера сегментов не совпадут, то реализуется полная процедура преобразования адресов, показанная на рис. 79, а.

Дополнительные обращения к ОП сопровождаются занесением информации о текущей странице в ассоциативную память. Если в ассоциативной памяти не оказывается свободного регистра (строки), то данные о новой странице записываются на место данных, которые дольше других не использовались в процессе преобразования адресов.

13. Технологические аспекты полупроводниковой технологии

Совершенствование технологического процесса изготовления микропроцессоров – это главный атрибут повышения их быстродействия и надежности. Переход на новые техпроцессы является очевидным шагом, но технологам это дается каждый раз все с большим трудом. Современные процессоры выполняются по технологии 0,13 и 0,09 мкм, причем последняя была введена в 2004 году. Как видно, для этих техпроцессов соблюдается закон Мура,

который гласит, что каждые два года частота кристаллов удваивается при увеличении количества транзисторов с них. С такими же темпами сменяется и техпроцесс. Правда, в дальнейшем «гонка частот» опередит этот закон. К 2006 году компания Intel планирует освоение 65-нм техпроцесса, а 2009 – 32-нм

Здесь пора вспомнить структуру транзистора (рис.80), а именно - тонкий слой диоксида кремния SiO_2, изолятора, находящегося между затвором и каналом, и выполняющего вполне понятную функцию - барьера для электронов, предотвращающего утечку тока затвора. Очевидно, что чем толще этот слой, тем лучше он выполняет свои изоляционные функции, но он является составной частью канала, и не менее очевидно, что если мы собираемся уменьшать длину канала (размер транзистора), то нам надо уменьшать его толщину, причем, весьма быстрыми темпами. За последние несколько десятилетий толщина этого слоя составляет в среднем порядка 1/45 от всей длины канала. Но у этого процесса есть свой конец - как утверждал пять лет назад все тот же Intel, при продолжении использования SiO2, как это было на протяжении последних 30 лет, минимальная толщина слоя будет составлять 2.3 нм, иначе ток утечка тока затвора приобретет просто нереальные величины.

Для снижения подканальной утечки до последнего времени ничего не предпринималось. Сейчас ситуация начинает меняться, поскольку рабочий ток, наряду со временем срабатывания затвора, является одним из двух основных параметров, характеризующих скорость работы транзистора, а утечка в выключенном состоянии на нем непосредственно сказывается - для сохранения требуемой эффективности транзистора приходится, соответственно, поднимать рабочий ток, со всеми вытекающими условиями.

Изготовление микропроцессора - это сложнейший процесс, включающий более 300 этапов. Микропроцессоры формируются на поверхности тонких круговых пластин кремния - подложках, в результате определенной последовательности различных процессов обработки с использованием химических препаратов, газов и ультрафиолетового излучения.

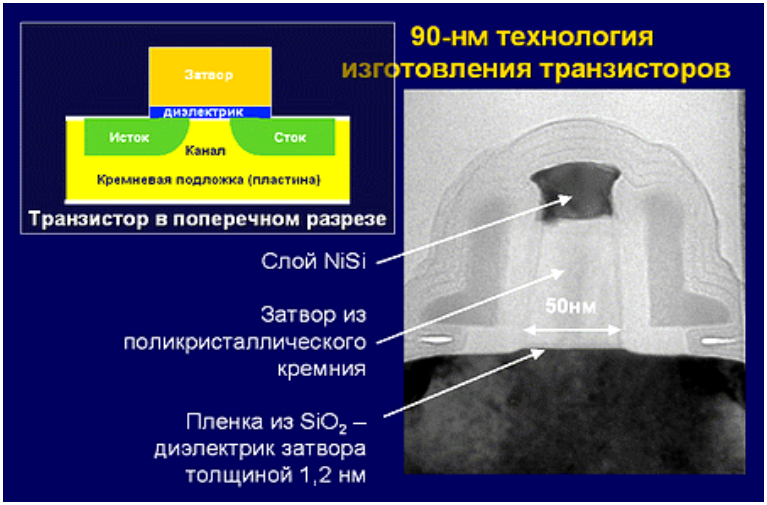


Рис. 80

Подложки обычно имеют диаметр 200 миллиметров, или 8 дюймов. Однако корпорация Intel уже перешла на пластины диаметром 300 мм, или 12 дюймов. Новые пластины позволяют получить почти в 4 раза больше кристаллов, и выход годных значительно выше. Пластины изготавливают из кремния, который очищают, плавят и выращивают из него длинные цилиндрические кристаллы. Затем кристаллы разрезают на тонкие пластины и полируют их до тех пор, пока их поверхности не станут зеркально гладкими и свободными от дефектов. Далее последовательно циклически повторяясь производят термическое оксидирование (формирование пленки SiO2), фотолитографию, диффузию примеси (фосфор), эпитаксию (наращивание слоя).

В процессе изготовления микросхем на пластины-заготовки наносят в виде тщательно рассчитанных рисунков тончайшие слои материалов. На одной пластине помещается до нескольких сотен микропроцессоров, для изготовления которых требуется совершить более 300 операций. Весь процесс производства процессоров можно разделить на несколько этапов: выращивание диоксида кремния и создание проводящих областей, тестирование, изготовление корпуса и доставка.

Процесс производства микропроцессора начинается с "выращивания" на поверхности отполированной пластины изоляционного слоя диоксида кремния. Осуществляется этот этап в электрической печи при очень высокой температуре. Толщина оксидного слоя зависит от температуры и времени, которое пластина проводит в печи.

Затем следует фотолитография - процесс, в ходе которого на поверхности пластины формируется рисунок-схема. Сначала на пластину наносят временный слой светочувствительного материала – фоторезист, на который с помощью ультрафиолетового излучения проецируют изображение прозрачных участков шаблона, или фотомаски. Маски изготавливают при проектировании процессора и используют для формирования рисунков схем в каждом слое процессора. Под воздействием излучения засвеченные участки фотослоя становятся растворимыми, и их удаляют с помощью растворителя (плавиковая кислота), открывая находящийся под ними диоксид кремния.

Открытый диоксид кремния удаляют с помощью процесса, который называется "травлением". Затем убирают оставшийся фотослой, в результате чего на полупроводниковой пластине остается рисунок из диоксида кремния. В результате ряда дополнительных операций фотолитографии и травления на пластину наносят также поликристаллический кремний, обладающий свойствами проводника. В ходе следующей операции, называемой "легированием", открытые участки кремниевой пластины бомбардируют ионами различных химических элементов, которые формируют в кремнии отрицательные и положительные заряды, изменяющие электрическую проводимость этих участков.

Наложение новых слоев с последующим травлением схемы осуществляется несколько раз, при этом для межслойных соединений в слоях оставляются "окна", которые заполняют металлом, формируя электрические соединения между слоями. В своем 0.13-микронном технологическом процессе корпорация Intel применила медные проводники. В 0.18-микронном производственном процессе и процессах предыдущих поколений Intel применяла алюминий. И медь, и алюминий - отличные проводники электричества. При использовании 0,18-мкм техпроцесса использовалось 6 слоев, при

внедрении 90 нм техпроцесса в 2004 году применили 7 слоев кремния.

Каждый слой процессора имеет свой собственный рисунок, в совокупности все эти слои образуют трехмерную электронную схему. Нанесение слоев повторяют 20 - 25 раз в течение нескольких недель.

Чтобы выдержать воздействия, которым подвергаются подложки в процессе нанесения слоев, кремниевые пластины изначально должны быть достаточно толстыми. Поэтому прежде чем разрезать пластину на отдельные микропроцессоры, ее толщину с помощью специальных процессов уменьшают на 33% и удаляют загрязнения с обратной стороны. Затем на обратную сторону "похудевшей" пластины наносят слой специального материала, который улучшает последующее крепление кристалла к корпусу. Кроме того, этот слой обеспечивает электрический контакт между задней поверхностью интегральной схемы и корпусом после сборки.

После этого пластины тестируют, чтобы проверить качество выполнения всех операций обработки. Чтобы определить, правильно ли работают процессоры, проверяют их отдельные компоненты. Если обнаруживаются неисправности, данные о них анализируют, чтобы понять, на каком этапе обработки возник сбой.

Затем к каждому процессору подключают электрические зонды и подают питание. Процессоры тестируются компьютером, который определяет, удовлетворяют ли характеристики изготовленных процессоров заданным требованиям.

После тестирования пластины отправляются в сборочное производство, где их разрезают на маленькие прямоугольники, каждый из которых содержит интегральную схему. Для разделения пластины используют специальную прецизионную пилу. Неработающие кристаллы отбраковываются.

Затем каждый кристалл помещают в индивидуальный корпус. Корпус защищает кристалл от внешних воздействий и обеспечивает его электрическое соединение с платой, на которую он будет впоследствии установлен. Крошечные шарики припоя, расположенные в определенных точках кристалла, припаивают к

электрическим выводам корпуса. Теперь электрические сигналы могут поступать с платы на кристалл и обратно.

В будущих процессорах компания Intel применит технологию BBUL, которая позволит создавать принципиально новые корпуса с меньшим тепловыделением и емкостью между ножками CPU.

После установки кристалла в корпус процессор снова тестируют, чтобы определить, работоспособен ли он. Неисправные процессоры отбраковывают, а исправные подвергают нагрузочным испытаниям: воздействию различных температурных и влажностных режимов, а также электростатических разрядов. После каждого нагрузочного испытания процессор тестируют для определения его функционального состояния. Затем процессоры сортируют в зависимости от их поведения при различных тактовых частотах и напряжениях питания.

Известно, что существующие КМОП-транзисторы имеют много ограничений и не позволят в ближайшем будущем поднимать частоты процессоров также безболезненно. В конце 2003 года специалисты Intel сделали очень важное заявление о разработке новых материалов для полупроводниковых транзисторов будущего. Прежде всего, речь идет о новом диэлектрике затвора транзистора с высокой диэлектрической проницаемостью (так называемый «high-k»-материал), который будет применяться взамен используемого сегодня диоксида кремния SiO_2 (рис. 81), а также о новых металлических сплавах, совместимых с новым диэлектриком затвора. Решение, предложенное исследователями, снижает ток утечки в 100 раз, что позволяет вплотную подойти к внедрению производственного процесса с проектной нормой 45 нанометров. Оно рассматривается экспертами как маленькая революция в мире микроэлектронных технологий.

Чтобы понять, о чем идет речь, взглянем сначала на обычный МОП-транзистор, на базе которого делаются МП. В нем затвор из проводящего поликремния отделен от канала транзистора тончайшим (толщиной всего 1,2 нм или 5 атомов) слоем диоксида кремния (материала, десятилетиями используемого в качестве подзатворного диэлектрика).

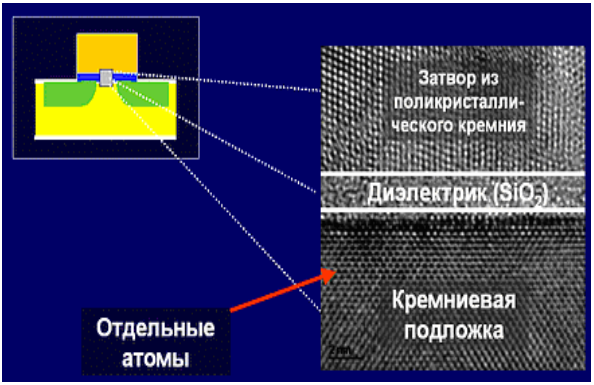


Рис. 81

Столь малая толщина диэлектрика необходима для получения не только малых габаритов транзистора в целом, но и для его высочайшего быстродействия (заряженные частицы передвигаются быстрее через затвор, в результате чего такой VT может переключаться до 10 миллиардов раз в секунду). Упрощенно - чем ближе затвор к каналу транзистора (то есть, чем тоньше диэлектрик), тем «большее влияние» в плане быстродействия он будет оказывать на электроны и дырки в канале транзистора.

Но с другой стороны, такой тонкий диэлектрик пропускает б*о*льшие паразитные токи электронов утечки из затвора в канал (идеальный МОП-транзистор должен пропускать ток от истока к стоку и не пропускать - от затвора к истоку и стоку).

И в современных высокоинтегрированных микросхемах с сотнями миллионов транзисторов на одном кристалле токи утечки затворов становятся одной из фатальных проблем, препятствующих дальнейшему наращиванию количества транзисторов на кристалле. Более того, чем меньше по размерам мы делаем транзистор, тем тоньше нужно делать подзатворный диэлектрик. Но при его толщинах менее 1 нм резко (по экспоненте) возрастают туннельные токи утечки, что делает принципиально невозможным создание традиционных транзисторов менее определенных «горизонтальных» размеров (если при этом мы хотим получить от них хорошие скоростные характеристики). По оценкам экспертов, в

современных чипах почти 40% энергии может теряться из-за утечек.

Поэтому важность открытия ученых Intel нельзя недооценивать. После пяти лет исследований в лабораториях корпорации разработали специальный материал, позволяющий заменить традиционный диоксид кремния в обычном маршруте производства микросхем. Требования к такому материалу весьма серьезны: высокая химическая и механическая (на атомарном уровне) совместимость с кремнием, удобство производства в едином цикле традиционного кремниевого техпроцесса, но главное - низкие утечки и высокая диэлектрическая проницаемость.

Если мы боремся с утечками, то толщину диэлектрика нужно повысить хотя бы до 2-3 нм (рис. 82).

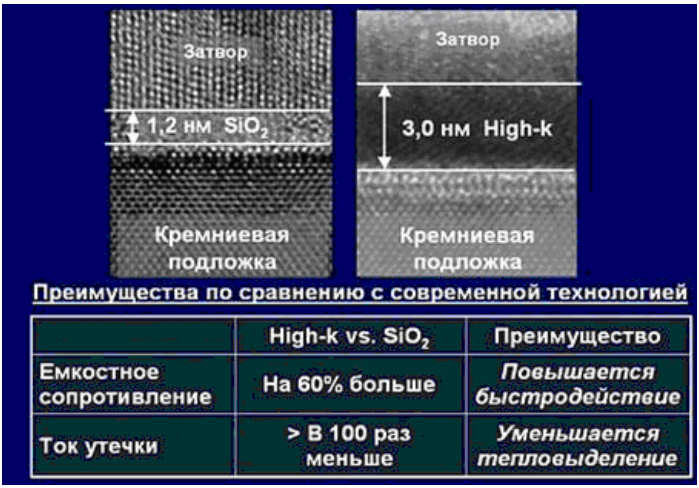


Рис. 82

Чтобы при этом сохранить прежнюю крутизну транзистора (зависимость тока от напряжения) необходимо пропорционально увеличить диэлектрическую проницаемость материала диэлектрика. Если проницаемость объемного диоксида кремния равна 4 (или чуть меньше в сверхтонких слоях), то разумной величиной диэлектрической проницаемости нового «интеловского» диэлектрика можно считать величину в районе 10-12. Несмотря на то, что материалов с такой диэлектрической проницаемостью немало (кон-

денсаторные керамики или монокристалл кремния), тут не менее важны факторы технологической совместимости материалов. Поэтому для нового high-k-материала был разработан свой высокоточный процесс нанесения, во время которого формируется один молекулярный слой этого материала за один цикл (рис. 83).

Исходя из этой картинки можно предположить, что новый материал - это тоже оксид. Причем монооксид, что означает применение материалов преимущественно второй группы, например, магния, цинка или даже меди.

Но диэлектриком дело не ограничилось. Потребовалось сменить и материал самого затвора - привычный поликристаллического кремния. Дело в том, что замена диоксида кремния на high-k-диэлектрик ведет к проблемам взаимодействия с поликристаллическим кремнием (ширина запрещенной зоны транзистора определяет минимально возможные для него апряжения).

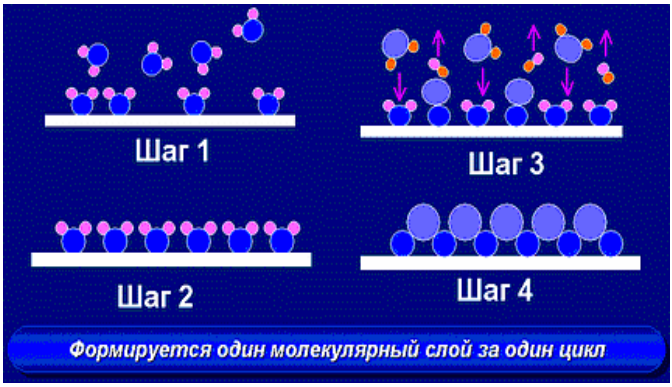


Рис. 83

Эти проблемы удается устранить, если использовать специальные металлы для затворов транзисторов обоих типов (n-МОП и p-МОП) в сочетании с особым технологическим процессом. Благодаря этой комбинации материалов удается достичь рекордной производительности транзисторов и уникально низких токов утечки, в 100 раз меньших, чем при использовании нынешних материалов (рис. 84).

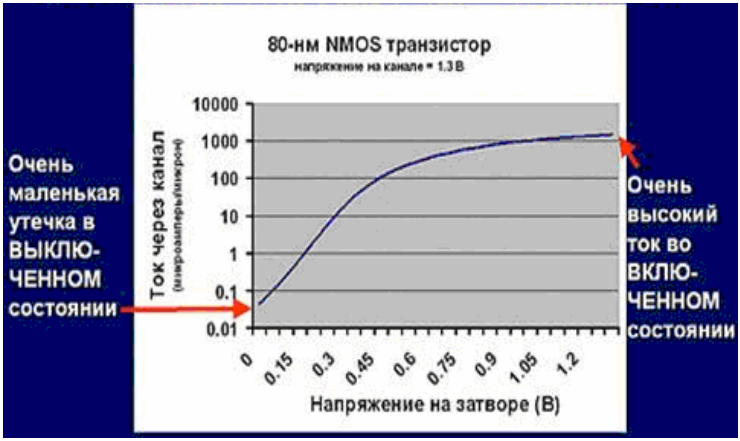


Рис. 84

В этом случае уже не возникает искушения использовать для борьбы с утечками значительно более дорогую технологию SOI (кремний на изоляторе), как это делают некоторые крупные производители микропроцессоров.

Отметим также еще одно технологическое новшество Intel - технологию напряженного (strained) кремния, которая впервые используется в 90-нанометровых процессорах Prescott и Dothan. Наконец-то, компания Intel в подробностях рассказала, каким именно образом происходит формирование слоев напряженного кремния в ее КМОП-структурах. КМОП-ячейка состоит из двух транзисторов - n-МОП и p-МОП (рис. 85).

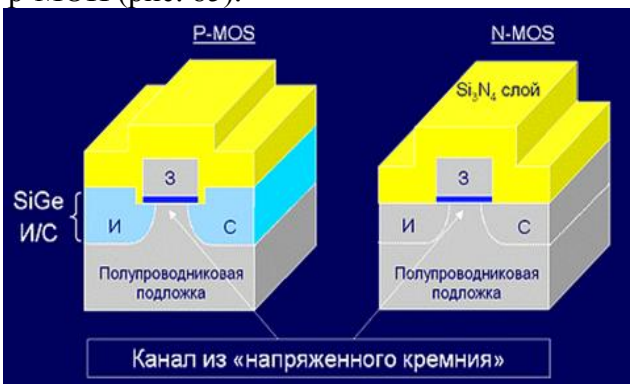


Рис. 85

В первом (n-MOS) канал транзистора (n-канал) проводит ток при помощи электронов (отрицательно заряженных частиц), а во втором (p-MOS) - при помощи дырок (условно положительно заряженных частиц). Соответственно, и механизмы формирования напряженного кремния у этих двух случаев различны. Для n-MOS-транзистора используется внешнее покрытие слоем нитрида кремния (Si_3N_4), который за счет механических напряжений немного (на доли процента) растягивает (в направлении протекания тока) кристаллическую решетку кремния под затвором, в результате чего рабочий ток канала возрастает на 10% (условно говоря, электронам становится более просторно двигаться в направлении канала). В p-MOS-транзисторах все наоборот: в качестве материала подложки (точнее - только областей стока и истока) используется соединение кремния с германием (SiGe), что немного сжимает кристаллическую решетку кремния под затвором в направлении канала. Поэтому дыркам становится «легче» «передвигаться» сквозь акцепторные атомы примеси, и рабочий ток канала возрастает на 25%. Сочетание же обеих технологий дает 20-30-процентное усиление тока.

Таким образом, применение технологии «напряженного кремния» в обоих типах устройств (n-MOS и p-MOS) приводит к значительному повышению производительности транзисторов при повышении себестоимости их производства всего лишь на ~2% и позволяет создавать более миниатюрные транзисторы следующих поколений. В планах Intel - использовать напряженный кремний для всех будущих техпроцессов вплоть до 22-нанометрового.

Материал с низкой диэлектрической проницаемостью используется в качестве диэлектрика медных соединений (рис. 86) во всех техпроцессах Intel, начиная с 0,13-микронного. Он уменьшает величину паразитной емкости, которая возникает между медными соединениями на кристалле, что повышает скорость передачи внутренних сигналов и уменьшает энергопотребление. Intel - первая и пока единственная компания, которая использует этот low-k-материал для изоляции межсоединений.

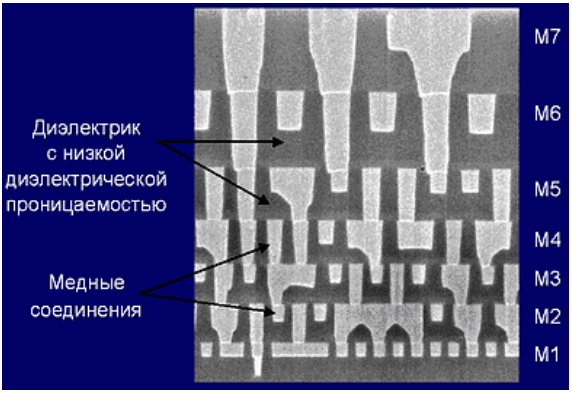


Рис. 86

Приложение 1. История развития компьютеров – период до появления первого ПК

• 1617 год. Джон Непер (John Napier) создал деревянную машину для выполнения простейших вычислений.

• 1642 год. Блез Паскаль (Blaise Pascal) описал машину для суммирования чисел.

• 1822 год. Чарльз Баббадж (Charles Babbage) представил механическое устройство, названное позднее аналитической машиной, которую можно считать первой настоящей вычислительной машинной.

• 1906 год. Ли Ди Форест (Lee DeForest) запатентовал вакуумный триод, использовавшийся в качестве переключателя в первых электронных компьютерах.

• 1945 год. Джон фон Нейман (John Von Newmann) написал статью First Draft of a Report on the EDVAC, в которой была описана архитектура современных программируемых компьютеров.

• 1946 год. Джоном Мошли (John Mauchly) и Дж. Преспером Экертом (J. Presper Eckert) создана электронно-вычислительная машина ENIAC.

• 1947год. 23 декабря Джоном Бардином (John Bardeen), Уолтером Браттейном (Walter Brattian) и Уильямом Шокли (William Shockley) был успешно протестирован первый транзистор, совершивший переворот в полупроводниковой технике.

• 1949 год. В Кембриджском университете Морис Вилке (Maurice Wilkes) создал первый практический программируемый компьютер EDSAC.

• 1950 год. Исследовательская организация в Миннеаполисе представила первый коммерческий компьютер ERA 1101.

• 1952 год. В U.S. Census Bereau был установлен компьютер UNIVAC I.

• 1953 год. Фирма IBM создала первый электронный компьютер 701.

• 1954 год. Впервые появился в продаже полупроводниковый транзистор стоимостью 2.5 доллара, созданный Гордоном Тилом (Gordon Teal) в фирме Texas Instruments, Inc.

• 1954 год. Фирма IBM выпустила первый массовый калькулятор 650; в течение этого же года было продано 450 экземпляров данной модели.

• 1955 год. Фирма Bell Laboratories анонсировала первый транзисторный компьютер TRADIC

• 1956 год. В Массачусетском технологическом институте создан первый многоцелевой транзисторный программируемый компьютер TX-0.

• 1956 год. С появлением модели IBM 305 RAMAC начинается эра устройств магнитного хранения данных.

• 1958 год. Джек Килби (Jack Kilby), сотрудник фирмы Texas Instruments, создает первую интегральную схему, состоящую из транзисторов и конденсаторов на одной полупроводниковой пластине.

• 1959 год. Фирма IBM создает серию мэйнфреймов 7000 - первых транзисторных компьютеров для крупных компаний.

• 1959 год. Роберт Нойс (Robert Noyce) - компании Fairchild Camera и Instruments Corp. - создает интегральную схему с помощью расположения соединительных каналов непосредственно на кремниевой пластине.

• 1960 год. В фирме DEC создан первый миникомпьютер PDP-1, стоимостью 120 тыс. долларов.

• 1961 год. По данным журнала Datamation, продукция фирмы IBM занимала 81,2% компьютерного рынка; в этом году IBM анонсирована серию систем 1400.

• 1964 год. Суперкомпьютер CDC 6600, созданный Сеймуром Креем (Seymour Cray) выполнял около 3 млн инструкций в секунду, что в три раза больше, чем у его ближайшего конкурента IBM Stretch.

• 1964 год. Фирма IBM анонсировала семейство компьютеров System/360 (шесть совместимых модификаций и 40 периферийных устройств).

• 1964 год. Впервые в мире была проведена транзакция в реальном времени на системе IBM SABRE.

• 1965 год. Фирма Digital Equipment Corporation анонсировала первый успешный коммерческий проект мини-компьютера PDP-8.

• 1966 год. Фирма Hewlett Packard представила компьютер для бизнеса HP-2115, который по производительности не уступал большим корпоративным системам.

• 1970 год. Впервые в мире осуществлена связь между двумя компьютерами; первые четыре узла сети ARPAnet - университет Калифорнии, UCLA, SRI International и университет штата Юта.

• 1971 год. В лаборатории фирмы IBM в Сан-Хосе создана 8-дюймовая дискета.

• 1971 год. В журнале Electronic News впервые появилась реклама микропроцессоров Intel 4004

• 1971 год. В журнале Scientific American впервые появилась реклама одного из первых персональных компьютеров Kenback-1 стоимостью 750 долларов.

• 1972 год. Фирма Hewlett Packard представила систему HP-35 с постоянной памятью.

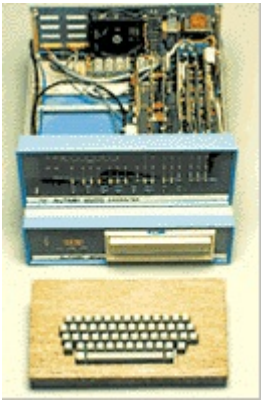
• 1972 год. Дебют микропроцессора Intel 8008.

• 1973 год. Роберт Меткалф (Robert Metcalfe) описал метод сетевого соединения Ethernet в исследовательском центре Пало Альто фирмы Хегох.

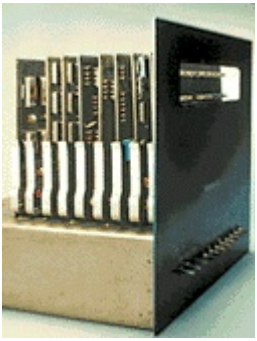
• 1973 год. Фирма Micral выпустила первый коммерческий персональный компьютер на основе микропроцессора Intel 8008.

• 1973 год. Дон Ланкастер (Don Lancaster) создал на основе телевизионного приемника первый буквенно-цифровой монитор TV Typewriter.

• 1974 год. Компания Micro Instrumentation Telemetry Systems (MITS), занимающаяся электроникой в городе Альбукерке (шт. Нью-Мексико) объявила о разработке небольшого компьютера для индивидуального пользования. Эд Робертс и двое его партнеров создали небольшой сборный компьютер. Он получил название Altair.



• 1974 год. Scelbi (SCientific ELectornic and Biological) Computer Consulting представила машину на базе процессора Intel — 8008. Она имела 1 кбайт программируемой памяти и была предназначена в основном для научного применения.



• 1974 год. В исследовательском центре Пало Альто фирмы Xerox создана рабочая станция, в качестве устройства ввода которой использовалась мышь.

• 1975 год. Появилась первая коммерческая сеть с пакетной коммутацией Telnet - гражданский аналог сети ARPAnet.

• 1975 год. В январском выпуске журнала Popular Electronics описан компьютер Altair - 8800, созданный на базе процессора Intel 8080.

• 1976 год. Стив Возняк (Steve Wozniak) создал одноплатный компьютер Apple I

• 1976 год. Фирмой Shugart Associates анонсирован первый 5,25-дюймовый гибкий диск и дисковод.

• 1976 год. Создан первый коммерческий векторный процессор Cray I

• 1977 год. Фирма Tandy Radio Snack выпустила компьютер TSR-80.

• 1977 год. Создан компьютер Apple II.

• 1977 год. Фирма Commodore выпустила компьютер PET (Personal Electronic Transactor).

• 1978 год. Фирмой Digital Equipment Corporation создан компьютер VAX 11/780, способный адресовать 4,3 Гбайт виртуальной памяти.

• 1979 год. Фирма Motorola выпустила микропроцессор 68000.

• 1980 год. Джон Шох (John Shoch) из исследовательского центра Пало Альто фирмы Хегох обнаружил первого компьютерного "червя" - небольшую программу, которая распространялась в сети в поиске свободных процессоров.

• 1980 год. Seagate Technologies выпустила первый жесткий диск для микрокомпьютеров.

• 1980 год. Разработан первый оптический диск, емкость которого в 60 раз превышала емкость 5,25-дюймового гибкого диска.

• 1981 год. Адам Осборн (Adam Osborne) выпустил первый портативный компьютер Osborne I стоимостью 1 795 долларов.

• 1981 год. Фирма IBM выпустила свой первый персональный компьютер PC.

• 1981 год. Фирма Sony анонсировала первую 3,5-дюймовую дискету и дисковод.

• 1983 год. Фирма Apple выпустила компьютер Lisa с первым графическим интерфейсом пользователя.

• 1983 год. Фирма Compaq Computer Corp. выпустила первый клон компьютера IBM PC.

• 1984 год. Фирма Apple стала выпускать первый самый успешный компьютер с графическим интерфейсом пользователя, который принес 1,5 млн долларов только за этот год.

• 1984 год. Фирма IBM создала компьютер PC-AT на базе процессора Intel 286.

• 1985 год. Выпущен первый музыкальный компакт-диск и накопитель CD-ROM.

• 1986 год. Фирма Compaq выпустила компьютер Deskpro 386, в котором впервые был установлен процессор Intel 386.

• 1987 год. Фирма IBM приступила к производству компьютеров семейства PS/2, в которых был установлен 3,5-дюймовый дисковод и VGA-видеоадаптер.

• 1988 год. Один из основателей Apple Стив Джобе (Steve Jobs) покидает эту фирму и создает собственную компанию NeXT.

• 1988 год. Фирма Compaq и другие производители PC - совместимых систем разработали новую, улучшенную архитектуру компьютера.

• 1988 год. Роберт Моррис (Robert Morris) создает и запускает своего "червя" в сеть ARPAnet; заражено по различным оценкам от 6 000 до 60 000 узлов.

• 1989 год. Фирма Intel выпускает процессор 486, который содержит 1 млн транзисторов.

• 1990 год. В Женеве в исследовательском центре CERN разработан язык разметки гипертекста (Hypertext Markup Language - HTML) и на свет появилась World Wide Web (WWW).

• 1993 год. Фирма Intel выпустила первый процессор Pentium из семейства P5. Кроме выпуска процессора, Intel разработала для него набор микросхем системной логики.

• 1995 год. Фирма Intel начала продавать процессор Pentium Pro - первого представителя семейства P6.

• 1995 год. Компания Microsoft представила первую 32-разрядную операционную системуWindows 95.

• 1997 год. Фирма Intel выпустила процессор Pentium II, построенный на базе Pentium Pro с поддержкой инструкции MMX.

• 1998 год. Компания Microsoft анонсировала новую версию своей операционной системы Windows 98.

• 1998 год. Фирма Intel представляет процессор Celeron - более дешевую версию Pentium II.

• 1999 год. Фирма Intel выпустила процессор Pentium III, построенный на базе Pentium II с поддержкой инструкции SSE (Streaming SIMD Extensions).

• 2000 год. Компания Microsoft выпустила операционную системуWindows 2000.

• 2000 год. Фирмы Intel и AMD объявили о выпуске процессоров с тактовой частотой 1 ГГц.

• 2000 год. Фирма Intel анонсирует процессор Itanium - первый процессор семейства P7.

Приложение 2. История развития МП Intel

"Intel - это процессоры"- Энди Гроув (Andy Grove). До начала 70-х годов вычислительные машины были доступны весьма ограниченному кругу специалистов, а их применение, как правило, оставалось окутанным завесой секретности и мало известным широкой публике. Однако в 1971 г. произошло событие, которое в корне изменило ситуацию и с фантастической скоростью превратило компьютер в повседневный рабочий инструмент десятков миллионов людей. В том вне всякого сомнения знаменательном году еще почти никому не известная фирма Intel из небольшого американского городка с красивым названием Санта-Клара (шт. Калифорния) создала новый полупроводниковый прибор, получивший название "МИКРОПРОЦЕССОР".

Сегодня, оглядываясь на 25 лет назад, мы не очень погрешили бы против истины, добавив к этому слову титул "Его Величество", - столь велико оказалось влияние крошечного полупроводникового кристалла практически на все сферы деятельности.Именно ему мы обязаны появлением нового класса вычислительных систем - персональных компьютеров, которыми теперь пользуются, по существу, все: от учащихся начальных классов и бухгалтеров до маститых ученых и инженеров. Этим машинам, не занимающим и половины поверхности обычного письменного стола, покоряются все новые и новые классы задач, которые ранее были доступны (а по экономическим соображениям часто и недос-

тупны - слишком дорого тогда стоило машинное время мэйнфреймов и мини-ЭВМ) лишь системам, занимавшим не одну сотню квадратных метров. Наверное, никогда прежде человек не имел в своих руках инструмента, обладающего столь колоссальной мощью при столь микроскопических размерах.

Начало В 1968 г. Гордон Мур (Gordon Moore) и Боб Нойс (Bob Noyce), одни из тех, кто закладывал фундамент известной полупроводниковой компании Fairchild Semiconductor, основали фирму Intel Corporation. Первой идеей нового предприятия было создание полупроводниковых запоминающих устройств, призванных заменить ЗУ на магнитных сердечниках. Поскольку к концу 60-х годов память этого типа практически исчерпала весь свой потенциал развития, проблема была весьма актуальной, а ее разработка сулила немалые прибыли. И хотя в данной области Intel добилась заметных успехов, тем не менее мировую славу ей принесли совсем другие изделия. Поворотным моментом в истории компании стал 1969 г., когда был получен заказ на создание ряда специализированных микросхем для калькуляторов от ныне уже несуществующей японской фирмы Busicom. В апреле того же года в штаб- квартиру Intel прибыли три инженера из Busicom, среди которых был идеолог нового проекта Масатоси Шима (Masatoshi Shima).

Через несколько лет этому человеку суждено будет сыграть одну из главных ролей в создании кристалла i8080, во многом предопределившего дальнейший путь развития корпорации Intel. Выполнение заказа японской компании поручили Марциану Хоффу (Marcian Hoff), в то время ведущему специалисту Intel. Проанализировав техническое задание (ТЗ), а также предложенный Шима вариант архитектуры и системы команд, Хофф пришел к выводу, что поставленную задачу можно решить и более простыми средствами, оптимизировав систему команд и дополнив схему устройства модулем памяти. Окончательная версия ТЗ была им разработана совместно с другим инженером компании, Стэном Мэзором (Stan Mazor), и предусматривала функционирование процессора на тактовой частоте 1 МГц. Любопытно, что на такой же частоте работал и компьютер модели 1620 фирмы IBM, но при этом ориентировочная стоимость изделия Intel составляла 30 - 40 дол., а одна

только арендная плата за пользование IBM 1620 доходила до 2000дол. в месяц. Темп работы над прибором был настолько высок, что уже осенью 1969 г. заказчику представили проект кристалла. Поскольку вариант, предложенный Intel, оказался более универсальным и имел потенциал для применения не только в калькуляторах, Busicom отдала предпочтение американской разработке. Для японского менеджмента того времени подобное решение было беспрецедентным, особенно если принять во внимание молодость компании из Соединенных Штатов - в 1969 г. фирме Intel исполнился всего лишь год.

Воплотить идеи Хоффа в кремнии выпало Федерико Фэггину (Federico Faggin), который и осуществил это менее чем за год. Кстати, спустя всего лишь несколько лет он станет одним из основателей и президентом фирмы Zilog, которая получит широкую известность благодаря микропроцессору Z80, и поныне выпускаемому рядом изготовителей. Вместе с тем, несмотря на успешный ход работ, одно обстоятельство беспокоило многих проектировщиков прибора. Согласно контракту, Busicom имела на кристалл исключительные права, но специалисты, его создавшие, прекрасно сознавали, что возможностей чипа вполне достаточно для гораздо более широкого круга применений, нежели только калькуляторы. Поэтому, когда в начале 1971 г. Busicom была вынуждена вступить с Intel в переговоры о снижении цен на поставляемые микросхемы (в связи с обострением конкуренции на рынке калькуляторов), Хофф рекомендовал сотрудникам отдела маркетинга согласиться на это в обмен на право продажи кристалла на открытом рынке. Японская сторона приняла предложение Intel.

Сразу после этого Эд Гелбах (Ed Gelbach), специалист по маркетингу компании, и его помощник Хэнк Смит (Hank Smith) приступили к исследованию потенциального рынка сбыта новых микросхем. Первые полученные результаты вполне обнадеживали. В ходе опроса потенциальных потребителей удалось установить, что стоимость - единственный фактор, удерживающий проектировщиков от программирования логических функций в их оборудовании. А на вопрос: "Будете ли вы делать это, если цена составит 5 дол.?" - был получен столь же краткий, сколь и категоричный ответ: "Безусловно". Резюме маркетинговому исследованию

подвел Боб Нойс: "Все, что нам надо - это довести стоимость комплекта интегральных схем до 30 или 40 дол., и тогда покупатели начнут писать собственные программы. Следует создать спрос и обеспечить приемлемую цену". В процессе исследования рынка стал очевиден еще один немаловажный фактор - потенциальным потребителям кристаллов необходимо оказать помощь в их применении.

Это натолкнуло команду Гелбаха на идею создания систем проектирования микроЭВМ, содержащих по крайней мере простейшие инструментальные средства разработки и отладки программного обеспечения. Идея оказалась настолько плодотворной, что уже через несколько лет поставки подобных систем стали приносить доход, соизмеримый с доходом от продажи самих микропроцессоров. В результате кристалл вышел на рынок в сопровождении соответствующих средств поддержки и началось триумфальное шествие микропроцессоров по всему миру. 15 ноября 1971 г. можно считать началом новой эры в электронике. В этот день компания приступила к поставкам первого в мире микропроцессора Intel 4004 - именно такое обозначение получил первый прибор, послуживший отправной точкой абсолютно новому классу полупроводниковых устройств.

Кристалл представлял собой 4-разрядный процессор с классической архитектурой ЭВМ гарвардского типа и изготавливался по передовой в те годы р-канальной МОП-технологии с проектными нормами 10 мкм. Электрическая схема прибора насчитывала 2300 транзисторов. Микропроцессор работал на тактовой частоте 750 кГц при длительности цикла команды 10,8 мкс. Чип i4004 имел адресный стек (счетчик команд и три регистра стека типа LIFO - Last In First Out), блок регистров общего назначения - РОН (регистры сверхоперативной памяти, или регистровый файл), 4-разрядное параллельное АЛУ, аккумулятор, регистр команд с дешифратором команд и схемой управления, а также схему связи с периферийными устройствами. Все эти функциональные узлы объединялись между собой 4-разрядной шиной данных (рис. 1). Для однокристального процессора i4004 имел весьма впечатляющие характеристики. Память команд достигала 4Кбайт (для сравнения: объем ЗУ мини-ЭВМ в начале 70-х годов редко превышал

16 Кбайт), а регистровый файл ЦП насчитывал шестнадцать 4-разрядных регистров, которые можно было использовать и как восемь 8-разрядных (восемь 4-разрядных пар). Такая организация РОНов сохранена и в последующих микропроцессорах фирмы Intel. Три регистра стека обеспечивали три уровня вложения подпрограмм. Конечно, эта цифра не вызвала особых восторгов у программистов, тем не менее они получили возможность создавать полноценные программы. Процессор i4004 монтировался в пластмассовый или металлокерамический корпус типа DIP (Dual In-line Package) всего с 16 выводами.

В систему его команд входило 46 инструкций. По своему функциональному составу она была универсальной, т. е. рассчитана на широкий круг решаемых задач и разрабатываемых приложений. Первоначальное назначение кристалла наложило определенный отпечаток на состав системы команд, поэтому присутствие в ней ряда инструкций, в частности десятичной коррекции, а также наличие соответствующих аппаратных средств не вызывает особого удивления.

Вместе с тем кристалл располагал весьма ограниченными средствами ввода/вывода, а в системе команд отсутствовали операции логической обработки данных (И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ), в связи с чем их приходилось реализовывать с помощью специальных подпрограмм, что в некоторых случаях чрезмерно усложняло создаваемое ПО. Модуль i4004 не имел возможности останова (команды HALT) и обработки прерываний. Впрочем, это объясняется не упущением создателей устройства, а тем, что в калькуляторах, где поначалу и планировалось использовать прибор, особой необходимости в этих средствах нет. Цикл команды процессора состоял из восьми тактов задающего генератора. Такое их количество вызывает удивление, но объясняется очень просто, хотя с позиций сегодняшнего дня и несколько неожиданно. Как уже отмечалось, чип i4004 монтировался в корпус всего с 16 выводами - самый распространенный (а значит, и самый дешевый) тип корпуса в начале 70-х годов. А поскольку в распоряжении инженеров оказался узкий интерфейс с "внешним миром", то пришлось пойти на применение мультиплексированной шины адреса и данных, причем 12- разрядный адрес выдавать порциями по четыре

разряда, что, конечно, не могло не сказаться на длительности машинного цикла. Прием команды по такому интерфейсу требовал еще двух тактов. На исполнение же самой инструкции из восьми тактов процессор затрачивал лишь три.

Таким образом, соотношение "накладные расходы/полезная работа" составило 5:3 в пользу накладных расходов. Узкое "окно" во внешний мир долгое время было бичом всех микропроцессоров без исключения. Забегая несколько вперед, можно сказать, что 40-выводной корпус во многом решил проблемы 8-разрядных систем, но уже первые 16-разрядные приборы опять поставили на повестку дня этот больной вопрос. Задача создания многовыводных корпусов оказалась крепким орешком и попортила немало крови конструкторам и технологам, не говоря уж о самих разработчиках процессоров, которых к тому времени число "40", наверное, приводило просто в бешенство.

Примерно к середине 70-х годов появились корпуса типов DIP и QUIP (QUad In-line Package), имевшие до 64 выводов, но и им по ряду причин не суждено было стать панацеей от всех бед. Во-первых, рост степени интеграции и тактовых частот БИС не могли не привести к увеличению потребляемой мощности. Пластмассовые корпуса позволяли рассеивать мощность не более 1,5 - 2 Вт. Металлокерамика увеличивала этот показатель до 3 - 4 Вт, но одновременно поднимала стоимость микросхемы на такую высоту, что ее массовый выпуск сразу оказывался под вопросом. Во-вторых, 64 вывода "спасали" разработчиков процессоров на крайне ограниченном временном отрезке. Как бы там ни было, но спрос рождает предложение и проблема корпусов с большим числом выводов и приемлемой ценой была решена в начале 80-х годов. Однако все это было впереди, а в самом начале 70-х пионерам пришлось исходить из существовавших реалий и принимать соответствующие технические решения. Говоря о первом в мире микропроцессоре, нельзя не вспомнить и о том, что с самого начала специалисты позаботились о простоте и удобстве построения систем на базе i4004. Компанией был разработан и выпущен не один кристалл центрального процессора, а целое семейство БИС, в которое вошли ПЗУ 4001, ОЗУ 4002, регистр сдвига 4003 и ряд других вспомогательных микросхем. Поскольку все они были рассчитаны

на совместное использование, разработка аппаратных средств системы заметно упрощалась, и это стало не последней причиной популярности i4004. i4004 + обработка прерываний + ... = i4040.

Опыт использования первого МП показал, что такие факторы, как отсутствие средств обработки прерываний, наличие трех уровней вложения подпрограмм и необходимость реализации логических операций И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ с помощью специальных подпрограмм, далеко не всегда удовлетворяют разработчиков. Создав новый рынок и захватив на нем господствующие высоты, Intel тем не менее стремилась расширить его границы, и решение о выпуске усовершенствованного варианта i4004, свободного от указанных недостатков, было вполне логичным ходом компании. Использование же в новом кристалле всех наработок, приобретенных при создании предыдущего процессора, дало существенную экономию времени, и неудивительно, что i4040 вышел в свет вскоре после появления i4004.

В новом приборе сохранены все функциональные возможности предшествующей модели и существенно улучшены как технические, так и программные средства. Система команд пополнилась 14 инструкциями, включая выполнение логических операций И и ИЛИ; кроме того, в процессор были введены средства останова и обработки прерываний. Претерпела некоторые изменения и архитектура устройства. Адресный стек процессора увеличен с трех до семи регистров, а количество РОНов возросло с 16 до 24, причем их разбили на две области, выбираемые при помощи специальных команд. Отчасти такая организация обусловлена тем, что процессор теперь мог обращаться к двум блокам памяти команд объемом 4 Кбайт и за каждым из них программист мог закрепить свою область регистров. Наряду с этим восемь РОНов были всегда доступны для использования. В итоге получилась достаточно гибкая и удобная структура, позволявшая разрабатывать самостоятельные программные модули, способные взаимодействовать через общую часть регистрового файла. Обработка одноуровневых прерываний - одно из наиболее существенных новшеств чипа i4040 - превратила его в полноценный процессор и сделала возможным использование в системах реального масштаба времени. Благодаря применению сигнала "останов" стала реальностью

синхронизация работы процессора с некоторыми внешними событиями.

Вместе с тем специальных операций со стеком (запись в стек содержимого РОН и извлечение его из стека) разработчики ПО так и не дождались. Несмотря на то что тактовая частота и машинный цикл i4040 не претерпели изменений, производительность процессора возросла за счет использования более совершенной архитектуры и эффективной системы команд. 60 инструкций, ориентированных на широкий спектр решаемых задач, обработка прерываний, до 8 Кбайт памяти команд, а также возможность быстрого перевода систем на базе i4004 на новый процессор вывели i4040 в безусловные лидеры рынка 4-разрядных устройств. Правда, полной совместимости с программным обеспечением, разработанным под i4004, получить не удалось, и некоторые программы пришлось переписывать.

Безусловно, необходимость доработки ПО при переводе систем с ЦП i4004 на новый процессор не вызвала особой радости у специалистов, но и не разочаровала их настолько, чтобы они отвергли новое детище Intel. Преимущества, которые i4040 имел перед своим предшественником, с лихвой компенсировали отмеченный недостаток. Кроме того, в то время проблема переносимости ПО еще не успела стать актуальной: создатели процессора i4040 не ощущали того давления со стороны потребителей своей продукции, которое они начнут испытывать спустя всего 5 - 7 лет, а потому могли себе позволить такую роскошь, как частичная несовместимость моделей процессоров в рамках одного семейства. Принимая во внимание особенности архитектуры и технические характеристики кристалла, i4040 можно считать первым прибором второго поколения 4-разрядных систем. Вот только век этого поколения оказался чрезвычайно коротким - новым процессором фирма Intel закрыла линию 4-разрядных устройств.

Следующая высота - 8 разрядов Выпустив на рынок первый микропроцессор, Intel не стала изменять привычкам лидера и 1 апреля 1972 г. начала поставки первого в отрасли 8-разрядного прибора Intel 8008. Создание этого чипа весьма напоминает историю с i4004. Как и 4-разрядный микропроцессор, он был разработан для нужд другой компании, только в этом случае заказчиком выступа-

ла американская фирма Computer Terminals Corporation of Texas, позднее известная как Datapoint. Кристалл проектировали четыре человека, трое из которых (Хофф, Фэггин и Мэзор) уже имели опыт создания МП i4004. Четвертым членом команды стал новый сотрудник компании Хэл Фини (Hal Feeney). Проектирование i8008 шло практически параллельно с работами над i4004. Когда же процессор был готов, заказчик от него отказался, мотивируя это тем, что прибор получился слишком медленным для решения поставленных перед ним задач, а также требовал для своей работы большого количества вспомогательных микросхем. Тогда Intel решила пустить i8008 в свободную продажу и в очередной раз не ошиблась. Кристалл изготавливался по p-канальной МОП-технологии с проектными нормами 10 мкм и содержал 3500 транзисторов. Процессор работал на частоте 500кГц при длительности машинного цикла 20 мкс (10 периодов задающего генератора).

В отличие от своих предшественников новый МП имел архитектуру ЭВМ принстонского типа, а в качестве памяти допускал применение комбинации ПЗУ и ОЗУ. Помимо увеличения разрядности и перехода на использование общего поля памяти для команд и данных, структура процессора претерпела еще ряд существенных изменений. Прежде всего это коснулось регистрового файла и устройства управления. По сравнению с i4004 число РОНов уменьшилось вдвое (с 16 до 8), причем два регистра в основном использовались для хранения адреса при косвенной адресации памяти. В связи с этим следовало бы ожидать снижения производительности, которого на самом деле не произошло, поскольку операции с памятью i8008 выполнял быстрее предыдущих моделей благодаря меньшему количеству состояний в машинном цикле и отсутствию необходимости исполнения минимум трех подготовительных команд (как в i4004 и i4040) при обращении к ОЗУ или ПЗУ.

Вместе с тем, объем блока регистров был ограничен и возможностями технологии, которая в то время еще не позволяла размещать на кристалле большие регистровые структуры (аналогично кристаллам i4004 и i4040 в МП i8008 блок РОНов был реализован в виде динамической памяти, необходимость регенерации

которой влечет за собой применение ряда дополнительных аппаратурных средств).

Почти вдвое (с восьми до пяти состояний) сократилась длительность машинного цикла. Теперь процессор выполнял команды за один - три машинных цикла, а некоторые инструкции - за один цикл из трех состояний. Для синхронизации работы ЦП с медленными устройствами был введен сигнал готовности (READY). Разработчики технических средств на базе i8008 не были ограничены жесткими требованиями в отношении быстродействия микросхем памяти и периферийных устройств и могли использовать те ИС, которые наиболее полно соответствовали конкретной системе. В ряде случаев это приводило к ощутимому сокращению стоимости оборудования. Такой гибкостью 4-разрядные кристаллы похвастаться не могли.

Система команд первого 8-разрядного ЦП насчитывала 65 инструкций, причем значительно увеличилось число команд условных переходов, а также логических инструкций и команд сдвига. Новый кристалл мог адресовать память объемом до 16 Кбайт (объем ЗУ для ЦП типа i4040 не превышал 8 Кбайт). Его производительность по сравнению с 4-разрядными системами возросла в 2,3 раза. Процессор с такими параметрами уже можно было рассматривать как серьезную заявку на многие очень перспективные секторы рынка, включая контрольно- испытательное оборудование, прецизионную измерительную технику и сложные промышленные контроллеры систем управления технологическими процессами.

Однако i8008 получил от своих предшественников и "тяжелое наследство". Объем и организация стека остались такими же, как и у чипа i4040, и реализация операций с ним по- прежнему возлагалась на программиста. Узкий интерфейс с "внешним миром" ограничил количество управляющих сигналов процессора: в результате специалистам Intel пришлось использовать их шифрацию, что повлекло за собой необходимость установки дополнительного внешнего оборудования для формирования сигналов управления. В среднем для сопряжения процессора с памятью и устройствами ввода/вывода требовалось около 20 схем средней степени интеграции.

Если спортсмены исповедуют древний олимпийский принцип "Citius, altius, fortius", то разработчикам он ближе в несколько измененном виде: "Citius, citius, citius". Вскоре после выхода чипа Intel 8008 появилась его усовершенствованная версия i8008-1. Модернизированный вариант работал уже на частоте 800 кГц при длительности машинного цикла 12,5 мкс.

Увеличение в 1,5 раза производительности центрального процессора, наряду с большим (по тому времени) объемом оперативной памяти, послужило лучшей рекомендацией для активного использования кристалла в различных областях, начиная от промышленности и медицины и кончая военной электроникой и торговлей. По мере расширения сферы влияния микропроцессора и усложнения систем на его базе возросли и требования к нему со стороны проектировщиков оборудования. Программное обеспечение уже с трудом вписывалось в 16 Кбайт, да и производительность прибора начинала "поджимать" многих разработчиков. Кроме того, некоторые области применения настойчиво требовали расширения не только количества, но и номенклатуры периферийных устройств. Системщики уже с трудом могли обходиться без такой традиционной для мэйнфреймов и мини-ЭВМ периферии, как дисплеи, принтеры, накопители на магнитной ленте и дисках и т. п. Стало очевидно, что технические характеристики изделия превратились в фактор, сдерживающий его дальнейшее распространение. Гордон Мур, один из основателей и первый президент Intel, в те годы писал, что микропроцессоры найдут еще более широкое применение, если улучшить их рабочие характеристики, ибо уже выявились области, где требуются предельно высокие параметры.

Возможности p-канальной МОП-технологии для создания сложных высокопроизводительных МП были уже практически исчерпаны, поэтому направление главного удара перенесли на технологию n-МОП. Перед проектировщиками стояли не менее сложные проблемы - разработка эффективной системы команд, рассчитанной на широкий круг решаемых задач, при сохранении программной совместимости с предыдущей моделью, расширение объема адресуемой памяти, поддержка интенсивного ввода/вывода без существенной потери производительности процессора, совер-

шенствование подсистемы обработки прерываний. i8080 - триумф 8-разрядных систем Работа над новым 8-разрядным процессором началась практически сразу после завершения опытно-конструкторского цикла, связанного с выпуском кристалла i8008, и некоторое время оба проекта шли практически параллельно. Костяк команды проектировщиков составили ставшие уже ветеранами "микропроцессорного фронта" Мэзор и Фэггин, а также бывший сотрудник японской фирмы Busicom Масатоси Шима, знакомый нам по кристаллу i4004.

Первоначально идея создания i8080 сводилась к повышению производительности чипа- предшественника только за счет перехода на новый технологический процесс. К этому времени Intel располагала технологией n-МОП, которая прошла обкатку на кристаллах памяти 2102. Стремясь сократить сроки проектирования, разработчики попытались использовать маски i8008 в производстве чипов ОЗУ 2102. Однако, как вспоминает Стэн Мэзор, после предварительного изучения вопроса стало ясно, что применить старые маски к новому технологическому процессу не удастся и их придется разрабатывать заново. Такой поворот событий привел к идее коренной модернизации схемы процессора и повышению его производительности примерно на порядок за счет сочетания преимуществ новой технологии и усовершенствованной архитектуры, включая расширенную систему команд. На проведение всего комплекса работ потребовалось более года, и 1 апреля 1974 г. (ровно через два года после выпуска i8008) микропроцессор Intel 8080 был представлен вниманию всех заинтересованных лиц.

Поставки начались по цене 360 дол. за кристалл. Дэйв Хаус (Dave House), один из ветеранов компании, так комментирует эту цифру: "По сути дела, прибор представлял собой настоящий компьютер, который стоил в то время тысячи долларов. Поэтому мы чувствовали, что предложили вполне разумную цену". По свидетельству Эда Гелбаха, расходы на исследования и разработки по программе i8080 окупились в течение первых пяти месяцев продаж. Новый ЦП практически по всем статьям разительно отличался от своих предшественников. Благодаря использованию технологии n-МОП с проектными нормами 6 мкм, на кристалле удалось разместить 6 тыс. транзисторов. При этом геометрические размеры

самого кристалла по сравнению с i8008 увеличились незначительно. Следовательно, процент выхода годных изделий и ряд экономических показателей производства, включая себестоимость, удалось сохранить на достаточно высоком уровне. Тактовая частота процессора была доведена до 2 МГц, что в 2,5 раза превышало аналогичный параметр для i8008, а длительность цикла команды составила уже 2 мкс.

Несмотря на чисто внешнее сходство структур i8080 и i8008, схема нового процессора существенно отличалась от предшествующей модели. К великой радости системщиков и программистов объем памяти, адресуемой процессором, был увеличен в четыре раза и достиг 64 Кбайт (кстати, в то время ОЗУ такой емкости предлагали потребитетелям минимальные конфигурации многих мини-ЭВМ). В сочетании с эффективным механизмом обработки прерываний это давало им карт-бланш для широкого применения нового МП в сложных системах сбора и обработки информации различного назначения, особенно функционирующих в реальном масштабе времени. За счет использования корпуса с 40 выводами удалось разделить адресную и информационную шины процессора, в результате отпала необходимость применения дополнительных внешних схем для разделения потоков адресов и данных. Общее же количество микросхем, требовавшихся для построения системы в минимальной конфигурации, сократилось с 20 до 6, т. е. более чем в три раза.

В регистровый файл были введены указатель стека, активно используемый при обработке прерываний, а также два программно-недоступных регистра для внутренних пересылок. Поскольку микропроцессор i8008 успешно продавался уже в течение двух лет и за этот период для него был наработан достаточно большой объем ПО, сохранение программной совместимости i8080 и i8008 было вполне естественным и разумным шагом компании (таким образом, Intel встала на тот же путь, что и корпорация IBM с компьютерами знаменитой серии System 360, оказавшись на долгие годы своего рода заложницей собственного творения). Именно поэтому в состав РОНов нового процессора были включены основные рабочие регистры предыдущей модели. Правда, полной совместимости с i8008 достичь опять не удалось, так как процедуры обраще-

ния к подпрограммам и инструкции ввода/вывода МП i8080 в значительной степени отличались от соответствующих процедур и операций кристалла i8008, и при переводе систем со старого процессора на новый в некоторых случаях программы приходилось полностью перерабатывать. Включение в систему команд ряда инструкций, адресующих память с использованием трех пар регистров (в i8008 для этого выделялась одна пара), придало дополнительную гибкость системе и существенно упростило жизнь программистам, реализация же блока РОНов на основе статической, а не динамической памяти дала дополнительную экономию площади кристалла для размещения других схем процессора. Исключение аккумулятора из регистрового файла и введение его в состав арифметико-логического устройства упростило схему управления внутренней шиной, поскольку при этом отпала необходимость в ее использовании для передачи данных между сверхоперативной памятью и АЛУ во время выполнения арифметических и логических операций.

Новым веянием в архитектуре микропроцессоров стало использование многоуровневой системы прерываний по вектору. Такое техническое решение позволило довести общее число источников прерываний в системе до 256. Правда, до появления специализированных БИС контроллеров прерываний схема формирования векторов прерываний требовала применения до десяти дополнительных чипов средней степени интеграции. В отличие от прерываний по вектору, размещение стека в оперативной памяти не было последним словом в архитектуре МП, но и здесь Intel не обошлась без "изюминки", добавив в схему микроЭВМ всего один триггер, в качестве стека можно было использовать отдельную память емкостью до 64 Кбайт, сэкономив тем самым ОЗУ для размещения программ и данных. Тот факт, что разработчики микропроцессоров воспользуются техническими решениями, которые уже нашли применение в мэйнфреймах и мини-ЭВМ, ни у кого не вызывал сомнений. Вопрос был лишь в том, что именно будет использовано и кто станет первым. Пионером в этом опять оказалась Intel.

Освобождение центрального процессора от управления внешними устройствами и обмен данными между памятью систе-

мы и периферией, минуя ЦП, были уже достаточно давно и успешно реализованы в универсальных ЭВМ (IBM System 360 и др.). Таким образом, появление в кристалле i8080 механизма прямого доступа к памяти при работе с внешними устройствами можно смело считать первым (но далеко не последним) ударом микропроцессоров по большим системам. ПДП открыл зеленую улицу для применения в микроЭВМ таких сложных устройств, как накопители на магнитных дисках и лентах, а также дисплеи на ЭЛТ, которые и превратили микроЭВМ в полноценную вычислительную систему.

Традицией компании, начиная с первого кристалла, стал выпуск не отдельного чипа ЦП, а семейства БИС, рассчитанного на совместное использование. Помимо микропроцессора, в новый набор микросхем вошли ИС системных генератора и контроллера. Вскоре их пополнили БИС контроллера ПДП и контроллера прерываний. Благодаря хорошо продуманному составу комплекта, проектирование микро-ЭВМ на его базе в ряде случаев упростилось настолько, что было подобно сборке домика из детских кубиков. Мимо такой техники пройти было трудно! Мощная система команд, высокое быстродействие, простота проектирования, прототипные комплекты и системы разработки вскоре превратили микропроцессор в стандарт де-факто. Более того, конкуренты отстали настолько, что около года Intel практически безраздельно господствовала на рынке. Однако спрос на ее кристаллы оказался чрезвычайно высок и не всегда удовлетворялся своевременно, так что для конкурентов появилась определенная ниша. Ряд полупроводниковых компаний, включая гигантов вроде Texas Instruments, купили у Intel лицензии на выпуск микропроцессоров, в результате возник институт так называемых вторых поставщиков. На первый взгляд могло показаться, что этот шаг отбирает прибыли у самой компании, на самом деле он только способствовал распространению ее продукции и укреплению позиций на рынке.

И все-таки рано или поздно любой монополии приходит конец. Высокие темпы роста рынка 8- разрядных систем не могли не привести к появлению на нем новых действующих лиц. Полупроводниковые компании, выпускавшие микропроцессоры собственной разработки, росли как грибы после дождя, и не было прак-

тически ни одной мало-мальски уважающей себя фирмы, которая не попробовала бы свои силы на этой стезе. Безусловно, все они отбирали у Intel определенную долю рынка, но самыми серьезными конкурентами стали Motorola и Zilog с кристаллами M6800 и Z80 соответственно (см. врезку "Motorola и Zilog против..."). Эти компании имели весьма серьезные намерения, и через некоторое время корпорации Intel пришлось потесниться.

Кристаллы от Motorola и Zilog обладали определенными преимуществами. Архитектура M6800 была более прозрачна для программистов, чем продукция Intel. Разработчики Z80 учли все недостатки микросхемы i8080, так как знали их лучше всех (и президент компании Федерико Фэггин, и ряд ведущих специалистов ранее работали в Intel и были не последними людьми в команде, создавшей МП i8080, что даже послужило основанием для обращения Intel в суд). Кроме того, для работы как M6800, так и Z80 требовался всего один номинал питания, а для конкурирующего изделия - три.

На вызов, брошенный фирмами Motorola и Zilog, компания Intel ответила разработкой серии периферийных контроллеров, которые существенно упростили построение сложных систем, а также ЦП i8085, уже свободного от указанных недостатков, имевшего ряд преимуществ перед конкурентами и обладавшего программной совместимостью со своим предшественником. Но, пожалуй, самым сильным ответным ходом компании стало создание системного ПО - однопользовательской ОС ISIS II и ОС реального времени iRMX-80. Мощь этого удара оказалась столь велика, что многие компании так и не смогли от него оправиться, а некоторым для восстановления былых рыночных позиций потребовался не один год и не один миллиард долларов. Intel обеспечила свои ЦП (а значит и потребителей) мощнейшей программной поддержкой, которую в то время больше никто не смог предложить. Это предрешило исход борьбы, и последствия той победы компания, по-видимому, ощущает до сих пор. Вторая половина 70-х годов была отмечена массированной атакой японских поставщиков полупроводниковых устройств на рынок США. Объемы производства американских компаний росли, а прибыли падали. Пожалуй, наибольшему давлению подвергся сектор ЗУ, которые занимали не

последнюю строку в списке приоритетов Intel. Но и к такому повороту событий фирма оказалась готовой, приступив в 1976 г. (к немалому удовольствию изготовителей законченных систем) к выпуску одноплатных микроЭВМ серии iSBC на базе своих микропроцессорных комплектов.

Операционная система iRMX-80 стала для этих машин базовым системным ПО. Таким образом, из изготовителя микросхем корпорация превратилась в производителя подсистем, обеспечив себе тем самым дополнительный стабильный рынок сбыта полупроводниковых изделий. Именно с тех пор в лексикон "кремниевых" компаний прочно вошел термин OEM (Original Equipment Manufacturer - изготовитель комплексного оборудования).

Ориентацию на перспективу можно смело отнести к характерной особенности стиля Intel Corporation. Вероятно, нельзя назвать ни одного прибора, работа над которым не проводилась бы одновременно с проектированием изделия следующего поколения (достаточно вспомнить историю создания кристалла i8008). Следуя установившейся традиции, в конце 1974 г. фирма приступила к созданию прототипа 32-разрядной системы iAPX-432, на которую возлагались большие надежды. Рост производительности планировалось получить не только благодаря увеличению разрядности, но и за счет использования сложной архитектуры с возможностью организации мультипроцессорной системы.

Следует отметить, что мультипроцессорные универсальные ЭВМ в ту пору еще не стали привычным атрибутом вычислительных центров и многие нюансы построения таких машин были известны весьма ограниченному кругу специалистов. Сыграло ли это свою роль или имелись другие обстоятельства - сегодня судить сложно, но работа над новым ЦП шла не так быстро, как хотелось. В то же время, несмотря на ряд удачных действий, давление на рынке 8- разрядных приборов со стороны главных конкурентов Motorola и Zilog хотя и ослабло, но продолжало оставаться объективной реальностью. Динамика выполнения проекта, связанного с iAPX-432, давала все основания полагать, что выпуск на рынок 32-разрядных процессоров будет задержан, а это означало потерю темпа. Вакуум, который мог образоваться в связи с проблемами 432-го кристалла, требовалось срочно заполнить. Поэтому руково-

дство компании сочло целесообразным попытаться развить успех i8080, и в самом начале 1976 г. стартовали работы по созданию 16-разрядного прибора, который впоследствии получил обозначение i8086.

Конечной целью нового проекта было получение 16-разрядного микропроцессора с производительностью, на порядок превышающей аналогичный параметр кристалла i8080. Поставленная задача решалась за счет дальнейшего совершенствования архитектурных концепций, положенных в основу его предшественника. Избранная стратегия эволюционного, а не революционного развития (как в случае iAPX-432) оказалась верной и скоро дала свои плоды. Менеджером работ был назначен Жан -Клод Корне (Jean Claude Cornet), занимавший в то время пост технического директора компании по микропроцессорам. Коллектив, ядро которого составили Боб Кохлер (Bob Koehler), Джон Бэйлис (John Bayliss), Джим Маккевит (Jim McKevitt), Чак Уайлдмэн (Chuck Wildman) и Стив Морз (Steve Morse), возглавил Билл Полмэн (Bill Pohlman).

Время и давление конкурентов были критическими факторами, и очень скоро численность команды увеличилась почти до 20 человек. По тем временам это выглядело весьма необычно. Но, пожалуй, еще любопытнее было то, что многие из них имели опыт работы не более года. Тем не менее они сумели создать сложный и исключительно удачный процессор в очень сжатые сроки. Новый кристалл был анонсирован 8 июня 1978 г. (через 2,5 года после начала выполнения проекта!). Прибор изготавливался по высококачественной трехмикронной МОП-технологии с кремниевыми затворами (H-MOS), позволившей разместить на чипе 29 тыс. транзисторов. Высокое быстродействие элементов (задержка 2 нс/вентиль) обеспечило тактовую частоту процессора 5 МГц, а 16-разрядная архитектура и 200-нс машинный цикл - производительность процессора, превыщающую аналогичный параметр i8080 на порядок величины.

Программная совместимость с i8080/i8085 была, пожалуй, единственной, но вместе с тем и исключительно важной характеристикой, которая объединяла 86-й кристалл с его предшественниками. Структура процессора оказалась полностью пересмотрен-

ной. Прежде всего, прибор был разбит на два функциональных блока (рис. 3) - операционный (Execution Unit, EU) и интерфейсный (Bus Interface Unit, BIU), которые могли работать одновременно. В результате исполнение одной команды совмещалось во времени с выборкой следующей инструкции или данных из памяти. Более того, в ЦП появился небольшой буфер команд, что давало дополнительную экономию времени при обращениях к памяти. Так, микропроцессоры позаимствовали у универсальных ЭВМ еще одно техническое решение, и именно с этого момента в них началась реализация принципов параллелизма.

Возможность адресации 1-Мбайт ОЗУ и сегментация памяти могут быть отнесены к одним из наиболее существенных новшеств, предложенных инженерами Intel. В частности, сегментация памяти и большое число уровней прерываний были ориентированы на работу систем в многозадачном режиме, весьма актуальном для приложений управления. (Правда, следует отметить, что механизм защиты памяти реализован не был, и в ряде случаев это существенно усложняло разработку ПО). Большая емкость ОЗУ позволяла перевести проекты построения сложных операционных и прикладных систем из области теории в сферу практической реализации.

Наряду с поддержкой ввода/вывода по каналу прямого доступа к памяти чип i8086 обеспечивал адресацию до 64К портов программно-управляемого ввода/вывода. Это снимало практически любые ограничения при формировании крупных систем сбора и обработки информации. Исключительно удачной находкой можно считать два режима работы процессора - минимальный и максимальный. Первый рассчитан на использование ЦП в небольших системах и предполагал работу кристалла без БИС контроллера шины. Максимальный режим был ориентирован на применение чипа в сложных крупномасштабных системах и требовал наличия указанного контроллера. Таким образом, один и тот же процессор с одинаковым успехом мог применяться в системах различного класса. Конкуренты же для этих целей выпускали разные модификации своих продуктов.

Однако, пожалуй, наиболее интересна система команд процессора. 147 инструкций позволяли решать задачи управления практически

любой сложности. Появление среди них таких операций, как умножение и деление 16-разрядных чисел со знаком и без знака, команд обработки массивов данных, а также программно-управляемых прерываний дает все основания назвать этот кристалл универсальным, рассчитанным на использование не только в сложных контроллерах, но и в качестве центрального процессора ЭВМ общего назначения.

Как и всегда, новый прибор вышел в мощном сопровождении средств поддержки: вспомогательных БИС, средств разработки и отладки аппаратуры и системного ПО, а также прототипных комплектов и одноплатных микроЭВМ серии iSBC86.

Несмотря на блестящие характеристики продукта и мощную поддержку со стороны изготовителя, потребовалось почти два года, чтобы кристалл завоевал признание разработчиков. Первое время даже его создатели стали опасаться - не слишком ли сложное детище они сотворили. Однако феномену медленного роста объема продаж нашлось разумное толкование: начальные темпы и не могли быть высокими, поскольку создаваемой системе требуется новое системное и прикладное ПО. Справедливость такого объяснения подтвердилась в начале 1980 г., когда закончился лабораторный период отработки компьютеров на базе ЦП i8086 и они вышли на стадию промышленного производства - кривая сбыта стала резко подниматься. Intel начала занимать на рынке 16-разрядных систем господствующие высоты.

Вместе с тем конкуренты тоже не сидели сложа руки. Компания Motorola извлекла уроки из битвы за сектор 8-разрядных систем и спустя примерно год после появления чипа i8086 предложила очень удачный микропроцессор M68000 (см. врезку "На рынке 16-разрядных систем..."). К концу 1979 г. его присутствие на рынке ощущалось весьма явственно. Некоторые сотрудники верхнего эшелона управления корпорации Intel пришли к выводу, что, если не сделать собственный кристалл стандартным прибором для отрасли, это может привести к потере рынка следующего поколения МП. Поэтому усилия были сосредоточены на ускорении выпуска процессора с 8-разрядной внешней шиной данных (i8088) и реализации плана агрессивного маркетинга приборов, получившего название Operation Crush. В основу кампании, которую воз-

главил Билл Дэвидоу (Bill Davidow), были положены два момента: во-первых, поставка пользователю готовых решений, включая системное ПО (ОС iRMX-86), широкий набор периферийных БИС и оперативную техническую поддержку; во- вторых, подготовка потребителей к будущим техническим решениям на базе платформы Intel. В соответствии с новой стратегией претерпел изменения и традиционный рекламный девиз компании, вполне закономерно превратившись из Intel delivers ("Intel поставляет") в Intel delivers solutions ("Intel поставляет решения").

В рамках Operation Crush от Стокгольма до Сиэтла фирма провела множество семинаров по различным аспектам практического использования микропроцессорного комплекта i8086. Просветительская деятельность сопровождалась массированной рекламной кампанией. Акция имела грандиозный успех - в течение года процессор i8086 стал базовым элементом более 2 тыс. проектов. IBM PC - новая эпоха в истории развития Intel В начале 1980 г. IBM приступила к реализации проекта создания персонального компьютера. Новая машина должна была представлять собой открытую систему и базироваться на стандартном для отрасли микропроцессоре.

Работы шли в г. Бока-Ратон (шт. Флорида). К этому моменту на рынке присутствовали кристаллы i8086, i8088, Z8000 и M68000. Сотрудники Intel с замиранием сердца ждали решения "Голубого гиганта". Дэйв Хаус, тогда генеральный менеджер компании по микропроцессорам и периферийным кристаллам, вспоминает, что выбор IBM в пользу кристалла i8088 как центрального процессора для ПК стал очевиден после поступившего от дилера Intel из Бока-Ратона сообщения о начале выполнения заказов на системы разработки и внутрисхемные эмуляторы ICE-88. Но тем не менее в течение еще пяти месяцев Intel не могла сделать никаких официальных заявлений на этот счет. Поскольку Apple выбрала для своих машин кристалл M68000, успокаиваться было рано и кампания Operation Crush была продолжена. И только после получения от IBM первых крупных заказов на процессоры стало окончательно ясно - это самая крупная победа Intel. Теперь уже вне всякого сомнения именно ее приборы стали истинным стандартом в электронной промышленности. В 1984 г. объемы продаж МП

i8086 в девять раз превысили соответствующий показатель для M68000.

Использование микросхем i8086 в IBM PC предопределило дальнейшее развитие корпорации Intel как разработчика и изготовителя универсальных процессоров общего назначения. Вычислительная мощь 16-разрядных приборов была поддержана арифметическим сопроцессором i8087, который позволил превратить ПК в достаточно мощный инструмент и для решения задач вычислительного характера. Более того, теперь и разработчики систем управления на базе 86-го ЦП получили возможность использовать интенсивную арифметическую обработку информации, для которой ранее служили мини-ЭВМ. Бурный рост объемов продаж персональных компьютеров явно свидетельствовал о том, что рождается новый рынок огромной емкости.

В конце 70 - начале 80-х годов основными сферами применения микропроцессоров продолжали оставаться различные системы управления (ПК еще не появились). Сложность решаемых задач требовала высокой производительности ЦП, большого объема ОЗУ и поддержки многозадачных сред, характерных для подобного рода комплексов. Для разработчиков программного обеспечения 1 Мбайт оперативной памяти достаточно быстро превратился в своеобразное прокрустово ложе, а отсутствие в i8086 аппаратных средств поддержки защиты памяти существенно усложняло создание ПО. Решение этих задач и было основной целью разработки следующего кристалла.

Процессор i80286 был анонсирован 1февраля 1982 г. и сразу приковал к себе пристальное внимание специалистов. Архитектура и характеристики чипа оказались весьма впечатляющими. Оставшись 16-разрядным прибором, по производительности новый ЦП в 3 - 6 раз превзошел своего предшественника при тактовой частоте первой модификации 8 МГц. Благодаря использованию многовыводного корпуса (к тому времени проблема их массового производства была успешно решена) разработчики смогли применить схему с раздельными шинами адресов и данных. 24 разряда адреса позволили обращаться к физической памяти объемом до 16 Мбайт - такую же емкость имели и старшие модели большинства мэйнфреймов. Встроенная система управления памя-

тью и средства ее защиты открывали широкие возможности использования МП в многозадачных средах. Кроме того, аппаратура i80286 обеспечивала работу с виртуальной памятью объемом до 1 Гбайт. Для поддержки устройства управления памятью система команд пополнилась еще 16 инструкциями.

Новый ЦП имел два режима работы - реальный и защищенный. В первом случае он воспринимался как быстрый ЦП i8086 с несколько расширенной системой команд и прекрасно подходил тем потребителям, для которых, помимо скоростных характеристик, жизненно важным было сохранение существующего задела ПО. Работа в защищенном режиме позволяла использовать преимущества прибора в полном объеме, и прежде всего - большой объем основной памяти. Этими качествами решила воспользоваться IBM, применив процессор в новой модели ПК типа AT, что с неимоверной скоростью превратило i80286 в бестселлер рынка 16-разрядных систем. Как видим, очередной передел этого сектора, начавшийся в 1982 г., компания встретила во всеоружии. Помимо 286-го, в арсенале Intel были еще две модели процессоров i80186 и i80188 - модифицированные варианты микросхем i8086 и i8088 соответственно.

Достигнутые в технологии успехи позволили разработчикам разместить на кристалле контроллеры прерываний и ПДП, а также таймер и системный генератор. Таким образом, покупателю были предложены мощные однокристальные ЭВМ, с помощью которых можно было реализовать законченную систему с числом ИС не более десяти. Уже в первый год после начала поставок количество проданных чипов i80186 превысило в 30 раз аналогичный показатель для i8086.

Спрос на i80286 также требовал постоянного увеличения объемов его производства. Длина слова - 32 разряда Использование корпорацией IBM микропроцессоров i80286 в новой модели персонального компьютера IBM PC/AT и стремительный рост объемов ее продаж стимулировали усилия специалистов Intel по разработке прибора следующего поколения. Увеличение тактовой частоты 286-го процессора сверх достигнутого предела в 16 МГц давалось уже слишком дорого, а кроме того, никак не устраняло узкого места системы, которым оставалась оперативная память.

Помимо прочего, 286-й решил далеко не все проблемы, характерные для многозадачных сред. У инженеров Intel было два пути кардинального повышения производительности процессора - 32-разрядная обработка данных и совершенствование тракта процессор - память. Эффективное же функционирование МП под управлением многозадачных ОС требовало усовершенствования устройства управления памятью. Таким образом, специалисты имели перед собой четкие задачи, которые необходимо было решить. (Любопытно, что примерно в это же время в американской прессе промелькнули сообщения о намерениях "Голубого гиганта" приобрести крупный пакет акций Intel и даже приводились оценки его размера, доходившие до 40%. Судя по всему, сделка не состоялась, но сам факт появления подобного рода информации свидетельствует о достаточно тесных контактах обеих компаний. Не исключено, что корпорация IBM выступала в качестве заказчика кристалла следующего поколения, а ее инженеры принимали непосредственное участие в формировании технических требований на процессор.)

Первенец 32-разрядных систем i80386 был представлен 17 октября 1985 г. и имел все права на звание процессора для ЭВМ общего назначения. Использование КМОП-технологии с проектными нормами 1 мкм и двумя уровнями металлизации позволило разместить на кристалле 275 тыс. транзисторов и реализовать полностью 32-разрядную архитектуру ЦП (рис.4). 32 разряда адреса обеспечили адресацию физической памяти объемом до 4 Гбайт и виртуальной памяти емкостью до 64 Тбайт. Встроенная в МП система управления памятью и защиты включала регистры преобразования адреса, механизмы защиты оперативной памяти и улучшенные аппаратные средства поддержки многозадачных ОС. Помимо работы с виртуальной памятью допускались операции с памятью, имевшей страничную организацию. Предварительная выборка команд, буфер на 16 инструкций, конвейер команд и аппаратная реализация функций преобразования адреса значительно уменьшили среднее время выполнения команды. Благодаря этим архитектурным особенностям, процессор мог выполнять 3 - 4 млн команд в секунду, что примерно в 6 - 8 раз превышало аналогичный показатель для МП i8086. Безусловно, новый прибор остался

совместимым со своими предшественниками на уровне объектных кодов. Одной из наиболее любопытных особенностей рассматриваемой разработки компании было использование кэш-памяти, позволившей существенно повысить производительность систем на базе 386-го процессора (еще один атрибут универсальных машин, который стал применяться в микропроцессорных системах).

Для управления работой буфера Intel создала БИС высокопроизводительного контроллера кэш-памяти типа i82385, с помощью которой формировался двухвходовой множественный ассоциативный кэш. Указанная БИС обеспечивала управление буфером емкостью до 32 Кбайт и высокий коэффициент удачных обращений. Поскольку степень интеграции была еще недостаточна для реализации на том же кристалле и математического сопроцессора, он по-прежнему выпускался в виде отдельного кристалла i80387, дополняя вычислительную мощь ЦП.

Особый интерес представляли три режима работы кристалла - реальный, защищенный и режим виртуального МП i8086. В первом обеспечивалась совместимость на уровне объектных кодов с устройствами i8086 и i80286, работающими в реальном режиме. При этом архитектура i80386 была почти идентична архитектуре 86-го процессора, для программиста же он вообще представлялся как ЦП i8086, выполняющий соответствующие программы с большей скоростью и обладающий расширенными системой команд и регистрами. Благодаря этим качествам 32- разрядного продукта компания сохранила прежних клиентов, которые хотели модернизировать свои системы, не отказываясь от имевшегося задела в области программного обеспечения, и привлекла тех, кому изначально требовалась высокая скорость обработки информации. Этот "двунаправленный" маркетинговый ход затронул как разработчиков и пользователей ПК, так и специалистов, занимающихся сложными системами управления. Одно из основных ограничений реального режима было связано с предельным объемом адресуемой памяти, равным 1 Мбайт. От него свободен защищенный режим, позволяющий воспользоваться всеми преимуществами архитектуры нового ЦП. Размер адресного пространства в этом случае увеличивался до 4 Гбайт, а объем поддерживаемых программ - до

64 Тбайт. Производителям ПО это позволяло задействовать достаточно гибкие методы разработки и создавать более крупные программные пакеты.

Для конечных пользователей выполнение приложений, рассчитанных на работу в реальном и защищенном режимах, происходило без каких-либо функциональных отличий, поскольку управление обоими режимами базировалось на средствах ОС и специальном прикладном ПО. Однако системы защищенного режима обладали более высоким быстродействием и возможностями организации истинной многозадачности. Наконец, режим виртуального МП открывал возможность одновременного исполнения ОС и прикладных программ, написанных для МП i8086, i80286 и i80386. Поскольку объем памяти, адресуемой 386-м процессором, не ограничен значением 1 Мбайт, он позволял формировать несколько виртуальных сред i8086. Немаловажно, что эти среды могли порождаться в одно и то же время, а механизм защищенного режима обеспечивал ОС и ее прикладным задачам использование различных областей памяти. Благодаря таким возможностям аппаратуры, можно было вместо нескольких ЦП типа i8086 использовать один процессор i80386, сохранив львиную долю имевшегося ПО.

Примерно в этот же период IBM и Microsoft приступили к разработке новой многозадачной ОС с графическим интерфейсом пользователя. Архитектура кристалла, явно ориентированная на многозадачные среды, заставляет предположить участие в формировании технических требований к первому 32-разрядному МП не только IBM (о чем уже говорилось), но и Microsoft. И хотя впоследствии пути этих компаний разошлись (достаточно упомянуть противостояние OS/2 и Windows), процессор одинаково успешно работал с первыми версиями обеих систем. Стремление удовлетворить запросы потребителей всех категорий привело руководство Intel к идее повторить свой удачный опыт с процессором i8088, выпустив вариант 386-го МП с 16- разрядной внешней шиной данных (при сохранении внутренней 32-разрядной архитектуры). Существующий прибор получил обозначение i80386SX и был анонсирован 16 июня 1988 г., а уже менее чем через полгода пользователям были предложены первые ПК на его основе. Поскольку

эти модели стоили дешевле компьютеров с ЦП 80386DX, многие потребители вполне справедливо рассматривали их как начальную ступень в применении вычислительной техники.
Так Intel создала еще один сектор рынка. В конце 80-х годов степень интеграции микросхем быстро приближалась к 1млн транзисторов на кристалле, и некоторые специалисты шутили, что еще неизвестно, что же делать с такой БИС, а журнал "Electronics" в одном из традиционных новогодних поздравлений пожелал Гордону Муру вдохновения, чтобы придумать ей достойное применение.

Вдохновение не оставило президента Intel, и 10 апреля 1989 г. корпорация объявила о начале выпуска 32-разрядного прибора второго поколения - i80486, ставшего после устройств i8080 и i8086 еще одним долгожителем. Архитектура нового ЦП (рис. 5) отчасти напоминала строение своего предшественника, но вместе с тем имела и ряд коренных отличий. 1,2 млн транзисторов позволили разработчикам реализовать на кристалле не только кэш-память, но и математический сопроцессор. Такое техническое решение свело к возможному минимуму число чипов на плате и самым благоприятным образом сказалось на стоимости готовых систем. В отличие от имевшегося в МП 80386 двухвходового множественного ассоциативного кэша в i80486 использовался более эффективный четырехвходовой буфер, который, будучи размещенным в чипе, мог работать на тактовой частоте процессора. Как и предыдущий МП, новый прибор функционировал в трех режимах и был ориентирован на многозадачные среды. За счет интеграции математического сопроцессора в БИС, а также модернизации его архитектуры производительность на задачах вычислительного характера возросла в 3 - 4 раза. Общая же производительность 486-го превышала аналогичный параметр своего предшественника в 4 - 5 раз.

Блестящие характеристики процессора i80486 позволили ему быстро завоевать симпатии производителей и пользователей, и через два года модели ПК на его базе полностью вытеснили изделия предыдущего поколения. В течение шести лет кристалл безраздельно господствовал на рынке. Институт "вторых поставщиков", возникший еще в эпоху 8-разрядных систем, продолжал су-

ществовать, и изготовлением микросхем класса 486 занимались такие гиганты электронной промышленности, как IBM, Texas Instruments и AMD. На совершенствовании архитектуры кристаллов Intel родились и сделали имя Cyrix и NexGen. Конкуренты оказались достаточно шустрыми и за ними требовался присмотр. Поэтому, даже приступив к проработкам процессора следующего поколения, компания не оставляла свое последнее творение без внимания.

Ровно через два года после выпуска i80486 появилась упрощенная версия кристалла (без сопроцессора), получившая по далеко не очевидной аналогии с предшествующей моделью обозначение i80486SX. Дальнейшее совершенствование пошло по пути увеличения тактовой частоты: были представлены версии на 50, 66, 75 и 100 МГц. Разработку вариантов с более высокими тактовыми частотами сочли нецелесообразной, и компании-конкуренты получили в этом отношении карт-бланш. Intel же была целиком устремлена в будущее.

Pentium - пятое поколение микропроцессоров Стремительное усложнение программного обеспечения и постоянное расширение сферы применения компьютеров настоятельно требовали существенного роста вычислительной мощи центральных процессоров ПК. Ко всему прочему на пятки стали наступать и RISC-процессоры. Хотя в конце 80-х годов некоторые эксперты предсказывали близкий конец кристаллов CISC, корпорация Intel вполне справедливо посчитала, что до этого еще далеко и в микропроцессорах использованы не все возможности CISC-архитектуры. Кроме того, фирме вряд ли простили бы отказ от программной совместимости с предшествующими моделями - стоимость накопленного системного и прикладного ПО уже измерялась в миллиардах долларов.

Как это случалось не раз, проработки нового процессора начались, когда проект создания 486-го МП вступил в заключительную стадию (начало 1989 г.). Менеджером команды "архитекторов" кристалла следующего поколения был назначен Дональд Альперт (Donald Alpert) из отделения микропроцессоров корпорации Intel, а работы по проектированию арифметического устройства с плавающей точкой возглавил Дрор Эвнон (Dror Avnon).

В основу продукта была положена суперскалярная архитектура (еще один атрибут из мира мэйнфреймов), которая и дала возможность получить пятикратное повышение производительности по сравнению с моделью 486DX.

Высокая скорость выполнения команд достигалась благодаря двум 5-ступенчатым конвейерам, позволявшим одновременно исполнять несколько инструкций. Для постоянной загрузки обоих конвейеров из кэша требуется широкая полоса пропускания. Совмещенный буфер команд и данных обеспечить ее не мог, и разработчики воспользовались решением из арсенала RISC-процессоров, оснастив Pentium раздельными буферами команд и данных. При этом обмен информацией с памятью через кэш данных осуществлялся совершенно независимо от процессорного ядра, а буфер инструкций был связан с ним через высокоскоростную 256-разрядную внутреннюю шину. Несмотря на то что новый кристалл был спроектирован как 32-разрядный, для связи с остальными компонентами системы использовалась внешняя 64-разрядная шина данных с максимальной пропускной способностью 528 Мбайт/с. Еще одной "изюминкой" архитектуры, позаимствованной у представителей универсальных ЭВМ стала схема предсказания переходов.

По оценкам специалистов компании, это дополнительно увеличило производительность ЦП на 20 - 25%. Обработка графической информации, мультимедиа-приложения, а также все более интенсивное использование ПК для решения задач вычислительного характера требуют высокой производительности при выполнении операций с плавающей точкой. Скоростные характеристики этого блока получены за счет применения конвейерной архитектуры устройства (8-ступенчатый конвейер FPU позволил выдавать результаты каждый такт), а также благодаря аппаратной реализации (вместо традиционной микропрограммной) основных арифметических операций. Сложение, умножение и деление выполнялись раздельными высокопроизводительными блоками умножителя, сложителя и делителя. Средства для работы с плавающей точкой поддерживали стандарт IEEE 754 и операции с одинарной, двойной и расширенной (80-битовый формат) точностью, а также набор базовых трансцендентных функций. В итоге по скорости вы-

полнения команд с плавающей точкой Pentium в пять - семь раз превзошел процессор 486DX2/50 и почти на порядок - микросхему 486DX/33. Выход новинки на рынок был обставлен корпорацией в ее лучших традициях (вспомним акцию Operation Crush), начиная от мощной рекламной кампании и кончая присвоением процессору собственного имени.

Предварительная информация о характеристиках этого ЦП, которая, кстати, незначительно разошлась с реальными параметрами прибора, подогрела к нему интерес как поставщиков компьютеров, так и пользователей. Пристальное внимание Intel традиционно уделила тесному взаимодействию с разработчиками ПО, которые получили спецификации процессора Pentium одними из первых и очередные версии приложений проектировали уже с учетом возможностей кристалла. Таким образом, процессор вышел в свет в сопровождении почетного эскорта новых программных продуктов.

Несмотря на ряд проблем, практически всегда возникающих при выпуске новых изделий, Pentium завоевал популярность и начал быстро теснить на рынке своего предшественника. Этому способствовала и политика самой компании, которая, стремясь обеспечить ускоренный переход потребителей на Pentium, вскоре прекратила работы по совершенствованию микросхем i80486 и даже начала сворачивать их производство, что, кстати, вызвало нескрываемое раздражение некоторых крупных потребителей продукции Intel, в частности Compaq. Ко всему прочему, теперь Intel уже никому не хотела продавать свое ноу-хау и институт вторых поставщиков приказал долго жить. С одной стороны, необходимости в утверждении архитектуры, и без того давно ставшей мировым стандартом, более не было, а производственные мощности корпорации стали вполне достаточны для удовлетворения спроса на ее изделия; с другой - создание Pentium оказалось настолько дорогим удовольствием (по разным оценкам, Intel вложила в этот проект 4 - 6 млрд дол.), что отдавать даже малую часть рынка становилось довольно рискованным. Хотя объемы продаж систем на базе нового ЦП росли буквально не по дням, а по часам (в 1994 г. их было продано 5,5 млн, а в 1995 г. - уже 18,4 млн), почивать на

лаврах компания не собиралась и постоянно улучшала рабочие характеристики чипов за счет увеличения тактовой частоты.

Начав в 1993 г. с отметки в 60 МГц и быстро миновав промежуточные барьеры в 66; 75; 90; 100; 120; 133 и 166 МГц, в этом году Intel довела тактовую частоту Pentium II - представитель второго поколения семейства P6. Помимо того что он дополнен MMX-командами, этот кристалл содержит в два раза большую кэш-память по сравнению с Pentium Pro - по 16 Кбайт (вместо восьми) для команд и данных - и взаимодействует с кэшем L2 не на свой полный тактовой, а на вдвое меньшей частоте.

Pentium II размещается не в двухотсековом керамическом корпусе вместе со специализированным кэшем L2, а представляет собой традиционную однокристальную конструкцию, в которой предусмотрены отдельные соединения с системной шиной и интерфейс кэша L2. Однако Intel не продает этот кристалл сам по себе; процессор устанавливается на небольшой печатной плате вместе с кэшем L2, построенном на кристаллах памяти SRAM. Затем эта сборка, которую Intel называет картриджем SEC (Single Edge Contact - с односторонним расположением контактов), помещается в корпус из пластмассы и металла. Именно эта конструкция Intel носит название Pentium II.

Интерфейс кэша L2 процессора Pentium II имеет в два раза меньшую скорость, чем у кэша Pentium Pro, что компенсируется до некоторой степени большей емкостью кэша L1 процессора. Гораздо важнее то, что благодаря двукратному уменьшению скорости интерфейса кэша становится значительно проще увеличить тактовую частоту ЦП; в настоящее время выпускаются 233-, 266- и 300-МГц модели Pentium II, а в будущем году намечается достичь отметки 400 МГц.

Применение для кэша Pentium II микросхем памяти, выпускаемых другими изготовителями, высвобождает производственные мощности фирмы Intel, а благодаря острой конкуренции на рынке SRAM сохраняется низкий уровень себестоимости. Это позволило Intel сделать 512-Кбайт кэш L2 стандартной принадлежностью Pentium II; вариант 512-Кбайт кэша для Pentium Pro обходится дорого и используется очень редко. В результате хотя, ин-

терфейс кэша L2 процессора Pentium II работает с меньшей скоростью, емкости его кэшей L1 и L2 в два раза больше.

Одно из ограничений, присущих нынешней архитектуре Pentium II, состоит в том, что она позволяет строить только двухпроцессорные системы, в то время как Pentium Pro может работать в четырехпроцессорной конфигурации. По этой причине до конца нынешнего года многие разработчики серверов будут по-прежнему ориентироваться на Pentium Pro. Даже в некоторых одно- и двухпроцессорных серверах будет применяться Pentium Pro, поскольку для выполнения прикладных задач, требующих обработки больших объемов данных, преимущества высокоскоростного кэша L2 могут оказаться гораздо важнее, нежели большая емкость кэшей и высокая тактовая частота. Кроме того, серверы, как правило, мало что выигрывают от технологии MMX. Однако в 1998 г. появится новый вариант Pentium II (описанный ниже в этой статье), который вытеснит Pentium Pro с рынка серверов.

Приложение 3. Сводные данные о МП Intel

Процессор Intel"Xeon"
Процессор Intel Xeon (2; 1,7; 1,5 и 1,4 ГГц)
Дата выпуска: 25 сентября 2001 г. (2 ГГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Organic Lan Grid Array 603 (OLGA 603)
Частота системной шины: 400 МГц
Потоковые SIMD расширения SSE2
Область применения: Двухпроцессорные серверы и рабочие станции
Процессор Intel "Pentium" 4
Процессор Intel Pentium 4 произведенный по 0.18-микронной технологии (2; 1,9; 1,8; 1,7; 1,6; 1,5 и 1,4 ГГц)
Дата выпуска: 27 августа 2001 г. (2, 1.9 ГГц); 2 июля 2001 г. (1.8, 1.6 ГГц); 23 апреля 2001 г. (1.7 ГГц); 20 ноября 2000 г. (1.5, 1.4 ГГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)

Тип корпуса: PGA423, PGA478
Частота системной шины: 400 МГц
Потоковые SIMD расширения SSE2
Область применения: офисные и домашние ПК, одно- и двухпроцессорные серверы и рабочие станции

Процессор Pentium III Xeon

Процессор Pentium III Xeon, произведенный по 0.18-микронной технологии (900 МГц)

Дата выпуска: 21 марта 2001 г. (900 МГц)
Кэш-память второго уровня: 2 МБ Advanced Transfer Cache (интегрированная)
Тип корпуса: SC330
Частота системной шины: 100 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: сервера конечного уровня, четырех-восьмипроцессорные рабочие станции

Процессор Pentium III Xeon, произведенный по 0.18-микронной технологии (933 МГц)

Дата выпуска: 24 мая 2000 г. (933 МГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: SC330
Частота системной шины: 133 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: офисные и домашние ПК, одно- и двухпроцессорные серверы и рабочие станции

Процессор Pentium III Xeon, произведенный по 0 .18-микронной технологии (700 МГц)

Дата выпуска: 21 марта 2001 г. (700 МГц)
Кэш-память второго уровня: 1 и 2 МБ Advanced Transfer Cache (интегрированная)
Тип корпуса: SC330
Частота системной шины: 100 МГц

Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: четырех-восьмипроцессорные рабочие станции

Процессор Pentium III Xeon, произведенный по 0.18-микронной технологии (866, 800, 733, 667 и 600 МГц)
Дата выпуска: 25 октября 1999 г. (733, 667 и 600 МГц); 12 января 2001 г. (800 МГц); 10 апреля 2000 г. (866 МГц)
Число транзисторов: 28 миллионов (0.18-микронная технология)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2)
Частота системной шины: 133 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: двухпроцессорные рабочие станции

Процессор Pentium III Xeon, произведенный по 0.18-микронной технологии (500 и 550 МГц)
Дата выпуска: 17 марта 1999 г.
Тактовая частота: 500 и 550 МГц
Число транзисторов: 9,5 миллионов (0.25-микронная технология)
Кэш-память второго уровня: 512 КБ и 1-2 МБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2)
Частота системной шины: 100 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: потребительские ПК, одно-и двухпроцессорные серверы и рабочие станции

Процессор Intel Pentium III с технологией сверхнизкого энергопотребления
Дата выпуска: 13 ноября 2001 г.
Тактовая частота: 700 МГц
Частота системной шины: 100 МГц

Технология производства: 0.13 микрон
Кэш-память второго уровня: 512 КБ
Тип корпуса: uFCBGA
SIMD Extensions
Наряжение ядра: 1.1 В
Область применения: рабочие станции

Процессор Intel Pentium III для мобильных ПК
Дата выпуска: 30 июля 2001 г.
Тактовая частота: 1,13; 1,06; 1 ГГц
Частота системной шины: 133 МГц
Технология производства: 0.13 микрон
Кэш-память второго уровня: 512 КБ
Тип корпуса: Micro FCBGA/PGA
SIMD Extensions
Наряжение ядра: 1.4 В в режиме максимальной производительности и 1.15 в режиме оптимальной производительности
Напряжение питания: 2 В в режиме оптимальной производительности
Область применения: полный спектр мобильных ПК

Процессор Intel Pentium III для мобильных ПК
Дата выпуска: 30 июля 2001 г.
Тактовая частота: 933, 866 МГц
Частота системной шины: 133 МГц
Технология производства: 0.13 микрон
Кэш-память второго уровня: 512 КБ
Тип корпуса: Micro FCBGA/PGA
SIMD Extensions
Наряжение ядра: 1.5 В в режиме максимальной производительности и 1.05 в режиме оптимальной производительности
Напряжение питания: 1 В в режиме оптимальной производительности
Область применения: полный спектр мобильных ПК

Процессор Pentium III со сверхнизким энергопотреблением и технологией Intel SpeedStep≥ (600, 500 МГц) для мобильных

ПК
Дата выпуска: 21 мая 2001 г. (600 МГц); 30 января 2001 г. (500 МГц)
Тактовая частота: 600, 500, 300(режим оптимальной производительности) МГц
Технология производства: 0.13 микрон
Кэш-память: 512 КБ (интегрированная)
Тип корпуса: Ball Grid Array (BGA)
Частота системной шины: 100 МГц
Наряжение ядра: 1.1 В при 600, 500 МГц; 1 В в режиме оптимальной производительности
Напряжение питания: < 1 В при 600, 500 МГц; < 0.5 при 300 МГц
Область применения: офисные и пользовательские мобильные ПК

Технология низкого энергопотребления

Процессор Pentium III с низким энергопотреблением и технологией Intel SpeedStep≥ (750, 700, 600 МГц) для мобильных ПК
Дата выпуска: 21 мая 2001 г. (750 МГц); 27 февраля 2001 г. (700 МГц); 19 июня 2000 г. (600 МГц)
Технология производства: 0.18 микрон
Кэш-память второго уровня: 512 КБ (интегрированная)
Тип корпуса: Ball Grid Array (BGA)
Частота системной шины: 100 МГц
Потоковые SIMD расширения
Наряжение ядра: 1.1 В в режиме оптимальной производительности
Напряжение питания: 1 В в режиме оптимальной производительности
Область применения: офисные и пользовательские мобильные ПК

Процессоры для мобильных ПК

Процессор Pentium III с технологией Intel SpeedStep≥ (1 ГГц, 900, 850, 800, 750 МГц)
Дата выпуска: 19 марта 2001 г. (1 ГГц, 900 МГц); 25 сентября 2000 г. (850, 800 МГц); 19 июня 2000 (750 МГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (ин-

тегрированная)
Тип корпуса: Mobile Module, Ball Grid Array (BGA) или Micro Pin Grid Array (micro PGA)
Частота системной шины: 100 МГц
Адресуемая память: 64 ГБ
Напряжение ядра: 1.6 В
Напряжение питания: <2 В
Область применения: офисные и пользовательские ПК

Процессор Pentium III с технологией Intel SpeedStep≥ (700, 650, 600 МГц)
Дата выпуска: 18 января 2000 г. (650, 600 МГц); 24 апреля 2000 г. (700 МГц); 19 июня 2000 г. (750 МГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Mobile Module, Ball Grid Array (BGA) или Micro Pin Grid Array (micro PGA)
Частота системной шины: 100 МГц
Адресуемая память: 64 ГБ
Напряжение ядра: 1.6 В
Область применения: офисные и пользовательские ПК

Процессор Pentium III (500, 450, 400 МГц), произведенный по технологии 0.18-микрон
Дата выпуска: 25 октября 1999 г. (500, 450, 400 МГц)
Число транзисторов: 28 миллионов (0.18-микронная технология)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Mobile Module, Ball Grid Array (BGA) или Micro Pin Grid Array (micro PGA)
Частота системной шины: 100 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Напряжение ядра: 1.6 В для 500 и 450 МГц; 1.35 В для 400 МГц
Область применения: пользовательские ПК и двухпроцессорные сервера

Для встраеваемых применений

Процессор Pentium III (700 МГц) для встраеваемых применений
Дата выпуска: 19 марта 2001 г.
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Технология производства: 0.18 микрон
Тип корпуса: Ball Grid Array (BGA)
Напряжение питания: <2 В
Частота системной шины: 100 МГц
Потоковые SIMD расширения
Адресуемая память: 64 ГБ
Напряжение ядра: 1.35 В
Напряжение питания:

Настольные ПК

Процессор Pentium III (933 МГц), произведенный по технологии 0.18-микрон
Дата выпуска: 24 мая 2000 г. (933 МГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2), Flip-Chip Pin Grid Array (FC-PGA)
Частота системной шины: 133 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: офисные и пользовательские ПК и двухпроцессорные сервера

Процессор Pentium III (1 ГГц, 933, 866/850 МГц), произведенный по технологии 0.18-микрон
Дата выпуска: 8 марта 2000 г. (1 ГГц); 20 марта 2000 г. (866, 850 МГц); 24 мая 2000 г. (933 МГц)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)

Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2), Flip-Chip Pin Grid Array (FC-PGA)
Частота системной шины: 100 и 133 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: офисные и пользовательские ПК и двухпроцессорные сервера

Процессор Pentium III (500, 533, 550, 600, 650, 667, 700 и 733 МГц), произведенный по технологии 0.18-микрон
Дата выпуска: 25 октября 1999 г. (500, 533, 550, 600, 650, 667, 700 и 733 МГц)
Число транзисторов: 28 миллионов (0.18-микронная технология)
Кэш-память второго уровня: 256 КБ Advanced Transfer Cache (интегрированная)
Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2), Flip-Chip Pin Grid Array (FC-PGA)
Частота системной шины: 100 и 133 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: офисные и пользовательские ПК и двухпроцессорные сервера

Процессор Pentium III (450, 500, 550 и 600 МГц)
Дата выпуска: 26 февраля 1999 г. (450, 500 МГц); 17 мая 1999 г. (500 МГц); 2 августа 1999 г. (600 МГц)
Тактовая частота: 450, 500, 550 и 600 МГц
Число транзисторов: 9,5 миллионов (0.25-микронная технология)
Кэш-память второго уровня: 256 КБ
Тип корпуса: Single Edge Contact Cartridge (S.E.C.C. 2)
Частота системной шины: 100 МГц
Разрядность системной шины: 64 разрядов
Адресуемая память: 64 ГБ
Область применения: офисные и пользовательские ПК и двухпроцессорные сервера

Процессор Pentium II Xeon(450 МГц)

Объявлен: 5 января 1999 г.
Тактовая частота: 450 МГц
Данные по производительности: **серверы**, **рабочие станции**
Кэш L2: 512 КБ, 1 МБ и 2 МБ
Количество транзисторов: 7.5 млн
Тип корпуса процессора: Картридж с односторонним контактом (S.E.C)
Частота шины: 100 МГц
Ширина полосы пропускания шины: 8 байт
Адресуемая память: 64 Гигабайт
Виртуальная память: 64 Терабайт
Габариты модуля: высота 4.8" x ширина 6.0" x глубина .73"
Применение: 4-процессорные серверы и рабочие станции

Процессор Pentium II Xeon (450 МГц)
Объявлен: 6 октября 1998 г.
Тактовая частота: 450 МГц
Данные по производительности: **рабочие станции**, **серверы**
Кэш L2: 512 КБ
Количество транзисторов: 7.5 млн
Тип корпуса: Картридж с односторонним контактом (S.E.C)
Частота шины: 100 МГц
Ширина полосы пропускания шины: 8 bytes
Адресуемая память: 64 Гигабайт
Виртуальная память: 64 Терабайт
Габариты модуля: высота 4.8" x ширина 6.0" x глубина .73"
Применение: Двухпроцессорные рабочие станции и серверы

Процессор Pentium II Xeon (400 МГц)
Объявлен: 29 июня 1998 г.
Тактовая частота: 400 МГц
Данные по производительности: **рабочие станции**, **серверы**
Кэш L2: 512 КБ и 1 МБ
Количество транзисторов: 7.5 млн
Тип коруса: Картридж с односторонним контактом (S.E.C)
Частота шины: 100 МГц
Ширина полосы пропускания шины: 8 bytes

Адресуемая память: 64 Гигабайт
Виртуальная память: 64 Терабайт
Габариты модуля: высота 4.8" x ширина 6.0" x глубина .73"
Применение: Серверы и рабочие станции среднего уровня и выше

Процессор Pentium II
Процессор Pentium II для мобильных ПК (400 МГц)
Объявление: 14 июня 1999 г.
Тактовая частота: 400 МГц; **данные по производительности**
Количество транзисторов: 27.4 млн (0.18-мкм процесс), 256 КБ встроенной кэш L2
Тип корпуса: Ball Grid Array (BGA): число выводов = 615
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.5 В
Питание зависит от частоты: 400 МГц = 7.5 Вт
Применение: мобильные ПК

Процессор Pentium II для мобильных ПК (400 МГц)
Объявление: 14 июня 1999 г.
Тактовая частота: 400 МГц; **данные по производительности**
Количество транзисторов: 27.4 млн (0.25-мкм процесс), 256 КБ встроенной кэш L2
Тип корпуса: Mini-Cartridge and MMC1, MMC2
Напряжение ядра: 1.5 В
Питание зависит от частоты: 400 МГц = 7.5 Вт
Применение: мобильные ПК

Процессор Pentium II для мобильных ПК (266, 300, 333 и 366 МГц)
Объявление: 25 января 1999 г.
Тактовые частоты: 266, 300, 333 и 366 МГц; **данные по производительности**
Количество транзисторов: 27.4 млн (0.25-мкм процесс), 256 КБ встроенной кэш L2
Тип корпуса: Ball Grid Array (BGA): число выводов = 615
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.6 В

Питание зависит от частоты: 366 МГц = 9.5 Вт; 333 МГц = 8.6 Вт; 300 МГц = 7.7 Вт; 266 МГц = 7.0 Вт
Применение: мобильные ПК

Процессор Pentium II для мобильных ПК (300 МГц)
Объявлен: 9 сентября 1998 г.
Тактовая частота: 300 МГц; **данные по производительности**
Количество транзисторов: 7.5 млн (0.25-мкм процесс), 512 K кэш L2
Разрядность шины: 64-bit
Адресуемая память: ~68 Гигабайт
Мобильный мини-картридж: 240 выводов
Мобильный модуль: 280(mmc1)/400(mmc2) выводов
Напряжение ядра: 1.6 В
Мощность: 9.0 Вт (включая ядро CPU и 512 КБ кэш L2)
Применение: мобильные ПК

Процессор Pentium II для мобильных ПК (233 и 266 МГц)
Объявлен: 2 апреля 1998 г.
Тактовая частота: 233, 266 МГц; **данные по производительности**
Количество транзисторов: 7.5 млн (0.25-мкм процесс), 512 K кэш L2
Разрядность шины: 64-bit
Адресуемая память: ~68 Гигабайт
Мобильный мини-картридж: 240 выводов
Мобильный модуль: 280(mmc1)/400(mmc2) выводов
Напряжение ядра: 1.7 В
Мощность: 7.5 Вт для 233 МГц и 8.6 Вт для 266 МГц (включая ядро CPU и 512 КБ кэш L2)
Применение: мобильные ПК

Настольные ПК

Процессор Pentium II (450 МГц)
Объявлен: 24 августа 1998 г.
Тактовая частота: 450 МГц;
Количество транзисторов: 7.5 млн (0.25-мкм процесс)

Картридж с односторонним контактом (S.E.C), 242 вывода
Габариты модуля: 5.505" x 2.473" x 0.647"
Частота шины: 100 МГц
Разрядность шины: 64-разрядная системная шина
Адресуемая память: 64 Гигабайт
Применение: Бизнес- и потребительские ПК, одно- и двухпроцессорные серверы и рабочие станции

Процессор Pentium II (350 и 400 МГц)
Объявлен: 15 апреля 1998 г.
Тактовая частота: 350, 400 МГц;
Количество транзисторов: 7.5 млн (0.25-мкм процесс), кэш L2 512K
Картридж с односторонним контактом (S.E.C), 242 вывода
Габариты модуля: 5.505" (12.82cm) x 2.473" (6.28cm) x 0.647" (1.64cm)
Частота шины: 100 МГц
Применение: ПК для бизнеса и для широкого круга потребителей, одно- и двухпроцессорные серверы и рабочие станции

Процессор Pentium II (333 МГц)
Объявлен: 26 января 1998 г.
Тактовая частота: 333 МГц (12.8 SPECint95, 9.14 SPECfp95, 8.32 SPECfpbase)
Количество транзисторов: 7,5 млн (0,25-мкм технология), кэш-память второго уровня 512 КБ
Разрядность шины: 64-битная системная шина с аппаратной коррекцией ошибок (ECC); 64-битная шина кэш с оптимизацией ECC
Адресуемая память: 64 гигабайт
Виртуальная память: 64 терабайт
Корпус с односторонним контактом (Single Edge Contact Cartridge - S.E.C), 242 вывода
Габариты модуля: 12,82 см x 6,28 см x 1,64 см
Частота шины: 66 МГц
Применение: ПК для бизнеса и для широкого круга потребителей, одно- и двухпроцессорные серверы и рабочие станции

Процессор Pentium II (300, 266, 233 МГц)
Объявлен: 7 мая 1997 г.
Тактовая частота: 300, 266, 233 МГц (11.7 SPECint95, 8.15 SPECfp95)
Количество транзисторов: 7,5 млн (0,35-мкм технология), кэш-память второго уровня 512 Кб
Ширина полосы пропускания шины: 64-битная системная шина с ECC; 64-битная шина кэш с оптимизацией ECC
Адресуемая память: 64 Гигабайт
Виртуальная память: 64 Терабайт
Корпус с односторонним контактом (S.E.C.), 242 вывода
Габариты модуля: 12,82 см x 6,28 см x 1,64 см
Применение: настольные компьютеры высшего уровня для бизнеса, рабочие станции и серверы

Процессор Intel Celeron
Технология сверхнизкого энергопотребления

Мобильный процессор Celeron со сверхнизким энергопотреблением и технологией Intel SpeedStep≥ (600 и 500 МГц)
Объявлены: 21 мая 2001 г. (600 МГц); 30 января 2001 г.
Тактовые частоты: 500 и 600 МГц; **данные по производительности**
Кэш-память второго уровня: 128 КБ
Тип корпуса: Ball Grid Array (BGA)
Частота шины: 100 МГц
Напряжение ядра: 1.1 В
Питание: <1 Вт
Применение: офисные и потребительские мобильные ПК

Мобильный процессор Celeron с низким энергопотреблением и технологией Intel SpeedStep≥ (600 МГц)
Объявлены: 21 мая 2001 г. (600 МГц); 30 января 2001 г.
Тактовые частоты: 600 МГц; **данные по производительности**
Кэш-память: 256 КБ
Технология производства: 0.18 микрон
Тип корпуса: Ball Grid Array (BGA)

Частота шины: 100 МГц
Напряжение ядра: 1.35 В
Питание: <2 Вт
Применение: ультра-портативные мобильные ПК

Процессоры для мобильных ПК

Мобильный процессор Intel Celeron (850, 800, 750, 700, 650, 550, 500, 450 МГц)
Объявлены: 2 июля 2001 г. (850 МГц); 21 мая 2001 г. (800 МГц); 19 марта 2001 г. (750 МГц); 25 сентября 2000 г. (700 МГц); 19 июня 2000 г. (650, 600 МГц); 24 апреля 2000 г. (550 МГц); 14 февраля 2000 г.(500, 450 МГц)
Тактовые частоты: 450-850 МГц; **данные по производительности**
Технология производства: 0.18 микрон; 128 КВ кэш-памяти второго уровня
Потоковые SIMD расширения Тип корпуса: Ball Grid Array (BGA2) and Pin Grid Array(PGA2)
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.6 В
Применение: недорогие мобильные ПК

Мобильный процессор Intel Celeron (466, 433 МГц)
Объявлены: 15 сентября 1999 г.
Тактовые частоты: 466, 433 МГц; **данные по производительности**
Количество транзисторов: 18,9 млн (0,25-мкм технология), кэш-память второго уровня 128 Кб
Тип корпуса: Ball Grid Array (BGA), 615 выводов
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.9 В
Питание: 466 МГц = 15.6 Вт; 433 МГц = 14.5 Вт
Применение: недорогие мобильные ПК

Мобильный процессор Intel Celeron (400 МГц)
Объявлен: 14 июня 1999 г.
Тактовая частота: 400 МГц; **данные по производительности**

Количество транзисторов: 18,9 млн (0,25-мкм технология), кэш-память второго уровня 128 Кб
Тип корпуса: Ball Grid Array (BGA), 615 выводов
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.6 В
Питание: 400 МГц = 9.2 Вт
Применение: недорогие мобильные ПК

Мобильный процессор Intel Celeron (366 МГц)
Объявлен: 17 мая 1999 г.
Тактовая частота: 366 МГц; **данные по производительности**
Количество транзисторов: 18,9 млн (0,25-мкм технология), кэш-память второго уровня 128 Кб
Тип корпуса: Ball Grid Array (BGA), 615 выводов
Габариты: ширина = 32 мм; длина = 37 мм
Напряжение ядра: 1.6 В
Питание: 9.0 Вт
Применение: недорогие мобильные ПК

Мобильный процессор Intel Celeron (333 МГц)
Объявлен: 5 апреля 1999 г.
Тактовая частота: 333 МГц; **данные по производительности**
Количество транзисторов: 18,9 млн (0,25-мкм технология), кэш-память второго уровня 128 Кб
Тип корпуса: Ball Grid Array (BGA), 615 выводов
Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.6 В
Питание: 333 МГц = 8.6 Вт
Применение: недорогие мобильные ПК

Мобильный процессор Intel Celeron (266 и 300 МГц)
Объявлен: 25 января 1999 г.
Тактовые частоты: 266 и 300 МГц; **данные по производительности**
Количество транзисторов: 18.9 млн (0.25-мкм процесс), 128K встроенной кэш L2
Тип корпуса: Ball Grid Array (BGA), число выводов = 615

Габариты: ширина = 31 мм; длина = 35 мм
Напряжение ядра: 1.6 В
Питание: 300 МГц = 7.7 Вт; 266 МГц = 7.0 Вт
Применение: недорогие мобильные ПК

Настольные ПК

Процессор Intel Celeron (1,2 ГГц)
Объявлен: 2 октября 2001 г.
Тактовая частота: 1.2 ГГц
Частота шины: 100 МГц
Технология производства: 0.13 микрон Кэш-память второго уровня: 128 КВ
Тип корпуса: FCPGA
SIMD Extensions Применение: недорогие ПК

Процессор Intel Celeron (1,1; 1,0 ГГц, 950, 900, 850, 800 МГц)
Объявлены: 13 ноября 2000 г. (1.1, 1.0 ГГц, 950 МГц); 2 июля 2001 г. (900 МГц); 21 мая 2001 г. (850 МГц);
Тактовые частоты: 800 МГц - 1.1 ГГц **данные по производительности**
Кэш-память второго уровня: 128 КВ
Тип корпуса: Flip-Chip Pin Grid Array (FC-PGA)
Частота шины: 100 МГц
Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (766, 733, 700, 667, 633, 600, 566 МГц)
Объявлены: 13 ноября 2000 г. (766, 733 МГц); 26 июня 2000 г. (700, 667, 633 МГц); 29 марта 2000 г. (600, 566 МГц);
Тактовые частоты: 766, 733, 700, 667, 633, 600, 566 МГц **данные по производительности**
Кэш-память: 128 КВ
Тип корпуса: Flip-Chip Pin Grid Array (FC-PGA)
Частота шины: 66 МГц
Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (533 МГц)
Объявлена: 4 января 2000 г.
Тактовые частоты: 533 МГц **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Кэш-память: 128 КВ
Тип корпуса: Flip-Chip Pin Grid Array (PPGA), 370 выводов
Частота шины: 66 МГц
Разрядность шины: 64-бит Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (466, 500 МГц)
Объявлены: 26 апреля 1999 г. (466 МГц); 2 августа 1999 г. (500 МГц)
Тактовые частоты: 466, 500 МГц **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Кэш-память: 128 КВ
Тип корпуса: Flip-Chip Pin Grid Array (PPGA), 370 выводов
Частота шины: 66 МГц
Разрядность шины: 64-бит Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (433 МГц)
Объявлены: 22 марта 1999 г. (466 МГц)
Тактовые частоты: 433 МГц **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Кэш-память: 128 КВ
Тип корпуса: Single Edge Processor Package (SEPP), 242 выхода; Plastic Pin Grid Array (PPGA), 370 выходов
Частота шины: 66 МГц
Разрядность шины: 64-бит Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (400, 366 МГц)
Объявлен: 4 января 1999 г.
Тактовая частота: 400, 366 МГц; **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Корпус с односторонним контактом (SEPP), 242 вывода

Корпус Plastic Pin Grid Array (PPGA), 370 вывода
Частота шины: 66 МГц
Разрядность шины: 64-bit системная шина
Адресуемая память: 4 Гигабайт
Применение: недорогие ПК

Процессор Intel Celeron (333 МГц)
Объявлен: 24 августа 1998 г.
Тактовая частота: 333 МГц; **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Корпус с односторонним контактом (SEPP), 242 вывода
Частота шины: 66 МГц
Разрядность шины: 64-bit системная шина
Адресуемая память: 4 Гигабайт
Габариты модуля: 5" x 2.275" x .208"
Применение: базовые ПК

Процессор Intel Celeron (300A МГц)
Объявлен: 24 августа 1998 г.
Тактовая частота: 300 МГц; **данные по производительности**
Количество транзисторов: 19 млн (0.25-мкм процесс)
Корпус с односторонним контактом (SEPP), 242 вывода
Частота шины: 66 МГц
Разрядность шины: 64-bit системная шина
Адресуемая память: 4 Гигабайт
Габариты модуля: 5" x 2.275" x .208"
Применение: базовые ПК

Процессор Intel Celeron (300 МГц)
Объявлен: 8 июня 1998 г.
Тактовая частота: 300 МГц; **данные по производительности**
Количество транзисторов: 7.5 млн (0.25-мкм процесс)
Корпус с односторонним контактом (SEPP), 242 вывода
Частота шины: 66МГц
Разрядность шины: 64-bit системная шина
Адресуемая память: 4 Гигабайт
Виртуальная память: 64 Терабайт

Габариты модуля: 5" x 2.275" x .208"
Применение: базовые ПК

Процессор Intel Celeron (266 МГц)
Объявлен: 15 апреля 1998 г.
Тактовая частота: 266 МГц; **данные по производительности**
Количество транзисторов: 7.5 млн (0.25-мкм процесс)
Корпус с односторонним контактом (SEPP), 242 вывода
Частота шины: 66МГц
Разрядность шины: 64-bit системная шина
Адресуемая память: 4 Гигабайт
Виртуальная память: 64 Терабайт
Габариты модуля: 5" x 2.275" x .208"
Применение: базовые ПК

Процессор Pentium Pro (200 МГц) с одним мегабайтом встроенной кэш-памяти второго уровня
Объявлен: 18 августа 1997 г.
Тактовая частота: 200 МГц (8.66 SPECint95, 6.80 SPECfp95)
Количество транзисторов: 5,5 млн (0,35-мкм технология), 1 Мб кэш-памяти второго уровня: 62 млн (0,35-мкм технология)
Ширина полосы пропускания шины: 300-битная шина внутреннего обмена; 64-битная шина кэш-памяти второго уровня
Адресуемая память: 64 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 387 (Dual Cavity Pin Grid Array Package)
Габариты модуля: 6,25 см x 6,76 см
Применение: Настольные ПК, рабочие станции и серверы высшего уровня

Процессор Pentium Pro (200, 180, 166, 150 МГц)
См. **фото**
Объявлен: 1 ноября 1995 г.
Тактовая частота: 200, 180, 166, 150 МГц (8.20 SPECint95, 6.21 SPECfp95 на системе Alder с 256 Кб кэш-памяти второго уровня)
Количество транзисторов: 5,5 млн (0,35-мкм технология), 256 КБ кэш-памяти второго уровня 256 Кб: 15,5 млн (0,6-мкм технология),

512 Кб кэш-памяти второго уровня: 31 млн (0,35-мкм технология)
Ширина полосы пропускания шины: 64-битная внешняя системная шина; 64-битная шина кэш-памяти второго уровня
Адресуемая память: 64 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 387 (Dual Cavity Pin Grid Array Package)
Габариты модуля: 6,25 см x 6,76 см
Применение: Настольные ПК, рабочие станции и серверы класса high-end

Процессоры для мобильных ПК

Процессор Pentium (300 МГц) с технологией MMX≥
Объявлен: 7 января 1999 г.
Тактовая частота: 300 МГц
Количество транзисторов: 4,5 млн (0,25-мкм технология)
Ширина полосы пропускания шины: 64 бит
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: ~68 Гигабайт
Тип корпуса: Tape Carrier Package (TCP), 320 выводов
Применение: мобильные ПК и ноутбуки

Процессор Pentium (266 МГц) с технологией MMX≥
Объявлен: 12 января 1998 г.
Тактовая частота: 266 МГц
Количество транзисторов: 4,5 млн (0,25-мкм технология)
Ширина полосы пропускания шины: 64 бит
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: ~68 Гигабайт
Тип корпуса: Tape Carrier Package (TCP), 320 выводов
Применение: мобильные ПК и ноутбуки

Процессор Pentium (200, 233 МГц) с технологией MMX≥
Объявлены: 8 сентября 1997 г.
Тактовая частота: 200, 233 МГц
Количество транзисторов: 4,5 млн (0,25-мкм технология)

Ширина полосы пропускания шины: 64 бит
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: ~68 Гигабайт
Тип корпуса: Tape Carrier Package (TCP), 320 выводов
Применение: мобильные ПК и ноутбуки

Настольные ПК

Процессор Pentium (233 МГц) с технологией MMX≥
Объявлен: 2 июня 1997 г.
Тактовая частота: 233 МГц (7.12 SPECint95, 5.21 SPECfp95. Показатель тестирования iCOMP≥ Index 2.0 - 203).
Количество транзисторов: 4,5 млн (0,35-мкм технология КМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (пластиковый корпус со штырьковыми выводами - PPGA)
Габариты модуля: 5 см x 5 см
Применение: высокопроизводительные настольные ПК и серверы

Процессор Pentium (200, 166 МГц) с технологией MMX≥
Объявлен: 8 января 1997 г.
Тактовая частота: 200, 166 МГц (6.44 SPECint95, 4.87 SPECfp95. Показатель тестирования iCOMP≥ Index 2.0 - 182).
Количество транзисторов: 4,5 млн (0,35-мкм технология КМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (пластиковый корпус со штырьковыми выводами - PPGA)
Габариты модуля: 5 см x 5 см
Применение: высокопроизводительные настольные ПК и серверы

Процессор Pentium (200 МГц)
Объявлен: 10 июня 1996 г.
Тактовая частота: 200 МГц (5.17 SPECint95,4.32 SPECfp95. Показатель тестирования iCOMP≥ Index 2.0 - 142).
Количество транзисторов: 3.3 млн (0,35-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (пластиковый корпус со штырьковыми выводами - PPGA)
Габариты модуля: 5 см x 5 см
Применение: высокопроизводительные настольные ПК и серверы

Процессор Pentium (166, 150 МГц)
Объявлен: 4 января 1996 г.
Тактовая частота: 166, 150 МГц (4.58 SPECint95, 3.92 SPECfp95 на системе Xxpress с 1 Мб кэш-памяти второго уровня)
Количество транзисторов: 3,3 млн (0,35-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (корпус со штырьковыми выводами - PGA)
Габариты модуля: 5 см x 5 см
Применение: высокопроизводительные настольные ПК и серверы

Процессор Pentium (133 МГц)
Объявлен: июнь 1995 г.
Тактовая частота: 133 МГц (218,9 млн. операций в секунду, 4.01 SPECint95, 3.50 SPECfp95 на системе Xxpress с 1 Мб кэш-памяти второго уровня)
Количество транзисторов: 3,3 млн (0,35-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)

(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (корпус со штырьковыми выводами - PGA)
Габариты модуля: 5 см x 5 см
Применение: высокопроизводительные настольные ПК и серверы

Процессор Pentium (120 МГц)
Объявлен: 27 марта 1995 г.
Тактовая частота: 120 МГц (203 млн. операций в секунду, 3.72 SPECint95, 2.81 SPECfp95 на системе Xxpress с 1 Мб кэш-памяти второго уровня)
Количество транзисторов: 3,2 млн (0,6 и 0,35-мкм технологии БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (корпус со штырьковыми выводами - PGA)
Габариты модуля: 5 см x 5 см
Применение: настольные ПК и ноутбуки

Процессоры Pentium (90 и 100 МГц)
Объявлен: 7 марта 1994 г.
Тактовая частота: 90 МГц (149,8 млн. операций в секунду, 2.74 SPECint95, 2.39 SPECfp95 на системе Gateway P5 с 256 Кб кэш-памяти второго уровня)
100 МГц (166,3 млн. операций в секунду, 3.30 SPECint95, 2.59 SPECfp95 на системе Xxpress с 1 Мб кэш-памяти второго уровня)
Количество транзисторов: 3,2 млн (0,6-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядные микропроцессоры)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 296 (корпус со штырьковыми выводами - PGA)

Габариты модуля: 5 см x 5 см
Применение: настольные ПК

Процессор Pentium (75 МГц)
Объявлен: 10 октября 1994 г.
Тактовая частота: 75 МГц (126,5 млн. операций в секунду, 2.31 SPECint95, 2.02 SPECfp95 на системе Gateway P5 с 256 Кб кэш-памяти второго уровня)
Количество транзисторов: 3,2 млн (0,6-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядный микропроцессор)
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 320 - корпус на ленточном носителе (TCP)
296 - плоский корпус с матрицей штырьковых выводов, расположенных в шахматном порядке (SPGA)
Габариты модуля: 5 см x 5 см
TCP: 2,4 x 2,4 см
Применение: настольные ПК и ноутбуки

Процессоры Pentium (60 и 66 МГц)
Объявлен: 22 марта 1993 г.
Тактовая частота: 60 МГц (100 млн. операций в секунду, 70.4 SPECint92, 55.1 SPECfp92 на системе Xpress с 256 Кб кэш-памяти второго уровня)
66 МГц (112 млн. операций в секунду, 77.9 SPECint92, 63.6 SPECfp92 на системе Xpress с 256 Кб кэш-памяти второго уровня)
Количество транзисторов: 3,1 млн (0,8-мкм технология БиКМОП)
Ширина полосы пропускания шины: 64 бит (внешняя шина данных), 32 бит (адресная шина)
(Примечание: это полностью 32-разрядные микропроцессоры).
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Число контактов: 273 (корпус со штырьковыми выводами - PGA)
Габариты модуля: 5,49 см x 5,49 см
Применение: настольные ПК

Процессоры Intel 486 и более ранние

Intel 486 SL CPU

Объявлен: 9 ноября 1992 г.
Тактовая частота: 20 МГц (15,4 млн. операций в секунду)
25 МГц (19 млн. операций в секунду)
33 МГц (25 млн. операций в секунду)
Адресуемая память: 64 Мбайт
Адресуемая виртуальная память: 64 терабайт
Формат процессора в мкм: 0,8
Количество транзисторов: 1,4 млн (0,8-мкм технология)
Путь доступа к внутренним данным: 32-битный
Путь доступа к внешним данным: 32-битный
Применение: первый процессор, специально предназначенный для ноутбуков

Процессор IntelDX2

Объявлен: 3 марта 1992 г.
Тактовая частота: 50 МГц (41 млн. операций в секунду, 29.9 SPECint92, 14.2 SPECfp92 на системе Micronics M4P с 256 Кб кэш-памяти второго уровня)
Процессор с тактовой частотой 66 МГц, о начале выпуска которого объявлено 10 августа 1992 г. (54 млн. операций в секунду, 39.6 SPECint92, 18.8 SPECfp92 на системе Micronics M4P с 256 Кб кэш-памяти второго уровня)
Количество транзисторов: 1,2 млн (0,8-мкм технология)
Ширина полосы пропускания шины: 32 бит
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Применение: недорогие настольные ПК
Благодаря применению технологии "удвоенной частоты" ядро микропроцессора работает в два раза быстрее шины.
Предназначен для самых дешевых настольных ПК с минимально необходимым уровнем производительности.

Intel486 SX CPU

Объявлен: 22 апреля 1991 г.

Тактовая частота: 16 МГц, начало выпуска процессора объявлено 16 сентября 1991 г. (13 млн. операций в секунду)
20 МГц (16,5 млн. операций в секунду)
25 МГц, начало выпуска процессора объявлено 16 сентября 1991 г. (20 млн. операций в секунду, 12 SPECint92)
33 МГц, начало выпуска процессора объявлено 21 сентября 1992 г. (27 млн. операций в секунду, 15.86 SPECint92)
Количество транзисторов: 1 185 000 (1 мкм); 900 000 (0,8-мкм технология)
Ширина полосы пропускания шины: 32 бит
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Применение: недорогие настольные ПК начального уровня класса Intel486≥.
Аналог процессора Intel486≥ DX, но без встроенного математического сопроцессора.
Имеется возможность модернизации путем установки процессора Intel OverDrive.
Этот процессор стал типовым при работе со встроенными приложениями.

Intel386 SL CPU
Объявлен: 15 октября 1990 г.
Тактовая частота: 20 МГц (4,21 млн. операций в секунду)
25 МГц - начало выпуска объявлено 30 сентября 1991 г. (5,3 млн. операций в секунду)
Количество транзисторов: 855 000 (1 мкм)
Внутренняя архитектура: 32 бит
Ширина полосы пропускания внешней шины: 16 бит
Адресуемая память: 4 гигабайта
Виртуальная память: 64 терабайт
Применение: первый микропроцессор, специально предназначенный для портативных ПК
Высокоинтегрированная архитектура включает в себя кэш-память, шину и средства управления памятью

Intel486 DX CPU

50-кратная производительность по сравнению с процессором 8088.
Объявлен: 10 апреля 1989 г.
Тактовая частота: 25 МГц (20 млн. операций в секунду, 16.8 SPECint92, 7.40 SPECfp92)
33 МГц - начало выпуска объявлено 7 мая 1990 г. (27 млн. операций в секунду, 22.4 SPECint92 на системе Micronics M4P с 128 Кб кэш-памяти второго уровня)
50 МГц - начало выпуска объявлено 24 июня 1991 г. (41 млн. операций в секунду, 33.4 SPECint92, 14.5 SPECfp92 на системе Compaq/50L с 256 Кб кэш-памяти второго уровня)
Количество транзисторов: 1 200 000 (1 мкм, в процессорах с тактовой частотой 50 МГц применялась 0,8-мкм технология)
Ширина полосы пропускания шины: 32 бит
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Применение: настольные ПК и серверы
По сравнению с процессором 8088, производительность возросла в 50 раз
Просмотр энциклопедии Britannica с применением этого процессора занял 3,5 секунды

Intel386 SX CPU

Объявлен: 16 июня 1988 г.
Тактовая частота: 16 МГц (2,5 млн. операций в секунду)
20 МГц - начало выпуска объявлено 25 января 1989 г. (2,5 млн. операций в секунду)
25 МГц (2,7 млн. операций в секунду)
33 МГц - начало выпуска объявлено 26 октября 1992 г. (2,9 млн. операций в секунду)
Количество транзисторов: 275 000 (1,5 мкм, в дальнейшем - 1 мкм)
Внутренняя архитектура: 32 бит
Ширина полосы пропускания внешней шины: 16 бит
Адресуемая память: 4 гигабайта
Виртуальная память: 64 терабайт
Применение: настольные и портативные ПК начального уровня

Intel386 DX CPU
Объявлен: 17 октября 1985 г.
Тактовая частота: 16 МГц (от 5 до 6 млн. операций в секунду)
20 МГц - начало выпуска объявлено 16 февраля 1987 г. (от 6 до 7 млн. операций в секунду)
25 МГц - начало выпуска объявлено 4 апреля 1988 г. (8,5 млн. операций в секунду)
33 МГц - начало выпуска объявлено 10 апреля 1989 г. (11,4 млн. операций в секунду, 9.4 SPECint92 на системе Compaq/i с 16 Кб кэш-памяти второго уровня)
Количество транзисторов: 275 000 (1,5 мкм, в дальнейшем - 1 мкм)
Ширина полосы пропускания шины: 32 бит
Адресуемая память: 4 гигабайт
Виртуальная память: 64 терабайт
Применение: настольные ПК
Способен обращаться к такому объему памяти, которого достаточно для составления восьмистраничной биографической справки на каждого обитателя Земли. Просмотр энциклопедии Britannica с применением этого процессора занял 12,5 секунд

80286
Объявлен: февраль 1982 г.
Тактовая частота: 6 МГц (0,9 млн операций в секунду)
10 МГц (1,5 млн операций в секунду)
12 МГц (2,66 млн операций в секунду)
Количество транзисторов: 134 000 (1,5 мкм)
Ширина полосы пропускания шины: 16 бит
Адресуемая память: 16 мегабайт
Виртуальная память: 1 гигабайт
Применение: в то время - стандартный микропроцессор для всех моделей ПК
По сравнению с процессором 8086, производительность возросла в три-шесть раз
Просмотр энциклопедии Britannica с применением этого процессора занал 45 секунд

80186
Объявлен: 1982 г.
Примечание: применялся, главным образом, в работе с управляющими приложениями

8088
Объявлен: июнь 1979 г.
Тактовая частота: 5 МГц (0,33 млн операций в секунду)
8 МГц (0,75 млн операций в секунду)
Внутренняя архитектура: 16 бит
Ширина полосы пропускания внешней шины: 8 бит
Количество транзисторов: 29 000 (3 мкм)
Применение: стандартный микропроцессор для всех ПК производства корпорации IBM и их клонов
Аналог процессора 8086, за исключением 8-битной внешней шины

8086
Объявлен: 8 июня 1978 г.
Тактовая частота: 5 МГц (0,33 млн операций в секунду)
8 МГц (0,66 млн операций в секунду)
10 МГц (0,75 млн операций в секунду)
Количество транзисторов: 29 000 (3 мкм)
Ширина полосы пропускания шины: 16 бит
Адресуемая память: 1 мегабайт
Применение: портативные ПК
По сравнению с процессором 8080, производительность возросла десятикратно

8085
Объявлен: март 1976 г.
Тактовая частота: 5 МГц
0,37 млн операций в секунду
Количество транзисторов: 6 500 (3 мкм)
Ширина полосы пропускания шины: 8 бит
Применение: весы Toledo. Электронное взвешивание и вычисление цены товара.

Высокий уровень интеграции, впервые применен единый 5-вольтовый источник питания (ранее - 12 вольт)

8080
Объявлен: апрель 1974 г.
Тактовая частота: 2 МГц
0,64 млн операций в секунду
Количество транзисторов: 6 000 (6 мкм)
Ширина полосы пропускания шины: 8 бит
Адресуемая память: 64 Кбайт
Применение: устройства управления уличным освещением, компьютеры Altair computer (первые ПК)
По сравнению с процессором 8008, производительность возросла в десять раз, а количество микросхем поддержки уменьшилось с 20-ти до шести

8008
Объявлен: апрель 1972 г. (разрабатывался одновременно с процессором 4004)
Тактовая частота: 200 килогерц
0,06 млн операций в секунду
Количество транзисторов: 3 500 (10 мкм)
Ширина полосы пропускания шины: 8 бит
Адресуемая память: 16 Кбайт
Применение: терминалы воода-вывода, калькуляторы общего назначения, автоматы бутылочного разлива
Обработка данных и текста

4004
Объявлен: 15 ноября 1971 г.
Тактовая частота: 108 килогерц
0,06 млн операций в секунду
Количество транзисторов: 2 300 (10 мкм)
Ширина полосы пропускания шины: 4 бит
Адресуемая память: 640 байт
Применение: калькуляторы Busicom
Первая компьютерная микросхема, арифметические вычисления

Приложение 4. Первая десятка самых мощных микропроцессорных систем (суперкомпьютеров)

№№	Производитель	Система	Год	Число процессоров
1.	IBM	ASCI White	2000	8192
2.	Compag	AlphaServer SC	2001	3024
3.	IBM	SP Power3	2001	3328
4.	Intel	ASCI Red	1999	9632
5.	IBM	ASCI Blue Pacific	1999	5808
6.	Compag	AlphaServer SC	2001	1536
7.	Hitachi	SR8000/MPP	2001	1152
8.	SGI	ASCI Blue Mountain	1998	6144
9.	IBM	SP Power3	2000	1336
10.	IBM	SP Power3	2001	1280

Микропроцессорная система ASCI White

Самый мощный суперкомпьютер современности - ASCI White, занимающий площадь размером в две баскетбольные площадки и установленный в Ливерморской национальной лаборатории. Он включает 512 SMP-узлов на базе 64-разрядных процессоров POWER3-II (в общей сложности 8192 процессора) и использует новую коммуникационную технологию Colony с пропускной

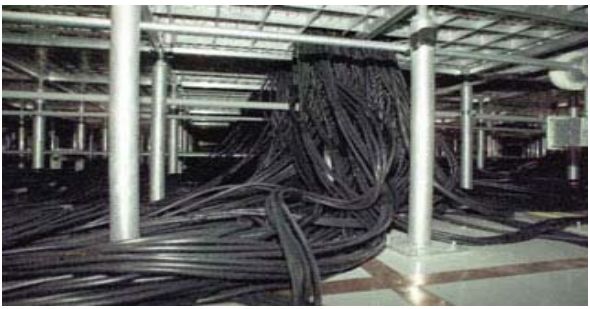
способностью около 500 Мбайт/с, что почти в четыре раза быстрее коммутатора SP high-performance switch.

Архитектура суперкомпьютера основана на зарекомендовавшей себя массивно-параллельной архитектуре RS/6000 и обеспечивает производительность в 12,3 Тфлопс (триллионов операций в секунду). Система включает в общей сложности 8 Тбайт оперативной памяти, распределенной по 16-процессорным SMP-узлам, и 160 Тбайт дисковой памяти. Доставка системы из лабораторий IBM в штате Нью-Йорк в Ливермор (Калифорния) потребовалось 28 грузовиков-трейлеров.

Все узлы системы работают под управлением ОС AIX. Суперкомпьютер используется учеными Министерства энергетики США для расчета сложных трехмерных моделей с целью поддержания ядерного оружия в безопасном состоянии. ASCI White состоит из трех отдельных систем, среди которых самой большой является White (512 узлов, 8192 процессора), а есть еще Ice (28 узлов, 448 процессоров) и Frost (68 узлов, 1088 процессоров).

Предшественником ASCI White был суперкомпьютер Blue Pacific (другое название ASCI Blue), включающий 1464 четырехпроцессорных узла на базе кристаллов PowerPC 604e/332 МГц. Узлы связаны в единую систему с помощью кабелей общей длиной почти в пять миль, а площадь машинного зала составляет 8 тыс. квадратных футов. Система ASCI Blue состоит в общей сложности из 5856 процессоров и обеспечивает пиковую производительность в 3,88 Тфлопс. Суммарный объем оперативной памяти составляет 2,6 Тбайт.

Американский национальный центр по исследованию атмосферы (NCAR) выбрал IBM в качестве поставщика самого мощного в мире суперкомпьютера, предназначенного для прогнозирования климатических изменений. Система, известная под именем Blue Sky (Синее небо) на порядок увеличивает возможности NCAR в области моделирования климата. Ядром Blue Sky являются суперкомпьютер IBM SP и системы IBM eServer p690, применение которых позволяет добиться пиковой производительности почти в 7 Тфлопс при объеме дисковой подсистемы IBM SSA в 31,5 Тбайт.



Кабельное соединение узлов суперкомпьютера ASCI White

Суперкомпьютер, получивший название "Синий шторм" (Blue Storm), создается по заказу Европейского центра среднесрочных прогнозов погоды (European Centre for Medium-Range Weather Forecasts - ECMWF). Blue Storm будет в два раза мощнее ASCI White. Для его создания необходимо 100 серверов IBM eServer p690, также известных как Regatta. Каждый системный блок размером с холодильник содержит более тысячи процессоров. В 2004 г. "Синий шторм" оснащен серверами нового поколения p960, которые делают его еще в два раза мощнее. Суперкомпьютер работает под управлением ОС AIX. Общая емкость накопителей Blue Storm составляет 1,5 петабайт, вычислительная мощь - около 23 Тфлопс. Система весит 130 т, а по мощи в 1700 раз превосходит шахматный суперкомпьютер Deep Blue.

Исследователи IBM совместно с Ливерморской национальной лабораторией ведут работы над компьютерами Blue Gene/L и Blue Gene/C. Эти компьютеры - часть начатого еще в 1999 г. с целью изучения белков 5-летнего проекта Blue Gene, в который было вложено 100 млн долл. Проектная производительность Blue Gene/L будет, таким образом, превышать суммарную производительность 500 самых мощных компьютеров в мире. При этом новый суперкомпьютер занимает площадь, равную всего половине теннисного корта. Инженеры IBM поработали и над снижением потребления энергии - его удалось уменьшить в 15 раз.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

М.М.Мичурина

Н.Н.Лисовская

ЦИФРОВЫЕ УСТРОЙСТВА И МИКРОПРОЦЕССОРЫ

Курс лекций

Красноярск

2007

УДК 621.382.049.77(07)

Курс лекций по дисциплине «Цифровык устройства и микропроцессоры» для студентов специальности 210300.62 – «Радиотехника» / Сост. М.М.Мичурина, Н.Н.Лисовская. Красноярск: СибФУ, 2007. 213 с.

Лекция № 1

1. ОБЩИЕ МЕТОДЫ ПРЕДСТАВЛЕНИЯ ОПЕРАЦИОННОЙ ИНФОРМАЦИИ В ЭЦВУ

1.1. Краткий исторический обзор

В развитии человеческого общества философы выделяют 3 основных этапа:

- появление языка;
- печать;
- компъютеризация.

Всеобщая компъютеризация – объективная необходимость. За один год в мире публикуется более 80 млн. страниц научно-технической информации, число этих страниц необратимо растёт. При современном темпе протекания научно-технической революции человечеству грозит информационный шок.

Если принять средний возраст жизни человека 62 года, то можно считать, что сменилось 800 поколений. Из них:

– 650 жили в пещерах;
– 70 имеют письменность;
– 6 пользуются печатным словом;
– 2 имеют электромотор;
– 1 работает с вычислительной техникой.

Вычислительная техника – объективный помощник человека, сегодня трудно представить свою жизнь без компъютера.

Анализируя раннюю историю вычислительной техники, нередко вспоминают механическое счётное устройство «абак» (4-ое тысячелетие до н.э.) Изначально – это глиняная дощечка с желобами, в которых раскладывались камни, однако, вряд ли можно назвать это предшественником, поскольку здесь нет автоматического выполнения вычислений. Одно из первых устройств – счёты, сохранившееся до сих пор.

Первое в мире автоматическое устройство для операции сложения было создано на базе механических часов. В 1623 г. его разработал Вильгельм Шикард (Германия), это так называемые суммирующие часы. В 1642 г. Французский учёный Блез Паскаль разработал довольно компактное суммирующее устройство, первый в мире механический калькулятор, который мог выполнять четыре основных арифметических операции. Основным элементом в нём было зубчатое колесо, изобретение которого уже было событием в развитии вычислительной техники. В 1671 г. немецкий философ и математик Густав Лейбниц разработал более совершенный вариант арифмометра, также выполнявшего четыре арифметических действия.

В арифмометрах операции над числами выполнялись с помощью колёс, которые при добавлении единицы поворачивались на 36 градусов и с помощью штифта приводили в движение следующее по старшинству колесо всякий раз, когда цифра 9 переходила к цифре 0 (накапливался десяток). Однако механические устройства громоздки, дороги и слишком инерционны.

Большинство механических устройств хранили лишь текущий результат. В качестве памяти использовался любой писчий материал, например, в древности – глиняная дощечка, позднее бумага. Пока скорость выполнения операций была небольшой и память использовалась медленно, отсутствовала заинтересованность в механизации управления последовательностями операций. XVIII и XIX века были временем, когда бурно развивались математика и астрономия, эти науки требовали длительных и трудоёмких вычислений.

Предвестники механизации управления последовательностями операций появились в ткацком станке изобретателя Жозефа Мари Жаккарда образца 1804 г. Он использовал перфокарточный метод управления механизмом, т.е. это был прообраз гибкого программирования.

Идея использования вычислительного устройства с программным управлением была впервые высказана английским математиком Чарльзом Бэббиджем ещё в 1833 г., однако реализовать её он не сумел. Он использовал идею Жаккарда при разработке замечательного устройства, которое он назвал «Аналитической машиной». Планировалось, что кроме вычислений машина будет выдавать результаты – печатать их на негативной пластине для фотопечати. Технические трудности не позволили ему до конца реализовать свой проект. Он умер в 1871 г., оставив более 37 кв. м подробнейших чертежей. Следует отметить, что вариант такой машины был всё же реализован другим учёным, С.Шойцом, в 1853 г. Многое из того, что известно о машине Жаккарда, дошло до нас благодаря научным трудам одарённого математика-любителя Огасты Ады Байрон (графини Лавлайс), дочери поэта лорда Байрона, Её называют первым программистом. В 1843 г. она перевела статью об Аналитической машине, написанную одним итальянским математиком, снабдив её собственными комментариями, которые касались потенциальных возможностей машины. Она описала приёмы управления последовательностями вычислений, которые используются в программировании и сейчас. Её именем назван один из языков программирования.

В дальнейшем на протяжении почти столетия ничего похожего на Аналитическую машину не появилось, однако идея использования перфокарт для обработки данных была опробирована довольно скоро. Спустя 20 лет после смерти Бэббиджа американский изобретатель Герман Холлерит создал электромеханическую счётную машину – табулятор, в которой перфокарты использовались для обработки результатов переписи населения, проводившейся в США в 1890 г. Табулятор получил столь широкое распространение, что Холлериту пришлось создавать собственную фирму. В конце концов, эта фирма превратилась в знаменитую корпорацию IBM (Internetional Business

Machines), которая сделала перфокарты стандартным средством программирования.

В 1938 г. центр разработок ненадолго смещается из Америки в Германию, где К.Цузе создаёт машину, оперирующую в отличие от своих предшественников, не десятичными числами, а двоичными. Если обратиться к истории появления двоичного кода, то уместно вспомнить, что двоичная система счисления была предложена в 1666 г. Г Лейбницем. Он пришёл к ней, занимаясь философской концепцией единства и борьбы противоположностей. Первая машина К.Цузе также была механической, но уже в 1941 году создаётся электромеханическая машина, арифметическое устройство которой выполнено на базе *реле*. В 1944 году в США Г.Айзен спроектировал машину, также работающую на реле, но оперирующую с данными в десятичной форме. Становится очевидным, что оптимальнее вместо реле использовать вакуумные лампы (первый триод был изобретён ещё в 1906 году Л.Форестом). В 1946 году в США в университете города Пенсильвания была создана первая универсальная ЭВМ – ENIAC. Она весила 30 тонн, занимала площадь 200 кв. м и содержала 18 тысяч ламп. С этой машиной связана разработка одной из важнейших идей вычислительной техники – *принципа хранимой программы*. Считается, что эта идея принадлежит математику Джону фон Нейману, хотя он скорее описал в статье то, что предложили талантливые инженеры-разработчики этой машины. Впрочем, история развития вычислительной техники, как и вся история, тоже изобилует драматическими ситуациями.

Следующая машина, названная EDVAC, разработана в 1951 году. В ней применяется двоичная арифметика и используется оперативная память, построенная на ультразвуковых ртутных линиях задержки. Память могла хранить 1024 слова размерностью в 44 двоичных разряда.

До середины 80-х годов, говоря о вычислительных машинах, активно пользовались термином «поколение». Чем же определяются поколения машин? – Видом элементной базы и реализуемой в машине архитектурой. Приведём краткие качественные характеристики этим поколениям.

1-е поколение (1945 – 1954 гг.) – активно развиваются фоннеймановские структуры. Машины этого поколения работали на ламповой элементной базе, что требовало больших энергетических затрат и имело невысокую надёжность. В СССР в это время были разработаны вычислительные машины типа МЭСМ, БЭСМ под руковдоством академика А.С.Лебедева (8000 операций в секунду). Большой влад в развитие отечественной вычислительной техники внесли академики Келдыш М.В., Глушков В.М., Семенихин В.С. Программы для машин первого поколения уже можно было составлять не на машинном языке, а на языке ассемблера.

2-е поколение (1955 – 1964 гг.). Вычислительные машины второго поколения были выполнены на полупроводниковых элементах. Совершенствование таких машин происходило по двум направлениям: создавались машины-гиганты, требующие больших помещений, систем охлаждения, дорогостоящие, поэтому необходимо было эксплуатировать эти машины круглосу-

точно. Для этих машин были разработаны 2 режима работы: пакетный и разделения времени. В пакетном режиме решается только одна задача, затем после окончания – другая и т.д. В режиме разделения времени одновременно решалось несколько задач. В это время появляются мини-ЭВМ, они являются не такими мощными, как ЭВМ-гиганты, но они значительно дешевле, меньше по габаритам (приблизительно, как письменный стол).В СССР это машины марки «Наири», «Мир», «Минск 22», «Минс 32», «Урал 14» и т. д. Появились языки высокого уровня – Algol, FORTRAN, COBOL. С появлением языков высокого уровня появились компиляторы для них.

3-е поколение (1965 – 1970 гг.) Вместо транзисторов в различных узлах стали использоваться интегральные микросхемы, это повысило производительность, снизило габариты и стоимость. Программное обеспечение становится дорогим, поэтому появляется тенденция к созданию семейств ЭВМ, то есть машины становятся совместимыми снизу вверх на программно-аппаратном уровне.

4-е поколение (1970 – 1984 гг.). В эти годы активно разрабатываются и производятся большие и сверхбольшие интегральные схемы (БИС и СБИС), позволяющие разместить на одном кристалле десятки тысяч элементов В начале 70-х годов был выпущен первый микропроцессор. В мире вычислительной техники появилось ещё одно направление – микропроцесорное.

5-е поколение можно назвать микропроцессорным. Что же такое микропроцессор (МП)? МП – это программно-управляемое устройство, предназначенное для обработки цифровой информации и управления процессом этой обработки, выполненное в виде одной или нескольких интегральных схем с высокой степенью интеграции. МП способен выполнять под программным управлением обработку информации, включая ввод и вывод, принятие решений, арифметические, логические и некоторые другие операции.

Машины 5-го поколения, конечно, повысят быстродействие, но и получат новые возможности: взаимодействие с помощью человеческой речи и графического изображения, способность самообучаться, делать логические суждения, вести беседу с человеком и т. д.

1.2. Общая характеристика микропроцессора, основные особенности современных микропроцессорных комплектов. Области применения

История микропроцессоров началась в 1971 году, когда фирма Intel выпустила первый микропроцессор. Он имел разрядность данных 4 бита и тактовую частоту 108 КГц, адресовал 640 байт памяти, содержал 2300 транзисторов.

В 1974 году появился 8-разрядный процессор **I8080** (в нашей стране это КР580ВМ80), который стал весьма популярен, он имел частоту 2 МГц и адресовал 64 Кб памяти. Следующим этапом стал процессор **I8085** (тактовая частота – 5МГц). Он сохранил совместимость с I8080, но в него добавили порт последовательного ввода и внесли некоторые другие изменения.

Вариацию на тему 8080 и 8085 представляет процессор **Z80** фирмы Zilog, имевший более высокую производительность.

Первый 8-разрядный процессор **8086** фирма Intel выпустила в 1978 году, частота 5 МГц (позже появились процессоры 8 и 10 МГц) адресуемая память 1 Мб. Массовое распространение и открытость архитектуры привели к лавинообразному появлению программного обеспечения.

Процессор **80286** появился только в 1982 году, он адресовал до 16 Мб физической памяти. Его принципиальные новшества – защищённый режим и виртуальная память размером до 1 Гб – не нашли массового применения, процессор использовался просто как быстрый 8088.

Класс 32-разрядных процессоров был открыт в 1985 году моделью 80386.

Разрядность шины данных достигла 32 бит, адресуемая память –4 Гб. С этого времени стала заметна тенденция «положительной обратной связи»: на появление нового процессора производители программного обеспечения реагируют выпуском новых интересных программ, которым скоро становится тесно в рамках этого процессора. Появляется новый процессор, но после непродолжительного восторга и его ресурсы «съедают» и т.д. Становится очевидным интересный факт: большие ресурсы расслабляют разработчика ПО, не принуждая его напрягаться в поисках эффективных решений.

История процессора 386 напоминает историю 8086: первую модель с 32-битной шиной данных (386 DX) сменил 386 SX с 16-битной шиной. Это делается с целью адаптации ранее разработанного программного обеспечения.

Процессор **Intel 486** разработан в 1989 году. В нём появляется кэш-память и встроенный математический сопроцессор

В1993 году появились первые процессоры Pentium с частотой 60 и 66 МГц –32-разрядные процессоры с 64-битной шиной данных. Pentium продолжил развитие идеи параллельной обработки. Его внутренний кэш достиг 8 Кбайт для кода и 8 Кбайт для данных. В устройство декодирования и исполнения команд был добавлен второй конвейер. Интерес к процессору со стороны производителей и покупателей сдерживался его очень высокой стоимостью. Кроме того, возник скандал с обнаруженной ошибкой сопроцессора. Хотя фирма Intel математически обосновала невысокую вероятность её появления (раз в несколько лет), она всё-таки пошла на бесплатную замену уже проданных процессоров на исправленные.

Процессоры Pentium с частотой 75, 90 и 100 МГц, появившиеся в 1994 году, представили уже второе поколение процессоров Pentium. В этих процессорах появляются новые команды, для поддержки которых несколько изменили программную модель микропроцессора, Эти команды, получившие название MMX-команд (MultiMediaExtention – мультемедийное расширение системы команд), позволили одновременно обрабатывать несколько единиц однотипных данных.

Параллельно с процессором Pentium развивался и процессор Pentium Pro, который отличался большей динамичностью исполнения команд, в частности был добавлен ещё один конвейер к имевшимся двум у микропроцессора Pentium. За один такт работы микропроцессор стал выполнять до трёх инструкций, совершенствуется его кэш-память. В Pentium Pro реализовали 36-разрядную адресную шину, что позволило адресовать до 64 Гбайт оперативной памяти.

Появление Pentium Pro разделило рынок на два сектора – высокопроизводительных рабочих станций и дешёвых домашних компъютеров.

Главное достоинство процессора Pentium III наличие новых 70 команд, Эти команды дополняют группу MMX-команд, но для чисел с плавающей точкой.

Лекция № 2

1.3. Кодирование операционной информации.

1.3.1.Системы счисления, используемые в вычислительной технике

Все операции в вычислительной технике выполняются над электрическими сигналами, закодированными двоичной цифрой 0 или 1. Поэтому перед занесением в память данные и команды должны быть тем или иным способом преобразованы в двоичную форму. Вычислители «питаются» однообразной пищей, им подавай информацию структурированную, в виде строго организованных последовательностей нулей и единиц, закодированные комбинации которых и составляют **машинный язык.** К счастью, программисту не нужно пытаться постичь значение различных комбинаций двоичных чисел, так как для программирования уже давно используется символический аналог машинного языка, называемый **языком ассемблера.** Он полностью отражает все особенности машинного языка. Именно поэтому, в отличие от языков высокого уровня, язык ассемблера для каждого типа процессора несколько отличается.

Для сокращения записи двоичных чисел (например, вы хотите указать какое-то данное) целесообразно использовать системы счисления с другими основаниями, например, с основанием 8 или 16. Для программиста такое кодирование более удобно. Если в языках высокого уровня применяются десятичные числа, то в языке ассемблера они изображаются в шестнадцатеричном виде. Следовательно, для успешной работы надо приучаться думать «по-шестнадцатеричному». В программах на языке ассемблера числа могут представляться также в двоично-десятичном виде, где каждая десятичная цифра от 0 до 9 представлена одной тетрадой двоичного кода. Десятичные числа, другое их название ВСД-числа (Binary Code Decimal – двоично-десятичный код), наиболее удобны для программирования прикладных задач. Недостаток этих чисел в том, что для них требуется разработка специфических алгорит-

мов. Итак, процессор поддерживает только двоичный и в определённом смысле двоично-десятичный формат. Однако ввод информации с клавиатуры и вывод её на экран осуществляется в символическом виде. Кодирование этой информации производится согласно таблице ASCII, где каждый символ кодируется одним байтом (см. * в конце раздела). Следовательно, при работе с клавиатурой нужно преобразовать символьную информацию к формату, поддерживаемому машинными командами. После такого преобразования нужно выполнить необходимые вычисления и преобразовать результат обратно к символьному виду. Затем следует отобразить информацию на мониторе.

1.3.2. Арифметические операции

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. Группа арифметических целочисленных команд работает с двумя типами чисел:

целыми двоичными числами. Числа могут, иметь знаковый разряд или не иметь такового, то есть быть числами со знаком или без знака;

целыми десятичными числами.

Целое двоичное число – это число, закодированное в двоичной системе счисления. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа, причём среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, – это команды целочисленного умножения и деления mul и div.

Десятичное число – специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырёх бит. При этом каждый байт числа содержит одну или две десятичные цифры в двоично-десятичном коде (BCD – Binary Coded Decimal). Микропроцессор хранит эти числп в двух форматах – упакованном и неупакованном.

Арифметические операции над числами в вычислительных устройствах используют специальные машинные коды: прямой, обратный (инверсия прямого) и дополнительный (обратный код плюс единица). Арифметические операции над числами со знаком производятся в дополнительном коде. Чтобы найти двоичное представление отрицательного числа (т. е. его дополнительный код), надо просто взять его положительную форму, обратить (т. е. инвертировать) каждый бит (заменить1 на 0, а 0 на 1), а затем добавить к полученному результату 1. Конечно, использование дополнительного кода приводит к тому, что отрицательное число становится трудно расшифровать. Однако только что описанной процедурой можно воспользоваться для того, чтобы получить положительную форму отрицательного числа, записанного в дополнительном коде.

*1.3.3. ASCII-код

С целью стандартизации в компьютерах используется американский национальный стандартный код для обмена информацией ASCII (Amerikan National Standard Code for Information Interchange). Наличие стандартного кода облегчает обмен данными между различными устройствами компьютера. Восьмибитовый расширенный ASCII-код обеспечивает представление 256 символов, включая символы для национальных алфавитов.

В ASCII-коде шестнадцатеричные цифры кодируются по следующему принципу в младшей тетраде для чисел от 0 до 9 помещается соответствующее двоичное число, а в старшей – двоичное число 0011 (признак ASCII-кода), для букв a, b, c, d, e, f – в младшей тетраде двоичные коды 0001, 0010, 0011, 0101 и т. д., а в старшей тетраде – код 0100 для заглавных букв и код 0110 – для строчных букв. Например:1h в ASCII – коде 31h, 4h – 34h; Ah – 41h; bh – 62h; Fh – 46h.

Из выше сказанного следует, что для преобразования кода ASCII в шестнадцатеричное число от 0 до 9 можно:

- выполнить двоичное вычитание (код ASCII)h – 30h;
- обнулить старшую тетраду байта в коде ASCII.

Не следует забывать после записи шестнадцатеричной цифры ставить символ «h». Это делается для того, чтобы транслятор мог отличить в программе одинаковые по форме записи десятичные и шестнадцатеричные числа. Таблица ASCII – кодов приведена в приложении 1.

2. ПРИНЦИПЫ ПОСТРОЕНИЯ, ОРГАНИЗАЦИИ И УПРАВЛЕНИЯ МИКРОПРОЦЕССОРНЫМ ВЫЧИСЛИТЕЛЕМ

2.1.Общие принципы построения микропроцессорного вычислителя. Структурная схема микроЭВМ

В основу построения вычислительных устройств заложены структуры фон Неймана (1902 – 1957 г.).

ЭВМ обрабатывает информацию так же, как и человек. Типичное вычислительное устройство состоит их процессора, памяти и устройств ввода-вывода (рис. 1).

«Сердцем» вычислительного устройства является центральный процессор, в состав которого входят:

- устройство управления выборкой команд из памяти и их выполнением;
- регистры, осуществляющие временное хранение данных и состояний процессора;
- схемы для управления и связью с памятью и устройствами ввода-вывода.

Устройства ввода обеспечивают считывание информации (данные и программа) с носителей информации (клавиатуры, магнитных дисков, телетай-

па и т. д.). Эти же устройства представляют информацию в необходимую для процессора форму.

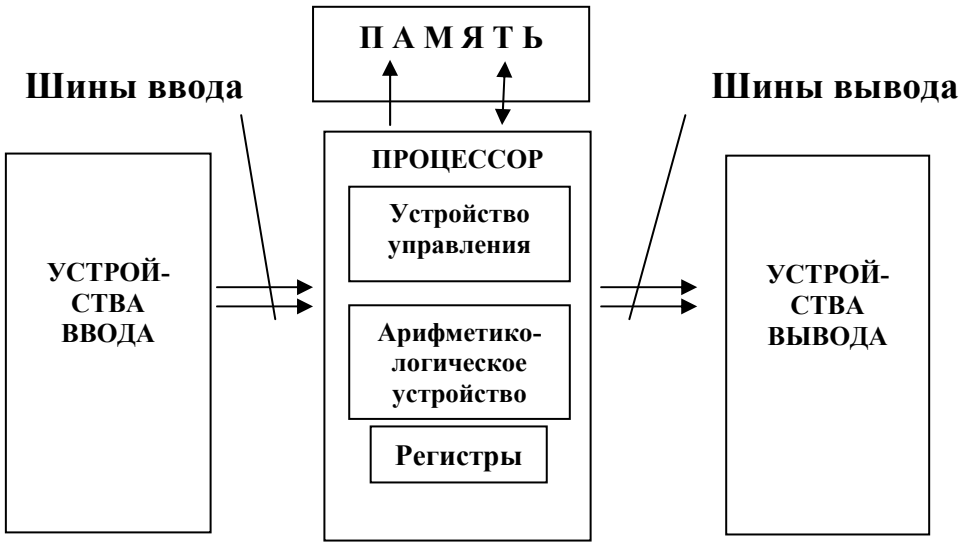


Рис.1. Структурная схема вычислительного устройства

Память представляет собой устройство, хранящее:

- совокупность команд, образующих программу;
- обрабатываемые данные.

Память состоит из ячеек, хранящих набор бит, составляющих информационные слова. Ячейки нумеруются числами, называемыми адресом ячейки. Если необходимо записать или считать информационное слово. Следует подать на шину адреса требуемый адрес. Таким образом, процессор связан с памятью однонаправленной шиной адреса (ША) и двунаправленной шиной данных (ШД). Кроме основной памяти в компъютерах имеется внешняя память для долговременного хранения программ и данных. Для этого используются накопители на магнитных дисках и т. д.

Устройства вывода представляют результаты обработки информации в форме, удобной для восприятия.

Лекция № 3

2.2. Классификация микропроцессоров

Микропроцессор характеризуется очень большим числом параметров и качеств, поскольку обеспечивает эффективное автоматическое выполнение операций обработки цифровой информации в соответствии с заданным алгоритмом.

По числу БИС в микропроцессорном комплекте различают микропроцессоры однокристальные и многокристальные секционированные (микропрограммируемые). Однокристальные получаются при реализации всех аппаратурных средств в виде одной БИС, они имеют фиксированную систему

команд. Секционированные микропроцессоры получаются в том случае, когда в виде БИС реализуются части (секции) структуры процессора при функциональном разбиении её вертикальными плоскостями. Для построения многоразрядных микропроцессоров при параллельном включении секций МП в них добавляются средства «стыковки», чем достигается возможность работы с требуемой разрядностью слов. (Принцип микропрограммирования приведён в конце раздела).

По назначению различают универсальные и специализированные микропроцессоры.

Универсальные микропроцессоры предназначаются для решения широкого круга задач и применения в различных вычислительных системах

Специализированные процессоры включают в себя цифровые сигнальные процессоры и микроконтроллеры. Цифровые сигнальные процессоры – рассчитаны на обработку в реальном времени цифровых потоков, образованных путем оцифровывания аналоговых сигналов. Это обусловливает их сравнительно малую разрядность и преимущественно целочисленную обработку. Однако современные сигнальные процессоры способны проводить вычисления с плавающей точкой над 32 – 40-разрядными операндами. Кроме того, появился класс медийных процессоров, представляющих собой законченные системы для обработки аудио- и видеоинформации.

Микроконтроллеры – обладают наибольшей специализацией и разнообразием функций. Используются во встроенных системах управления, в том числе в бытовых приборах.

CISC (Complete Instruction Set Computer) – Микропроцессоры с полным набором команд. Такие микропроцессоры используют программирование на языке команд и выполняются на одном кристалле. Они имеют фиксированную разрядность слова данных и фиксированный набор команд. Каждая команда представляет собой определенную последовательность микрокоманд. На ее выполнение может затрачиваться несколько машинных циклов (обращений к внешней памяти), каждый из которых включает в себя 1…5 (а иногда и более) рабочих тактов.

RISC (Reduce Instruction Set Computer) – Микропроцессоры с сокращенным набором команд. Основная особенность RISC-процессоров состоит в использовании небольшого набора часто используемых команд одинакового формата, которые могут быть выполнены за один командный цикл (такт). Более сложные, редко используемые команды реализуются на программном уровне. Однако за счет значительного повышения скорости выполнения сокращенного набора команд средняя производительность RISC-процессоров оказывается выше, чем у CISC- процессоров.

2.3. Принцип микропрограммного управления

Секционированные микропроцессоры работают не на программном, а на микропрограммном уровне, поэтому не имеют своей системы команд. Они

могут использовать систему команд какого-либо однокристального процессора, разбивая его команды на микрокоманды или просто разрабатываются программы на микропрограммном уровне. Микропрограммное устройство управления состоит из микропрограммной памяти и схемы формирования адресов микрокоманд. Микрокоманда содержит в общем случае три поля: поле управления генерацией адреса следующей микрокоманды, поле управления длительностью такта, поле управления операционной частью микропроцессора.

Для эффективного применения комплекта К1804 необходимо знать принципы микропрограммного управления в ЭВМ. В микропрограммной ЭВМ для выполнения различных операций используется однородная последовательность микрокоманд. Выполнение машинной команды интерпретируется набором микрокоманд, образующих микропрограмму. Элементарные функции, реализуемые при выполнении микрокоманды, инициируются микрокомандами. Обычно микрокоманда выполняет две главные функции: определение и управление всеми микрооперациями, определение и управление адресом –следующей микрокоманды. Первая функция связана с выбором операндов для АЛУ, заданием операции АЛУ, выбором получателя результата АЛУ, управлением переносом, сдвигом, прерываниями, вводом и выводом данных и т.д. Вторая функция – выбор источника адреса следующей микрокоманды, иногда этот адрес определяется явно.

В блоке микропрограммного управления (БМУ) имеется микропрограммная память **M x N**. Диапазон адресов составляет от 0 до ***N-1*** . Каждое слово (микрокоманда) состоит из ***M*** бит, разделенных на поля различной длины. Определение полей называется форматом микрокоманды. В типичной ЭВМ микрокоманда содержит следующие поля: 1 — общего назначения, 2 — адрес перехода (адрес микропрограммной памяти), 3 — функция управления адресом следующей микрокоманды, 4 — управление прерыванием, 5 — управление выбором синхронизации, 6 — управление переносом, 7 — управление источниками операндов АЛУ, 8 — управление функцией АЛУ, 9 — управление получателем результата АЛУ.

Более подробно вопросоы микропрограммирования будут рассмотрены далее в разделе 9 при изучении секционированных микропроцессоров.

3. АРХИТЕКТУРА 16-РАЗРЯДНЫХ ПРОЦЕССОРОВ I8086/88

3.1. Архитектура микропроцессора I8086/88 (К1810ВМ86/88)

3.1.1. Введение

Микропроцессор 8086 фирмы Intel содержит на кристалле около 29000 транзисторов и производится по МОП-технологии. Процессоры I8086 и выпущенный годом позже его модифицированный вариант I8088 выполняют 8/16-битные логические и арифметические операции, включая умножение и деление, операции со строками и операции ввода-вывода. Процессоры имеют 20-

разрядную шину адреса (ША), позволяющую адресовать 1 Мб памяти. Шина данных у микропроцессоров I8086, у I8088 – 8-разрядная. Это сокращение, сделанное с целью удешевления системы, поскольку предоставляет возможность использования ранее разработанного программного обеспечения, оборачивается некоторым снижением производительности: I8086 за счёт большей разрядности ШД работает примерно на 20-60 % быстрее, чем I8086 с такой же тактовой частотой. С программной точки зрения эти процессоры идентичны, их система команд и набор регистров включены во все процессоры PC-совместимых процессоров. От процессора I8086 пошло общее обозначение семейства: x 86.

Процессоры поддерживают аппаратные и программные прерывания, предусмотрено использование математического сопроцессора, повышающего производительность вычислений.

В процессорах применена конвейерная архитектура, позволяющая выполнить выборку кодов команд из памяти и их дешифрацию во время выполнения внутренних операций. Это сокращает время простоя его операционных узлов.

Лекция № 4

3.1.2. Структурная схема

Структурная схема микропроцессора представлена на рисунке 2.

Структуру микропроцессора можно условно разбить на три части: операционное устройство, устройство сопряжения с шиной и управляющее устройство.

Операционное устройство. В операционном устройстве декодируются команды и обрабатываются данные, т. е. осуществляется сам процесс выполнения

команды. В состав операционного устройства входят: 16-разрядное арифметико-логическое устройство (АЛУ) с тремя регистрами временного хранения и регистром признаков и восемь 16-разрядных регистров. АЛУ выполняет различные арифметические и логические операции, а также операции сдвига.

Поскольку АЛУ является комбинационной схемой, регистры временного хранения необходимы для хранения операндов во время выполнения операций, программисту они не доступны.

Блок регистров состоит из четырёх 16-разрядных регистров A, B, C, D, двух указательных (SP,BP) и двух индексных регистров (SI, DI). Для первых четырёх регистров общего назначения (РОНов) существует возможность их использования как раздельных восьмиразрядных. Если РОН используется как 16-разрядный – добавляется буква X, если как 8-разрядный – символ H для левой группы (от английского слова high) и буква L для правой группы (от слова low), т. е. старшие и младшие байты. Аккумулятор (A) называют собственным регистром АЛУ, его функции довольно разнообразны. *Операционное устройство*. В операционном устройстве декодируются команды и обрабатываются данные, т. е. осуществляется сам процесс выполнения

команды. В состав операционного устройства входят: 16-разрядное арифметико-логическое устройство (АЛУ) с тремя регистрами временного хранения и регистром признаков и восемь 16-разрядных регистров. АЛУ выполняет различные арифметические и логические операции, а также операции сдвига.

Поскольку АЛУ является комбинационной схемой, регистры временного хранения необходимы для хранения операндов во время выполнения операций, программисту они не доступны.

Блок регистров состоит из четырёх 16-разрядных регистров A, B, C, D, двух указательных (SP,BP) и двух индексных регистров (SI, DI). Для первых четырёх регистров общего назначения (РОНов) существует возможность их использования как раздельных восьмиразрядных. Если РОН используется как 16-разрядный – добавляется буква X, если как 8-разрядный – символ H для левой группы (от английского слова high) и буква L для правой группы (от слова low), т. е. старшие и младшие байты. Аккумулятор (A) называют собственным регистром АЛУ, его функции довольно разнообразны. *Операционное устройство*. В операционном устройстве декодируются команды и обрабатываются данные, т. е. осуществляется сам процесс выполнения

команды. В состав операционного устройства входят: 16-разрядное арифметико-логическое устройство (АЛУ) с тремя регистрами временного хранения и регистром признаков и восемь 16-разрядных регистров. АЛУ выполняет различные арифметические и логические операции, а также операции сдвига.

Поскольку АЛУ является комбинационной схемой, регистры временного хранения необходимы для хранения операндов во время выполнения операций, программисту они не доступны.

Блок регистров состоит из четырёх 16-разрядных регистров A, B, C, D, двух указательных (SP,BP) и двух индексных регистров (SI, DI). Для первых четырёх регистров общего назначения (РОНов) существует возможность их использования как раздельных восьмиразрядных. Если РОН используется как 16-разрядный – добавляется буква X, если как 8-разрядный – символ H для левой группы (от английского слова high) и буква L для правой группы (от слова low), т. е. старшие и младшие байты. Аккумулятор (A) называют собственным регистром АЛУ, его функции довольно разнообразны. *Операционное устройство*. В операционном устройстве декодируются команды и обрабатываются данные, т. е. осуществляется сам процесс выполнения

команды. В состав операционного устройства входят: 16-разрядное арифметико-логическое устройство (АЛУ) с тремя регистрами временного хранения и регистром признаков и восемь 16-разрядных регистров. АЛУ выполняет различные арифметические и логические операции, а также операции сдвига.

Поскольку АЛУ является комбинационной схемой, регистры временного хранения необходимы для хранения операндов во время выполнения операций, программисту они не доступны.

Блок регистров состоит из четырёх 16-разрядных регистров A, B, C, D, двух указательных (SP,BP) и двух индексных регистров (SI, DI). Для первых четырёх регистров общего назначения (РОНов) существует возможность их ис-

пользования как раздельных восьмиразрядных. Если РОН используется как 16-разрядный – добавляется буква X, если как 8-разрядный – символ H для левой группы (от английского слова high) и буква L для правой группы (от слова low), т. е. старшие и младшие байты. Аккумулятор (A) называют собственным регистром АЛУ, его функции довольно разнообразны. *Операционное устройство*. В операционном устройстве декодируются команды и обрабатываются данные, т. е. осуществляется сам процесс выполнения
команды. В состав операционного устройства входят: 16-разрядное арифметико-логическое устройство (АЛУ) с тремя регистрами временного хранения и регистром признаков и восемь 16-разрядных регистров. АЛУ выполняет различные арифметические и логические операции, а также операции сдвига.

Поскольку АЛУ является комбинационной схемой, регистры временного хранения необходимы для хранения операндов во время выполнения операций, программисту они не доступны.

Блок регистров состоит из четырёх 16-разрядных регистров A, B, C, D, двух указательных (SP,BP) и двух индексных регистров (SI, DI). Для первых четырёх регистров общего назначения (РОНов) существует возможность их использования как раздельных восьмиразрядных. Если РОН используется как 16-разрядный – добавляется буква X, если как 8-разрядный – символ H для левой группы (от английского слова high) и буква L для правой группы (от слова low), т. е. старшие и младшие байты. Аккумулятор (A) называют собственным регистром АЛУ, его функции довольно разнообразны.

Кроме того, что в РОНах могут храниться данные, участвующие в операциях, каждый регистр имеет ещё своё специфическое назначение, неявно подразумеваемое в некоторых командах:

AX – умножение, деление, ввод и вывод слова;

AL – умножение, деление, ввод и вывод байта, десятичная арифметика, трансляция (XLAT);

AH – умножение и деление слова;

BX – трансляция, адресация по базе;

CX – счётчик циклов и указатель длины строковых операций;

ДХ – умножение и деление слова, ввод и вывод с косвенной адресацией.

Регистры – указатели SP, BP и индексные регистры SI и DI главным образом хранят адресную информацию.

SP (Stack Pointer – указатель стека) – операции со стеком;

BP (Base Pointer – указатель базы) – обращение к стеку;

SI (Sours Index – индекс источника) – смещение адреса в текущем сегменте данных DS;

DI (Destination Index – индекс приёмника) – смещение адреса в дополнительном сегменте ES.

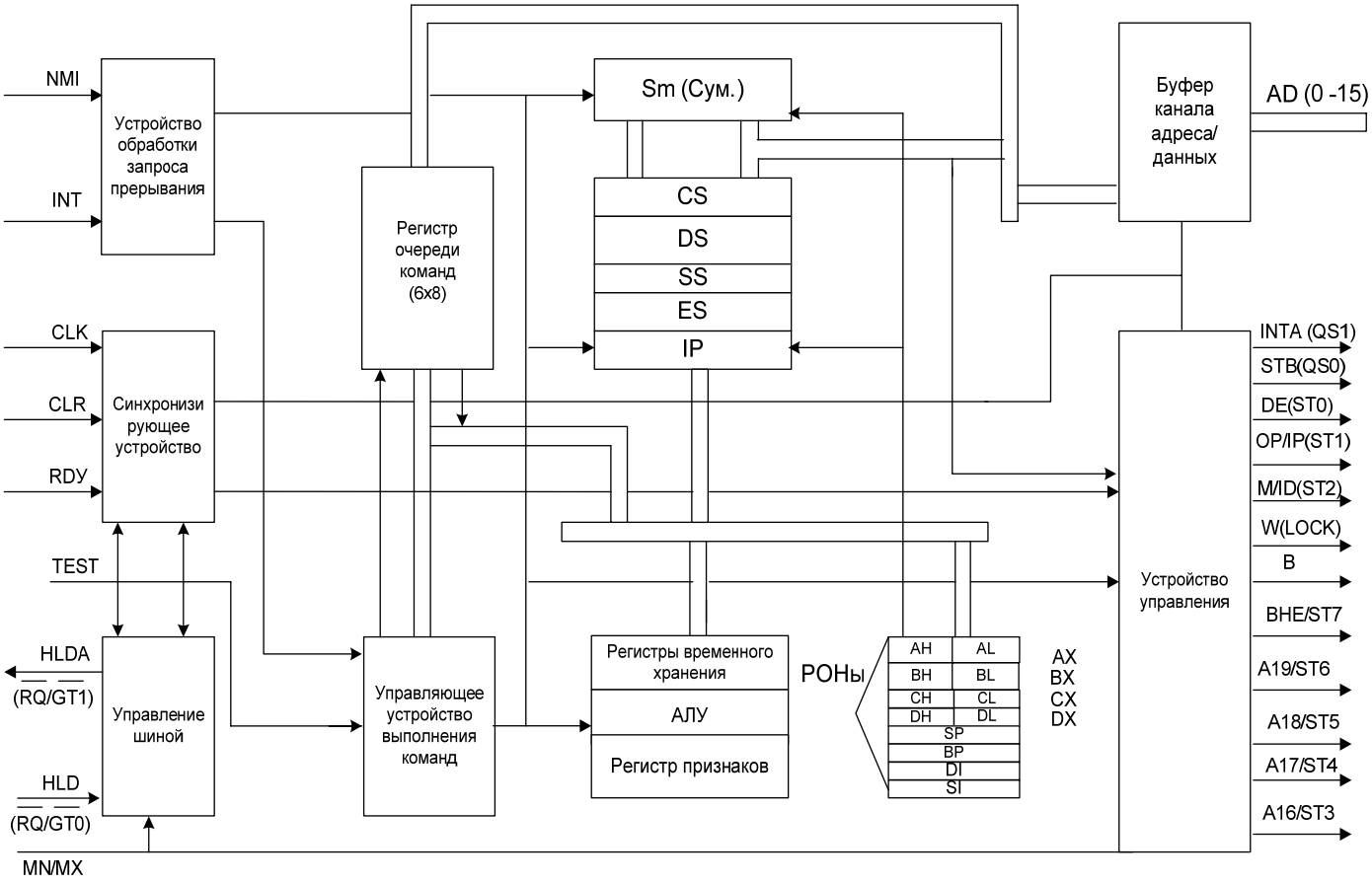


Рис. 2. Структурная схема микропроцессора I 8086

Регистр признаков (F) состоит из нескольких триггеров, называемых флажками (рис. 3). Он хранит признаки результатов выполнения арифметических и логических операций и управляющие триггеры. Команды пересылок на флаги не воздействуют. Назначение флагов:

CF (Carri Flag) – флаг переноса (заёма) старшего бита в арифметических операциях;

PF (Pariti Flag) **–** флаг паритета, устанавливается при чётном числе единиц в результате;

15р	14р	13р	12р	11р	10р	9р	8р	7р	6р	5р	4р	3р	2р	1р	0р
				OF	**DF**	**IF**	**TF**	**SF**	**ZF**		**AF**		**PF**		**CF**

Рис. 3. Регистр флагов

AF (Auxiliary Flag) – флаг дополнительного переноса (заёма) в тетраде для десятичной арифметики;

ZF (Zero Flag) – флаг нулевого результата.

SF (Sign Flag) – флаг знака. Если в триггере устанавливается единица, флаг отрицательный.

TF (Trap Flag) – флаг трассировки (пошагового режима). При установке этого флага после выполнения каждой команды вызывается внутреннее прерывание 1-го типа (INT 1).

IF (Interrapt Flag) – флаг управления прерываниями. При единичном значении разрешается выполнение маскируемых аппаратных прерываний.

DF (Direction Flag) – флаг управления направлением в строковых операциях. При единичном значении индексные регистры, участвующие в строковых операциях, автоматически декрементируются на количество байт операнда, при нулевом – инкрементируются.

OF (Overflow Flag) – флаг переполнения. Устанавливается, если результат арифметической команды не умещается в операнде назначения.

Устройство сопряжения с шиной осуществляет связь с ВУ, обеспечивает выборку команд и данных из памяти, формирует очередь команд.

В состав устройства сопряжения с шиной входят: сегментные регистры CS, DS, SS и ES, указатель команд IP, сумматор адреса, регистры очереди команд и буферы, обеспечивающие связь с шинами адреса и данных.

Блок очереди команд у I8086 имеет 6-байтную организацию. Блок предварительной выборки при наличии двух свободных регистров в очереди старается её заполнить в то время, когда внешняя шина процессора не занята операциями обмена. Очередь у процессора I8088 сокращена до 4 байт, а предварительная выборка выполняется уже при наличии одного свободного байта. Очередь обнуляется при выполнении любой команды передачи управления. Конвейерный принцип, при котором пока одна команда исполняется в операцион

ном устройстве, другая команда выбирается из памяти, повышает скорость выполнения программы.

Буферы представляют собой 16 двунаправленных усилителя с тремя выходными состояниями для линий AD0 – AD15. Для линий A16 – A19 буферы содержат четыре однонаправленных усилителя с тремя выходными состояниями.

Особое место занимают 16-битные регистры – указатели сегментов CS (Code Segment – сегмент кодов команд), DS (Date Segment – сегмент данных), ES (Extra Segment – дополнительный сегмент данных), SS (Stack Segment – сегмент стека). Содержимое этих регистров не может быть изменено никакими командами, кроме команд их загрузки.

Регистры CS, DS, SS, ES предназначены для указания начала области (сегмента) памяти и используются при обращении к памяти для вычисления адресов ячеек.

CS – регистр сегмента программы, определяет начальный адрес сегмента памяти, в котором располагается программа. Выборка очередной команды осуществляется относительно содержимого CS с использованием значения указателя команд IP.

DS – регистр сегмента данных, определяет начальный адрес текущего сегмента данных. Вычисление физического адреса в этом сегменте зависит от способа адресации.

ES – регистр дополнительного сегмента данных

SS – регистр сегмента стека, определяет начало стекового сегмента и используется в командах обращения к стеку, при обработке прерываний и подпрограмм. (Стек – это сегмент памяти, работающий по принципу LIFO (Last in first out – вошедший последним выходит первым)).

IP – указатель команд, является смещением в сегменте кода относительно начала сегмента.

Более подробно описание функций этих регистров приведено в следующем разделе.

Сумматор адресов формирует 20-разрядный физический адрес, суммируя два логических адреса: базовый логический адрес, находящийся в одном из сегментных регистров и смещение в сегменте. Более подробно описание описание этого процесса приведено в следующем разделе.

Буферы обеспечивают связь с шинами адреса и данных для передачи логического нуля и единицы о обладая возможностью переходить в так называемое «третье», высокоимпедансное состояние.

Управляющее устройство дешифрирует команды, формирует управляющие сигналы, обеспечивающие функционирование вычислительной системы, а также микроприказы для внутренних действий МП.

Лекция № 5

3.2. Сегментация памяти, вычисление адресов

Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к четырём сегментам: кода CS, данных DS, стека SS и дополнительного сегмента данных ES (для 32-х разрядных процессоров серии Intel вводятся ещё два дополнительных сегмента FS и GS). В компъютере программа никогда не знает, по каким физическим адресам будут размещены её сегменты, этим занимается её операционная система. Она размещает сегменты программы в оперативной памяти по определённым физическим адресам, после чего помещает значения этих адресов в соответствующие сегментные регистры. В самостоятельно разрабатываемом вычислителе проблема инициализации сегментных регистров и распределение ресурсов памяти лежит на разработчике аппаратуры и программного обеспечения. Внутри сегмента программа обращается к адресам относительно начала сегмента, то есть, начиная с нуля и заканчивая адресом, равным размеру сегмента (его максимальный объём может быть 64 кб). Этот относительный адрес или смещение, используемое микропроцессором для доступа к данным внутри сегмента, называется эффективным (EA) или исполнительным. (В руководствах фирмы Intel термин «эффективный адрес» применяется в контексте машинного языка, а термин «смещение» (offset) – в контексте языка ассемблера. Слово «смещение» (displasement) означает величину, которая прибавляется к содержимому регистра(ов) для образования EA.)

Сегменты могут быть смежными, следовать с интервалом или перекрываться. Каждый сегмент начинается на 16-байтной границе(так называемая граница параграфа).

Для вычисления 20-разрядного физического адреса содержимое каждого сегментного регистра (CS, DS, SS, ES) рассматривается как 16 старших разрядов A4–A19 начального адреса соответствующего сегмента. Младшие разряды A0–A3 полагаются равными нулю и приписываются справа при вычислении физического адреса. Вычисление производится в сумматоре и состоит в сложении 20-разрядного начального адреса сегмента с 16-разрядным смещением, которое дополняется четырьмя старшими разрядами, равными нулю (рис. 4).

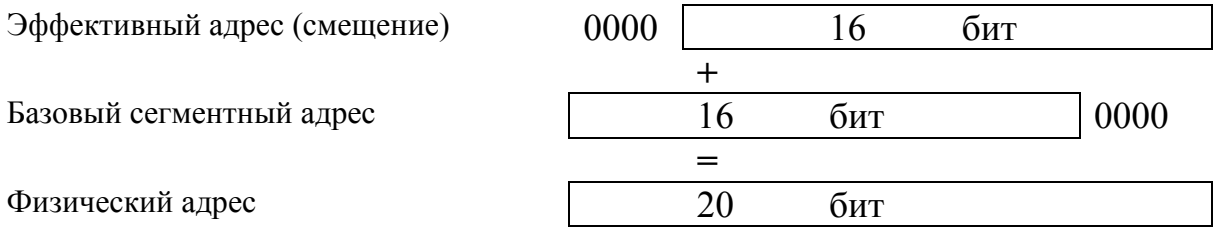


Рис. 4. Формирование физического адреса

Наличие сегментных регистров обеспечивает следующие преимущества:

ёмкость памяти может доходить до 1 Мбайт, хотя команды оперируют 16-битными адресами;

сегменты кода, данных и стека могут иметь длину более 64 Кбайт благодаря использованию нескольких сегментов кода, данных или стека.

Источники смещения для различных типов обращения к памяти приведены в таблице 1. Из таблицы видно, что команды всегда выбираются из сегмента кода в соответствии с логическим адресом CS : IP. Стековые команды всегда обращаются к сегменту стека по адресу SS:SP. Если при вычислении адреса EA используется регистр BP, то обращение производится также к стековому сегменту, однако в этом случае стековый принцип работы «вошедший последним выходит первым» игнорируется и ячейки стекового сегмента рассматриваются просто как ячейки памяти. Обычно операнды располагаются в текущем сегменте данных DS и обращение к ним организуется по адресу DS:EA. Однако, можно обратиться к переменным, находящимся в других сегментах. Цепочку (строку) – источник, как правило, располагают в текущем сегменте данных, а её смещение задаётся регистром SI. Цепочка-получатель находится в дополнительном сегменте ES, а смещение – в регистре DI.

Таблица 1

Тип ссылки к памяти	Сегмент по умолчанию	Альтернативный сегмент	Смещение
Выборка команды	CS	Нет	IP
Стековая операция	SS	Нет	SP
Переменная	DS	CS, SS, ES	EA
Цепочка-источник	DS	CS, SS, ES	SI
Цепочка-приёмник	ES	Нет	DI
BP как базовый регистр	SS	CS, SS, DS	EA

Смена сегментного регистра осуществляется с помощью однобайтового префикса замены сегмента 001SR110, который ставится перед первым байтом команды. Поле SR содержит код сегментного регистра: 00 – ES, 01 – CS, 10 – SS, 11 – DS.

3.3. Организация ввода – вывода во внешние устройства

Ввод и вывод информации в порты можно осуществить двумя способами: с использованием общего с памятью адресного пространства, т.е. с отображением на память (совмещённая адресация) и с использованием адресного пространства ввода – вывода (раздельная адресация). При раздельной адресации применяются специальные команды IN (ввод) и OUT (вывод), обеспечивающие передачу между аккумулятором и портами. При выполнении этих команд в минимальном режиме вырабатывается сигнал M/IO = 0, позволяющий вместе с сигналами WR и RD сформировать системные сигналы IOW и IOR для организации записи данных в порт и чтения из порта. В максимальном режиме системные сигналы IOW и IOR вырабатываются системным контроллером I8288 на базе сигналов состояния S0, S1, S2.

3.4. Назначение выводов микропроцессоров

Расположение выводов микропроцессоров I8086 и 8088 совпадает, отличие заключается лишь в использовании старшего байта адресной шины, в процессоре I8088 шина данных однобайтная, поэтому мультиплексирован только младший байт ША/ШД. Кроме этого, сигнал M/IO имеет инверсное значение относительно процессора I8086. Назначение выводов микропроцессора I8086 приведено в таблице 2. В зависимости от уровня сигнала на входе MN/MX процессоры могут работать в минимальном и максимальном режимах. В **минимальном режиме** микропроцессор сам вырабатывает сигналы управления для системы. Этот режим предназначен для построения небольших устройств, не использующих сопроцессоры или другие процессоры. В **максимальном режиме** сигналы управления системной шиной IOR (чтение внешнего устройства), IOW (запись во внешнее устройство), MEMR (чтение памяти), MEMW (запись в память), INTA (подтверждение прерывания) и ALE (или STB – строб записи адреса в регистр-защёлку) вырабатываются контроллером шины I8288 (К1810ВГ88) по сигналам состояния S0-S2.

Таблица 2.

Сигнал	Вх/Вых	Назначение в мин. Режиме	Назнач.в макс. реж.
AD[0:7]	Вх/Вых (z)	Address / Data – мультиплексированные сигналы шины адреса и данных. Адрес присутствует в начале каждого машинного цикла. В I8088 эти сигналы через регистр-защелку выводятся на линии Addr[0:7] шины адреса и двунаправленным буферным регистром соединяются с линиями Data[0:7] шины данных. У процессора 8086 мультиплексирована 16-разрядная шина AD[0:15]	Аналогично
AD[8:15]	Вх/Вых (z)	Address / Data – сигналы шины адреса/данных в микропроцессоре I8086 (не мультипликсированные у процессора I8088).	Аналогично
A[16:19] / S[3:6]	Вых (z)	Address / Status – старшие биты шины адреса, мультиплексированные с сигналами состояния. Адрес присутствует в первом такте каждого машинного цикла (а при адресации ВУ – нули), после чего сменяется признаками состояния. В битах состояния S6=0, S5 отражает состояние флага прерываний IF, S4 и S3 указывают, какой сегментный регистр используется в данном цикле (при обращении к ВУ на S3 =0, на S4=1). В I8086 информация состояния не используется, а эти сигналы через регистр-защелку поступают на линии	Аналогично

		Addr[16:19] шины адреса	
CLK	Вх	Clock – сигнал синхронизации процессора частотой 4,77 МГц (8МГц в Turbo-XT)	Аналогично
RDY	Вх	Аппаратная проверка готовности, используется для синхронизации работы МП и медленно действующей периферии, 1 – есть готовность, 0 – нет готовности,вводятся такты ожидания; проверяется в каждом машинном цикле.	Аналогично
TEST	Вх	Программная проверка готовности. Вход проверяется, если есть команда WAIT, 0 – есть готовность, 1 – нет готовности, вводятся такы ожидания	Аналогично
CLR	Вх	Сброс: регистры DS, ES, SS, флагов, указатель команд IP обнулены, конвейер команд пуст, в регистре CS все разряды устанавливаются в 1.	Аналогично
HLD RQ / GT0	Вх Вх/Вых	HLD – запрос захвата шины от внешней подсистемы (ВУ или контроллера прямого доступа памяти)	RQ/GT0 – Request / Grant – сигнал запроса (Request) и предоставления (Grant) управления локальной шиной, используется для связи с сопроцессором I8087
HLDA RQ / GT1	Вых Вх/Вых	HLDA – подтверждение захвата шины, выдаётся в ответ на сигнал HLD после приостанова вычислительного процесса в МП и перевода шин и некоторых управляющих сигналов в z- состояние.	RQ/GT1 – аналогично RQ/GT0, но с меньшим приоритетом. В I8086 не используется (на него подается лог. «1»)
WR LOCK	Вых (z) Вых	WR – запись, указывает на выполнение цикла записи в память или ВУ.	LOCK – Сигнал монополизации управления шиной, вырабатывается на время выполнения команды по префиксу инструкции LOCK
NMI	Вх	Non Mascable Interrupt – сигнал, высокий уровень которого вызывает немаскируемое прерывание NMI.	Аналогично
INTR	Вх	Interrupt Request – сигнал запроса (высоким уровнем) маскируемого прерывания.	Аналогично
RESET	Вх	Сигнал аппаратного сброса (высоким уровнем). Синхронизированный сигнал сброса поступает через микросхему (генератор тактовых импульсов) I8284.	Аналогично
INTA ALE	Вых	INTA – подтверждение запроса прерывания, формируется в ответ на приня-	QS0, QS1 – Queue Status – состояние

QS0, QS1		тый запрос прерывания INT, выполняет функцию сигнала RD в цикле подтверждения прерывания и стробирует чтение вектора прерывания. ALE (STB) – строб адреса, выдаётся в начале каждого машинного цикла для записи адреса в регистр-защёлку	внутренней очереди команд. Эти сигналы поступают на одноименные входы сопроцессора I8087
DE ST0	Вых (z) Вых (z)	DE – Сигнал разрешения передачи данных шинному формирователю по ШД	ST0–Status – используется совместно с сигналами ST1, ST2 (табл. 3).
OP/IP ST1	Вых (z) Вых (z)	OP/IP (DT/R) – передача/приём данных, определяет направление передачи по шине данных. 1 – запись данных , 0 – чтение. Предназначен для управления шинными формирователями по ШД и действуют на протяжении всего цикла шины.	Это сигналы состояния, идентифицирующие выполняемый шинный цикл. Начало и конец цикла индицируется переходом бит состояния из пассивного (111) в активное состояние и обратно .
M/IO ST2	Вых(z) Вых(z)	M/IO – определяет обращение процессора к памяти или к ВУ и используется для разделения адресного пространства памяти и внешних устройств.	Сигналы поступают на входы контроллера шины I8288, который их декодирует в сигналы управления системной шиной IOR, IOW, MEMR, MEMWR, INTA и ALE .
RD	Вых (z)	Чтение, идентифицирует выполнение цикла чтения из памяти или ВУ.	Аналогично
BHE / ST7	Вых (z)	Byte High Enable / Status 7 (только для процессора 8086) – разрешение старшего байта. Сигнал BHE указывает на присутствие данных на линиях AD[8:15]. Совместное использование BHE и младшей линии адреса A0 для дешифрации адресов позволяет осуществлять передачу слов или отдельных байтов по шине AD (табл. 4). ST7 – резервный выход.	Аналогично
MN/MAX	Вход	Минимальный/максимальный – обеспечивает соответствующий режим работы микропроцессора.	Аналогично

В таблице 3 приведены состояния сигналов на выходах процессора ST0, ST1, ST2, используемых в максимальном режиме, и их декодирование. Они показывают тип машинного цикла, исполняемого в данный момент. Сигналы с этих выводов подаются на системный контроллер I8288 для формирования системных управляющих сигналов.

Таблица 3.

ST0	ST1	ST2	Тип цикла
0	0	0	Подтверждение прерывания (INTA)
0	0	1	Чтение порта (IORD)
0	1	0	Запись в порт (IOWR)
0	1	1	Останов (Halt)
1	0	0	Выборка кода
1	0	1	Чтение памяти (MEMRD)
1	1	0	Запись в память (MEMWR)
1	1	1	Пассивное состояние

Лекция № 6

3.5. Организация адресного пространства памяти и ввода-вывода

Память логически организована как одномерный массив байтов, каждый из которых имеет 20-битовый физический адрес в диапазоне 00000 — FFFFF. (Для записи адресов здесь и далее используется 16-ричная система счисления.) Любые два смежных байта в памяти могут рассматриваться как 16-битовое слово. Младший байт слова имеет меньший адрес, а старший — больший. Такое размещение байтов слова используется также в I8080 и в большинстве современных микроЭВМ. Адресом слова считается адрес его младшего байта. Таким образом, 20-битовый адрес памяти может рассматриваться и как адрес байта, и как адрес слова.

Полная информация, необходимая для определения физического адреса, содержится в адресном объекте «сегмент : смещение», который называется указателем адреса и содержит адрес сегмента и внутрисегментное смещение. Для запоминания указателя адреса требуется два слова памяти, причем слово с меньшим адресом всегда содержит смещение, а слово с большим адресом — базовый адрес сегмента. Каждое слово хранится обычным образом, т. е. по принципу «младший байт — по меньшему адресу».

Команды, байты и слова данных можно свободно размещать по любому адресу, что позволяет экономить память благодаря ее плотной упаковке. Однако для экономии времени выполнения программы целесообразно размещать слова данных в памяти по четным адресам, так как МП передает такие слова за один цикл шины. Слово с четным адресом называется выравненным на границе слов. Слова с нечетными адресами (невыравненные) также допустимы, но для их передачи требуются два цикла шины, что снижает производительность МП. (Каждый цикл имеет четыре обязательных такта Т.) Отметим, что шинный интерфейс инициирует необходимое для выборки слова число обращений к памяти автоматически, так что двукратное обращение к памяти не требует специального указания в программе. Особенно важно иметь выравненные слова для операций со стеком, так как в них участвуют

только слова. Следовательно, указатель стека SP необходимо всегда инициализировать на четный адрес.

Команды всегда выбираются словами по четным адресам, за исключениием первой выборки после передачи управления по нечетному адресу, когда выбирается один байт. Поток команд разделяется на байты при заполнении очереди команд внутри МП, так что выравнивание команд не влияет на производительность и поэтому не используется.

Подключение блоков памяти.

При подключении ЗУ к шинам МПС необходимо обеспечивать передачу как двухбайтовых слов, так и отдельных байтов. С этой целью память выполняется в виде двух банков (рис. 5): младшего, подключаемого к линиям данных D7 – DO и содержащего байты с четными адресами (A0 = 0), и старшего, соединенного с D15 – D8 и содержащего байты с нечетными адресами (A0 = 1). Чтобы каждое слово передавалось за один цикл шины, слова располагают только с четных адресов. Напомним, что адресная линия AO совместно с линией разрешения старшего банка BHE обеспечивает следующие варианты пересылок по шине данных:

A0 = 0, BHE = 0 – пересылается слово;
A0 = 0, BHE = 1 – пересылается только младший байт;
A0 = l, BHE = 0 – пересылается только старший байт,
A0 = 1, BHE = 1 – устройство не выбрано.

Выработка сигнала BHE и указанный порядок пересылок реализуются микропроцессором автоматически.

При чтении из ЗУ в любом случае на шину данных будет подаваться слово, из которого МП при необходимости выберет требуемый байт и поместит его в регистр, указанный в выполняемой команде. Поэтому сигналы BHE и AO на ПЗУ не подаются. При записи в ЗУ необходимо различать старший и младший байты (иначе может происходить разрушение информации, хранящейся в соседнем байте). Для этого сигналы BHE и AO подаются на входы CSH и CSL выбора старшего и младшего банков ОЗУ.

Процесс обращения к ПЗУ стробируется сигналом MEMR, а к ОЗУ – сигналами MEMR и MEMW, объединенными с помощью логического элемента И-НЕ. В примере, показанном на рис. 5, емкость каждого блока (ПЗУ и ОЗУ) составляет 8 Кбайт. Блок ПЗУ может быть, например, выполнен на основе двух включенных параллельно БИС К573РФ4 емкостью 8 Кбайт каждая, а блок ОЗУ – на основе восьми БИС К537РУ10 емкостью 2 Кбайт каждая. Адресные входы A12 – AO каждой пары БИС соединены параллельно и подключены к адресным линиям A13 – A1. Оставшаяся свободной линия A14 используется для различения блоков ПЗУ (A14 = 0) и ОЗУ (A14 = 1). В более общем случае для различения блоков ПЗУ и ОЗУ, а также для раздельной адресации страниц этих блоков осуществляется дешифрация старших адресных линий, например, с помощью ИС К155ИД4.

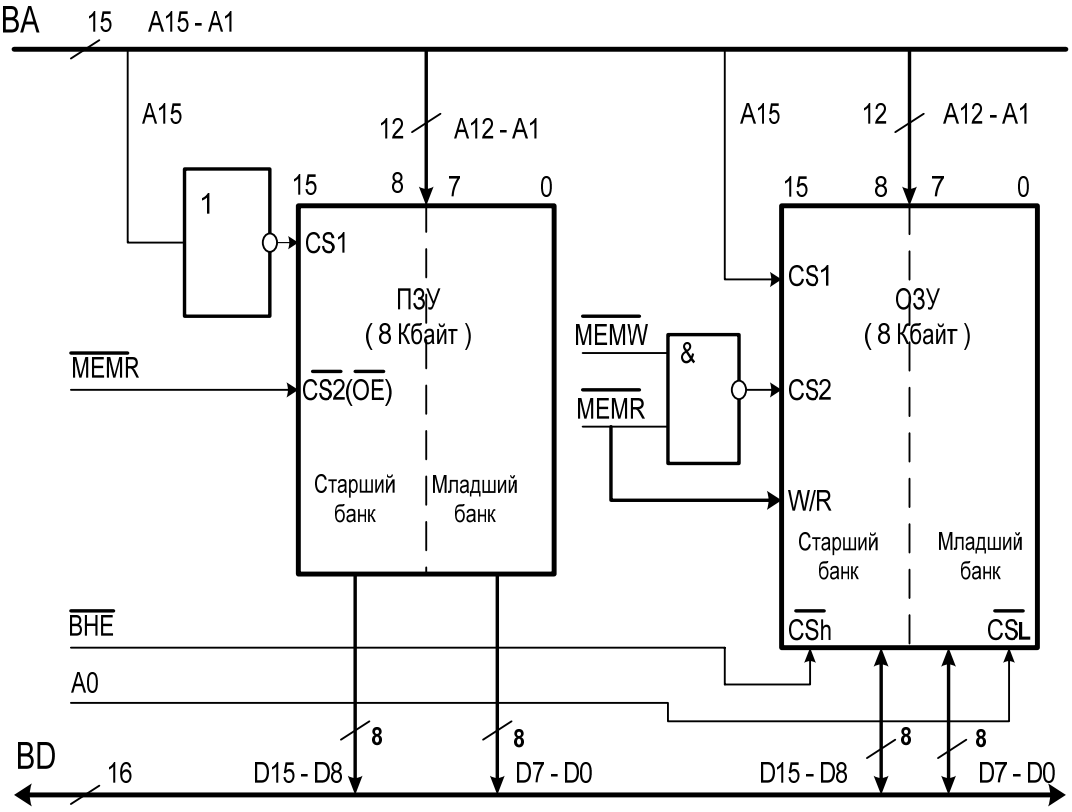


Рис. 5. Схема подключения банков памяти

*3.5.2. Средства и способы доступа к элементам структурной схемы

При подключении внешних устройств также возникает проблема передачи слов или отдельных байтов по шине данных (ШД) К устройствам, которые осуществляют обмен байтами и поэтому подключаются к младшей или старшей половине ШД, относятся, в частности, все программируемые БИС. В этом случае отдельно дешифруются четные и нечетные адреса ВУ (рис 6.), и полученные таким образом сигналы выбора CS подаются на соответствующие входы БИС. Устройства с четными адресами подключаются к младшей половине ШД (или ко всей ШД), а устройства с нечетными адресами — к старшей половине ШД. Если, например, необходимо передавать слова с помощью БИС интерфейса К580ВВ55, то параллельно включается две такие БИС, входы CS которых соединены с разными дешифраторами адреса. При этом имеется возможность передавать не только слова, но и отдельные байты, т е. обращаться индивидуально к каждой БИС. Если в этом нет необходимости, то входы CS обеих БИС можно подключить к одному выходу верхнего дешифратора DC

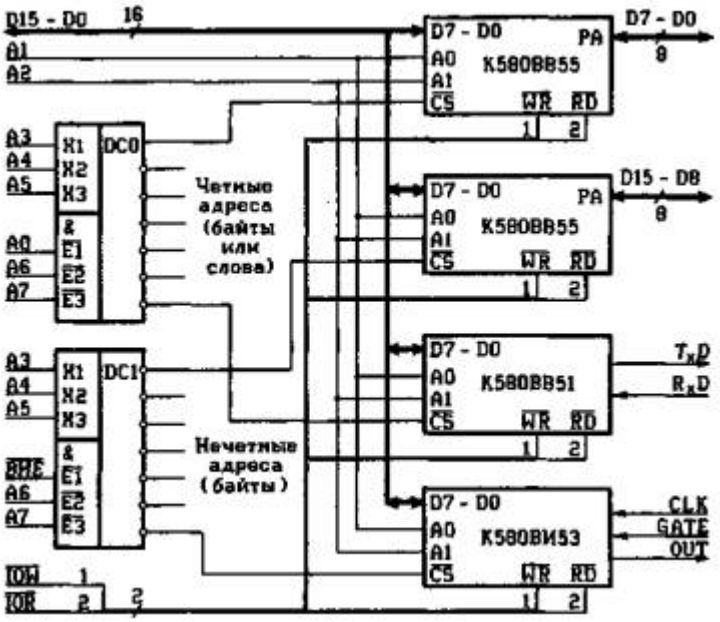


Рис. 6. Подключение программируемых БИС к микропроцессорной системе

Несколько иной способ выработки сигналов CS показан на (рис. 7), где возможна передача слов по нечетному адресу, при которой МП последовательно передает два байта

Если в системе реализуется ввод – вывод, отображенный на память, т.е. используется совмещённая адресация, то могут потребоваться дополнительные дешифраторы.

При использовании ВУ, ориентированных на передачу байтов, может оказаться целесообразным преобразование двухбайтовой ШД в однобайтовую (рис. 8). Это, в частности, требуется для организации прямого доступа к памяти с помощью БИС контроллера ПДП К1810ВТ37 и для пересылки блоков данных между ЗУ и ВУ с помощью цепочечных команд. В последнем случае ввод – вывод должен быть организован с отображением на адресное пространство памяти. Устройства, подключаемые к однобайтовой шине данных, могут получать четные или нечетные адреса произвольно.

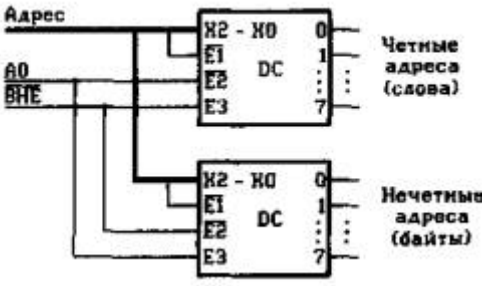


Рис.7 . Вариант дешифрации адресов ВЧ

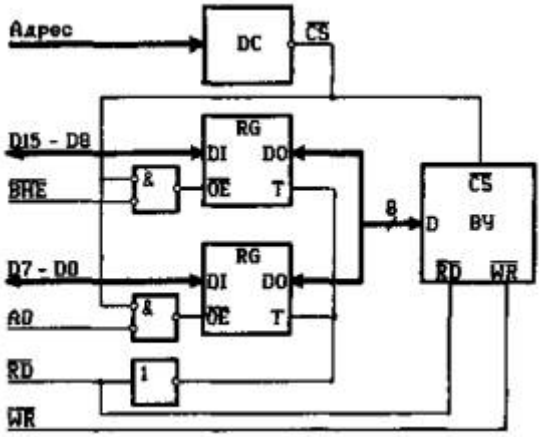


Рис. 8. Схема преобразования двухбайтовой шины данных в однобайтовую для подключения ВУ

3.6.Основные особенности архитектуры микропроцессора I8088. Шинные циклы процессоров

Интегральная схема I8088 **(**К1810ВМ88) представляет собой 16-битовый микропроцессор с 8-битовой внешней шиной данных. Он предназначен прежде всего для перевода аппаратных средств, построенных на базе МП К580ВМ80 и К580ВМ85, на программную среду МП К1810ВМ86 с целью повышения производительности этих средств. Его нередко используют и в оригинальных разработках, поскольку при этом упрощаются построение блоков памяти и разводка проводника на печатной плате по сравнению с МП К1810ВМ86. Микропроцессоры ВМ86 и ВМ88 имеют аналогичную архитектуру и одинаковую систему команд. В ВМ88 сохранены 16-битовые общие и сегментные регистры, АЛУ для обработки 16-битовых операндов, сумматор для вычисления 20-битового физического адреса и средства поддержки многопроцессорных систем. Различия между этими двумя МП состоят в изменении разрядности шины данных и соответствующих изменениях структуры и работы шинного интерфейса.

Назначение и нумерация выводов БИС ВМ88 такие же, как БИС ВМ86, но линии адреса А15 – А8 используются только для выдачи адресов, а линия ВНЕ заменена линией состояния SSO, так как ВМ88 может обращаться только к байтам и надобность в сигнале разрешения старшего байта шины ВНЕ отпадает. При работе ВМ88 в минимальном режиме сигнал SSO логически эквивалентен сигналу состояния SO, который вырабатывается МП ВМ86 в максимальном режиме. Сочетание сигналов Ю/М, DT/R и SSO позволяет однозначно определить тип выполняемого цикла шины (табл. 4).

Таблица 4

Ю/М	DT/R	STO	Тип цикла шины
0	0	0	Подтверждение прерывания
0	0	1	Чтение из порта ввода – вывода
0	1	0	Запись в порт ввода – вывода
0	1	1	Останов
1	0	0	Выборка команды
1	0	1	Чтение данных из памяти
1	1	0	Запись данных в память
1	1	1	Пассивное состояние (нет цикла шины)

При работе ВМ88 в максимальном режиме на выводе 34 постоянно присутствует высокий потенциал.

Необходимо отметить, что инвертировано значение сигнала выбора памяти и внешних устройств, чтобы он совпадал с сигналом IO/M микропроцессора ВМ85. Однако остальные управляющие сигналы ВМ88 и ВМ85 различны, что требует существенного изменения логики управления шиной при введении ВМ88 в системы, выполненные на базе ВМ85. Изменения в подсистемах памяти и ввода – вывода могут быть небольшими (если не требуется увеличения емкости памяти), поскольку адреса и данные в ВМ88 мультиплексируются, как и в МП ВМ85. Значения адресных сигналов А15 – А8 запоминаются во внутреннем регистре МП, и их не нужно защелкивать во внешнем регистре с помощью строба ALE.

Длина очереди команд МП ВМ88 выбрана равной 4 байт, поскольку в отличие от ВМ86 ВМ88 может считывать только байты (а не слова) за один цикл шины и соответствующее увеличение времени выборки команд не позволяет полностью использовать 6-байтовую очередь для повышения производительности. Порядок опережающей выборки команд отличается тем, что ВМ88 инициирует цикл шины для выборки команды, когда в очереди оказывается один свободный байт, а не два, как в ВМ86. Передача в ВМ88 одного байта за цикл шины приводит к увеличению времени выполнения команд на четыре такта при передаче каждого слова. Поэтому при оценке времени выполнения команд необходимо учитывать число обращений к памяти и к портам ввода – вывода для передачи слов. Временные диаграммы сигналов МП ВМ88 (кроме А15 – А8) совпадают с диаграммами сигналов ВМ86 (рис. 9).

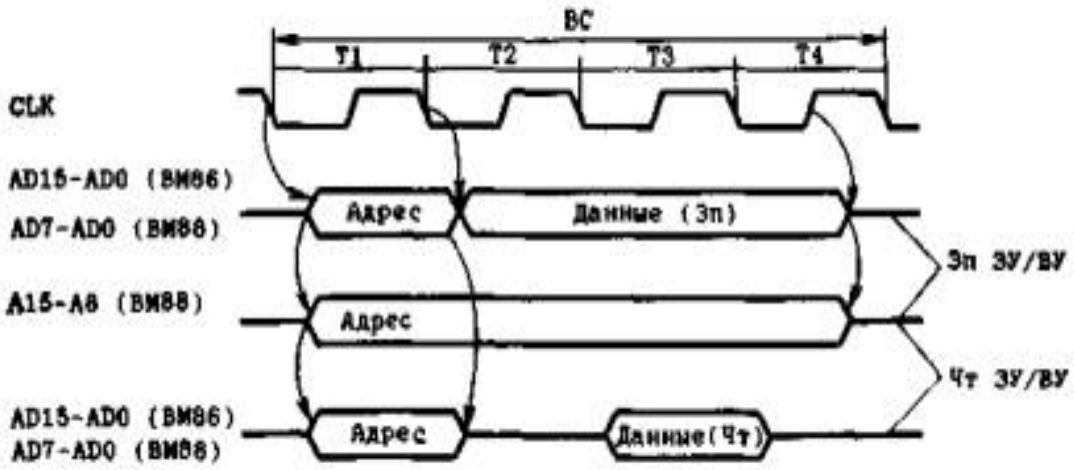


Рис. 9. Временная диаграмма работы МП ВМ88

При построении систем, содержащих ЦП ВМ86/ВМ88 и арифметический сопроцессор ВМ87, необходимо, чтобы сопроцессор имел информацию о типе используемого ЦП и соответственно о разрядности шины данных и длине очереди команд. Для этого сопроцессор проверяет уровень сигнала на выводе 34 непосредственно после окончания сигнала RESET, на котором ВМ88 в максимальном режиме устанавливает постоянное напряжение высокого уровня, а ВМ86 — сигнал BHE/S7. Так как в ВМ86 первой выбирается команда по четному адресу FFFFO, то первоначально устанавливается сигнал BHE = 0. Естественно, что ЦП и сопроцессор должны иметь общую линию сброса. Когда к ЦП подключены сопроцессор и независимый процессор, который выбирает свои команды, сопроцессор должен определять, каким процессором выбирается команда, чтобы правильно модифицировать свою очередь команд. Для этого сопроцессор контролирует бит состояния ST6, на который МП ВМ88, как и ВМ86, всегда выводит напряжение низкого уровня, а процессор ввода – вывода ВМ89 – высокого уровня.

3.6. Инициализация, сброс и синхронизация микропроцессоров.

Тактовая синхронизация работы микропроцессорной системы производится с помощью генератора тактовых импульсов К1810ГФ84, сигналы синхронизации поступают на вход CLK микропроцессора с периодом повторения, равным 200-500 нс.

Сброс микропроцессора производится по входу CLR (RESET) сигналом длительностью не менее 50 мкс. При действии сигнала сброса МП прекращает все свои действия и переводит шины и некоторые управляющие сигналы в высокоимпедансное состояние. При этом производится инициализация регистров микропроцессора следующим образом: все регистры, в том числе регистр IP, обнуляются, а сегментный регистр CS устанавливает все свои разряды в «1».

Таким образом, МП начинает свою работу с адреса FFFF0h. Обычно здесь находится команда перехода на начало программы Jmp.

Поскольку после сброса прерывания по входу INT запрещены (так как регистр флагов сброшен, в том числе и флаг IF программа должна разрешить прерывания в нужном месте.

3.7. Прерывания работы микропроцессора

В обыденной жизни всем нам время от времени приходится сталкиваться с прерываниями, какие-то нам приятны (например, позвонил друг), и мы с удовольствием прерываем своё занятие и отвлекаемся, некоторые нам не нравятся, и мы позволяем себе не реагировать на них, на другие мы просто обязаны ответить. По такому же принципу организована вычислительная система. Прерывания существенно увеличивают эффективность вычислительной системы, если бы в системе не было прерываний, то процессору пришлось бы самому периодически проверять, не требует ли какое-либо устройство обслуживания (в микропроцессорной технике это называется «режим опроса»).

Прерывания могут поступить от внешних устройств, они могут быть инициированы специальными командами или самим микропроцессором. Итак, прерывания – это вызываемый определённым образом процесс, переключающий МП на выполнение другой программы с последующим возвратом к прерванной программе. Прерываний в МПС на базе I8086/88 256 типов (0…255). Они делятся на внешние аппаратные, внутренние аппаратные и программные. Любые прерывания требуют новых адресов подпрограмм прерываний, которые выбираются из таблицы векторов. Подробно эти вопросы рассмотрены в разделе 5.2.

Лекция № 7

4. СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА I8086

Ассемблер позволяет общаться с компъютером на его собственном языке и непосредственно управлять аппаратными средствами. Вам кажется, что компъютер «понимает» языки высокого уровня, на самом деле программы исполняются на машинном языке, который и управляет процессором. Следовательно, программы должны предварительно переводиться в машинный код (компилироваться). Программы на языке ассемблера также переводятся в машинный код программой – транслятором, но одной команде в этом случае соответствует машинный код величиной от одного до шести байт, а каждый оператор языков высокого уровня обычно преобразуется в целый набор машинных кодов. У каждого языка есть свои достоинства, но только на языке ассемблера обеспечивается максимальный доступ к процессору, что позволяет достичь высоких скоростей при выполнении программ, на нём можно писать более гибкие программы. Основной недостаток связан с

тем, что ассемблерная программа может исполняться только на том процессоре, для которого она была написана.

4.1. Форматы команд

Команды и данные представляются в одинаковой формат т. е. в виде двоичных чисел, которые по шине данных передаются в центральный процессор. Команда должна определить:

– саму операцию, эта часть называется кодом операции (КОП);

– информацию об адресе или адресах (в зависимости от типа команды) данных, с которыми нужно произвести операцию.

Кроме того, нужно явно или неявно определиться с местом помещения результата и адресом следующей команды.

Последние процессоры серии Intel имеют максимальную длину машинной команды – 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного – только КОП. В дальнейшем будем рассматривать форматы команд только для микропроцессора I8086/88.

Команды микропроцессора I8086/88 (К1810 ВМ86/88) могут адресовать один или два операнда. Однако в таких командах один из операндов должен обязательно располагаться в регистре, поскольку имеются команды типа регистр – регистр, регистр – память и память – регистр, но команды типа память – память отсутствуют (за исключением команды обработки строк данных). В общем виде формат команд приведён на рис. 10.

КОП	Биты d, w или s, w	mod	Reg	R/m	Disp 8/16	Date 8/16
1 байт		1 байт			1-2 байта	1-2 байта

Рис. 10. Формат команд

Первый байт команды содержит код операции КОП и однобитовые поля **d** и **w** или **s** и **w** или один бит **w**. При d = 1 осуществляется передача операнда или результата операции в регистр, который определяется полем **reg** второго байта команды, при d = 0 – передача из указанного регистра. Поле w определяет разрядность операнда: при w = 1 команда оперирует словами, при w = 0 – байтом.

Второй байт называется постбайтом. С его помощью определяются участвующие в операции регистры и ячейки памяти. Постбайт состоит из трёх полей: двухбитовое поле mod и два трёхбитовых поля reg – регистр и r/m – регистр/память. Поле mod используется для определения способа адресации, поле reg определяет операнд, который обязательно находится в регистре микропроцессора (далее – МП) и условно считается вторым операндом. Это поле используется для указания имени регистра только в двухоперанд-

ных командах. Если в команде один операнд, то он определяется полем r/m, а поле reg используется для расширения кода операции. В таблице 5 представлены закодированные значения регистров микропроцессора.

Табл. 5.

Поле r/m или reg	W=0	W=1	Поле r/m или reg	W=0	W=1
000	AL	AX	100	AH	SP
001	CL	CX	101	CH	BP
010	DL	DX	110	DH	SI
011	BL	BX	111	BH	DI

Если поле mod = 11, используется регистровая адресация операндов, тогда в поле r/m помещается код регистра, в котором находится первый операнд. Когда адресуется память, поле mod определяет вариант использования смещения disp, находящегося в третьем и четвёртом байтах, или показывает его отсутствие (табл.6). Смещение disp, входящее в состав команды, интерпретируется как знаковое целое, участвующее в вычислении эффективного (или исполнительного) адреса EA. В свою очередь, из-за сегментной организации памяти весь эффективный адрес EA является смещением (offset) относительно базового адреса сегмента (рассматривается как беззнаковое целое при вычислении физического адреса). Использование одного и того же термина «смещение» и для disp, участвующего в вычислении эффективного логического адреса EA, и для самого EA, являющегося смещением при вычислении физического адреса, может внести путаницу. Давайте в необходимых случаях будем называть “offset” смещением в сегменте.

Таблица 6

r/m	Mod		
	00	01	10
00	BX+SI	BX+SI+disp 8	BX+SI+disp 16
01	BX+DI	BX+DI+disp 8	BX+DI+disp 16
10	BP+SI	BP+SI+disp 8	BP+SI+disp 16
11	BP+DI	BP+DI+disp 8	BP+DI+disp 16
00	SI	SI+disp 8	SI+disp 16
01	DI	DI+disp 8	DI+disp 16
10	disp16	BP+disp8	BP+disp 16
11	BX	BX+disp 8	BX+disp 16

4.2.Способы адресации

В МП I8086/88 разрядность адресов равна 20, однако процессор манипулирует 16-разрядными логическими адресами, к которым относятся начальные (базовые) адреса сегментов памяти и значения смещений в этих сегментах. Физический адрес вычисляется следующим образом: содержимое каждого сегментного регистра рассматривается как 16 старших разрядов A19

– A4 начального адреса соответствующего сегмента. Младшие разряды A3 – A0 этого адреса всегда полагаются равными разрядам во время операции вычисления физического адреса. Эта операция выполняется сумматором адреса, расположенным в МП, и состоит в сложении 20-разрядного начального адреса сегмента с 16-разрядным смещением, которое дополняется четырьмя старшими разрядами A19 – A16, равными нулю. Наибольшая ёмкость памяти, отводимой под один сегмент, определяется максимальным значением 16-разрядного смещения и составляет 64Кбайт.

Наличие разнообразных способов адресаций упрощает организацию сложных структур данных и повышает гибкость их применения.

Регистровая адресация. Регистровые операнды указываются именами регистров, определёнными в байте кода операции или в постбайте, в котором выделены 3-битовые поля reg и r/m (при mod = 11).Команды, содержащие только регистровые операнды, являются наиболее короткими и выполняются за наименьшее время, так как не требуют вычисления EA и выполняются машинного цикла для обращения к памяти.

Например, команда DEC DI уменьшает содержимое регистра DI.

Непосредственная адресация. Непосредственные операнды – это постоянные данные, определяемые как часть машинной команды (поле date). Данные могут быть однобайтными или двухбайтными. Непосредственные операнды могут быть выбраны быстро, так как их можно получить прямо из очереди команд без обращения к памяти. Они могут быть заданы только как операнды-источники. Если приёмником операнда служит аккумулятор, машинная команда более компактна. Например, команда CMP AL,02 (машинный код 3C02 сравнивает содержимое регистра AL с шестнадцатеричным числом 02 и устанавливает определённые флаги в соответствии с результатом сравнения. Эта команда занимает только два байта в памяти и может быть выполнена за четыре такта.

Прямая адресация. Эффективный адрес EA берётся непосредственно из 16-битового поля смещения машинной команды (поле disp).Этот прямой адрес определяет байт или слово памяти, расположенное внутри сегмента. По умолчанию прямая адресация приводит в сегмент данных, однако для его модификации возможно применение префиксных команд, в этом случае данные можно помещать в любой из четырёх сегментов. При использовании прямой адресации в поле mod содержится число 00, а в поле r/m – число 110 (табл. 6).

Например, пусть регистр сегмента данных содержит 04И5H, а байт 1400H текущего сегмента данных – символическое имя ALFA. Машинная команда C606001402 (MOV ALFA, 02) предписывает запомнить по перемещаемому адресу 1400 сегмента данных шестнадцатеричное число 02.

В этой пятибайтной команде C6 – это код команды, 06 – постбайт, 0014 – смещение (disp 16) 1400H, интерпретируемое как эффективный адрес EA, 02 – шестнадцатеричный операнд 02.Действительный физический 20-битовый адрес здесь соответствует 05F50H (4B50H + 1400H).

Кроме обычной прямой адресации используют два специальных вида адресации: относительную и абсолютную (или длинную) прямую адресацию.

При относительной адресации поле смещения представляется 8-битовым числом со знаком. Эффективный адрес EA при этом определяется в результате сложения содержимого поля смещения и регистра указателя команд IP. Относительную адресацию используют для команд условных переходов, таких, как JE (переход по равенству), JO (переход по переполнению). При использовании относительной адресации постбайт не требуется.

При абсолютной (длинной) прямой адресации часть команды представляет собой 32-битовый указатель, определяющий физический адрес в памяти МП. Младшее слово указателя рассматривается при этом как перемещаемый адрес сегмента, базовым адресом которого является старшее слово указателя, в этом случае постбайт также не требуется.

Косвенно-регистровая адресация. При этом способе в качестве эффективного адреса EA выступает содержимое базового или индексного регистров (BX, SI или DI). Изменяя содержимое базового или индексного регистров, можно обратиться к различным участкам памяти. Можно использовать дополнительные префиксные команды для переназначения эффективного адреса в другие сегменты памяти.

При косвенной адресации существует одно исключение: для команд

IMP (переход) и CALL (вызов процедур) в качестве эффективного адреса может выступать содержимое любого из 16-битовых регистров общего назначения (AX, BX, CX, DX, SI, DI, BP, SP).

Например, в программе регистр DS содержит число 14C5H, регистр SI – число 28FFH, команда C60410 (MOV [SI], 10H) запоминает шестнадцате ричное число 10 в байте сегмента данных с физическим адресом 1754FH (14C50H + 28FFH).

Базовая адресация. При такой адресации эффективный адрес EA определяется как сумма содержимого одного из базовых регистров BP или BX и смещения disp 8 или 16. При использования регистра БР за эффективный адрес принимается перемещаемый адрес теущего сегмента стека, базовый адрес которого находится в регистре сегмента стека SS.

Индексная адресация. При такой адресации эффективный адрес EA определяется как сумма содержимого одного из индексных регистров SI или DI и смещения disp 8 или 16.

Базово-индексная адресация. В этом случае эффективным адресом является сумма содержимого базового регистра (BX или BP) и индексного регистра (SI или DI). Базово-индексная адресация является очень гибким средством доступа к самым различным участкам памяти благодаря возможности оперативного изменения содержимого базового или индексного регистров. При базово-индексной адресации поле mod = 00.

Базово-индексная адресация со смещением. При таком способе эффективный адрес определяется сложением содержимого базового регистра (BX или BP), индексного регистра (SI или DI) и смещения, являющегося ча-

стью команды. Адресуемая ячейка памяти может находиться сегменте данных или стека в зависимости от того, какой из базовых регистров используется: BX или BP.

Неявная адресация. Операнд указывается первым байтом вместе с кодом операции без выделения специальных битов.

4.3. Описание команд

Система команд более подробно разобрана в методическом пособии по циклу лабораторных работ. Таблица с системой команд приведена в приложении 2.

4.3.1. Пересылка данных

Эти команды осуществляют пересылку данных из одного места в другое, запись и чтение информации из портов ввода-вывода, преобразование информации, манипуляции с адресами, обращение к стеку. Различают следующие типы пересылок: общего назначения, с участием аккумулятора, пересылка адреса операнда, флагов и стековые пересылки. Необходимо подчеркнуть, что все команды пересылок не устанавливают флаги.

Пересылки общего назначения задаются с помощью двух мнемокодов: это команды **MOV** (переслать) и **XCHG** (обменять).

Команда MOV часто используется в программах и является основной командой пересылки данных. В качестве источника и приёмника данных может служить регистр, память или сегментный регистр, а также источником могут являться данные, непосредственно представленные в формате команды.

Выбор подходящего формата в определённой степени может оптимизировать программу по времени её выполнения. Отметим некоторые особенности выполнения этой команды:

- командой MOV нельзя осуществить пересылку из одной области памяти в другую;
- нельзя загрузить в сегментный регистр значение непосредственно из памяти, для этого приходится использовать регистр общего назначения;

нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр, выполнить такую пересылку можно, используя в качестве промежуточных всё те же регистры общего назначения

- нельзя использовать сегментный регистр CS в качестве приёмника данных, это означало бы операцию перехода в программе.

Команда XCHG применяется для двунаправленной пересылки данных. Команда может быть двух форматов, в первом случае она вызывает обмен байтами или словами между регистрами или регистром и памятью (№20 и 21 приложения), второй формат – более короткий: пересылка между одним из регистров и аккумулятором (команда № 22 приложения 1).

Примечания: при обмене нельзя использовать сегментные регистры; команду XCHG AX, AX можно применить как пустую операцию.

Пересылки с участием аккумулятора состоят из трёх команд: **IN** (ввод), **OUT** (вывод) и **XLAT**.

Вся работа системы с внешними устройствами выполняется с использованием портов ввода-вывода. Команда IN служит для пересылки байта или слова из порта ввода в аккумулятор (в AL или AH). Номер порта ввода может быть задан как непосредственно, во втором байте команды, так и косвенно, в регистре DX (команды № 23 – № 26 в таблице системы команд). Двухбайтные команды IN A, port и OUT port,A позволяют адресоваться к 256 портам, однобайтные команды IN A,DX и OUT DX,A адресуют 65536 портов. Косвенное задание порта хотя и требует предварительной загрузки его адреса в DX, однако позволяет организовывать программные циклы, в которых используется изменяющийся адрес портов.

Команда XLAT производит замещение значения в регистре AL другим байтом из таблицы в памяти, размером не более 256 байт. Слово «таблица» весьма условно, практически это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра AL , определяется суммой (BX + AL), т. е. содержимое AL выполняет роль индекса в байтовом массиве. Разумеется, составляющие адресов предварительно должны быть занесены в указанные регистры.
Пересылки адреса операнда. При написании программ производится активная работа с адресами операндов, находящимися в памяти. Для поддержки таких операций в процессорах I8086 есть специальная группа команд, в которую входят следующие команды: **LEA** (загрузка эффективного адреса в указанный регистр), **LDS** (загрузка указателя в DS), **LES** (загрузка указателя в ES). В более современных процессорах имеются дополнительные команды LFS и LGS (загрузка указателя в регистр дополнительного сегмента данных FS и GS), а также команда LSS (загрузка указателя в регистр сегмента стека SS). Команда LEA производит вычисление эффективного адреса EA и передаёт его в 16-разрядный регистр, код которого указан в поле reg. Команды LDS и LES применяются в основном при обращении к данным, находящимся вне текущих сегментов DS и ES, так что возникает необходимость изменить базовый адрес сегмента. Два 16-разрядных адреса – базовый сегментный и смещение в сегменте, называемые указателем, предварительно загружаются в память. Значение смещения содержится в двух первых байтах указателя, а базовый адрес сегмента – в третьем и четвёртом байтах. По команде LDS (или LES) происходит обращение к указателю и осуществляется загрузка регистра DS (или ES) базовым адресом. А смещение пересылается в регистр, указанный полем reg постбайта команды. Аналогичные действия производят команды LFS, LDS и LSS.

Пересылки флагов. Эта группа включает четыре однобайтные команды: LAHF (загрузить AH флагами), SAHF (запомнить AH в регистре F), PUSH F (занести F в стек) и POP F (извлечь из стека).

4.3.2. Арифметические команды

Группа арифметических целочисленных команд работает с двумя типами чисел:

целыми двоичными числами. Числа могут иметь знаковый разряд или не иметь такового, то есть быть числами со знаком или без знака;

целыми десятичными числами.

Целые двоичные числа с фиксированной точкой – это числа, закодированные в двоичной системе счисления. Размерность целого двоичного числа может составлять 8,16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, – это команды целочисленного умножения и деления IMUL и IDIV. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста.

Десятичные числа – специальный вид представления информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырёх бит, двоично-десятичный код (BCD – Binari-Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах:

- упакованный формат – каждый бит содержит две десятичные цифры. Код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;
- неупакованнй *формат* – каждый бит содержит одну десятичную цифру в четырёх младших битах. Старшие 4 бита имеют нулевое значение. Следовательно, диапазон представления неупакованного числа в одном байте составляет от 0 до 9.

Сложение двоичных чисел без знака

В системе команд микропроцессора имеются три команды двоичного сложения:

- **ADD операнд 1, операнд 2** – команда сложения двух операндов, результат помещается на место 1 операнда;
- **ADC операнд 1, операнд 2** – команда сложения с учётом флага переноса CF;
- **INC операнд** – операция увеличения значения операнда на 1.

Команда ADC является средством микропроцессора для сложения длинных двоичных чисел.

Сложение двоичных чисел со знаком

При выполнении сложения микропроцессор не различает числа со знаком и без знака, сами команды сложения чисел со знаком те же, что и для чисел без знака. Старший бит в числах со знаком является знаковым: если он равен нулю, значит это число положительное, если единице – отрицательное. Очевидно, что ответственность за правильность действий с получивши-

мися числами ложится на программиста. У микропроцессора есть средства фиксирования возникновения различных ситуаций, возникающих при вычислениях, к ним относятся установка флага переноса CF, свидетельствующего о выходе за пределы разрядности операндов, и флага переполнения OF. Если рассмотреть различные варианты сложения чисел, можно выяснить, что переполнение (т. е. установка флага OF в 1) происходит при переносе:

- из 14-го разряда (для положительных чисел со знаком);
- из 15-го разряда (для отрицательных чисел).

Переполнения не происходит (т.е. флаг OF сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах, в этом случае результат вычислений правильный.

Команды вычитания

Операции вычитания включают пять мнемокодов: **SUB** (беззнаковое вычитание), **SBB** (вычесть со знаком), **DEC** (уменьшить на единицу) **NEG** (изменить знак) и **CMP** (сравнить).

Вычитание двоичных чисел без знака

По команде SUB вычитается операнд источника из операнда приёмника.

Команда SBB используется при вычитании многобайтных чисел и учитывает флаг CF, который теперь играет роль индикатора заёма 1 из старшего разряда при вычитании чисел.

После выполнения команды вычитания чисел без знака нужно анализировать состояние флага CF. Если он установлен в 1, то это говорит о том, что произошёл заём из старшего разряда и результат получился отрицательным, т. е. в дополнительном коде.

Команда DEC вызывает уменьшение на единицу (декрементирование) содержимого регистра или памяти и имеет два формата, как и команда INC.

Команда NEG изменяет знак операнда, причём используется представление операнда в дополнительном коде. Например, если есть операнд

– 2 (11111110), то команда NEG изменит его на +2 (00000010).

Команда CMP служит для сравнения двух операндов путём вычитания значения операнда приёмника из операнда источника, при этом полученная разница никуда не заносится, а результатом операции сравнения являются значения флагов, которые устанавливаются в зависимости от соотношения сравниваемых операндов.

Вычитание двоичных чисел со знаком

При вычитании чисел со знаком, как и при сложении, необходимо анализировать значение флага переполнения OF. Если он установился в 1, значит результат вышел за диапазон представления знаковых чисел (т.е. изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Команды умножения

В арифметических командах есть две команды умножения: **MUL** – умножить и **IMUL** – умножение со знаком. По команде MUL умножается без

знака содержимое аккумулятора (AL или AX) на операнд источника, а результат двойной длины возвращается в аккумулятор и регистр, используемый для его расширения (в AL и AH в случае 8-разрядных операндов или в AX и DX в случае 16-разрядных операндов). По результату устанавливаются только два флага: CF и OF, которые покажут 1, если старшая половина результата отлична от нуля.

Команды деления

Для деления двоичных чисел без знака имеется команда **DIV** делитель, который может находиться в памяти или в регистре и иметь размер 8, 16, или 32 бита (начиная с процессора I286). Местонахождение делимого так же, как в команде умножения, зависит от размера операндов (табл. 7).

Таблица 7.

Делимое	Делитель	Частное	Остаток
16 бит в рег.AX	Байт в регистре или ячейке памяти 16 бит	Байт в регистре AL	Байт в регистре AH
32 бита DX - старшая часть, AX-младшая часть	16 бит в регистре или ячейке памяти	Слово 16 бит в регистре AX	Слово 16 бит в регистре DX
64 бита EDX-старшая часть, EAX-младшая часть	Двойное слово 32 бита в регистре или ячейке памяти	Двойное слово 32 бита в регистре EAX	Двойное слово 32 бита в регистре EDX

При выполнении операции деления флаги принимают произвольные значения, но возможно возникновение прерывания с номером 0. Эта разновидность прерывания 0 (исключения) возникает внутри микропроцессора из-за некоторых ситуаций:

- делитель равен нулю;
- частное не входит в отведённую под него разрядную сетку.

Для деления чисел со знаком предназначена команда IDIV. Особенностью команды деления IDIV является то, что частное и остаток всегда имеют одинаковые знаки. Прерывание типа 0 возникает в следующих случаях:

- делитель равен нулю;
- при делении на делитель величиной в байт со знаком частное находится вне диапазона от – 128 до +127;
- при делении на делитель величиной в слово со знаком частное находится вне диапазона от – 32768 до = 32768;
- при делении на делитель величиной в двойное слово со знаком частное находится вне диапазона от –2 147 483 648 до + 2 147 483 647.

Арифметические операции над неупакованными двоично-десятичными числами

Отдельных команд сложения, вычитания, умножения и деления двоично-десятичных чисел не существует, потому что размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-

десятичные числа как в упакованном формате, так и в неупакованном, а делить и умножать можно только неупакованные BCD-числа.

Для коррекции операции сложения двух однозначных неупакованных BCD-чисел существует специальная команда **AAA**, которая ставится после операции сложения и корректирует результат сложения. Команда работает с регистром AL и анализирует значение его младшей тетрады. Если оно меньше 9, то флаг CF сбрасывается в 0, и осуществляется переход к следующей команде. Если это значение больше 9, то выполняется следующее:

- к содержимому младшей тетрады AL прибавляется 6, чтобы получившееся число было бы десятичным;
- флаг CF устанавливается в 1, фиксируя перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Не лишним будет напомнить, что при записи в память числа в BCD-коде записываются в памяти, начиная с младшего байта.

При вычитании неупакованных BCD-чисел используется команда **AAS,** которая также ставится после команд вычитания SUB и SBB.

Для коррекции команд умножения неупакованных BCD-чисел применяют команду **AAM**. Эту же команду можно применять для преобразования двоичного числа в регистре AL (от 0 до 99) в неупакованное BCD-число, которое будет размещено в регистре AX: старшая цифра результата – в AH, а младшая – в AL.

При выполнении деления неупакованных BCD-чисел корректирующей командой является команда **AAD**, но коррекция выполняется до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Команду AAD также, как и команду AAM, можно использовать для перевода неупакованных BCD-чисел из диапазона 0…99 в их двоичный эквивалент.

Арифметические операции над упакованными двоично-десятичными числами

Упакованные BCD-числа можно только складывать и вычитать, для выполнения других действий их нужно преобразовывать или в неупакованный формат, или в двоичное представление.

Коррекцию результатов сложения производит команда **DAA**. Она преобразует содержимое регистра AL в две упакованные десятичные цифры. Если результат сложения больше 99, то получившаяся в результате сложения единица запоминается в флаге CF, тем самым учитывается перенос в старший разряд.

Для коррекции результатов вычитания используется команда DAS.

По результатам выполнения арифметических операций устанавливаются все флаги!

К группе арифметических операций относятся также две команды, осуществляющие расширение со знаком 8- и 16- разрядных операндов. Эти команды играют вспомогательную роль при подготовке операнда используемого в качестве делимого. При выполнении деления в зависимости от раз-

рядности делителя делимое размещается либо в 16-разрядном регистре AX, либо в регистрах DX-AX. При делении чисел со знаком перед заполнением указанных регистров требуется анализировать знак делимого и заполнять регистр или нулями, если делимое является положительным числом, или единицами, если делимое отрицательное. Для этого используются команды **CBW** (Расширение со знаком байта до слова) и **CWD** (расширение со знаком слова до двойного слова). По этим командам старший разряд числа, находящегося в регистре AL (или AX), записывается во все разряды регистра AH (или DX).

4.3.3. Логические команды

Система команд микропроцессора содержит пять команд, поддерживающих реализацию четырёх булевых функций:

- поразрядное логическое умножение (логическое И) выполняет команда AND;
- поразрядное логическое сложение (логическое ИЛИ) – команда OR;
- поразрядное логическое исключающее сложение (ИСКЛ. ИЛИ) – команда XOR;
- поразрядное отрицание (логическое НЕ) – команда NOT.

К этой же группе команд относится команда TEST (проверка), которая состоит в поразрядном логическом умножении (И) операндов без занесения результатов умножения в приёмник и служит для анализа содержимого источника по значениям флагов PF, SF и ZF.

Все двухоперандные команды AND, OR, XOR и TEST имеют одинаковые форматы, однооперандная команда NOT осуществляет инвертирование операнда и имеет один формат.

4.3.4. Команды сдвига

Все команды сдвига перемещают биты операнда влево или вправо, они имеют следующую структуру: КОП *операнд, счётчик сдвига*.

Счётчик циклов может задаваться двумя способами:

- статически, когда значение числа сдвигов задаётся с помощью непосредственного операнда;
- динамически, при этом значение числа сдвигов заносится в регистр CL перед выполнением команды сдвига. Следует отметить, что несмотря на то, что регистр AL восьмибитовый, микропроцессор воспринимает только значения пяти младших битов счётчика, то есть значение лежит в диапазоне от 0 до 31.

Все команды сдвига устанавливают флаг переноса CF за счёт сдвига очередного бита. По принципу действия команды сдвига делятся на два типа: команды линейного сдвига и команды циклического сдвига.

Линейный сдвиг. При линейном сдвиге осуществляется следующий алгоритм:

– очередной сдвигаемый бит устанавливает флаг CF;
– бит, вводимый в операнд с другого конца, получает значение 0;

Команды линейного сдвига делятся на логические и арифметические.

К командам линейного логического сдвига относятся следующие:

SHL (Shift Logical Left) – логический сдвиг влево.

SHR (Shift Logical Right) – логический сдвиг вправо.

Команды арифметического линейного сдвига особым образом работают со знаковым разрядом операнда:

SAL (Shift Arithmetic Left) – арифметический сдвиг влево. Команда практически совпадает с командой SHL.

SAR (Shift Arithmetic Right) – арифметический сдвиг вправо.

Команды арифметического сдвига позволяют выполнить умножение и деление операнда на степени двойки, скорость их выполнения выше, чем у команд умножения и деления.

Циклический сдвиг. Подобные команды бывают двух типов:

– команды простого циклического сдвига;
– команды циклического сдвига через флаг переноса

К командам простого циклического сдвига относятся:

ROL (Rotate Left) – сдвигаемые влево биты записываются в тот же операнд справа.

ROR (Rotate Right) –Сдвигаемые вправо биты записываются в тот же операнд.

Команды циклического сдвига через флаг переноса CF отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а сначала записывается во флаг переноса CF. К таким командам относятся следующие:

RCL (Rotate through Carry Left) – циклический сдвиг влево через перенос. Сдвигаемые биты поочерёдно становятся значением флага CF.

RCR (Rotate through Carry Right) – циклический сдвиг вправо через перенос. Сдвигаемые биты поочерёдно становятся значением флага CF

4.3.5. Команды обработки строк данных

Строкой называют последовательность байтов, размещаемую в смежных ячейках памяти. Команды обработки строк данных также называют цепочечными командами. Цепочка – понятие несколько более широкое, чем строка, и предполагает размерность операнда большую, чем байт (слово или двойное слово). Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют ещё и автоматическое продвижение к следующему элементу данной цепочки. Команды, обрабатывающие двойное слово, по понятным причинам в системе команд I8086-процессора отсутствуют.

Перечислим эти команды:

- пересылка цепочки, производится копирование элементов из одной области памяти в другую:

  MOVS адрес приёмника, адрес источника
  MOVSB
  MOVSW
  MOVSD

- сравнение цепочек, производится сравнение элементов цепочки-источника с элементами цепочки-приёмника:

  CMPS адрес приёмника, адрес источника
  CMPSB
  CMPSW
  CMPSD

- сканирование цепочки, производится поиск некоторого значения в области памяти. Искомое значение предварительно должно быть загружено в аккумулятор.

  SCAS адрес приёмника
  SCASB
  SCASW
  SCASD

- загрузка элементов из цепочки, извлекается элемент цепочки и помещается в аккумулятор:

  LODS адрес источника
  LODSB
  LODSW
  LODSD

- сохранение элемента в цепочке, переносится значение аккумулятора в элемент цепочки:

  STOS адрес приёмника
  STOSB
  STOSW
  STOSD

  Следующие ниже перечисленные команды также отсутствуют в системе команд I8086.

- получение элементов цепочки из порта ввода-вывода

  INS адрес приёмника, номер порта
  INSB
  INSW
  INSD

- вывод элементов цепочки в порт ввода-вывода

  OUTS адрес приёмника, номер порта
  OUTSB
  OUTSW
  OUTSD

В сочетании со многими цепочечными командами используют префиксы повторения:

REP
REPE или **REPZ**
REPNE или **REPNZ**

Цепочечная команда без префикса выполняется один раз. Размещение префикса перед командой (в поле метки) заставляет её выполняться в цикле.

Префикс REP используется с командами MOVS и STOS, при этом данные команды выполняются до тех пор, пока содержимое в регистре CX не станет равным нулю (после выполнения каждой команды содержимое регистра CX (ECX) автоматически декрементируется).

Префиксы REPE и REPZ являются синонимами, они заставляют цепочечную команду выполняться до тех пор, пока содержимое регистра CX (ECX) не равно нулю или флаг ZF равен 1. Наиболее эффективно использовать эти префиксы с командами CMPS и SCAS..

Префиксы повторения REPNE и REPNZ также синонимы, они заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое регистра CX (ECX) не равно нулю или флаг ZF равен нулю. При невыполнении одного из этих условий работа команды прекращается.Данные префиксы часто используют с командами CMPS и SCAS, но для поиска совпадающих элементов цепочки.

Важно отметить формирование физического операндов: цепочка- источник находится в текущем сегменте данных, определяемых регистром DS, цепочка-приёмник должна быть в дополнительном сегменте данных, адресуемом сегментным регистром ES. Таким образом, полные физические адреса операндов цепочечных команд следующие:

- адрес источника – пара DS:EI;
- адрес приёмника – пара ES:DI.

Для цепочечных команд существует понятие направления обработки строк:

- от начала цепочки к её концу, т. е. в направлении возрастания адресов;
- от конца цепочки к её началу, т. е. в направлении убывания адресов.
- С этой целью используется флаг направления DF в регистре флагов:
- если DF = 0, то значения индексных регистров SI, DI будут автоматически увеличиваться;
- если DF=1, то значения индексных регистров SI, DI будут автоматически уменьшаться.

Лекция №8

4.3.6. Команды передачи управления

При выполнении программ весьма часто возникают ситуации, когда принимается решение о ветвлении в программе, т. е. о переходе в другую

точку программы. Это решение может быть безусловным, когда переход к другой команде, находящейся на некотором удалении от текущей команды, обязателен, или условным, когда переход зависит от выполнения некоторых условий. Команды передачи управления изменяют содержимое регистров CS и IP (дальний переход) или только IP (ближний переход).

Группу команд передачи управления можно разделить на четыре типа:

- безусловные переходы;
- условные переходы;
- циклы;
- прерывания.

Безусловные переходы. Команды безусловных переходов включают три мнемокода: JMP (безусловный переход), CALL (вызов подпрограмм) и RET (возврат из подпрограммы).

По команде JMP можно осуществить переход в любую точку программы, расположенную как в текущем сегменте данных (ближний переход), так и в другом сегменте (дальний переход). При переходе в пределах текущего сегмента используются первые три формата команды JMP (см. приложение 1). Первый формат позволяет перейти к любой команде внутри текущего сегмента, для чего к содержимому IP добавляется в дополнительном коде 16-разрядное смещение, старший разряд которого является знаковым. Второй, укороченный формат позволяет перейти к точке программы, отстоящей не более чем на – 128 – +127 адресов от команды JMP. Третий формат осуществляет загрузку указателя команд IP 16-разрядным числом, размещённым по исполнительному адресу EA, которое определяется постбайтом (косвенный переход). Четвёртый и пятый форматы позволяют производить межсегментные переходы: в четвёртом формате во втором и третьем байтах формата указан относительный адрес точки перехода, а в четвёртом и пятом – новое значение CS.

Команда CALL используется в тех случаях, когда некоторый участок программы требует неоднократного повторения, CALL – это команда вызова подпрограммы или процедуры. Она имеет такие же форматы, что и команда JMP, за исключением укороченного. При выполнении команды CALL происходит автоматическое запоминание в стеке текущих значений регистров CS и IP (адрес возврата из подпрограммы). В конце подпрограммы ставится команда RET, по которой из стека возвращаются сохранённые значения в Cs и IP. При возврате из подпрограмм, расположенных в текущем сегменте кода, применяются первые два формата команды RET, причём второй формат отличается от первого тем, что к содержимому указателя стека добавляется константа, записанная во 2-м и 3-м байтах команды. Этот приём позволяет сбрасывать значения, записанные в стек при выполнении этой подпрограммы и не используемые в дальнейшем.

Для межсегментного возврата применяют третий и четвёртый форматы RET, которые обеспечивают восстановление регистров CS и IP.

Условные переходы. Команды условных переходов позволяют принимать решения о переходе в зависимости от некоторых условий. Микропроцессор имеет 18 команд условных переходов, позволяющих проверить:

- отношение между операндами со знаком («больше– меньше»);
- отношение между операндами без знака («выше – ниже»);
- состояние арифметических флагов ZF, SF, CF, OF, PF.

Термины «больше-меньше» и «выше-ниже» происходят от соответсвующих английских терминов «greater-less» и «above-below». Первые буквы этих терминов входят в состав мнемонических обозначений соответствующих команд условного перехода. Мнемокод команд начинается буквой J. Для того, чтобы принять решение о переходе, сначала должно быть сформировано условие, на основании которого и будет принято решение о передаче управления:

- это может быть любая команда, изменяющая состояние регистра флагов;
- команда сравнения CMP, сравнивающая значения двух операндов;
- состояние регистра CX.

Все команды условных переходов имеют одинаковый формат, в первом байте задаётся код операции, а во втором – 8-разрядное смещение, которое рассматривается как число со знаком, позволяющее осуществить изменение адреса в диапазоне от –128 до +127. При необходимости более отдалённого перехода по выполнению условия используется дополнительно команда безусловного перехода.

В таблице 8 приведены значения аббревиатур в названиях команд условных переходов.

Таблица 8

Мнемоническое обозначение	Английский	Русский	Тип операндов
E e	Egual	Равно	Любые
N n	Not	Не	Любые
G g	Greater	Больше	Числа со знаком
L l	Less	Меньше	Числа со знаком
A a	Above	Выше, в смысле «больше»	Числа без знака
B b	Below	Ниже, в смысле «меньше»	Числа Без знака

В таблице 9 приведён перечень команд условного перехода для команды CMP операнд 1, операнд 2.

Таблица 9

Типы операндов	Мнемоническое обозначение	Критерий условного перехода	Значение флагов для осуществления перехода
Любые	je	операнд _1= операнд _2	Zf = 1
Любые	jne	операнд _1<> операнд _2	Zf = 0
Со знаком	jl /jnge	операнд _1< операнд _2	Sf < >of
Со знаком	jle /jng	операнд _1<= операнд _2	Sf < >of or zf =1
Со знаком	jg /jnle	операнд _1> операнд _2	Sf = of and zf = 0
Со знаком	jge /jnl	операнд _1= >операнд _2	Sf = of
Без знака	Jb/jnae	операнд _1<операнд _2	Cf =1
Без знака	jbe /jna	операнд _1<= операнд _2	Cf =1 or Zf =1
Без знака	ja /jnbe	операнд _1> операнд _2	Cf = 0 and Zf = 0
Без знака	jae /jnb	операнд _1= >операнд _2	Cf = 0

Мнемоническое обозначение части команд условного перехода отражает название флага, с которым они работают. В таблице 10 приведены названия флагов и соответствующие команды условного перехода.

Таблица 10

Название флага	Номер бита в регистре флагов	Команда условного перехода	Значение флага для осуществления перехода
Флаг переноса CF	1	JC	CF=1
Флаг чётности PF	2	JP	PF=1
Флаг нуля ZF	6	JZ	ZF=1
Флаг знака SF	7	JS	SF=1
Флаг переполнения OF	11	JO	OF-1
Флаг переноса CF	1	JNC	CF=0
Флаг чётности PF	2	JNP	PF=0
Флаг нуля ZF	6	JNZ	ZF=0
Флаг знака SF	7	JNS	SF=0
Флаг переполнения OF	11	JNO	OF=0

Регистр CX, кроме того, что он является регистром общего назначения, имеет определённое функциональное назначение – он выполняет роль счётчика в командах управления циклами и при работе с цепочками символов.В связи с этим существуют две команды условного перехода:

- **JCXZ –** переход, если CX нуль (для I8086);
- **JECXZ** – переход, если ECX нуль.

Команды организации циклов. Для удобства выполнения вычислительных циклов в систему команд введена группа из трёх команд, облегчающая

программирование циклов. Эти команды используют регистр CX(ECX) как счётчик цикла.

- **LOOP** – повторить цикл.
- **LOOPE/LOOPZ –** повторить цикл, пока CX< >0 или ZF=0.
- **LOOPNE/LOOPNZ –** повторить цикл, пока CX< >0 или ZF=1.

Работа команды LOOP заключается в выполнении следующих действий:

- декремента регистра CX (ECX);
- сравнение регистра CX (ECX) с нулём.

Если CX (ECX) > 0, то управление передаётся на метку перехода;

если CX (ECX) = 0, то управление передаётся на следующую после LOOP команду.

Алгоритм работы команды LOOPE/LOOPZ таков:

- декремент регистра CX (ECX);
- сравнение регистра CX (ECX) с нулём;
- анализ состояния флага нуля ZF.

Если CX (ECX) > 0 и ZF = 1, то управление передаётся на метку перехода;

если CX (ECX) = 0 или ZF = 0, то управление передаётся на следующую после LOOP команду.

Работа команды LOOPNE/LOOPNZ заключается в выполнении следующих действий:

- декремента регистра CX (ECX);
- сравнение регистра CX (ECX) с нулём.
- анализа состояния флага нуля ZF.

Если CX (ECX) > 0 и ZF = 0, то управление передаётся на метку перехода;

если CX (ECX) = 0 или ZF = 1, то управление передаётся на следующую после LOOP команду.

Команды прерывания. Программные прерывания бывают трёх типов:

- **INT type** – прерывание по номеру вектора (type);
- **INTO** – прерывание при переполнении;
- **IRET** – возврат из подпрограммы прерывания.

Во втором байте команды INT type помещается 8-разрядный номер вектора прерываний, полученный процессором номер вектора умножается на 4, чтобы найти точку входа в таблицу векторов. В таблице векторов, максимальный размер которой один килобайт, на каждый вектор отводится по 4 байта: в первых двух байтах помещается значение IP, во вторых двух байтах – значение CS. Новые значения регистров CS и IP помещаются в соответствующие регистры микропроцессора, таким образом адрес соответствующей подпрограммы определён. Для того, чтобы адрес возврата в основную программу не был утрачен, предварительно производятся следующие действия: текущие значения регистра CS и указателя команд записываются в стек, так-

же в стек записываются значения регистра флагов F и сбрасываются флаги IF и TF. Обычно пользовательский резерв начинается с вектора № 32.

Однобайтная команда INT, отличающаяся от двухбайтной одним битом, не требует специального указания уровня прерывания. Она автоматически воспринимается процессором как прерывание третьего уровня (type 3), и используется в программах для организации контрольной точки.

Однобайтная команда INTO вызывает переход на обслуживание прерывания четвёртого уровня, связанного с переполнением, когда значение флага переполнения OF = 1. Команда INTO обычно используется после арифметических команд над числами со знаком.

4.3.7. Команды управления процессором

Существует три типа команд управления процессором: команды работы с флагами, установки МП в особые состояния и синхронизации с сопроцессором.

Команды операций с флагами. Эти команды включают семь мнемокодов:

- **STC** – установка флага переноса CF;
- **CMC** – инвертирование флага переноса;
- **CLC** – сброс флага переноса;
- **STD** – установка флага направления передачи DF;
- **CLD** – сброс флага направления DF;
- **STI** – установка флага разрешения передачи;
- **CLI** – сброс флага разрешения прерывания.

Назначение этих команд очевидно и комментарий не требует.

Команды установки микропроцессора в особые состояния. Это команды **HLT** (останов) и **WAIT** (ожидание). По той и другой команде процессор начинает отсчитывать такты ожидания TW, которые прекращаются при определённых внешних воздействиях. Из команды останова HLT процессор может быть выведен двумя способами: подачей начального сброса RESET или внешним прерыванием, что широко используется при организации прерываний. В первом случае процессор перейдёт к выполнению основной программы сначала, во втором – к выполнению подпрограммы обслуживания прерывания. После выполнения подпрограммы процессор обязан перейти к следующей за командой HLT команде, следовательно, при необходимости возвращения к команде останова в подпрограмме необходимо поставить команду JMP с указанием адреса команды HLT.

При выполнении команды WAIT процессор аппаратно проверяет вход TEST, когда процессор обнаружит на этом входе сигнал нулевого уровня, состояние ожидания прекращается. Управление ожиданием с помощью тандема WAIT–TEST позволяет осуществить синхронизацию, т.е. сопряжение во

времени работы процессора с внешними устройствами (например, с арифметическим сопроцессором).

Другой способ вывода процессора из состояния ожидания заключается в подаче запроса прерывания по входу INT. Необходимо отметить, что по команде WAIT не происходит автоматического увеличения содержимого указателя команд IP, поэтому после выполнения программы обслуживания прерывания процессор вновь перейдёт к выполнению команды WAIT, т.е. перейдёт в состояние ожидания. Таким образом можно выполнять прерывающие программы во время ожидания сигнала готовности TEST от ВУ.

Команды синхронизации с арифметическим сопроцессором. Для организации совместной работы основного процессора с сопроцессором служит команда **ESC.** Более подробно эта команда будет рассмотрена при изучении арифметического сопроцессора.

Лекция № 9

4.4. Основные составляющие ассемблерных программ

4.4.1. Определение сегментов и групп

Основные составляющие ассемблерных программ подробно рассмотрены в пособии к циклу лабораторных работ.

Любая программа должна обязательно состоять из сегментов, без сегментов программ не бывает. Обычно в программе задаются три сегмента: команд, данных и стека, при необходимости организуют дополнительный сегмент. Сегменты могут быть самостоятельными блоками памяти или объединяться в зависимости от используемой модели памяти (модели памяти рассматриваются в учебном пособии по циклу лабораторных работ).

Большинство программ на языке ассемблера можно разделить на пять основных частей со следующими условными названиями: *заголовок, макроопределения, данные, тело программы и заключение.*

Заголовок

Любая программа начинается с заголовка. В нём содержатся команды и директивы, которые не приводят к созданию машинного кода при трансляции, но помогают при генерации исполняемого файла.

Макроопределения, директивы

После заголовка следуют различные описания переменных и констант. Для описания пользуются некоторыми *директивами (псевдооператорами*). Директивы управляют работой программы – ассемблера, а не микропроцессора. С помощью директив можно определять сегменты и подпрограммы (процедуры*),* давать имена командам и данным, резервировать области памяти и выполнять другие важные задачи. В отличие от команд языка ассемблера большинство директив не генерирует объектный код. В ассемблере константы часто называют *макроопределениями,* использующими директиву **EQU**, которая связывает значение с определённым именем (идентификато-

ром), вместо которого Turbo Assembler подставит указанное значение. Для числовых значений, кроме директивы EQU, можно использовать знак равенства **(=).**

Использование имён позволяет обращаться по именам к выражениям, что упрощает программу для чтения и отладки. Ниже приводится несколько примеров макроопределений, которые могут следовать за заголовком.

```
Count       EQU         6
Tent        =           Met + Count
```

Существует несколько правил, которые надо помнить при создании макроопределений в языке ассемблера.

- После описания имени константы с помощью директивы EQU нельзя изменять его значение.
- Это ограничение не распространяется на имена – идентификаторы, описанные с помощью знака равенства (=), – можно свободно изменять их значения. Это позволяется сделать в любом месте программы.
- EQU может описывать все типы равенств, включая числа, выражения и символьные строки. Знак равенства может описывать только числа либо арифметические выражения типа приведённого Met + Count.
- Имена-идентификаторы не являются переменными – ни они, ни их значения не содержатся в сегменте данных. Команды ассемблера не могут изменять их значения независимо от того, были они описаны с помощью директивы EQU или (=).
- Предпочтительнее располагать макроопределения в начале программы. Выражения, описанные с помощью EQU, вычисляются, когда соответствующее имя используется программой. Выражения, описанные через знак равенства (=), вычисляются непосредственно в месте определения.

Процедуры

Процедурой в ассемблере является всё то, что в других языках называют подпрограммами, функциями и т.д. Ассемблер не накладывает на процедуры никаких ограничений – на любой адрес программы можно передать управление командой CALL, и оно вернётся к вызвавшей процедуре, как только встретится команда RET. Такая свобода выражения легко может приводить к трудно читаемым программам, и в язык ассемблера были включены директивы логического оформления процедур.

Данные

Описание сегмента данных в программе можно расположить между макроопределениями и кодовым сегментом, в котором находятся команды программы, но допускается его размещение и в другом месте программы.

Сегмент данных программы должен начинаться с директивы **SEGMENT**, перед которой ставится имя, а после директивы можно поставить параметры. Директива даёт указание ассемблеру разместить в памяти переменные, указанные в этом сегменте. Сегмент данных может содержать два типа

переменных: инициализированных (т.е. определённых) и неинициализированных, т.е. не имеющих определённых значений перед запуском программы. Для неинициализированных переменных на месте определяющей память константы используют знак вопроса (?), при необходимости получения большего неинициализированного пространства знак вопроса ставится в скобках на месте DUP-выражений, что является полезным для создания больших буферов, например:

```
Big          db        8000 DUP (?)       ;  8000-байтовый буфер
```

Когда DOS загрузит вашу программу, в памяти создастся 8000-байтовый буфер, в котором будут находиться какие-то прежние значения. Рекомендуется все неинициализированные значения ставить в сегменте данных последними.

Для инициализации переменных чаще всего используются три директивы: **DB** (определить байт), **DW** (определить слово), **DD** (определить двойное слово). Для инициализации данных операнд должен быть константой, выражением, вычисление которого даёт константу, или цепочкой. Указанные директивы применяются для размещения конкретных данных в ячейках или просто для резервирования пространства без инициализации.

Операторы инициализации и резервирования данных имеют следующий формат:

Переменная Мнемоника Операнд...,Операнд ; Комментарий

Здесь переменная не обязательна, но при её наличии ей назначается смещение первого байта, резервируемого директивой. Переменная должна заканчиваться пробелом (в отличие от метки, которая должна заканчиваться двоеточием.)

Например, директивы

```
D-BYTE    DB      104,10H
D-WORD    DW      100, 100H, –5
D-ABC     DB      0,?,?,10
```

Вероятно, вы обратили внимание, что числа в программе могут быть записаны в любой системе счисления, но в памяти число будет только в двоичном коде.

Тело программы

Кодовый сегмент содержит вашу исполняемую программу, которая имеет четыре колонки: *метки, мнемоника, операнды и комментарии.* Начинается сегмент также с директивы SEGMENT, перед которой ставится имя. Оно может быть любым, но удобнее использовать именно слово CODE. Метки помечают места в программе, на которые могут ссылаться другие команды и директивы. В кодовом сегменте метка всегда заканчивается двоеточием (:), в сегменте данных двоеточие не используется. Во второй колонке содержатся мнемоники, т.е. машинные команды. Третья колонка содержит

операнды, которые обрабатываются мнемоническими командами. Некоторые команды не требуют операндов. Четвёртая колонка используется для комментариев. Из перечисленных колонок обязательной является только вторая (мнемоника команды). Перед комментариями обязательно ставится точка с запятой. Многие программисты начинают свои программы многострочными пояснительными комментариями.

После строчки с директивой SEGMENT идёт строчка с первой командой, начинающейся с метки, которая определяет точку входа в программу. Завершается программа директивой END с именем метки, определяющей точку входа.

5. РЕАЛИЗАЦИЯ МИКРОПРОЦЕССОРНОЙ СИСТЕМЫ НА БАЗЕ 16-РАЗРЯДНЫХ МИКРОПРОЦЕССОРОВ

5.1. Функционирование микропроцессора

5.1.1. Способы обмена информацией в микропроцессорной системе. Формирование системных управляющих сигналов в минимальном и максимальном режимах, совмещённая и раздельная адресации

При обмене информацией микропроцессора, работающего в минимальном режиме, с памятью и внешними устройствами используют один из двух вариантов: совмещённую или раздельную адресацию. Это выражается в разных способах формирования системных управляющих сигналов для блоков памяти и внешних усройств. Задача может быть решена с помощью комбинационных логических схем, которые формируют требуемые управляющие сигналы на основе сигналов RD, WR и M/IO, вырабатываемых МП. Если в системе используется адресное пространство ввода – вывода, изолированное от пространства памяти (раздельная адресация), то целесообразно сформировать сигналы MEMR, MEMW, IOR, IOW. Эти сигналы управляют запоминающими и внешними устройствами отдельно, независимо от информации на ША, тем самым увеличивается адресное пространство памяти и внешних устройств. Обращаться к внешним устройствам в этом случае нужно только с помощью команд IN или OUT. Роль формирователей сигналов могут выполнять элементы ИЛИ-НЕ (рис. 11) или дешифратор на три входа (например, К155ИД7).

Если же в МПС ввод – вывод организован с отображением на память (совмещённая адресация), то сигнал M/IO не используется, и на ЗУ, и ВУ подаются сигналы RD и WR после усиления. Преимущество такого способа подключения ВУ в том, что появляется возможность использования большого числа команд, предназначенных для обработки данных из памяти, вместо двух – команд *IN* и *OUT* .

В максимальном режиме управляющие сигналы MEMR, MEMW, IOR, IOW вырабатывает системный контроллер К1810ВГ88.

5.1.2. Структура и формирование системных магистралей (шин) микропроцессорной системы

При разработке структуры блока ЦП (рис. 11) возникают задачи разделения (демультиплексирования) шины адреса/данных (ШАД) и буферирования шин адреса (ША) и шин данных (ШД).

Первая задача решается с помощью ИС К1810ИР82/83, выполняющих функции адресной защелки. Так как сигнал BHE формируется в том же интервале времени, что и адресные сигналы, то его также необходимо зафиксировать в защелке. Поэтому изображенные на рис. 11 два 8-битовых регистра К1810ИР82 обеспечивают запоминание 15 разрядов адреса. Для доступа к памяти максимальной емкостью 1 Мбайт необходимо подключить еще один регистр, на который подаются оставшиеся старшие разряды AD15, A19/S 6 – A16/S3.

Вторая задача решается с помощью двунаправленных 8-битовых шинных формирователей К1810ВА86/87, которые усиливают сигналы системной шины данных.

Используемые усилители и формирователи должны обеспечивать три выходных состояния, чтобы можно было организовать прямой доступ к памяти. В этом случае после перевода МП в состояние захвата эти усилители переходят в третье состояние по сигналу РМ (BUSEN), поступающему от контроллера ПДП. Если захват шин и обмен данными по ПДП не предусмотрен, то необходимость в таком переключении отпадает.

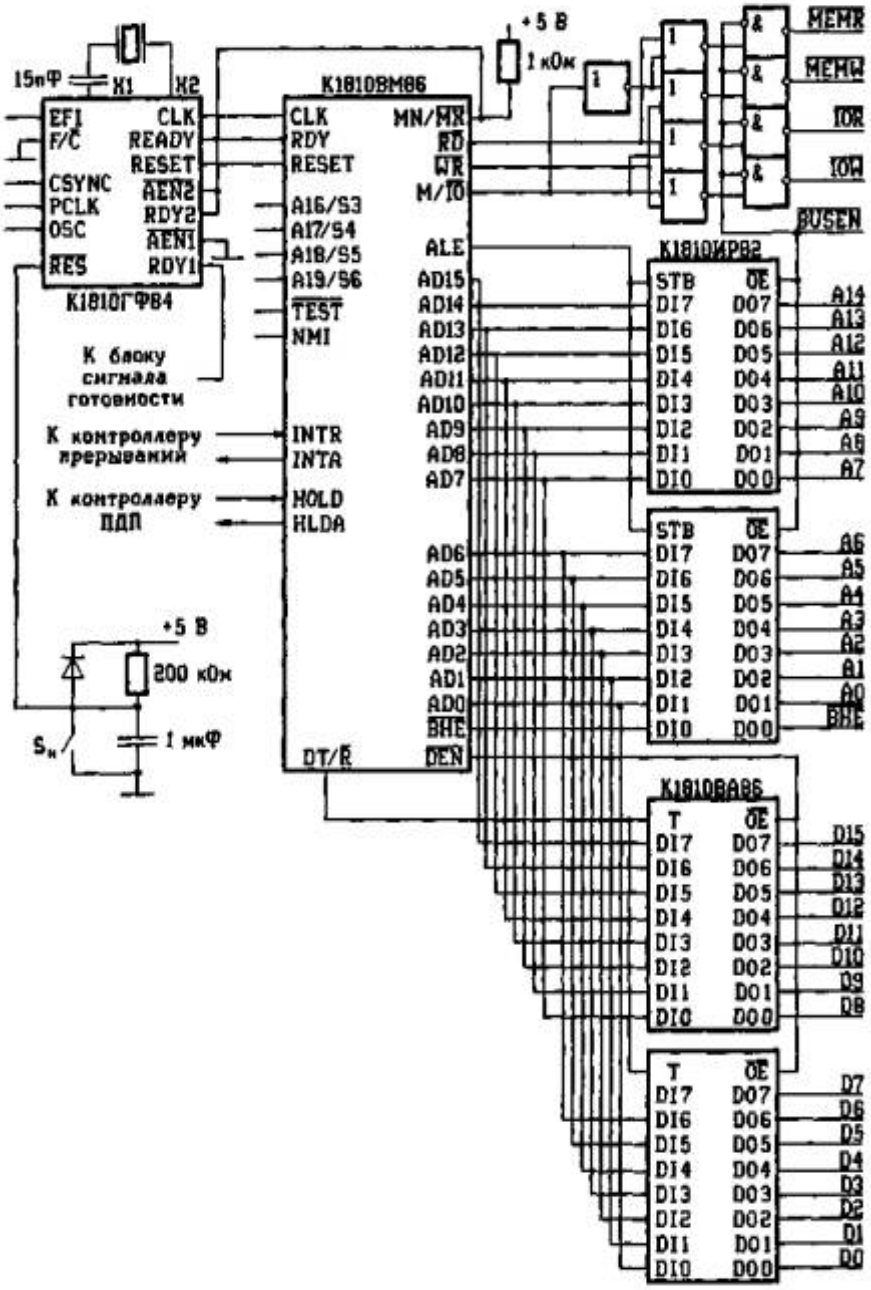


Рис. 11. Структурная схема ЦП на основе ВМ86

Лекция № 10

5.1.3. Минимальный режим

Минимальный и максимальный режимы отличаются по своим функциональным возможностям. Для введения минимального режима контакт *MN / MX* подключается к «1». Ещё раз обратимся к рис.11, на котором представлена типичная конфигурация минимального режима. Поскольку 16 младших разрядов ША мультиплексированы с ШД, адрес необходимо фиксировать в регистрах-защёлках.

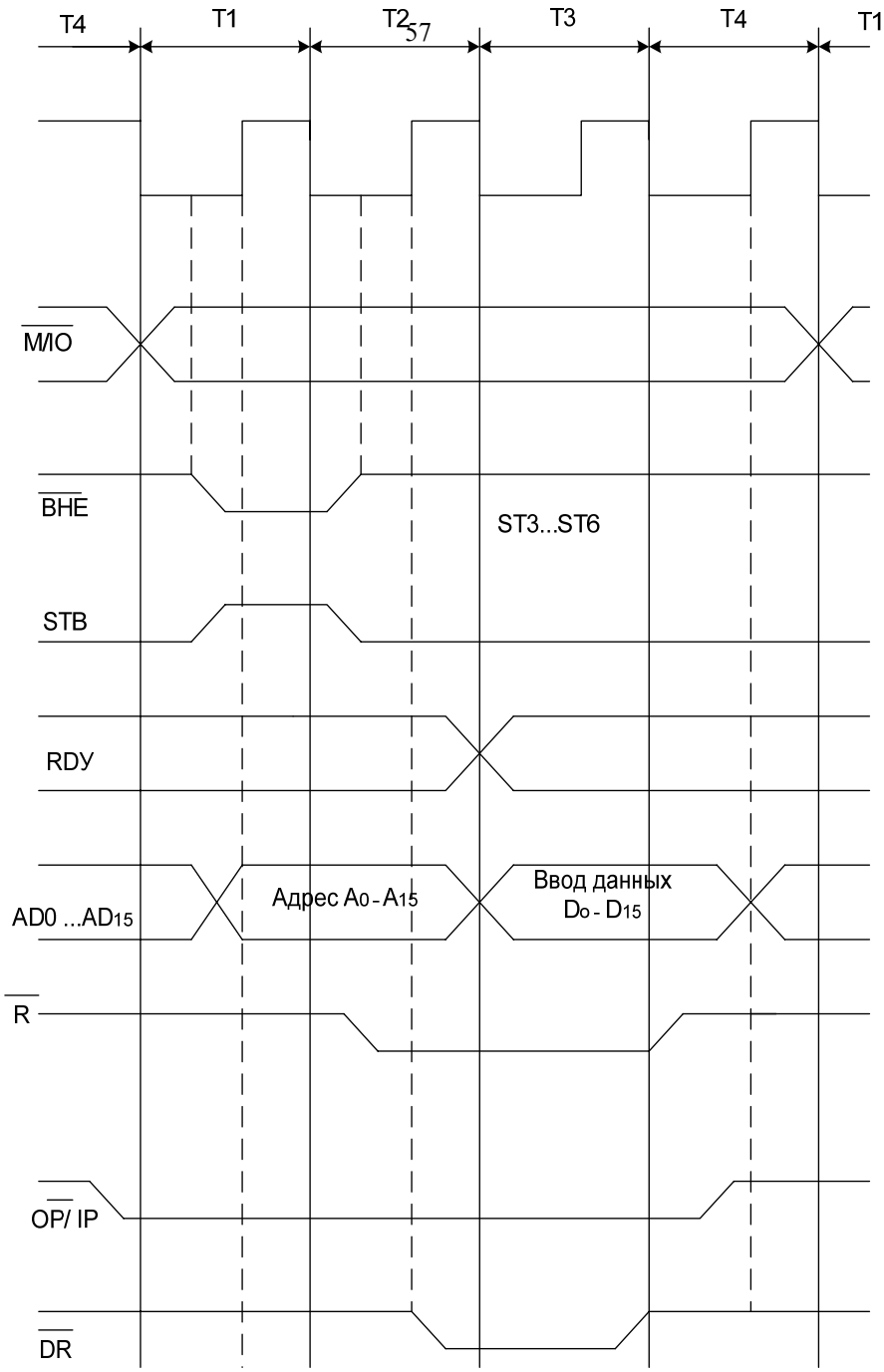


Рис. 12. Временные диаграммы последовательности чтения для min режима КР1810ВМ86

Для 8-разрядных регистра К1810ИР82 (18282), каждый из которых рассчитан на 8 бит, запоминают адреса, один из разрядов используется для фиксирования сигнала разрешения старшего байта шины $\overline{BHE}$. Для доступа ко всему объему памяти (1 Мбайт) необходимо включение в систему еще одного регистра, на который следует подать пять старших адресных разрядов: *АД*15 и $A16 \div A19$. Для фиксирования адреса используется сигнал *STB*. Вход *ДЕ* регистров-защелок позволяет перевести регистры в высокоимпедансное

состояние в режиме ПДП. В однопроцессорных системах без контроллера ПДП этот контакт заземляется.

Для управления взаимодействием с памятью и внешними устройствами МП генерирует сигналы чтения $\overline{RD}$, записи $\overline{WR}$ и обращения к памяти или к ВУ $M/\overline{IO}$. Как уже говорилось, с помощью внешней логики, показанной на рис. 11, можно сформировать сигналы $\overline{MRD}, \overline{MW}, \overline{IOR}, \overline{IOW}$, позволяющие разделить адресное пространство памяти и ВУ.

В системе с несколькими интерфейсами требуются приемопередатчики на ШД. Для этого предназначаются микросхемы К1810ВА86 (18286). Каждая микросхема содержит 16 тристабильных элементов – 8 приемников и 8 передатчиков (драйверов). Следовательно, для обслуживания 8 линий ШД 18086 – две таких микросхемы.

Сигнал *ДЕ* разрешения пересылки данных подается на вход разрешения вывода *ДЕ* шинных формирователей К1810ВА86. Направление пересылки определяется сигналом $OP/\overline{IP}$, поступающим от процессора на вход T ($ДТ/\overline{R}$) приемопередатчика. Процессор переводит линии $\overline{ДЕ}$ и $OP/\overline{IP}$ в высокоимпедансное состояние в ответ на запрос шины по линии HLD.

Иногда системная шина проектируется так, что сигналы адреса или данных инвертируется. Поэтому микросхемы К1810ИР82 (18282) и К1810ВА86 (18286) имеют свои полные аналоги 18283 и 18287, осуществляющие инверсию сигналов при передаче со входов на выходы.

Все типы циклов по обмену информации могут быть объединены в 2 базовых цикла: чтения и записи. На рис. 12 представлены диаграммы работы микропроцессора, работающего в минимальном режиме, для машинного цикла «чтение». Машинный цикл (м. ц.) чтения начинается в такте $T1$ с установки адресной информации и с подачи разрешения фиксации адреса STB. Адрес запишется в буферах К1810ИР82 (или ИР83).

Сигнал $\overline{BHE}$ показывает, будет ли пересылаться только младший байт («1») или все слово («0»). В тактах $T1$ – $T4$ сигнал $M/\overline{IO}$ указывает, что является источником данных – память или ВУ.

В такте $T2$ завершается выдача адреса, и выходы буферов $A_0 - A_{14}$ переводятся в 3-е состояние, на линиях $A16 - A19$ появляются сигналы $ST3...ST6$, которые сохраняются до конца $T4$, начинается управляющий сигнал чтения R. Данные передаются в тактах *Т3*, $T4$, при этом в такте $T3$ адресуемое устройство выдает сигнал готовности RDУ, позволяющий синхронизировать скорость работы памяти и МП введением между $T3$ и $T4$ дополнительных тактов ожидания, если на входе RДУ окажется «0». Затем в такте $T4$ сигнал чтения Rустанавливается в «1», тем самым выводы адресуемого устройства устанавливаются в 3-е состояние и освобождают системную ШД.

Сигнал $OP/\overline{IP}$ в $T1$ – $T4$ определяет сигналы формирователям ВА86 режим приема (т.е. направление), а сигнал *ДЕ* разрешает передачу данных в точке $T3$. Чтение кода команды и данных осуществляется аналогично, отли-

чие в том, что данные направляются в блок регистров, а код операции в регистр очереди команд.

5.1.4. Максимальный режим

Максимальный режим используется в структурах микропроцессорных систем повышенной сложности. Он устанавливается при заземлении вывода $MN / \overline{MX}$. Режим характеризуется тем, что устанавливаются дополнительные управляющие сигналы, обеспечивающие совместную работу микропроцессора с другими микропроцессорами, арифметическими сопроцессорами или процессорами ввода-вывода. При установке максимального режима изменяется назначение восьми выводов микропроцессора. Еще раз акцентируем внимание на этих выводах, рассмотрев их в таблице 11.

На рис. 13 приведена типичная конфигурация максимального режима, на рис. 14 временные диаграммы работы.

На выводы $\overline{ST0}$, $\overline{ST1}$, $\overline{ST2}$ в тактах $T1$, $T2$ и $T4$ выводится код состояния микропроцессора, характеризующий текущий цикл и определяющий способ использования ША/ШД. Коды состояний приведены в табл. 11. За счет кодирования кодов состояния освобождается ряд выводов. Во время тактов $T3$ и TW, когда шины не используются МП, на выводах $\overline{ST0}$, $\overline{ST1}$, $\overline{ST2}$ присутствует код 111. Код состояния позволяет выработать все необходимые управляющие сигналы при помощи внешнего средства – системного контроллера К1810ВГ88.

Выводы $\overline{RQ} / \overline{GTO}$ и $\overline{RQ} / \overline{GT1}$ являются двунаправленными линиями, причем вывод $\overline{RQ} / \overline{GTO}$ имеет более высокий приоритет. Вывод служат для захвата системной шины другими устройствами. Захват начинается с поступления на один из входов $\overline{RQ} / \overline{GTO}$ и $\overline{RQ} / \overline{GT1}$ запроса захвата, во время очередного такта $T1$ или $T4$ микропроцессор подает на этот же вывод ответственный сигнал разрешения захвата. В следующем такте микропроцессор входит в режим захвата и устанавливает шины и ряд других выводов в высокоимпедансное состояние. После окончания использования системных шин ВУ сообщает об этом микропроцессору по тем же линиям. Получив этот сигнал, микропроцессор возобновляет работу.

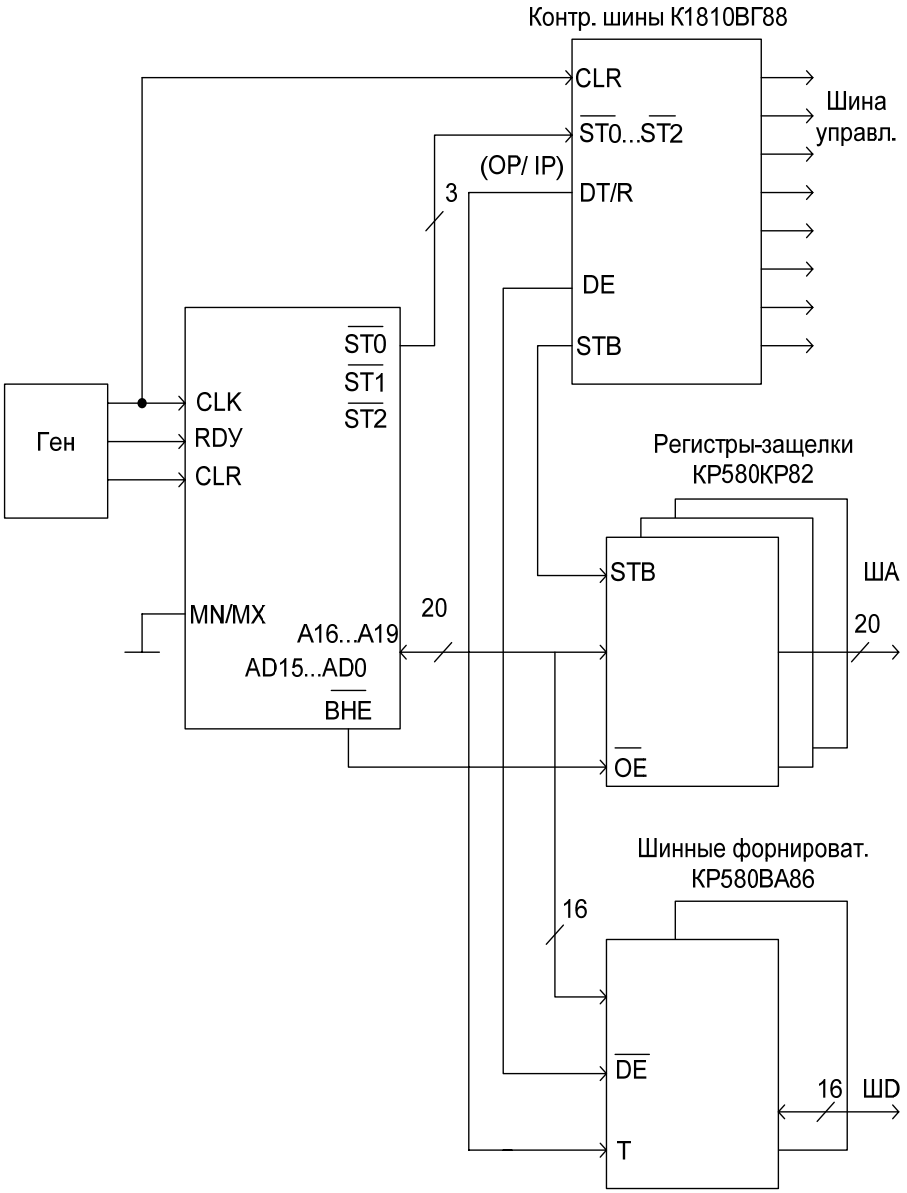


Рис. 13. Максимальный режим

Активный сигнал на выводе $\overline{LOCK} = 0$ вырабатывается с помощью специальной команды (префикса) $\overline{LOCK}$ и позволяет запретить захват шин микропроцессора внешними устройствами на время выполнения следующей за префиксом $\overline{LOCK}$ команды. Выводы $QS0$, $QS1$ несут информацию о состоянии очереди команд микропроцессора, комбинации сигналов на этих выводах приведены с таблице 7.

Сигналы $QS0$, $QS1$ действуют в течение такта синхронизации после выполнения операции над очередью, они предназначены для сопроцессора, который воспринимает команды и операции с помощью команды ECS. Подробнее эта ситуация будет рассмотрена в разделе 4.2. «Сопроцессорные конфигурации».

Таблица 11.

Минимальный режим ($MN/\overline{MX}=1$)	Максимальный режим ($MN/\overline{MX}=0$)	Назначение выводов в максимальном режиме
$\overline{WR}$	$\overline{LOCK}$	Блокировка
$\overline{INTA}$	$QS0$	Состояние очереди команд
STB	$QS1$	
$M/\overline{IO}$	$\overline{ST2}$	Состояние МП
$OP/\overline{IP}$	$\overline{ST1}$	
$\overline{DE}$	$\overline{ST0}$	
HLD	$\overline{RO}/\overline{GT0}$	Запрос/предоставление
$HLDA$	$\overline{RO}/\overline{GT1}$	

Для декодирования сигналов $\overline{ST0}, \overline{ST1}, \overline{ST2}$ и в систему включается контроллер системной шины К1810ВГ88 (18288), позволяющий сделать полную развязку ША, ШД и управляющей шины. С его помощью реализуется раздельная адресация ВУ и памяти.

Схема системного контроллера приведена на рис. 14. В схему входят дешифратор состояний, формирователи управляющих и командных сигналов и схема управления. Работа схемы тактируется импульсами с тактового генератора. Входы $\overline{AEN}, CEN$ и IOB необходимы для организации работы в многопроцессорной системе. Сигнал $\overline{AEN}$ служит для разрешения выдачи семи управляющих сигналов, поступающих с формирователей командных сигналов. Сигнал CEN разрешает как выдачу командных сигналов, так и выдачу сигналов управления $\overline{DEN}$ и $\overline{PDEN}$.

Сигналы $\overline{AEN}$ и CEN подаются с арбитра шин К1810ВБ89. Вход IOB используется для задания режима работы системного контроллера на него подается постоянный потенциал логической «1» или «0». При $IOB=1$ системный контроллер функционирует в режиме шины ввода – вывода, а при $IOB=0$ – в режиме системной шины. Сигналы $\overline{DEN}$ (DE) и $\overline{PDEN}$ используются, как и в минимальном режиме, для разрешения или запрещения работы приемопередатчиков шин данных в зависимости от потенциалов на входе IOB.

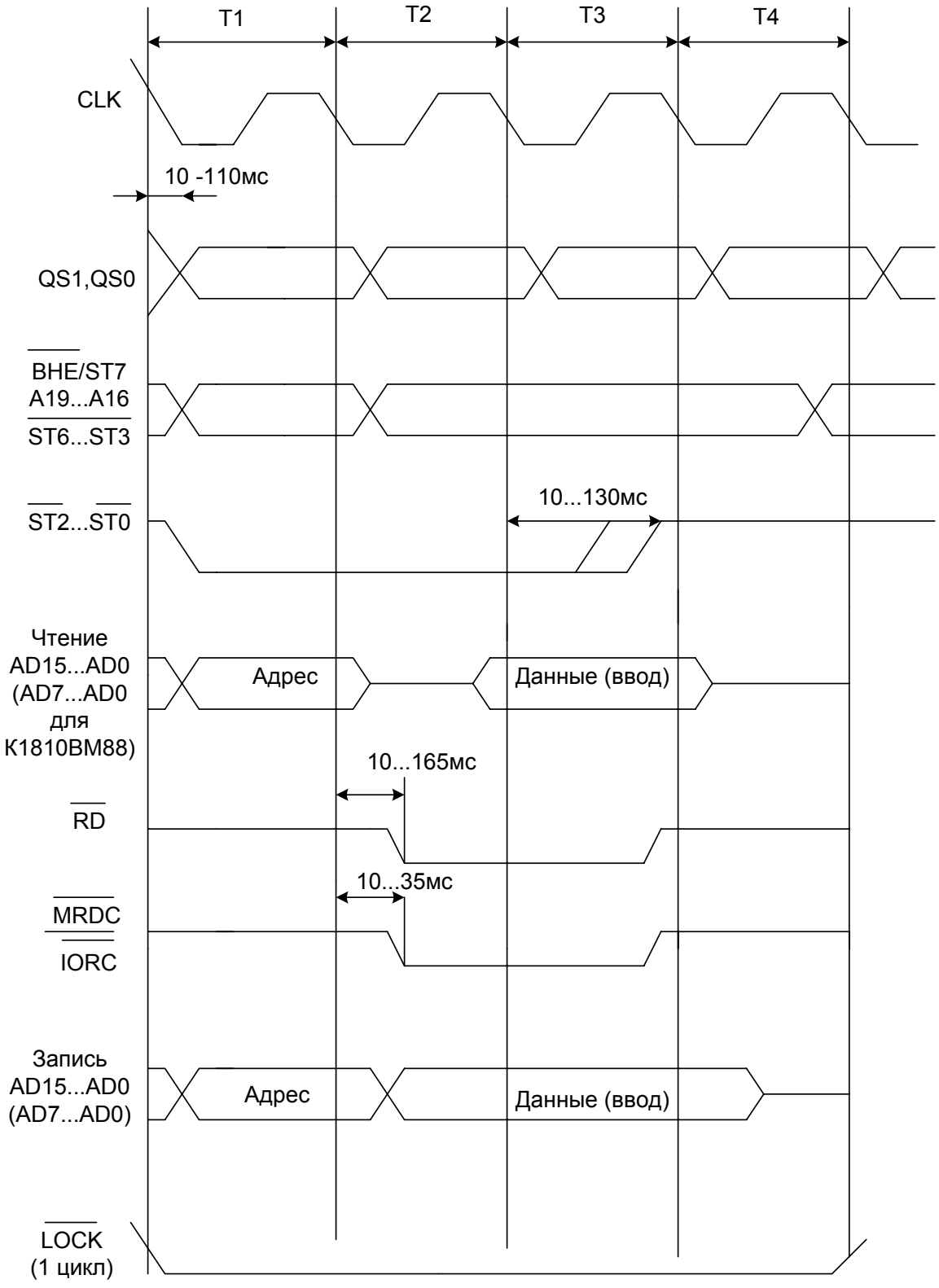


Рис. 14. Временные диаграммы работы МП К1810ВМ86 в максимальном режиме

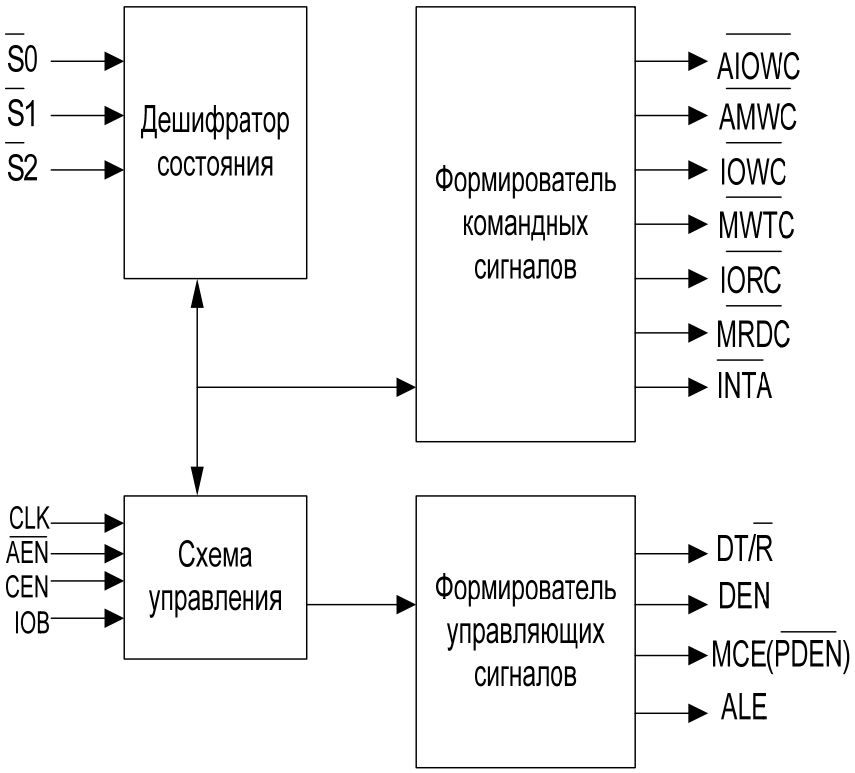


Рис. 15. Структура системного контроллера К1810ВГ88

Командные сигналы вырабатываются системным контроллером (рис. 15) в режиме системной шины (при $IOB = 0$) в соответствии с кодом состояния $\overline{ST0}$, $\overline{ST0}$, $\overline{ST2}$ при наличии разрешающего сигнала $\overline{AEN}$. Сигнал MCE используется только при каскадировании контроллеров прерываний и служит для считывания адреса ведомого контроллера приоритетных прерываний. Сигналы $\overline{IOWC}$ и $\overline{IORC}$ служат для ВУ командами записи и чтения информации с шины данных, сигналы $\overline{MRDC}$ и $\overline{MWTC}$ – командами чтения и записи соответственно. Сигналы $\overline{AIOWC}$ и $\overline{AMWC}$ – сигналы опережающей записи для ВУ и памяти и служат для их подготовки к записи информации.

Выходной сигналы $\overline{INTA}$ определяет начало обработки запроса прерывания от ВУ.

Назначение сигнала $ALE(STB)$ такое же, как и в минимальном режиме: защелкивание адреса в буферном регистре.

Лекция № 11

5.2. Генерация и обработка прерываний

5.2.1. Внешние, внутренние и программные прерывания

Ранее говорилось, что прерывания – это вызываемый определённым образом процесс, переключающий МП на выполнение другой программы с по-

следующим возвратом к прерванной программе. Прерываний в МПС на базе I8086/88 256 типов (0…255). Они делятся на внешние аппаратные, внутренние аппаратные и программные. Любые прерывания требуют новых адресов подпрограмм прерываний, которые выбираются из таблицы векторов. Каждый вектор представляет собой два 16-разрядных слова, расположенных в соседних ячейках памяти, причём в ячейке с меньшим адресом (чётным) содержится смещение в кодовом сегменте (новое значение IP), а в ячейке с большим адресом новое значение базового адреса кодового сегмента (значение CS). Под векторы (указатели) прерываний в общем пространстве адресов памяти зарезервирована область 0-3FFh.

Внешние аппаратные прерывания вызываются внешними по отношению к МП событиями, возникающими асинхронно по отношению к исполняемой программе. Процессор может воспринимать прерывания после выполнения каждой команды, длинные строковые команды имеют для этого специальные окна. Аппаратные прерывания подразделяются на маскируемые и немаскируемые. Для этого у процессора есть два входа: INTR (Interrupt Requist) и NMI (Non Mascable Interrupt). **Маскируемые прерывания** вызываются переходом в единицу сигнала на входе INTR и выполняются при установленном флаге разрешения прерываний (IF=1). Управление флагом IF производят команды CLI – сброс флага IF =0 и STI – установка флага IF=1(однобайтные команды, выполняемые за два такта). В случае перехода к подпрограмме процессор сохраняет в стеке адрес возврата в основную программу (содержимое CS и IP), регистр флагов, сбрасывает флаг IF и вырабатывает два следующих друг за другом цикла подтверждения прерываний, в которых генерируются два соответствующих сигнала INTA (Interrupt Acknolidge). Высокой уровень сигнала на входе INTR должен сохраняться по крайней мере до появления сигнала INTA. Первый цикл подтверждения прерываний и сигнал INTA холостой, предупреждающий ВУ о принятии запроса и необходимости подготовки номера вектора. По второму сигналу INTA, служащему сигналом чтения в этом цикле, происходит считывание по шине данных байта, содержащего номер вектора, соответствующего данному типу аппаратного прерывания. Полученный номер вектора умножается в процессоре на 4, это значение помещается в регистр IP. Тем самым определяется точка входа в таблицу векторов, откуда считывается адрес □жет□ый□я□ммы, соответствующий полученному запросу. Обработка текущего прерывания может быть, в свою очередь, прервана немаскируемым прерыванием, а если в подпрограмме разработчик программы установит флаг IF, то и другим маскируемым аппаратным прерыванием (вложенные прерывания). После аппаратного сброса по сигналу CLR (RESET) флаг IF сброшен, и маскируемые прерывания не возможны.

По окончании выполнения подпрограммы по команде IRET МП извлекает из стека адрес возврата и сохранённое значение регистра флагов. При использовании команды RET (FAR) регистр флагов из стека не восстанавливается.

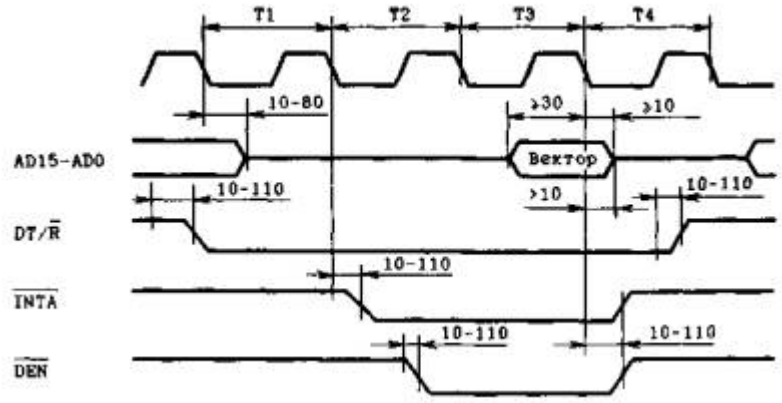


Рис. Диаграмма машинного цикла «Прерывание»

Немаскируемые прерывания выполняются независимо от состояния флага *IF* по сигналу *NMI* . Высокий уровень на этом входе вызовет прерывание с фиксированным номером 2 (см. табл. 4), которое выполняется так же, как и маскируемое. Этот вход используется для обработки в связи с экстренной ситуацией, например, сбоем по питанию, обнаружением ошибки памяти. Немаскируемое прерывание не требует номера вектора, поэтому в ответ на сигнал на входе *NMI* нет цикла «подтверждение прерывания», и сигнал на выходе *INTA* не формируется. Это помогает достигнуть высокой реакции, так как МП сразу же входит в таблицу векторов и считывает адрес соответствующей подпрограммы прерывания.

Внешние прерывания могут появляться в произвольный момент. Время реакции МП, определяющее запаздывание обслуживания прерывания, зависит от времени выполнения текущей команды. Наибольшее время выполнения у команд умножения, деления и команд сдвига на много бит.

Внутренние прерывания могут возникнуть по особым условиям в двух случаях:

прерывание (тип 0) при выполнении команд *DIV* и *IDIV* , вызывающее переполнение разрядной сетки;

пошаговые (отладочные) прерывания (тип 1).

Прерывание из-за ошибки деления возникает. Когда при выполнении команд *DIV* или *IDIV* формат частного превышает формат получателя или в случае деления на нуль.

Прерывание типа 1 обеспечивает пошаговую работу, им управляет флажок *TF* (флаг трассировки). Если *TF* =1, то по окончанию следующей команды возникает прерывание, вызывающее обращение к таблице векторов с адресами 00004…00007. При этом, как и в предыдущем случае. В стек автоматически записываются значения из регистров *CS*, *IP* и регистр флагов. При выполнении подпрограммы флаги *IF* и *TF* сбрасываются, чтобы при выполнении подпрограммы процессор работал обычным способом, а не в пошаговом режиме, но при выходе из подпрограммы исходное значение регистра флагов, включая и разрешенный флажок *TF* , восстанавливается. Таким обра-

зом, сразу после команды, выполненной за возвратом из подпрограммы, снова появится прерывание. Значит, до тех пор, пока флажок *TF* установлен, прерывание возникает после каждой команды. Флажок *TF* можно установить или сбросить посредством включения в стек регистра и извлечения этой информации в процессор с последующим необходимым маскированием. Для включения регистра флагов *F* в стек и извлечения его из стека предусмотрены команды *PUSHE* и *POPF* соответственно.

Таблица 16

Программные прерывания не зависят от состояния флагов *F*, они бывают трех видов:

пользовательские *INT* тип (номер вектора);

специальное прерывание *INT3*;

прерывание по переполнению *INTO*.

В прерываниях, определяемых пользователем при составлении программы, вторым байтом задается номер вектора прерывания. Считанный

жерой байт множится в процессоре на 4, тем самым определяется точка входа в таблицу векторов, где хранится адрес подпрограммы прерывания. При переходе к подпрограмме в стеке запоминается адрес возврата в программу (*CS* и *IP*) и регистр флагов. При этом может быть выполнении межсегментный переход.

Однобайтная команда *INT3* определена как прерывание контрольной точки (точки разрыва). Контрольные точки обычно используются как средства отладки. Эту команду также можно использовать, чтобы вставить дополнительный фрагмент программы без ее повторной трансляции.

Прерывания по переполнению (тип 4) выполняются по однобайтной команде *INTO*, которую обычно ставят в программе после арифметических команд над числами со знаком.

5.2.2 . Программируемый контроллер прерываний I8259 (К1810ВН59)

Для обработки внешних прерываний микропроцессоров I8086/88 предлагается программируемый контроллер прерываний 8259A фирмы Intel.

Система прерываний с одним контроллером. Микросхема 8259A выпускается в 28-контактном корпусе типа DIP и требует один источник питания напряжением +5 В. Ее организация и подключения в системе с максимальным режимом показаны на рис. 17. Ниже даются определения контактов микросхемы:

D7-D0. Для взаимодействия с ЦП по шине данных. В больших системах могут потребоваться шинные драйверы, а в малых системах достаточно прямых соединений.

INT. Для выдачи в ЦП сигнала запроса прерывания.

INTA#. Для приема сигналов подтверждения прерывания от ЦП. Предполагается, что подтверждение состоит из двух импульсов, что делает контроллер совместимым с микропроцессорами 8086/8088.

RD#. Сигнализирует контроллеру поместить на шину данных содержимое регистров IMR, ISR или IMR или приоритетный уровень. Что выдается на шину, зависит от состояния контроллера (см. далее).

WR# Сигнализирует контроллеру, что он должен принять данные с шины данных и использовать их для установки бит в словах приказов. Распределение принятых данных рассматривается далее.

CS#. Идентифицирует обращение к контроллеру. Подключается к шине адреса через дешифратор, который сравнивает старшие биты адреса 8259A с адресом, находящимся на шине адреса.

AO. Указывает порт 8259A, к которому производится обращение. Для каждого контроллера в адресном пространстве ввода-вывода системы необходимо зарезервировать два адреса.

IR7-IRO. Для восприятия запросов от интерфейсов ввода-вывода или других (ведомых) контроллеров.

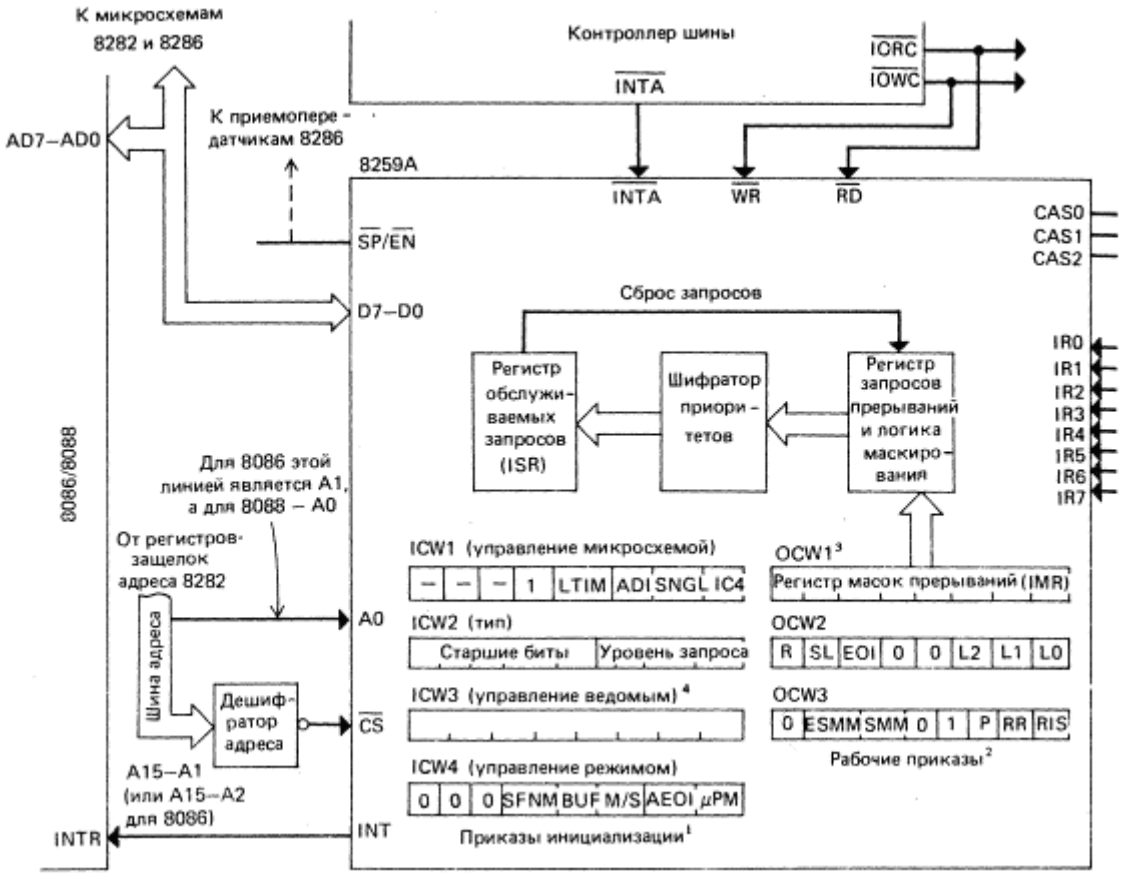


Рис.17. Организация программируемого контроллера прерываний

CAS2-CASO. Для идентификации конкретного ведомого контроллера. **SP#/EN#.** Выполняет две функции. Как вход определяет, каким может быть контроллер: ведущим (SP#/EN# == 1) или ведомым (SP#/EN# == 0). Как выход осуществляет запрещение приемопередатчиков шины данных, когда данные передаются из контроллера в ЦП. Использование в качестве входа или выхода рассматривается далее.

В системе на базе микропроцессора 8088 два адреса 8259A обычно соседние и линия адреса AO подключается на вход AO. Так как контроллер имеет всего восемь линий данных, а микропроцессор 8086 всегда вводит указатель прерывания по младшим 8 битам своей 16-битной шины данных, все передачи данных в (из) 8259A должны производиться по младшей половине шины. Наиболее просто гарантировать удовлетворение этого требования: подключить линию A1 на вход AO и использовать два соседних четных адреса, первый из которых кратен 4. Однако ради упрощения последующего обсуждения в обоих случаях будем считать второй адрес нечетным.

Секция управления 8259A имеет несколько программируемых бит, которые можно считать находящимися в семи 8-битных регистрах. Эти регистры разделены на две группы, первая из которых содержит слова приказов инициализации (ICW), а вторая — слова рабочих приказов (OCW) . *Слова приказов инициализации* обычно устанавливаются процедурой инициализации при включении вычислительной системы и при работе не изменяются.

Слова рабочих приказов применяются для динамического управления обработкой прерываний.

Регистр IRR (и логика маскирования), шифратор приоритетов и регистр ISR предназначены для восприятия и управления прерываниями на входах IR7-IRO. Регистр IRR фиксирует входные запросы и вместе с шифратором приоритетов разрешает незамаскированным запросам с достаточным приоритетом сформировать сигнал 1 на выходе INT. Шифратор приоритетов определяет приоритеты запросов в IRR, а регистр ISR предназначен для хранения текущих обрабатываемых запросов.

После того как бит в IRR установлен в состояние 1, он сравнивается с соответствующим битом маски из IMR. Если бит маски содержит 1, запрос передается в шифратор приоритетов, в противном же случае запрос блокируется. Когда запрос прерывания подается в шифратор приоритетов, □жет□ый□ется его приоритет, и если в соответствии с текущим состоянием шифратора приоритетов необходимо выдать прерывание в процессор, формируется сигнал INT.

Если флажок IF = 1, процессор реализует последовательность прерывания по завершению текущей команды и возвращает два отрицательных импуль□ж по линии INTA#. При действии первого импульса триггеры-защелки в IRR запрещаются, поэтому IRR игнорирует последующие сигналы на входах IR7-IRO. Такое состояние сохраняется до окончания второго импульса INTA#. Кроме того, первый импульс INTA# устанавливает соответствующий бит ISR и сбрасывает соответствующий бит в IRR. Второй импульс INTA# передает на линии D7-DO текущее содержимое ICW2 и этот байт процессор использует как тип прерывания. Если бит автоматического окончания прерывания (AEOI) в ICW4 содержит 1, по окончанию второго импульса INTA# сбрасывается бит ISR, который был установлен первым импульсом INTA#; в противном случае бит ISR не сбрасывается до выдачи в OCW2 соответствующего приказа окончания прерывания (EOI).

Как уже говорилось, слова приказов инициализации обычно загружает процедура инициализации при включении системы, и они содержат биты управления, не изменяющиеся при работе системы. Контроллер 8259A имеет четный адрес (AO = 0) и нечетный адрес (AO = 1) и слова приказов инициализации должны загружаться последовательно с использованием четного адреса для ICW1 и нечетного для остальных ICW.

Биты ICW1 определяются следующим образом:

Биты 7-5. Предназначены только для системы на базе микропроцессоров 8080/8085.

Бит 4. Всегда установлен в 1. Он направляет принятый байт в ICW1 в отличие от OCW2 или OCW3, которые также используют четный адрес (AC) = 0).

Бит 3 (LTIM). Определяет режим запуска фронтом (LTIM = 0) или уровнем (LTIM = 1). Режим запуска фронтом вызывает сброс бита IRR, когда устанавливается соответствующий бит ISR.

Бит 2 (ADI). Предназначен только для систем на базе микропроцессоров 8080/8085.

Бит 1 (SNGL). Показывает, каскадируется ли 8259A с другими контрол□жет□ы. Бит SNGL = 1, когда в системе прерываний имеется один контроллер.

Бит 0 (IC4). Устанавливается в 1, если в последовательности инициализации выводится ICW4. В системе с микропроцессорами 8086/8088 этот бит должен всегда содержать 1, так как бит 0 в ICW4 должен быть установлен в 1.

Биты 7-3 ICW2 загружаются из соответствующих бит второго байта, выводимого ЦП при инициализации 8259A, а биты 2-0 устанавливаются в соответствие с уровнем приоритета запроса прерывания, например запрос на ли□же IR6 загружает в эти биты код 110. Слово ICW3 предназначено для систем с несколькими контроллерами прерываний и выводится, если только SNGL== = 0 (см. раздел 8.3.2). Слово ICW4 выводится, если только бит 0 (IC4) установлен в 1; в противном случае содержимое ICW4 сбрасывается.

Биты ICW4 имеют следующие определения:

Биты 7-5. Всегда содержат 0.

Бит 4 (SFNM). Если установлен в 1, применяется специальный вложенный режим, предназначенный для систем с несколькими 8259A и рассматриваемый далее.

Бит 3 (BUF). Состояние BUF = 1 означает, что SP#/EN# служит выходом для запрещения системных приемопередатчиков 8286, пока ЦП вводит данные из 8259A. Если приемопередатчиков нет, BUF должен быть сброшен в О и в системах с одним контроллером на вход SP#/EN# следует подать 1.

Бит 2 (M/S). Этот бит игнорируется, если BUF = 0. В системах с одним контроллером этот бит должен содержать 1; в противном случае он должен быть в состоянии 1 для ведущего контроллера и в состоянии 0 для ведомых контроллеров.

Бит 1. (AEOI). Если AEOI = 1, в конце второго импульса INTA# сбрасывается бит ISR, который вызвал прерывание.

Бит 0 (JuPM). Состояние JuPM == 1 показывает, что контроллер находится в системе па базе микропроцессоров 8086/8088. Нулевое состояние подразумевает систему на базе микропроцессоров 8080/8085.

Типичный фрагмент установки содержимого ICW имеет следующий вид (четный адрес 8259A равен 0080):

```
MOV  AL,13H
OUT  80H,AL
MOV  AL,18H
OUT  81H,AL
MOV  AL,0DH
OUT  81H,AL
```

Первые две команды определяют запуск запросов фронтом, наличие в системе одного контроллера и необходимость вывода ICW4. Следующие две команды задают 5 старших бит типа прерывания равными 00011. Слово

ICW3 не выводится, так как SNGL = 1; следовательно, последние две команды определяют ICW4 = 0D, которое сообщает следующую информацию: специальный вложенный режим не применяется, сигнал SP#/EN# используется для запрещения приемопередатчиков, контроллер является ведущим, для сброса бита ISR необходим приказ EOI, контроллер 8259A работает в системе на базе микропроцессоров 8086/8088.

Имеется три слова OCW рабочих приказов. Слово OCW1 применяется для маскирования запросов прерываний; если бит маски, соответствующий запросу прерывания, содержит 1, запрос блокируется. Слова OCW2 и OCW3 предназначены для управления режимом контроллера и приема приказов EOI В OCW1 байт выводится с указанием нечетного адреса 8259A, а в OCW2 и OCW3 – с указанием четных адресов. Слово OCW2 отличается от OCW3 содержимым бита 3 байта данных. Если бит 3 содержит 0, байт помещается в OCW2, а если он содержит 1 – в OCW3. Оба слова OCW2 и OCW3 отличаются от 1CW1, которое также использует четный адрес, значением бита 4 данных. Если бит 4 равен 0, то байт помещается в OCW2 или OCW3 в зависимости от состояния бита 3. Неоднозначности интерпретации слов ICW2, ICW3, ICW4 и OCW1, использующих нечетный адрес, не возникает, так как слова инициализации должны всегда следовать за ICW1, как этого требует последовательность инициализации, и в середине этой последовательности вывод в OCW1 производить нельзя.

Обратимся к рис. 17. Биты L2-LO в OCW2 обозначают уровень IR, бит 5 предназначен для задания приказов EOI, а биты 6 и 7 управляют уровнями IR. Напомним, что, когда бит AEOI в ICW4 содержит 1, установленный запросом бит ISR автоматически сбрасывается в конце второго импульса INTA#. Если же AEOI = 0, бит ISR необходимо явно сбрасывать приказом EOI, который заключается в выдаче OCW2 с установленным в 1 битом 5. Когда выдается приказ EOI, четыре возможные комбинации бита 7 (бит R – ротации) и бита 6 (бит SL — установки уровня) приведены в табл. 17.

Таблица 17

R	SL	Действие
0	0	Режим обычных приоритетов
0	1	Сбрасывает бит ISR
1	0	Циклически изменяет приоритеты на одну позицию
1	1	Назначает низший приоритет с циклическим изменением остальных приоритетов

Биты OCW2 сохраняются в 8259A только на время выполнения OI.

Обычно запрос на линии IRO имеет наивысший приоритет, на линии IR1 — следующий меньший и т. д. Когда появляется первый импульс INTA, шифратор приоритетов разрешает только незамаскированному запросу с

наибольшим приоритетом установить свой бит ISR. Так как три младших бита ICW2 (слово ICW2 указывает тип прерывания и определяет адрес указателя прерывания) определяются тем, какой бит ISR установлен, то и адрес процедуры прерывания зависит от установленного бита ISR. Следовательно, первой начинается процедура прерывания устройства, подключенного к входу IR с наибольшим приоритетом, а остальные запросы должны ожидать разрешения дальнейших прерываний.

В режиме обычных приоритетов при установленном бите ISRn шифратор приоритетов не распознает запросов на линиях IR7-IR(n + 1), но распознает незамаскированные запросы на входах IR(n – 1)-IRO. Следовательно, если в процессоре флажок IF == 1, запросы с приоритетами выше обрабатываемого вызывают прерывание текущей процедуры прерывания, а запросы с меньшими приоритетами ожидают и обрабатываются в соответствии со своими приоритетами по мере сброса бит ISR с большими приоритетами. Если AEOI = 1, соответствующий прерыванию бит ISR автоматически сбрасывается в конце второго импульса INTA#. Когда же AEOI == 0, бит ISR должна сбрасывать процедура прерывания посредством установки бита 5 в OCW2.

Рассмотрим пример режима обычных приоритетов, приняв первоначально, что AEOI == 0 и все биты ISR и IMR сброшены. Предположим также, что, как показано на рис. 8.18, одновременно появляются запросы IR2 и IR4, затем появляется запрос на IR1 и последним появляется запрос на IR3. Сначала устанавливается бит ISR2 и начинает выполняться процедура прерывания, ассоциируемая с запросом IR2. После того как эта процедура устанавливает IF = 1 и появляется запрос IR1, устанавливается бит ISR1 и полностью выполняется процедура обслуживания запроса IR1. При своем выполнении она должна установить IF = 1 и выдать необходимый приказ для сброса ISR1. При возврате в процедуру IR2 сбрасывается бит ISR2. Затем устанавливается ISR4 и начинается его процедура, в течение которой возникает запрос IR3. Он подтверждается сразу же после установки IF = 1, устанавливается бит ISR3 и инициируется процедура обслуживания IR3. До своего завершения эта процедура должна сбросить ISR3 и установить IF = 1. Осуществляется возврат в процедуру IR4, которая перед возвратом в процедуру IR2 должна сбросить ISR4. Процедура IR2, которая уже сбросила бит ISR2, осуществляет обычный возврат в прерванную программу. (Отметим, что если флажок IF не устанавливается в самой процедуре, дальнейшие прерывания не обрабатываются до завершения процедуры, т. е. до выполнения команды IRET.)

Единичное состояние бита 5 в OCW2 обычно осуществляет сброс бита ISR с максимальным приоритетом (т. е. последнего установленного бита ISR). Но бит ISR можно сбросить явно, выдавая OCW2, в котором биты R, SL и EOI содержат комбинацию 011, а поле L2-LO идентифицирует номер сбрасываемого бита. Если, например, в OCW2 посылается байт 01100011, сбрасывается бит ISR3.

Кроме рассмотренного режима обычных приоритетов, приказ OCW2 может циклически изменять приоритеты, назначая низший приоритет любому из уровней IR. В этом случае остальные приоритеты изменяются так, как будто обычное упорядочивание « поворачивается вкруговую». Если, например, низший приоритет назначен IR4, получается следующий порядок приоритетов:

IR5, IR6, IR7, IR0, IR1, IR2, IR3, (R4.

Здесь IR5 «поворачивается» в позицию высшего приоритета. Циклическое изменение определяет комбинация 10 бит R и SL. Если эти биты содержат 11, низший приоритет назначается IR, определяемому полем L2-LO. Если, например, высший приоритет имеет IR5 и в OCW2 загружается байт 10100000, получается следующий порядок приоритетов:

IR6, IR7, IR0, IR1, (R2, IR3, IR4, IR5.

Когда же в OCW2 посылается байт 11100010, приоритеты упорядочиваются таким образом:

IR3, IR4, !R5, IR6. IR7, IR0, iR1, (R2.

Биты R и SL влияют на работу и когда EOI = 0. В этом случае комбинация R = 1 и SL = 0 вызывает автоматическое циклическое изменение приоритетов, когда AEOI == 1, а комбинация R = SL = 0 выключает это действие. Комбинация R == SL = 1 и EOI = 0 назначает низший приоритет запросу, определяемому полем L2-LO, без выдачи приказа EOI = 0. Оставшаяся комбинация R = 0 и SL == 1 не производит никаких действий.

В слове OCW3 биты ESMM (разрешение режима специальной маски) и SMM (режим специальной маски) можно использовать для отмены рассмотренных выше режимов. Если в OCW3 посылается байт, в котором ESMM = = SMM = 1, незамаскированные запросы прерываний обрабатываются по мере их появления (если флажок IF в процессоре содержит 1) и порядок их приоритетов игнорируется. Посылка в OCW3 байта, в котором ESMM = 1 и SMM = = 0, восстанавливает приоритетное упорядочивание запросов. Если, наконец, в OCW3 посылается байт с битом ESMM = 0, бит SMM не действует и режим специальной маски не изменяется.

Бит P (полинга, опроса) переводит контроллер в режим опроса. В этом режиме предполагается, что процессор не воспринимает прерываний (IF = 0) и необходимо опрашивать запросы прерываний в IRR. Когда P = 1, следующий сигнал RD вызывает установку соответствующего бита в ISR, как будто получен импульс INTA#, и передает в регистр AL процессора байт со следующим форматом:

1 – – – – W2 W1 W0

Здесь I=1 показывает наличие запроса прерывания, а поле W2 = WO содержит уровень IR прерывания с наибольшим приоритетом. Пусть, например. P = 1, существует такое упорядочивание приоритетов

IR3, IR4, IR5, IR6, IR7, IR0, IR1, IR2

и имеются незамаскированные прерывания на IR4 и IR1. Тогда команда **IN** AL,80H (где 0080 – четный адрес контроллера) загружает в регистр AL следующий байт:

1 – – – – 1 0 0

Когда P = 0, содержимое IRR или ISR можно считать в регистре AL установкой RR = 1 и выполнением команды IN AL,80H. Если во время выполнения команды IN бит R1S = 0, вводится содержимое IRR, а в противном случае – содержимое ISR. Содержимое IMR можно считать в любой момент времени, пользуясь нечетным адресом контроллера 8259A, например, при прежнем назначении адресов команда IN AL,81H вводит в AL содержимое IMR.

Так как биты OCW3 (кроме бита ESMM) определяют, находится ли контроллер в режиме специальной маски и какая информация выдается на шину данных при считывании, они сохраняются до изменения следующим выводом в OCW3. Например, когда P = 0, RR == 1 и R1S = 0, любое считывание по четному адресу до посылки в OCW3 нового байта осуществляет ввод в процессор содержимого IRR.

В заключение рассмотрим, что произойдет, когда на запросы влияют помехи. На входе IR должен сохраняться высокий уровень до фронта первого импульса INTA#. Если это условие не удовлетворяется, контроллер «моделирует» 1 на входе IR7. Таким образом, если устройство к входу IR7 не подключено, запросы по этой линии показывают наличие помех на других линиях запросов и процедура прерывания IR7 служат процедурой «очистки» помех. Если на вход IR7 подключено устройство, этот способ обнаружения помех мех все же можно использовать, так как запрос от устройства установит г ISR7, а «запрос» помехи на линии IR7 не воздействует на ISR7. Следовательно, процедура IR7 может различить эти два события, считывая ISR и проверяя бит 7.

Система прерываний с несколькими контроллерами. Схема системы прерываний с несколькими контроллерами 8259A представлена на рис.18. На ней не показаны шинные драйверы, но их можно подключить в соответствии с рис. 18. Выход SP#/EN# ведущего контроллера подключен к приемопередатчикам шины данных, а на входы SP#/EN# ведомых приборов подан уровень 0. Показан только один ведомый 8259A, но аналогично подключаются еще до 7 контроллеров, что обеспечивает максимум 64 линии запросов прерываний. При разработке дешифратора адреса каждому контроллеру необходимо назначить свою пару адресов в пространстве ввода-вывода. Драйверы на линиях CAS2-CASO могут и не потребоваться в зависимости от расстояний между ведущим и ведомыми контроллерами.

В рассматриваемой системе необходимо инициализировать ведущий и ведомые контроллеры. Ведущий инициализируется, как описано выше, но бит SNGL должен быть 0 и потребуется загружать слово ICW3. В каждый бит ICW3, для которого соответствующий вход IR подключен к ведомому прибору, необходимо записать I, а в остальные биты записываются нули. Бит

SFNM устанавливается в 1 для организации специального вложенного режима. При инициализации ведомых контроллеров бит SNGL должен содержать 0. Также образом, для каждого ведомого прибора потребуется ICW3, но у каждого прибора это слово несет различное содержание. Слово ICW3 имеет следующий формат:

0 0 0 0 0 ID2 ID1 ID0

Три младших бита определяют код идентификации ведомого контроллера. Он должен совпадать с номером той линии запроса ведущего контроллера, к которой подключается выход INT.

Сигнал INT == 1 ведомого контроллера подается на соответствующий вход IR ведущего 8259A. Если IMR и шифратор приоритетов не блокируют этот сигнал, он посылается в процессор через выход INT ведущего контроллера. Когда процессор возвращает сигнал INTA#, ведущий контроллер не только устанавливает бит ISR и сбрасывает бит IRR, но и проверяет соответствующий бит в ICW3 — возникло прерывание от ведомого прибора или нет. В случае прерывания от ведомого контроллера ведущий выдает на линии CAS2-CASO номер уровня IR; в противном случае он помещает содержимое ICW2 на шину данных и не выдает сигналов на линии CAS2-CASO. Сигнал INTA# поступает во все ведомые 8259A, но его воспринимает только тот прибор, код идентификации которого соответствует номеру, выданному ведущим прибором на линии CAS2-CASO. В выбранном ведомом приборе соответствующий бит ISR устанавливается, соответствующий бит IRR сбрасывается, а содержимое его ICW2 выдается на шину данных. Так как 1CW2 содержит тип прерывания, в процессе инициализации важно загрузить однозначные комбинации в ICW2 ведущего и ведомых приборов. Для этих приборов требуются приказы EOI, если их биты AEOI содержат 0.

За исключением реакции на сигнал INTA действия всех приборов в системе одинаковы. Аналогично же производится управление их режимами и считывание регистров. Однако имеется одно исключение. Если бит SFNM в ICW4 ведущего контроллера инициализирован на 1, он вводит специальный режим, который применяется с режимом обычных прерываний и AEOI = 0. В этом случае ведущий контроллер разрешает незамаскированным запросам с достаточным приоритетом проходить на выход INT, даже если соответствующий бит ISR уже установлен в 1. Это означает, что, если в ведомом контроллере появляется запрос с большим приоритетом в то время, когда обрабатываются запросы одного или нескольких ведомых контроллеров, новый запрос сформирует сигнал INT через ведущий контроллер. При использовании специального вложенного режима процедура прерывания может выдавать два приказа EOI. Сначала приказ неконкретного (неадресуемого) EOI выдается в ведомый прибор, который вызвал прерывание, а затем проверяется его ISR. Если и только если ISR содержит нули, в ведущий прибор выдается приказ неконкретного EOI.

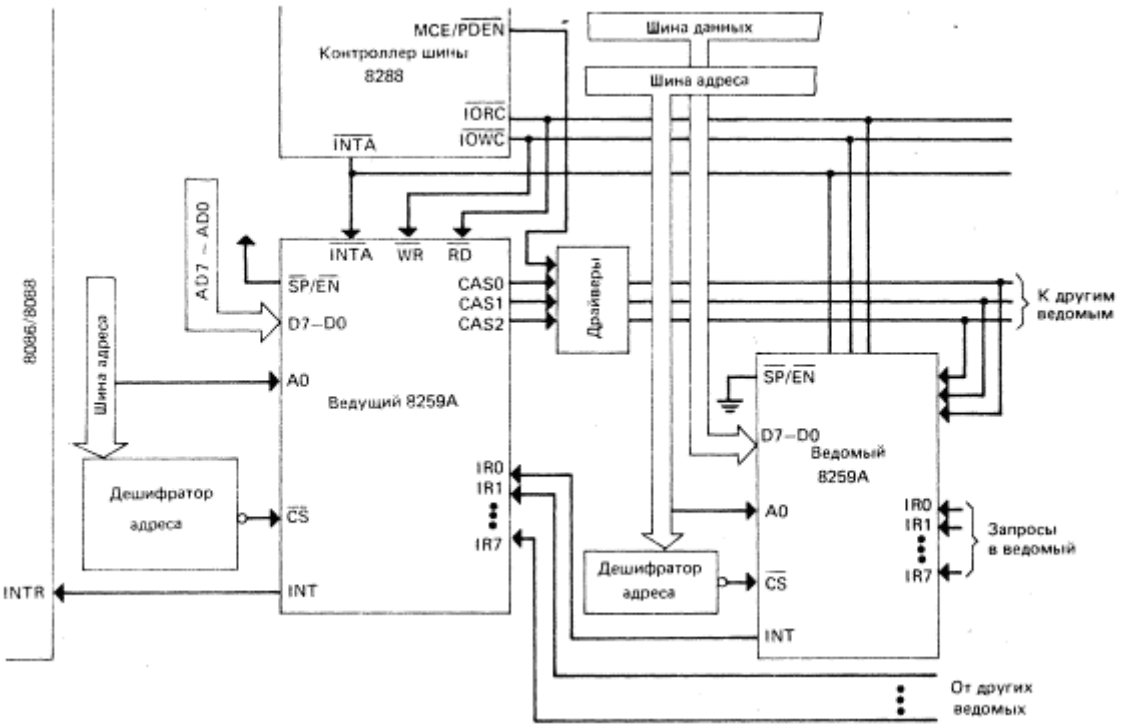


Рис.18 . Каскадирование контроллеров прерываний

Конечно, для блокирования некоторых запросов в ведущем и ведомых контроллерах можно применять маски.

Лекция № 12

5.3. Управление вводом – выводом. Интерфейсы ввода – вывода

При работе микропроцессорных систем возникает необходимость обмена информацией с различными устройствами ввода-вывода. Для обеспечения такого обмена данными требуются определённые средства – система команд, сигналов и соответствующие устройства сопряжения. Эти средства объединяются под наименованием *интерфейс ввода-вывода.* Интерфейсы ввода-вывода, как и интерфейсы памяти, связаны с логикой управления шиной. Интерфейс ввода-вывода должен выполнять следующие основные функции:

определять, когда обращение производится именно к нему, и в случае такого обращения понять, к каким регистрам происходит обращение; выяснить, требуется ли выполнение ввода или вывода. При выводе воспринять с шины выходные данные или управляющую информацию, а при вводе поместить на шину входные данные или информацию о состоянии.

Внешние устройства могут принимать и передавать информацию в параллельном или последовательном коде. Параллельные каналы обычно используются при расстоянии до 10–15 м. При больших расстояниях стоимость

многопроводного кабеля достаточна высока, повышается вероятность возникновения ошибки. Кроме того, некоторые внешние устройства не могут выдавать или получать информацию в параллельном коде (телефонная, телеграфная связь) и передают её по одному биту. При передаче данных в последовательном коде пропускная способность канала меньше, чем при передаче в параллельном. Интерфейс может иметь отдельные линии для передачи и приёма информации. Когда для двух направлений сигналов применяются различные линии, связь называется *дуплексной*. Такая система может передавать и принимать одновременно. В *полудуплексной* связи для ввода и вывода применяется одна и та же линия.

Последовательная связь настолько сложна, что были разработаны специальные микросхемы, выполняющие работу по формированию и синхронизации строк битов, составляющих последовательные данные. Такие микросхемы называют *универсальные синхронно-асинхронные программируемые приёмо-передатчики (УСАПП).*

Организацию обмена данными по каналу связи в последовательном коде можно осуществить с помощью УСАПП I8251 (КР 580ВВ51).

5.3.1. Последовательная связь. Универсальный синхронно-асинхронный программируемый последовательный приемопередатчик (УСАПП) КР580ВВ51 (I 8251)

Микросхема КР580ВВ51 предназначена для аппаратной реализации последовательного протокола обмена между микропроцессором, способным запрограммировать данную микросхему на требуемый режим работы, и каналами последовательной передачи дискретной информации бит за битом. Он преобразует параллельный формат данных, получаемый от МП, в последовательный поток символов со служебными битами и выдаёт этот поток в последовательный канал связи с различной скоростью. Микросхема также выполняет обратное преобразование: последовательный поток символов – в параллельное 8-разрядное слово. Передаваемая и принимаемая информация может контролироваться на чётность или нечётность.

Универсальность микросхемы состоит в следующем:

в изменении формата слов (от 5 до 8 разрядов);

в диапазоне скоростей от 0 до 19,2 килобод (т. е. 19200 бит/сек) при асинхронной и до 64 килобод (64000 бит/сек) при синхронной передаче;

в различных скоростях передачи, равных 1, 1/16, 1/64 от частоты тактирования;

в различных режимах работы, а их пять: асинхронная передача, асинхронный приём, синхронная передача, синхронный приём данных с внешней синхронизацией и синхронный приём данных с внутренней синхронизацией.

Микросхема переходит в один из пяти режимов работы после записи инструкции режима, синхросимвола (синхросимволов) – в синхронном режиме работы и инструкции команд.

Микросхема имеет 48 выводов, напряжение питания U = 5В, $P_{потр}$ = 400 МВт.

Контроллер КР580ВВ51 программируется на выполнение почти всех применяющихся протоколов последовательной передачи данных и работает в двух режимах: синхронном и асинхронном.

Асинхронный режим характеризуется одиночными посылками информации, инициализация которых определяется либо микропроцессором, либо внешним устройством. Формат данных при асинхронном режиме представлен на рис. 19.

В начале каждой посылки устанавливается отрицательный импульс «старт-бит», длительность которого равна биту данных. «Старт-бит» служит для ввода в синхронизацию передатчика/приёмника КР580ВВ51А. До начала передачи стартового символа линия должна находиться в состоянии «1», которое часто называют состоянием «маркера». Затем следуют 5-8 информационных битов, первым из которых является младший бит. В конце каждой посылки устанавливается положительный импульс «стоп-бит», длительность которого может равняться 1; 1,5 и 2 длительностям бита информации (устанавливается программно); «стоп-бит» служит для определения конца посылки.

Выход передатчика	Старт-биты	Биты данных	Бит четности	Стоп-биты

Рис. 19. Формат данных при асинхронном режиме

Синхронный режим характеризуется непрерывным потоком передаваемой/принимаемой информации. На рис. 20 приведён формат данных для синхронного режима. Для установления синхронизации между передатчиком/приёмником микросхемы КР580ВВ51А и приёмником / передатчиком внешнего устройства и выделения из последовательного потока символов полезной информации в поток информации вводятся кодирующие слова (синхросимволы). Информационная (5–8 бит) и временная длины синхросимволов и слова данных равны.

Если между словами данных имеются временные промежутки, то они заполняются синхросимволами. Синхросимволов может быть один или два (устанавливаются программно). Если запрограммирован контроль данных на чётность (нечётность), то после каждого слова данных вставляется бит контроля.

1-ый символ синхронизации	2-ой символ синхронизации	Байт данных	Бит контроля на четность(нечётн.)	2-ой байт данных

Рис. 20. Формат данных при синхронном режиме

На рис. 21 представлена укрупнённая структурная схема УСАПП, состоящая из двух каналов: приёмного и передающего, имеющих общую связь с МП по ШД и полностью раздельными каналами связи с приёмопередающими устройствами. Приведём краткое описание основных узлов микросхемы и соответствующих выводов. (Для облегчения восприятия излагаемого материала наименование выводов производится на русском языке, одновременно указывается принятая в справочной литературе аббревиатура на английском языке).

Буфер данных (Буф.дан.) представляет собой параллельный 8-разрядный двунаправленный регистр с трёхстабильными каскадами. Он служит для обмена данными и управляющими словами между МП и УСАПП. Принимаемый с ШД микропроцессора байт данных фиксируется в этом регистре буфера данных (Буф.дан.), откуда он через внутреннюю шину передаётся в буфер передатчика (Б.Пд). Основу передатчика составляет 13-разрядный сдвиговый регистр, хранящий очередное выходное слово. В этом буфере происходит преобразование параллельного кода в последовательный и добавляется служебная информация. Путём серии сдвигов содержимое этого регистра выдвигается в последовательной форме через буфер передатчика (Б.Пд.) на выход передатчика (Вых.пд. – TxD). При вводе данных в микропроцессор поступающие на вход приёмника (Вх.пр. – RxD) в последовательной форме данные фиксируются в регистре данных буфера приёмника (Б.пр.), где производится преобразование данных из последовательного кода в параллельный. Данные через буфер данных в параллельной форме выдаются на ШД МП. Служебная информация, предназначенная для программирования УСАПП, из буфера данных передаётся в устройство управления записью/чтением.

Устройство управления записью/чтением (УУзп/чт). принимает управляющие сигналы от МПС и вырабатывает внутренние сигналы управления. В нем есть два регистра: регистр команд, определяющий тип выполняемой операции, и регистр режима, определяющий режим работы: синхронный или асинхронный. Под типом выполняемой операции понимается следующее: прием данных, выдача данных, чтение регистра слова состояния; выдача команд или высокоимпедансное состояние.

Рассмотрим выводы, связанные с устройством управления записью/чтением (УУзп/чт).

Основными управляющими сигналами являются: чтение (Чт. – RD), запись (Зп. – WR), управление/данные (У/Д – CO/D), выбор кристалла (ВК – CS). Сигналы ЧТ и ЗП подаются на соответствующий сигнал ЧтВУ и ЗпВУ МПС и показывают направление передачи информации по ШД. Сигнал У/Д – указывает тип вводимой по ШД информации, т.е. данные или служебные слова. Обычно этот вывод подключается к младшему разряду ША А0, «0» – соответствует передаче данных, «1» – запись управляющих слов или чтение слова состояния ВВ51. ВК – выбор ВВ51, подается к одному из разрядов ША прямо или через дешифратор.

Возможные варианты сочетания управляющих сигналов и направления передачи информации в системе приведены в табл.18.

Таблица 18

Входные сигналы				Направление передачи	Вид информ.
У/Д(A_0)	Чт	Зп	ВК		
0	0	1	0	ВВ51→ШД	Данные
0	1	0	0	ШД→ВВ51	Данные
1	0	1	0	ВВ51→ШД	Слово состояния
1	1	0	0	ШД→ВВ51	Управляющие слова

Кроме рассмотренных, к устройству управления записью/чтением подводится вывод синхронизации **(СИ – С),** на который подаются синхроимпульсы. При работе в синхронном режиме их частота должна быть min в 30 раз выше частоты приема или передачи; при работе в асинхронном режиме – min в 5 раз.

Сброс (Сбр. – С) **–** установка микросхемы в исходное состояние, сигнал подаётся от генератора ГФ– 84.

При программировании в ВВ51 заносится инструкция режима и инструкция команды, координирующая работу ВВ51. Кроме того, для синхронной связи необходимы регистры для хранения символов синхронизации, которые тоже заносятся программно. В системах передачи данных часто необходимо контролировать то состояние микросхемы, которое устанавливается в процессе работы, сбоев, ошибок или других ситуаций. Микросхема содержит регистр состояний, позволяющий программисту читать её состояние в любой момент времени в процессе выполнения операции. Содержимое регистра состояния не изменяется во время чтения слова состояния. Формат регистра слова состояния приведён в табл. 19 и будет рассмотрен далее.

Таким образом, процессор может обращаться к семи регистрам микросхемы: регистру входных данных, регистру выходных данных, регистру режима, регистру команд, регистру слова состояния, двум регистрам синхросимволов).

Как же производится адресация, если для МП контроллер ВВ51 ассоциируется с адресами всего 2 портов (на вход У/D подаётся 0 или 1 с адресной линии A_0). Дело в том, что выбор регистров режима, управления или символов синхронизации зависит от последовательности обращения при программировании микросхемы. Последовательность программирования приведена на рис.22.

Рассмотрим следующие блоки структурной схемы.

Устройство управления передатчиком УУПд вырабатывает внутренние и внешние управляющие сигналы для передачи. Для синхронизации передачи данных по выходу «Вых.пд» служит вход Си.Пд (ТхС)(синхроимпульсы передатчика), на который подаются синхросигналы от приёмника данных.

При синхронном режиме данные поступают на выход передатчика при каждом срезе сигнала на входе Си. Пд., т. е. частота выдачи информации на выходе передатчика равна частоте синхросигнала на входе Си.Пд. Для асинхронного режима частота выдачи данных может программно (как определено в инструкции режима) устанавливаться равной частоте синхросигнала или же в 16 или 64 раза меньше её. Сигналы тактирования поступают с приемника данных от специального генератора скорости в бодах или через делитель от тактовых импульсов общего генератора, хотя синхронности с сигналом CLK не требуется. Например, скорость передачи 100 бод, тогда при кратности 1 – частота тактирования должна быть 100 Гц, при кратности 1/16 – частота тактирования равна 1,6кГц; при кратности 1/64 – частота тактирования 6,4 кГц. Максимальная частота на этом входе может быть 615 кГц.

Устройство управления передатчиком имеет ещё два выхода: готовность передатчика Г.Пд (Tx.Rd) и конец передачи К.Пд (Tx.End).

Выход Г.Пд используется для информирования МП о готовности контроллера КР580ВВ51 принять новые данные или команды управления. При работе по запросу сигнал с этого выхода используется как сигнал запроса прерывания МП, после чего МП переходит на подпрограмму для передачи данных. При работе по опросу (считываемому типу прерываний) МП определяет состояние сигнала на этом выходе с помощью чтения слова состояния из регистра слова состояния. Микропроцессор программно определяет состояние разряда D0 (Г.Пд). В этот разряд запишется «1», как только в буфере передатчика (Б.Пд) не будет полезной информации, причём в разряде D0 управляющего слова (разрешение передачи) при программировании записана «1», а на входе готовности приёмника терминала (Г.Пр.Т) – «0» (об этом блоке будем говорить далее). После определения «1» в разряде D0 слова состояния МП выполнит передачу данных. Как только в контроллер запишется новое данное из МП, на выходе Г.Пд. установится «0», (соответственно, разряд D0 слова состояния тоже установится в «0»).

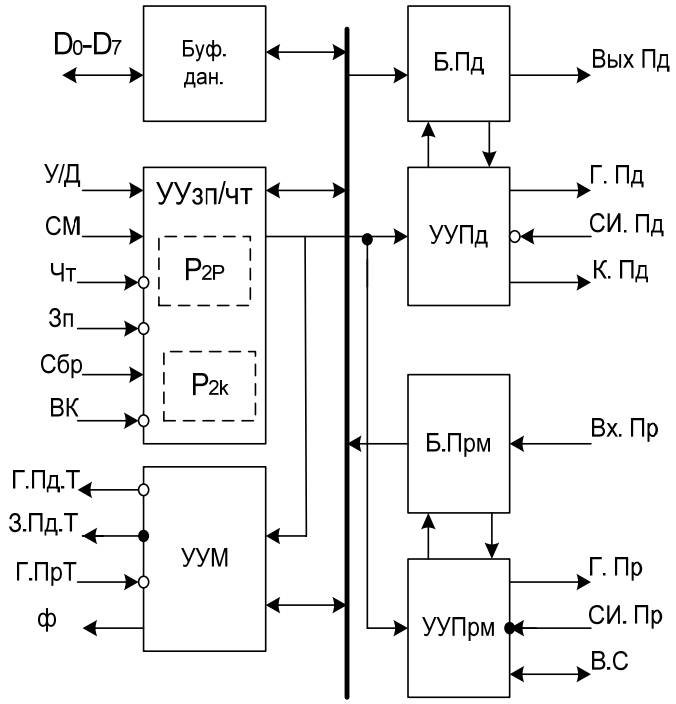


Рис. 21. Структурная схема КР580ВВ51А

Выход конец передачи КПд (Tx.End) установится в «1», как только преобразователь параллельного кода в последовательный в канале передатчика не будет содержать данных. Сброс сигнала на этом выходе происходит в момент записи новых данных в микросхему.

Асинхронная передача. Как уже говорилось, в асинхронном режиме работы формат данных (рис. 19) включает нулевой старт-бит, биты данных, контрольный бит и стоп-биты. Число битов данных и стоп-битов, а также наличие или отсутствие бита контроля задаются инструкцией режима. Более подробно инструкция режима рассматривается ниже в разделе «Программирование микросхемы». Если микросхема не содержит информацию для передачи, то на выходе Вых.Пд (TxD) устанавливается напряжение высокого уровня. Если в инструкции команды запрограммирована «пауза», на выходе устанавливается «0».

Синхронная передача. После записи в микросхему инструкции режима, синхросимволов, инструкции команды и данных передатчик не начнёт передачу до тех пор, пока на входе готовности приёмника терминала Г.Пр.Т. (CTS) не установится «0». Если на этом входе установился «0» и в разряд D0 инструкции команды записана «1», то передатчик начинает трансляцию по выходу передатчика Вых.Пд. (Tx.D) со скоростью синхроимпульсов, поступающих на вход синхронизации передатчика Си.Пд. (Tx.C).

Каждый раз после сигнала сброса (RS) программируются инструкция режима, синхросимвол (синхросимволы) и инструкция команды. Для начала передачи информации с выхода передатчика в него необходимо записать любые данные, которые будут потеряны, так как в это время приёмник внешне-

го устройства будет работать в режиме поиска синхросимволов. Может получиться, что МП не запишет очередную информацию в микросхему ВВ51А до того, как она передаст предыдущую информацию. В этом случае для предотвращения потери синхронизации между УСАПП и внешним устройством в поток данных автоматически вставляются синхросимволы. При этом на вывод «Конец передачи» К.Пд. (Тх.End) подаётся напряжение высокого уровня, показывающее, что микросхема не имеет информации для передачи и синхросимволы посланы внешнему устройству. Когда МП начинает записывать информацию в ВВ51А, на выходе К.Пд. устанавливается ноль.

Буфер приёмника (Б.Пр.) получает последовательность символов по выводу Вх.Пр.(RxD) – вход приёмника. В этом блоке данные выделяются и преобразуются в параллельный код.

В *асинхронном режиме* перед началом приёма данных на выводе Вх.Пр. должен быть выставлен уровень «1». В соответствии с форматом данных при работе в асинхронном режиме, нулевой уровень воспринимается схемой как стартовый разряд, сообщаемый перед каждым данным. Определив стартовый разряд и получив данные, разряд чётности и стоп биты, приёмник преобразует их в параллельный код, после чего устанавливает сигнал «1» на выходе Г.Пр.–RxRDY (готовность приёмника). Сигнал на нём указывает о готовности данных приёмника ко вводу. Сигнал при работе по запросу может быть использован как сигнал запроса прерывания. Можно использовать его как сигнал запроса захвата в каналах ПДП. При работе по считываемому типу прерываний состояние сигнала на этом выходе определяется с помощью чтения слова состояния (табл. 19) и определения содержания его разряда D1 (ГПр).

Ошибки работы адаптера устанавливают соответствующие биты в слове состояния (четности, формата или переполнения). Бит переполнения устанавливается, если к моменту окончания приёма очередного данного МП не считал ранее записанные данные из регистра принимаемых данных. В этом случае ранее записанная информация теряется.

При *синхронном приёме* данных по входу Вх.Пр **с**уществует два вида режима работы, отличающиеся методом синхронизации приёма данных: режим внутренней синхронизации и режим внешней синхронизации. Формат данных при синхронном режиме уже рассматривался выше (рис. 20).

При *синхронном приёме с внутренней синхронизацией* работа микросхемы начинается с поиска синхросимолов. Информация принимается по входу Вх.Пр.(Rx.D) в буфер приёмника и непрерывно сравнивается с содержимым регистра первого синхросимвола. Если содержимое двух регистров не одинаково, то регистр прёмника принимает следующий бит информации и сравнение повторяется. Когда содержимое сравниваемых регистров становится одинаковым, микросхема заканчивает поиск и переходит в режим синхронизации. При этом, если не запрограммирован контроль на чётность (нечётность), на выводе В.С – SYNDET/BD (вид синхронизации), работающем как выход, после приёма последнего бита синхросимвола устанавливается

уровень «1», сигнализируя внешнему устройству о том, что произошёл захват синхронизации.

Если УСАПП запрограммирован на работу с двумя синхросимволами или с контролем на чётность (нечётность), то описанная ситуация произойдёт во время последнего бита второго синхросимвола или бита контроля соответственно

Сигнал «0» на выводе В.С. устанавливается автоматически после каждого чтения слова состояния. Состояние сигнала на выводе В.С. отображается в этом слове.

В режиме *синхронного приёма с внешней синхронизацией* вывод В.С. используется как входной для подачи напряжения синхронизации, которое разрешает приём информации по входу Вх.Пр. со скоростью синхросимволов, поступающих на вход синхронизации приёмника СИ.Пр (Rx.C)**.** Длительность входных сигналов, поступающих на вход «вид синхронизации (В.С.)» должна быть больше или равна периоду частоты синхронизации сигналов, поступающих на вход синхронизации приёмника Си.Пр. Этот же контакт (В.С.), но только в качестве выхода применяется и при асинхронной работе. Сигнал на нём называется сигналом обнаружения разрыва, и он высоким уровнем отмечает прием символа, состоящего из одних нулей.

Рассмотрим последний блок структурной схемы: устройство управления модемом (УУМ). Этот блок осуществляет обмен управляющими сигналами со связными модемами различных типов, которые включаются между ВВ51А и линией связи с удаленными ВУ, с целью проверки состояния внешних устройств. Для каждого канала (приёмного и передающего) в устройстве управления модемом имеется по одному одноразрядному входу и выходу.

Запрос передатчика терминала (З.Пд.Т. – DTR) – выход сигнала запроса передатчика терминала. Это сигнал общего назначения. В частности, сигнал запрашивает внешнее устройство о его готовности передавать данные через вход приёмника Вх.Пр. микросхемы ВВ51А к МП. Выход сигнала запроса управляется программно установкой «0».

Готовность передатчика терминала (Г.Пд.Т. – DSR) – входной сигнал общего назначения. В частности, по этому входу внешнее устройство сообщает о готовности передатчика передать данные. Состояние входа фиксируется в слове состояния (D7) и может быть проанализировано.

Запрос приёмника терминала З.Пр.Т.(RTS) – выходной сигнал общего назначения, в частности, используется как сигнал запроса приемника терминала о его готовности принять через ВВ51А данные. Нуль на выходе устанавливается программно.

Готовность приёмника терминала Г.Пр.Т.(CTS) – входной сигнал общего назначения, в частности, вход может быть использован для информирования микросхемы о готовности приемника модема принять данные в ответ на сигнал Г.Пр.Т. Нулевой сигнал на этом входе при наличии «1» в разряде D0 управляющего слова, записанного при начальной установке схемы, разрешает передачу данных на выходе передатчика Вых.Пд.

Часто возникает необходимость контролировать состояние схемы, устанавливаемое в процессе обмена данными. Для этой цели используется регистр состояния, в разрядах которого отображаются сбои, ошибки и наличие сигналов на управляющих выводах микросхемы. Микропроцессор может считывать содержание регистра состояния в любой момент времени в процессе обмена данными. Как уже говорилось выше, с помощью программной проверки разрядов D0, D1 организуется работа по опросу, т. е. без использования сигнала запроса прерывания МП. Формат регистра состояния приведён в табл. 19. Наличие ошибок при обмене информации по каналам связи фиксируется в разрядах ошибок регистра состояния, но не воздействует на режим работы схемы. Разряды ошибок сбрасываются при записи инструкции команды.

Программирование микросхемы КР580ВВ51А

После начальной установки (внешней или внутренней) в схему должны быть записаны команды начальной установки из микропроцессора. Различаются управляющие слова двух видов: инструкция режима и команды. Инструкция режима задаёт синхронный

Таблица 19

Разряд	Обозначение	Пояснение
D0	ГПд	В разряд записывается «1», если буфер данных передатчика не содержит данных
D1	ГПр	При «1» приёмник готов выдать данные МП
D2	КПд	При «1» передатчик закончил преобразование данных
D3	ОЧ	В разряд ошибки чётности записывается «1», если обнаруживается ошибка чётности. Разряд сбрасывается в «0» записью «1» в разряд D4 инструкции команды.
D4	ОП	В разряд ошибки переполнения записывается «1», если МП не считывает данные из приёмника к моменту записи новой посылки. Разряд сбрасывается в «0» записью «1» в разряд D4 инструкции команды
D5	ОФ	В разряд ошибки формата записывается «1» (только при асинхронном режиме), если в конце данного не обнаруживается в стоп-бите «1». Разряд сбрасывается в «0» записью «1» в разряде D4 инструкции команды
D6	Вид синхр	Состояние такое же, как состояние сигнала на выходе Вид СИНХР
D7	ГПдТ	Состояние разряда такое же, как состояние сигнала на входе ГПдТ.

или асинхронный режим работы, формат данных, скорость приёма или передачи, необходимость контроля. При синхронном режиме приёма данных указывается также тип синхронизации (внутренний или внешний). Инструкция заносится сразу после установки УСАПП в исходное состояние программно

или по сигналу сброса и заменяется лишь при смене режима. Команда осуществляет управление установленным режимом обмена и может многократно задаваться в процессе обмена, управляя различными его этапами.

При асинхронном режиме команда загружается сразу же после инструкции режима, а при синхронном обмене перед ней располагаются один или два синхросимвола. Ограничения на последовательность загрузки управляющих слов связаны с внутренней адресацией УСАПП. Порядок записи инструкции режима, синхросимволов, инструкции команды и алгоритм работы приведены на рис. 22.

Формат инструкции режима при асинхронном и синхронном режимах работы схемы приведён на рис. 5.

Инструкция команды должна записываться в схему всегда после инструкции режима и синхросимволов (в синхронном режиме). Она может записываться в любое время в процессе обмена информацией по каналам связи. Формат инструкции команды приведён в таблице 20.

Начальная установка схемы по сигналу СБРОС (внешняя) или с помощью «1» в разряде D6 инструкции команды (внутренняя) возвращает схему в исходное состояние и приводит к необходимости записи новой инструкции режима.

Рассмотрим фрагмент программы, который инициализирует регистр режима и содержит приказ разрешения работы передатчика и запуска асинхронной передачи со следующим форматом: 7 бит символа, бит чётного паритета и 2 стоповых бита. Предположим, что регистры режима и команд имеют адрес 71H, а частоты синхронизации в 16 раз больше соответствующих скоростей в бодах.

```
MOV  AL, 11111010B;  инструкция режима
OUT  71H, AL
MOV  AL, 00110011B;  инструкция команды
OUT  71H, AL
```

Следующий пример показывает программирование микросхемы в синхронный режим с внутренней синхронизацией и поиском одного символа синхронизации. Запрограммированы 7 информационных бит и бит чётного паритета. Адреса сохраним прежние.

```
MOV  AL, 00111000B  ; инструкция режима
OUT  71H, AL
MOV AL, 16H
OUT  71H, AL
MOV  AL, 10010100B  ; инструкция команды
OUT  71H, AL
```

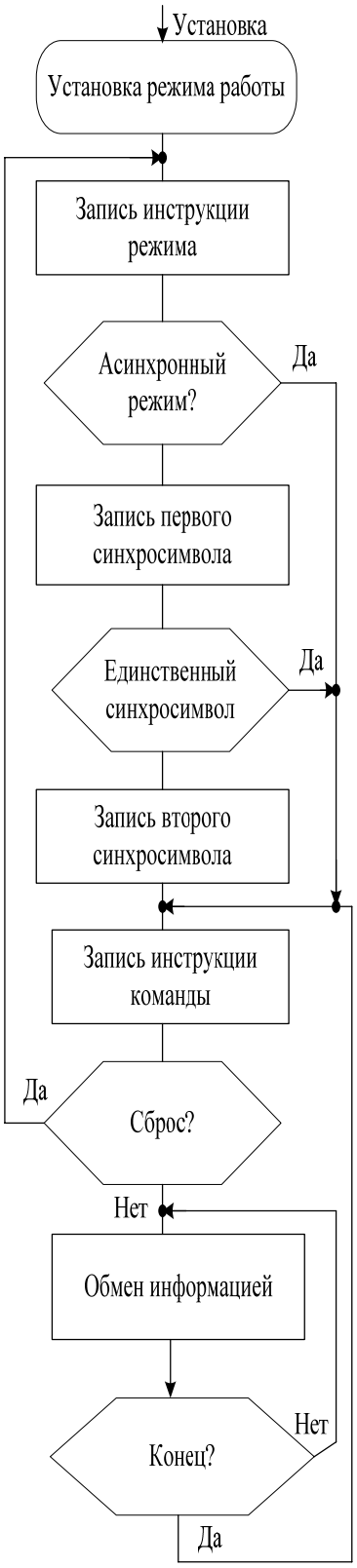


Рис. 22. Последовательность программирования и алгоритм работы КР580ВВ51А

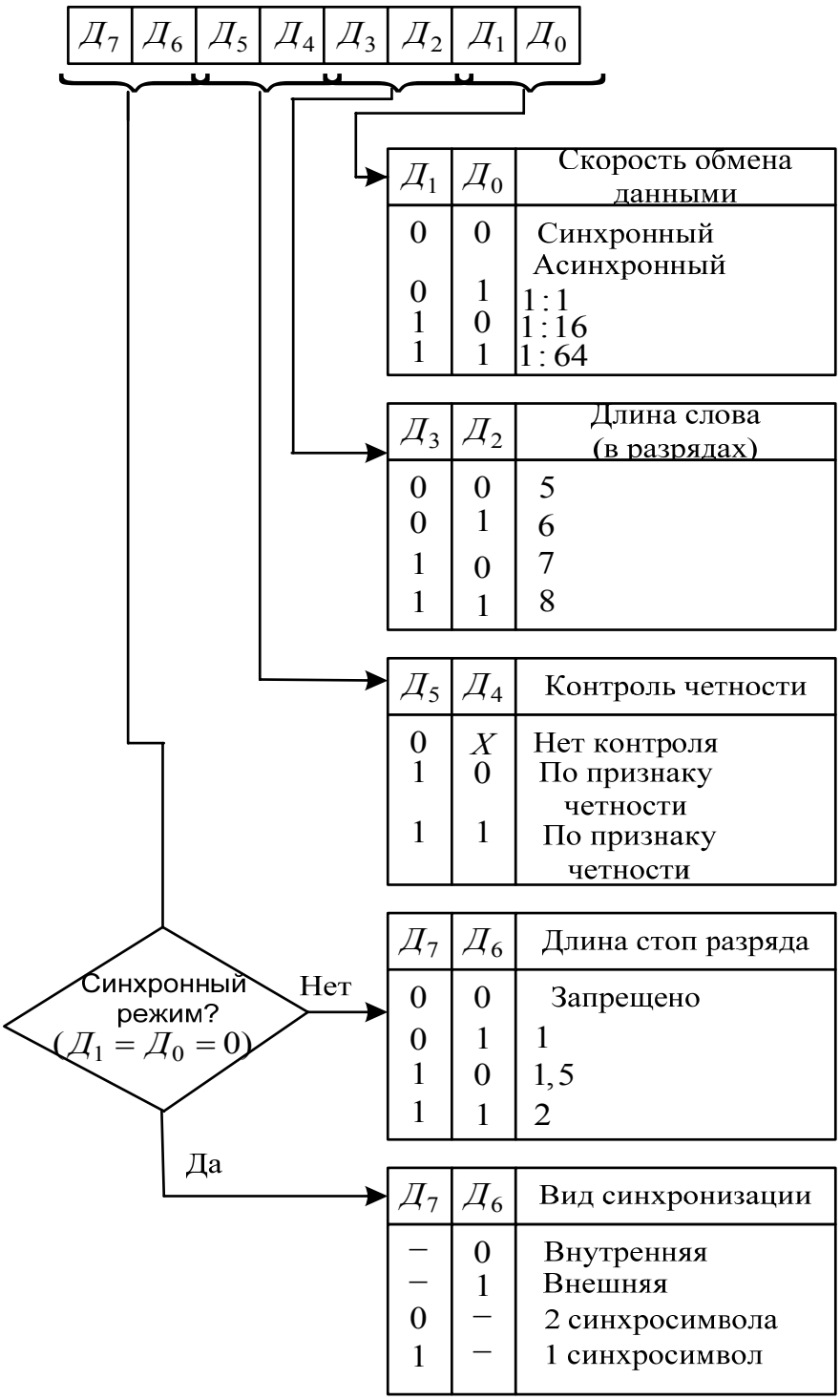


Рис. 23. Формат инструкции режима

Таблица 20

Разряд	Обозначение	Содержание	Пояснение
$Д_о$	Разрешение передачи информацией	0	Передача информации невозможна
		1	Передача информации возможна
$Д_1$	Запрос о готовности передатчика терминала передать данные	0	–
		1	Записывает «0» на выходе ЗПдТ
$Д_2$	Разрешение приема информации (Р.Прм)	0	Приём информации невозможен
		1	Приём информации возможен
$Д_3$	Отключение канала передатчика	0	Нормальная работа канала
		1	Уровень «0» на выходе канала
$Д_4$	Начальная установка разрядов ошибки	0	–
		1	Установка разрядов ошибки регистра состояния в исходное состояние
$Д_5$	Запрос о готовности приемника терминала принять данные	0	–
		1	Записывает «0» на выходе ЗПрТ
$Д_6$	Программный сброс схемы в исходное состояние	0	–
		1	Возвращает схему к установке режима работы
$Д_7$	Установка режима поиска синхроимпульсов	0	–
		1	Устанавливает режим поиска синхроимпульсов

Рассмотрим ещё один пример, где с помощью словосостояния (программный ввод, т.е. работа микропроцессорной системы « по опросу») производится ввод 80 символов с внешнего устройства через последовательный интерфейс КР580ВВ51А. (Будем считать, что микросхема была ранее уже запрограммирована). Примем адрес буферного регистра данных равным 0070Н. Введённые символы разместить в ячейках памяти, начинающихся с BUF. После считывания слова состояния производится проверка бита D1 ГПр. (RxRDY) до тех пор, пока он не установится символом, принятым в буферный регистр входных данных. Затем символ пересылается в ячейки памяти, начинающейся с адреса BUF.

```
        MOV   AL, 00010110B ; инструкция команды, разрешающая
        OUT   71H, AL       ; приём
        MOV   SI, 0         ; инициализация индексного регистра
        MOV   CX,80         ; счётчик
MET:    IN    AL, 71H       ; прочитать слово состояния
        TEST  AL, 02H
        JZ    MET
```

```
IN      AL, 70H         ; ввести символ
MOV     BUF[SI], AL     ; и отправить его в память
INC     SI
 LOOP   MET
  …
```

При необходимости можно произвести контроль битов ошибок. Для этого слово состояния вводится снова и проверяются соответствующие биты. Если текущий символ появился в заполненный буфер данных контроллера или во время передачи возникли ошибки паритета или кадра, ввод прекращается и вызывается процедура обработки ошибки.

Подключение микросхемы

На рис. 24 показан возможный вариант подключения микросхемы в микропроцессорной системе.

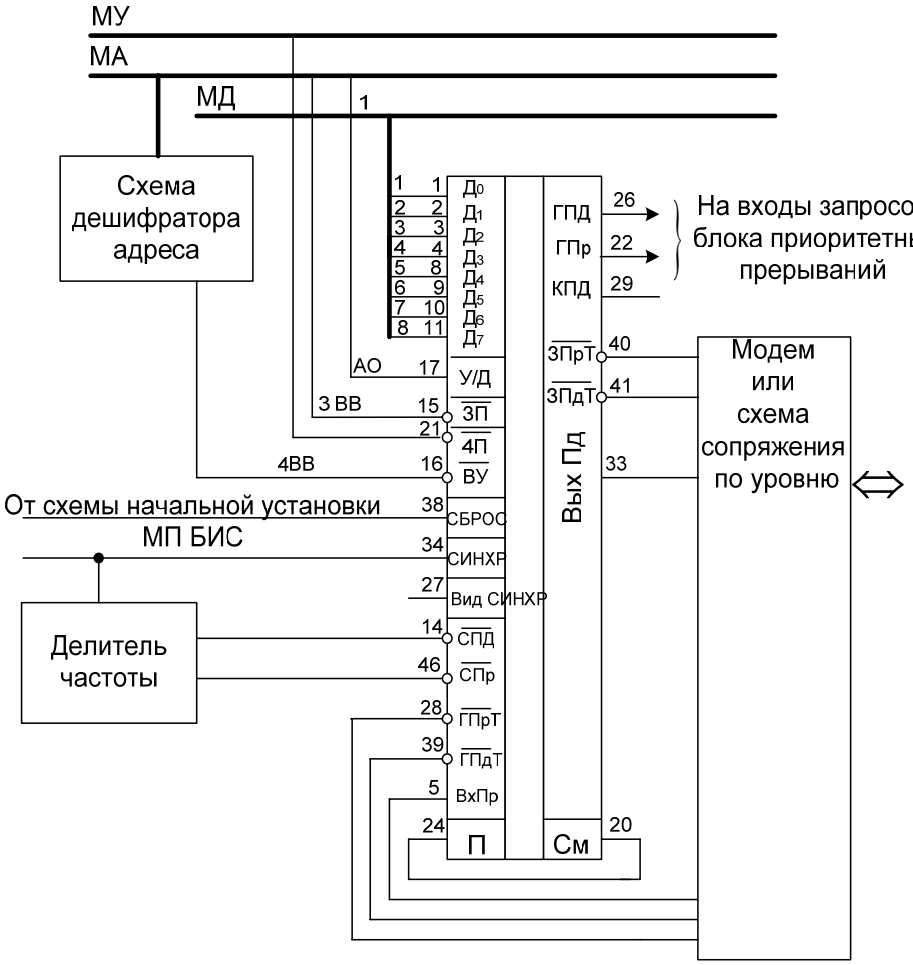


Рис. 24. Схема подключения КР580ВВ51А в микропроцессорную систему

Делитель частоты обеспечивает деление тактовых импульсов до частоты, необходимой для работы передатчика или приёмника. Выводы ГПд и ГПр подключаются ко входам запросов прерывания контроллера прерыва-

ний, если предполагается организация работы «по запросу». Выход передатчика, вход приёмника, сигналы запросов приёмника и передатчика терминала подключаются либо к модему, либо к схеме сопряжения по уровню, позволяющей осуществлять преобразование ТТЛ-уровней входных и выходных сигналов в уровни, необходимые для работы с ВУ (дисплеем и т. д.).

Каждый компъютер обычно оборудован, по крайней мере, двумя последовательными портами, которые чаще всего используются для подключения мыши и модема или для соединения компъютеров между собой.

Лекция № 13

5.3.2. Параллельная связь. Программируемый параллельный интерфейс 8255 (КР580ВВ55)

Параллельная связь осуществляется одновременной передачей нескольких бит по отдельным линиям. Её преимущество по сравнению с последовательной связью заключается в том, что для линий с заданной максимальной двоичной скоростью обеспечивается более высокая скорость передачи информации. Недостатком же является, конечно, стоимость дополнительных линий, поэтому параллельная связь применятся на большие расстояния, если только требуется высокая скорость передачи.

Программируемым устройством ввода-вывода параллельной информации различного формата является микросхема **КР580ВВ55**. Микросхема представляет собой программируемое устройство, используемое для ввода – вывода. БИС программируемого параллельного интерфейса (ППИ) может использоваться для сопряжения микропроцессора со стандартным периферийным оборудованием (дисплеем, телетайпом и т. д.). В микросхеме программируется:

направление обмена информацией (ввод или вывод);

режим обмена информацией;

разрешение обмена с прерыванием программы.

Структурная схема ППИ приведена на рис. 25.

В схему входят: буфер данных, блок управления с регистром управляющего слова, три канала (А, В и С). Регистры канала А и В – восьмиразрядные, регистр канала С может быть подразделён на два четырёхразрядных регистра ввода-вывода данных, к которым осуществляется доступ как к отдельным независимым регистрам. Имеется возможность подразделения трёх каналов на две группы. В этих группах при определённых режимах каналы А и В используются для обмена данными, а отдельные линии канала С – для управляющих сигналов.

Двунаправленный тристабильный буфер данных предназначен для подключения внутренней шины данных микросхемы к системной ШД. Приём в буфер информации или выдача происходит при выполнении МП команд

ввода или вывода, через него же производится программирование и считывание слова состояния контроллера.

Блок управления чтением / записью обеспечивает сопряжение внутренних командных сигналов с системными управляющими сигналами.

Для связи с периферийными устройствами используются 24 линии ввода/вывода, сгруппированные в три 8-разрядных канала A, B и C, направление передачи информации и режимы работы которых определяются программным способом.

Микросхема может функционировать в трёх основных режимах: режим 0, режим 1 и режим 2.

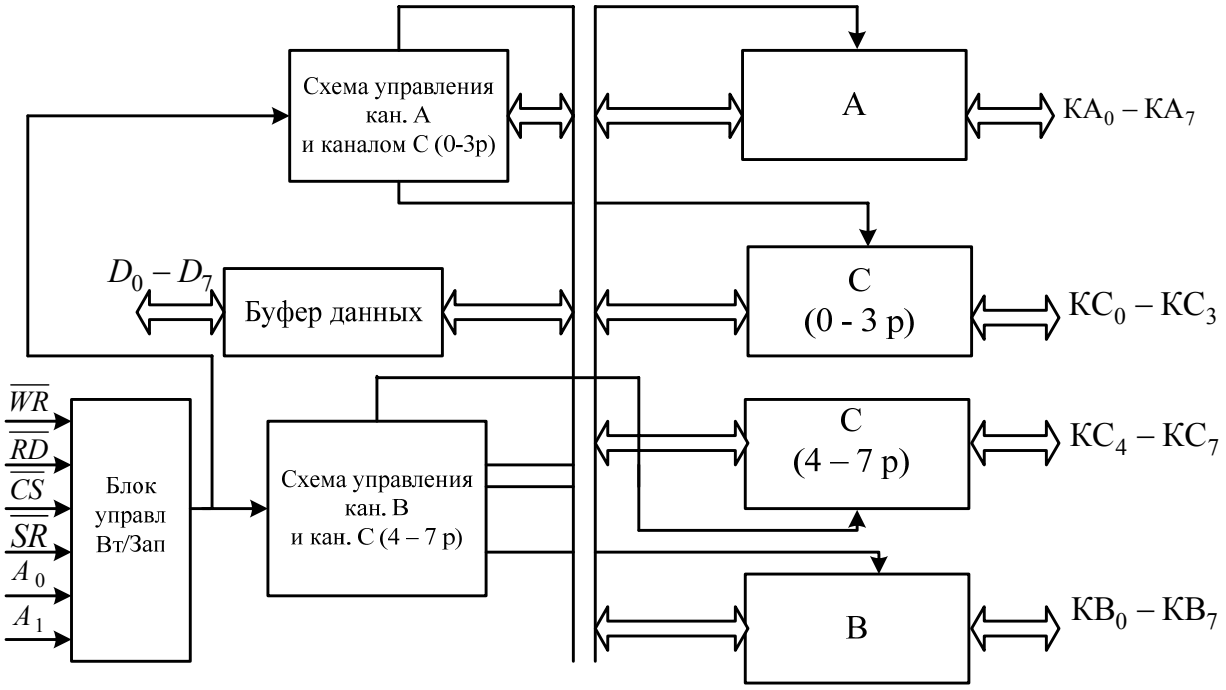


Рис. 25. Структурная схема КР580ВВ55

Режим 0 – это простой ввод-вывод информации. В этом режиме обеспечивается возможность синхронной программно управляемой передачи данных через все три независимых 8-разрядных канала A, B и C, причём канал C может быть разбит на два 4-разрядных канала.

Режим 1 – стробируемый ввод-вывод информации. В этом режиме обеспечивается возможность ввода или вывода информации в или из периферийного устройства через два независимых 8-разрядных канала A и B по сигналам квитирования. При этом по три линии канала C для каждого из каналов A и B используются в качестве сигналов управления обменом.

Режим 2 – двунаправленный обмен информацией по стробу. В режиме 2 обеспечивается возможность обмена информацией с периферийными устройствами только через двунаправленный 8-разрядный канал A по сигналам квитирования. Для передачи и приёма управляющих сигналов используются 5 линий канала C. Выбор соответствующего канала и направление передачи информации через канал определяются сигналами A0, A1 (которые

обычно соединяются с младшими разрядами шины адреса системы) и сигналами RD, WR, CS в соответствии с табл. 20.

Итак, канал А может работать в любом из трёх режимов работы на ввод и на вывод, канал В – в режимах 0 и 1 на ввод и вывод, канал С используется для передачи данных только в режиме 0, а в 1 и 2 режимах он передаёт управляющие сигналы.

Канал А имеет 2 регистра, один из них используется для приёма данных с ШД микропроцессора и выдачи их во внешнее устройство, другой – для приёма данных из внешнего устройства и выдачи их на ШД микропроцессора. В каналах В, С1 и С2 имеется по одному регистру, который обеспечивает передачу данных в нужном направлении за счёт входных и выходных формирователей.

Регистр порта С в режимах 1 и 2 выполняет функции регистра состояния.

Таблица 20

Сигнал на входах					Вид информации	Направление передачи
A1	A0	$\overline{WR}$	$\overline{RD}$	$\overline{CS}$		
0	0	1	0	0	Данные	К А → ШД
0	1	1	0	0	Данные	К В → ШД
1	0	1	0	0	Данные	К С → ШД
0	0	0	1	0	Данные	ШД → К А
0	1	0	1	0	Данные	ШД → К В
1	0	0	1	0	Данные	ШД → К С
1	1	0	1	0	Управляющие слова	ШД → Рег. слов управления
x	x	x	x	1		Микросхема не выбрана

Режим работы каждого из каналов определяется при программировании после записи управляющего слова (рассматривается ниже) в регистр управляющего слова контроллера.

В табл. 21 приведено назначение выводов ППИ.

Таблица 21

Вывод	Обозначение	Тип вывода	Функциональное назначение
1–4 37–40	КА3–КА0 КА7–КА4	Входы/выходы	Информационный канал А
18-25	КВ0–КВ7	Входы/выходы	Информационный канал В
10–17	КС7–КС4 КС0–КС3	Входы/выходы	Информационный канал С
8,9	A0, A1	Вход	Входы для адресации внутренних регистров ППИ
5	RD	Вход	Чтение информации; низкий уровень сигнала разрешает считывание информации из регистра, адресуемо-

			го по входам A0, A1 на ШД МП
36	WR	Вход	Запись информации; низкий уровень сигнала разрешает запись информации с ШД МП в регистр ППИ, адресуемый по входам A0, A1
6	CS	Вход	Выбор микросхемы
35	SR	Вход	Сброс. По этому сигналу все каналы настраиваются на режим 0 для ввода информации. Все шины портов переходят в высокоомное состояние
7	GND	–	Общий
26	Ucc	–	Напряжение питания

Программирование ППИ производится с помощью двух управляющих слов, их форматы приведены на рис. 26. и 27.

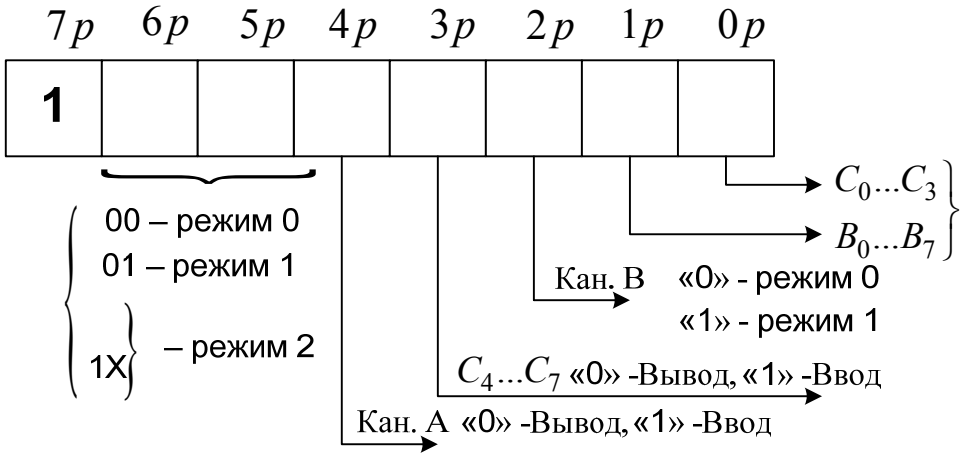


Рис. 26. Первый формат управляющего слова

Первый формат управляющего слова определяет режимы, каналы и направление передачи. Когда ППИ принимает байт, направляемый в его регистр управления, он анализирует бит 7 данных. Если этот бит содержит 1, первое управляющее слово передаётся в регистр управления; если же бит = 0, второе управляющее слово считается командой установки/сброса и применяются для установки или сброса определяемого командой бита порта С. Биты 3-1 дают номер изменяемого бита, а бит 0 показывает сброс или установку. Остальные биты не используются.

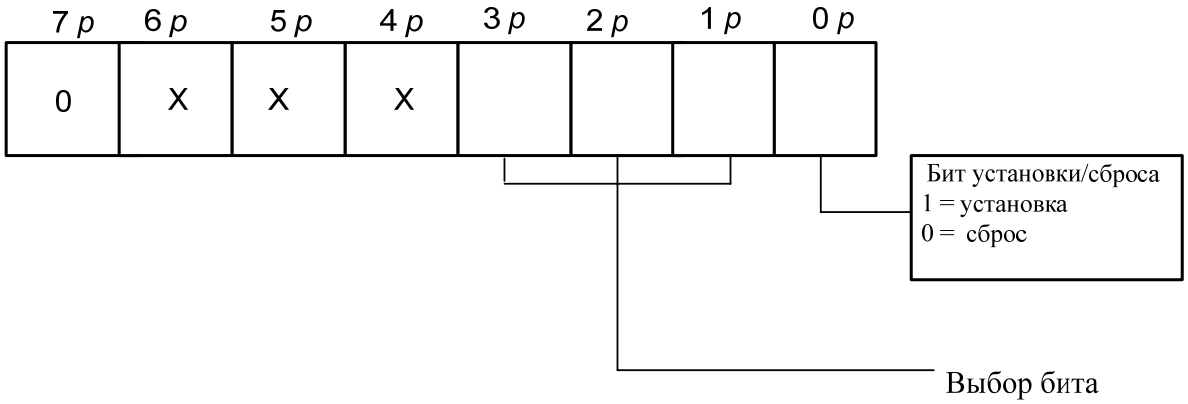


Рис. 27. Второй формат управляющего слова

Рассмотрим работу микросхемы в различных режимах более подробно.

Режим 0. В этом режиме вывод информации осуществляется по команде OUT микропроцессора с фиксацией выводимых данных в регистрах каналов, а ввод – по команде IN без запоминания информации. Дополнительных сигналов управления обменом информации с периферийных устройств не требуется.

Режим 1 обеспечивает стробируемый однонаправленный обмен информацией с внешним устройством. Передача данных производится по каналам А и В, а линии канала С управляют передачей. Работу канала в режиме 1 сопровождают три управляющих сигнала. Если один из каналов запрограммировать на режим 1, то остальные 13 линий можно использовать в режиме 0. Если оба канала запрограммированы на режим 1, то оставшиеся две интерфейсные линии канала С могут быть настроены на ввод или вывод.

Временные диаграммы для ввода информации в режиме 1 представлены на рис. 28.

В режиме 1 для ввода информации используются следующие управляющие сигналы: (см. рис. 28, 29) строб приёма STB – (разр. С3 для канала А и С2 для канала В) – входной сигнал, формируемый внешним устройством; указывает на готовность ВУ к вводу информации; подтверждение приёма IBF (С5 – для канала А, С1 – для канала В) – выходной сигнал ППИ, сообщающий ВУ об окончании приёма данных в канал; формируется по спаду STB; запрос прерывания INTR (С3 для канала А и С0 для канала В) – выходной сигнал контроллера, информирующий МП о завершении приёма информации каналом.

Высокий уровень сигнала запроса прерывания устанавливается при STB = 1 и IBF = 1, он переводит МП на соответствующую подпрограмму, обеспечивающую считывание информации из ППИ. По завершении считывания спадом сигнала RD сигнал запроса прерывания сбрасывается. Разрешения на выдачу сигналов запроса прерывания INTR выдают внутренние триггеры разрешения прерывания (на рис. 29 – Тг.пр.1 и Тг.пр.2). Состояния триггеров устанавливаются лишь программно с помощью второго управ-

ляющего слова (линия С4 для канала А и линия С2 – для канала В). Наличие таких внутренних триггеров позволяет при необходимости на время запрещать «отвлекать» процессор. Аналогично производится работа при выводе информации через программируемый параллельный интерфейс.

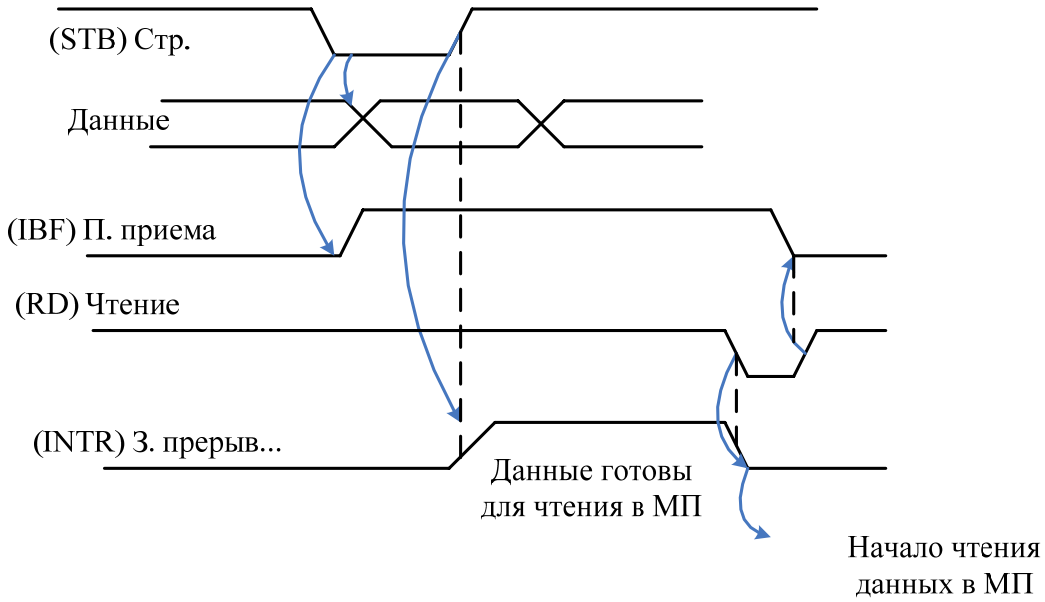


Рис.28. Временные диаграммы работы К580ВВ55 в режиме 1 (ввод)

Режим 2 обеспечивает двунаправленную передачу информации по каналу А к внешнему устройству и обратно без программирования направления передачи. Процесс обмена сопровождают пять управляющих сигналов, подаваемых по линиям С7 – С3. Управляющие сигналы являются комбинацией сигналов, необходимых при работе канала в режиме 1. Распределение сигналов по интерфейсным линиям приведено на рис.30. Функции управляющих сигналов аналогичны рассмотренным выше сигналам для режима 1. Управление установкой внутреннего сигнала INTR для операции ввода осуществляется по линии С4, а для операции вывода – по линии С6.

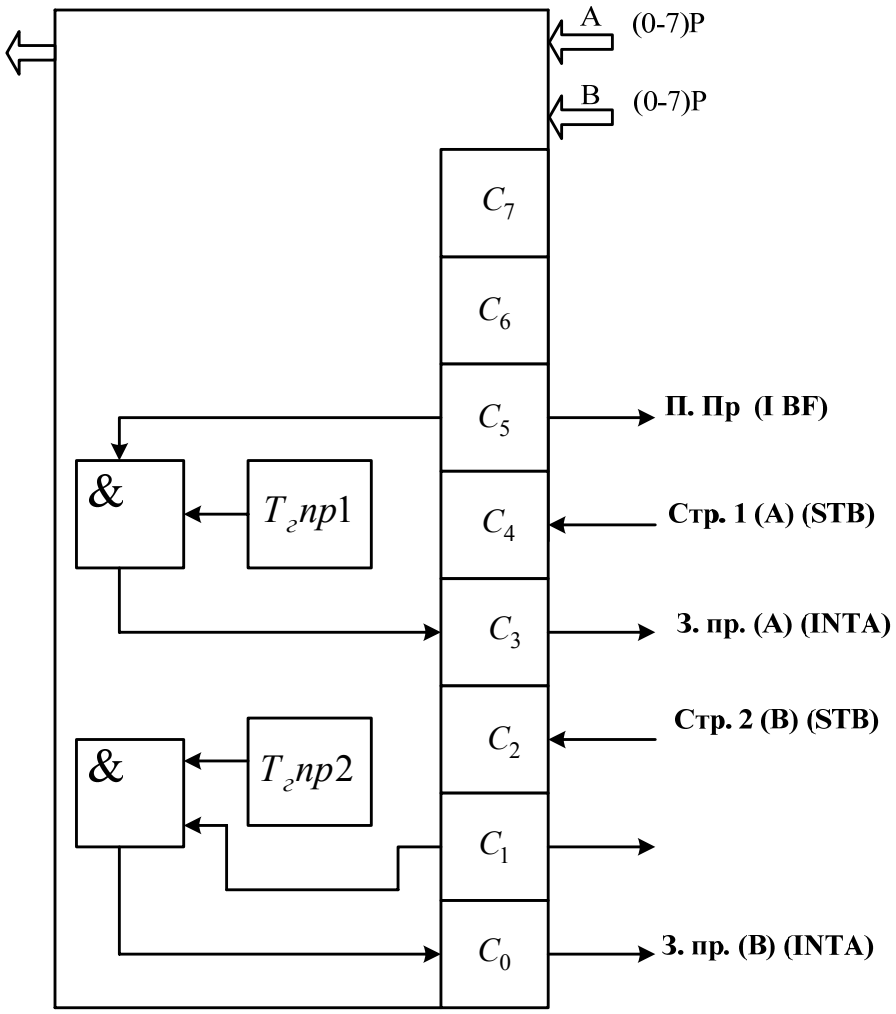


Рис. 29. Передача управляющих сигналов по каналу С в режиме 1 (ввод)

Поскольку микросхема не имеет внутреннего регистра, указывающего на состояние схемы, то для его определения командой IN считывают содержимое регистра канала С, являющееся слово-состоянием контроллера, и интерпретируют отдельные его разряды. Формат слова-состояния показан на рис. 31.

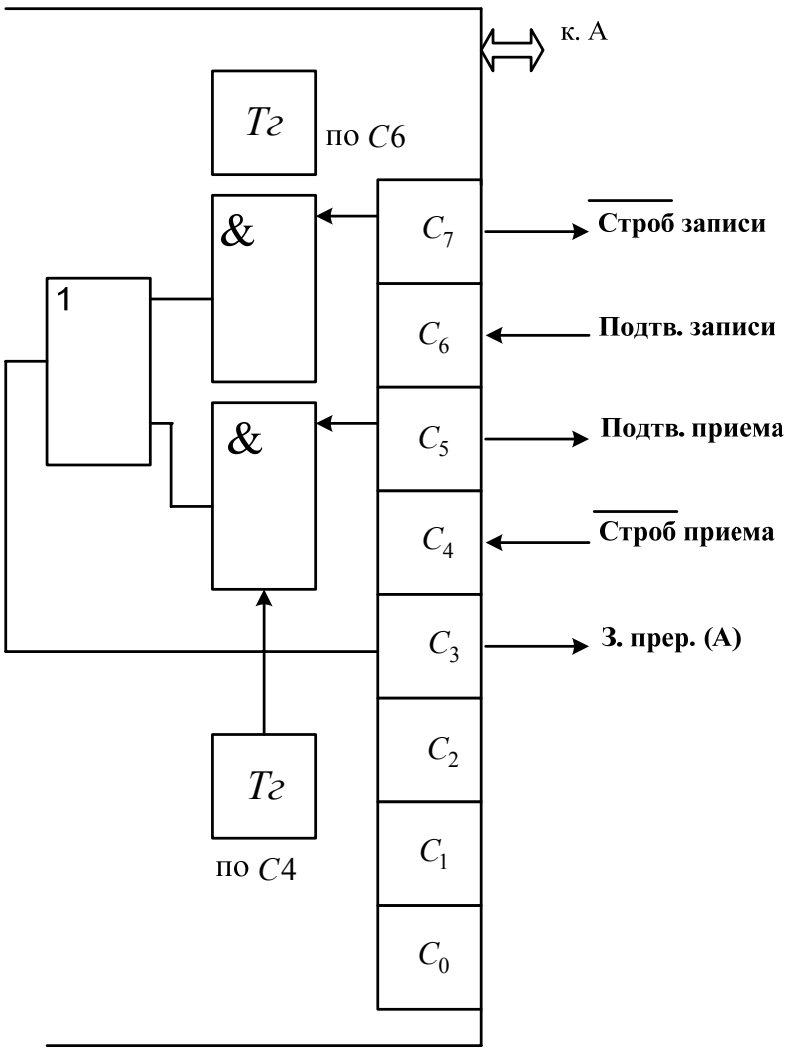


Рис. 30. Передача управляющих сигналов по каналу С в режиме 2

Использование слова-состояния позволяет работать микропроцессору работать с ППИ не только по запросу, но и по опросу. Для этого после считывания слова-состояния программным путём производится анализ разрядов D0 (для канала В) или D3 (для канала А). Разряды D7, D6 показывают напрвление передачи: «1» – ввод, «0» – вывод.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1/0	1/0	Подтв. приема (А)	Разр. прер. (А)	3 пр. (А)	Разр. прер. (В)	Подтв. приема (В)	3 пр. (В)

Рис. 31. Слово-состояние контроллера

Ниже приводится фрагмент программы проверки готовности ВУ при обмене с обслуживанием по программе.

```
M:      IN AL, port ;      чтение слова-состояния при A1, A0 = 11
        AND MASK;          выделение флага готовности.
                           MASK =  00001000 для кан.A
                           MASK =  00000001 для кан.B
```

Лекция № 14

5.5. Программируемые таймеры и счётчики событий

5.5.1. Принцип работы программированного таймера

Довольно часто требуется устройство формирования временных интервалов для процессора и внешних устройств, подсчета внешних событий и ввода показаний в процессор, а также генерирования внешней синхронизации, которую может программировать процессор. Такое устройство называется *программируемым интервальным таймером / счетчиком событий.* Некоторыми областями применения такого устройства являются:

прерывание операционной системы с разделением времени через равномерные интервалы, чтобы она осуществляла переключение программ;

вывод точных временных сигналов с программируемыми периодами в устройство ввода-вывода (например, в аналого-цифровой преобразователь);
программируемая генерация скорости передачи в бодах;
измерение временной задержки между внешними событиями;
подсчет числа появлений событий во внешнем эксперименте и ввод показания в компьютер;
прерывание процессора после появления запрограммированного числа внешних событий.

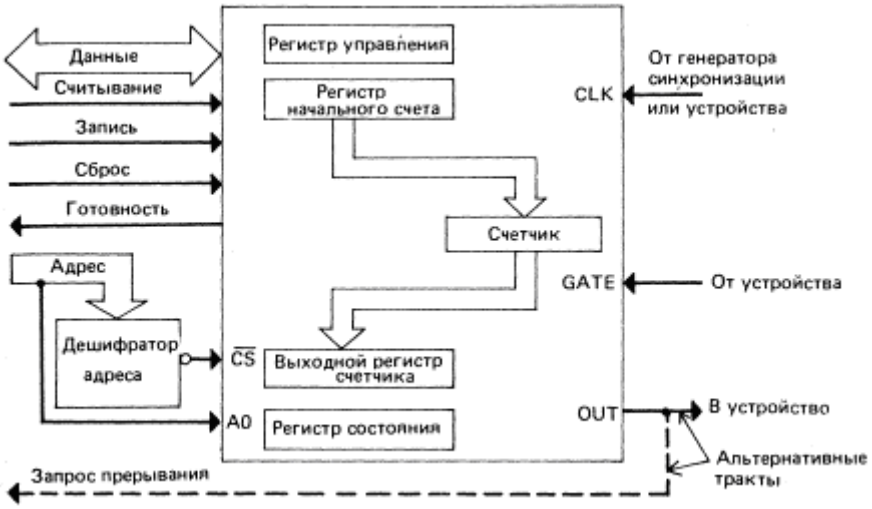


Рис. 32. Типичный интервальный таймер/счётчик событий

Типичная организация интервального таймера / счетчика событий показана на рис.32. Слева находятся четыре доступных компьютеру регистра: два верхних являются выходными портами, а два нижних – входными. Сам счетчик прямо процессору не доступен, но может инициализироваться из регистра начального счета и считывается посредством передачи его содержимого в выходной регистр счетчика. Счетчик запускается с начального значения и отсчитывает до 0. Вход CLK определяет скорость счета, сигнал GATE разрешает и запрещает вход CLK и, возможно, выполняет другие функции, а выход OUT становится активным при достижении счетчиком 0 или, возможно, при подаче сигнала GATE. Выход OUT подключается к линии запроса прерывания в системной шине, поэтому прерывание возникает при достижении счетчиком 0; его же можно подключить к устройству ввода-вывода для инициирования необходимых действий.

Устройство вводит значение в регистр начального счета, передает его в счетчик и выполняет счет "назад" (т. е. вычитание) импульсами со входа CLK. Текущее содержимое счетчика в любой момент можно ввести в процессор, не нарушая работы счетчика, посредством передачи его в выходной регистр счетчика с последующим считыванием из этого регистра. При буферировании содержимого счетчика не требуется вводить его в процессор немедленно. Индикация нуля в счетчике обычно фиксируется на выходе OUT и в одном бите регистра состояния. Поэтому для обнаружения нуля допускается применять программный ввод-вывод и ввод-вывод по прерываниям.

Регистр управления определяет режим работы и выполняет другие функции. Режим точно определяет, что происходит при достижении счетчиком 0 и (или) при подаче сигнала на вход GATE. Возможными действиями являются:

- вход GATE применяется для разрешения и запрещения входа CLK;
- вход GATE вызывает реинициализацию счетчика;
- вход GATE прекращает счет и формирует высокий уровень на выходе OUT;
- при достижении 0 счетчик выдает сигнал OUT и останавливается;
- при достижении 0 счетчик выдает сигнал OUT и автоматически реинициализируется из регистра начального счета.

Режимы могут также определяться комбинациями перечисленных возможностей. Рассмотрим, например, применение интервального таймера в операционной системе с разделением времени. В этом случае на вход CLK подаются сигналы синхронизации, а выход OUT подключается к линии запроса прерывания, возможно, немаскируемого прерывания. Вход GATE здесь не требуется. При включении системы в регистр начального счета загружается значение

начальный счет = частота синхронизации x T,

где T – продолжительность каждого временного кванта в секундах. Задается такой режим, в котором при достижении счетчиком 0 содержимое регистра начального счета вновь загружается в счетчик, а выход OUT становится ак-

тивным. Поскольку сигнал OUT используется как запрос прерывания, процедура прерывания для переключения программ будет выполняться с интервалом T секунд.

5.5.2. Программируемый таймер I8254 (К1810ВИ54)

Программируемый таймер (ПТ) К1810ВИ54 предназначен для генерации времязадающих функций, программно-управляемых временных задержек с возможностью программного контроля их выполнения. Программируемые таймеры применяются в МПС, выполненных на базе МПК БИС К580, К1810, К1821, используемых в задачах управления и измерения в реальном масштабе времени с тактовой частотой до 8 МГц. Конструктивно эти ПТ совместимы с ПТ типа К580ВИ53, отличаются от них повышенным быстродействием и расширенными функциональными возможностями.

Программируемый таймер К1810ВИ54 включает три независимых канала, каждый из которых может быть запрограммирован на работу в одном из шести режимов для двоичного или двоично-десятичного счета. Структурная схема ПТ показана на рис. 33, его условное графическое обозначение – на рис. 34.

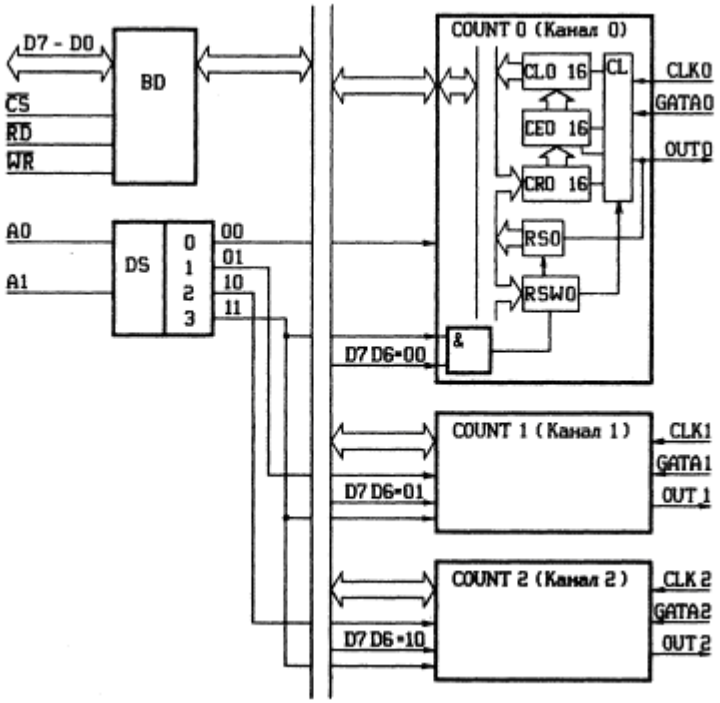


Рис. 33. Структурная схема программируемого таймера К1810ВИ54

Назначение выводов.

CS# – выборка кристалла. Сигнал управляет входным буфером BD. При CS = 0 разрешается работа буфера.

RD# – чтение. Сигнал RD =0 ориентирует входной буфер BD на вывод. ПТ выдает информацию в ЦП.

WR# – запись. Сигнал WR=0 ориентирует входной буфер BD на ввод. ПТ принимает информацию от ЦП.

AO, Al – адресные входы, по которым осуществляется адресация к одному из каналов:

AO = A1 = 00 – адрес канала 0;

AO = A1 = 01 – адрес канала 1;

AO = A1 = 10 – адрес канала 2;

AO = A1 = 11 – признак загрузки управляющего слова или команд.

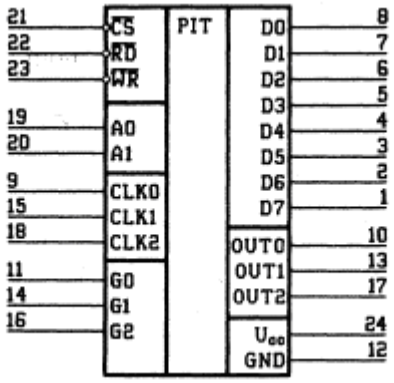


Рис. 34. Условное графическое обозначение К1810ВИ54

CLK2 – CLKO – входы тактовых сигналов для управления счетчиком/таймером. Срез сигнала на входе CLK приводит к уменьшению содержимого счетчика/таймера CE на единицу.

GATA2 – GATAO – входы разрешения счета. При OATA = 1 разрешается выполнение функций; для некоторых режимов работы разрешается поступление тактовых сигналов на вход счетчика/таймера, для других (импульсный генератор и генератор меандра) открывается выходной буфер OUT.

OUT2 – OUTO – выходы счетчика/таймера.

Структурная схема ПТ (рис. 33) включает:

- буфер шины данных (BD) и логические схемы управления чтением/записью:
- дешифратор DS, с помощью которого выбирается один из трех каналов либо формируется признак загрузки управляющих слов или команд;
- три идентичных канала **COUNT2 – COLJNTO**, реализующих запрограммированную функцию.

Каждый канал включает:

❖ *16-разрядный буферный регистр* **CL,** служащий для запоминания и хранения мгновенного значения счетчика CE, которое в любое время может быть записано командой *Защелка* или *Чтение состояния* канала. После выполнения этих команд содержимое OL может быть считано в ЦП без остановки дальнейшего счета в регистре CE;

- *16-разрядный счетчик/таймер* **CE,** работающий в режиме вычитания. Изменение содержимого CE осуществляется по срезу сигнала CLK при GATA== 1;
- *16-разрядный регистр констант пересчета* **CR,** служащий для хранения констант пересчета. Содержимое CR загружается в CE для счета в зависимости от запрограммированного режима;
- *8-разрядный регистр состояния канала* **RS,** содержимое которого можно считывать в ЦП с помощью команды *RBC – Чтение состояния канала.* Содержимое этого регистра является словом состояния канала, формат которого представлен на рис. 6.22.
- *8-разрядный регистр управляющего слова* **RSW,** предназначенный для его хранения. Слово загружается в RSW командой OUT с адресом, формирующим на входах A0, A1 код 11. Выбор конкретного канала осуществляется с помощью двух старших разрядов самого управляющего слова.

Схема управляющей логики канала CL осуществляет управление входом / выходом счетчика / таймера в зависимости от запрограммированного режима.

По правилам загрузки счетчика / таймера CE содержимым регистра CR все шесть режимов работы ПТ можно разделить на три группы.

1. Режимы 0, 4 – режимы однократного выполнения функций. Константы из CR передаются в CE по первому тактовому сигналу CLK при GATA = 1. С приходом последующих сигналов на входе CLK происходит уменьшение содержимого CE. Если во время счета на вход GATA подать нуль, то это приведет к останову счета. Новый положительный сигнал на GATA не вызывает перезагрузку счетчика / таймера, а только разрешает продолжение счета. По окончании счета выполнение действий заканчивается. При необходимости повторения функции требуется новое программирование – загрузка новой константы.

2. Режимы 1,5 – режимы с перезапуском. Здесь характерна возможность повторения запрограммированных функций без нового перепрограммирования. Загруженная константа сохраняется в CR, а ее передача в CE осуществляется по фронту сигнала GATA независимо от завершения счета.

3. Режимы 2, 3 – режимы автозагрузки. Загрузка CE содержимым CR осуществляется автоматически при выполнении условий счета (импульсный генератор и генератор меандра), поскольку это режимы с зацикливанием счета. Выход OUT открывается положительным сигналом на GATA.

Программирование. ПТ относится к классу функционально ориентированных программно управляемых интерфейсных БИС, поэтому перед началом работы в него необходимо загрузить управляющее слово (УС) и константу пересчета. УС задает один из шести режимов работы, тип счета (двоичный или двоично-десятичный), порядок загрузки и размерность (один или два байта) константы. Времяимпульсная функция формируется на выходе OUT при GATA = 1. Формирование функции осуществляется с помощью

счетчика-таймера СЕ, работающего в режиме вычитающего счетчика по срезу сигнала CLK.

Программист может опросить состояние каналов ПТ с помощью специальных команд *Защелка (Чтение на лету)* (CLC) или *Чтение состояния канала* (RBC). Эти команды позволяют, не прерывая счета, опросить состояние счетчика / таймера СЕ. Кроме того, команда RBC позволяет прочитать содержимое регистра состояния канала, разряды которого несут информацию о запрограммированном режиме, состоянии выхода OUT и флага «обновления».

Управляющее слово **CW.** Формат УС показан на рис. 35. Управляющие слова загружаются в регистры RSW каналов ПТ по командам вывода, формирующим на входах ПТ коды AOA1 = 11, CS# = 0, WR# = 0, RD# = 1. Управляющие слова загружаются в тот канал ПТ, адрес которого указывается в самом формате УС, и сохраняются там во все время работы или до следующего программирования. Загрузка УС должна предшествовать загрузке констант.

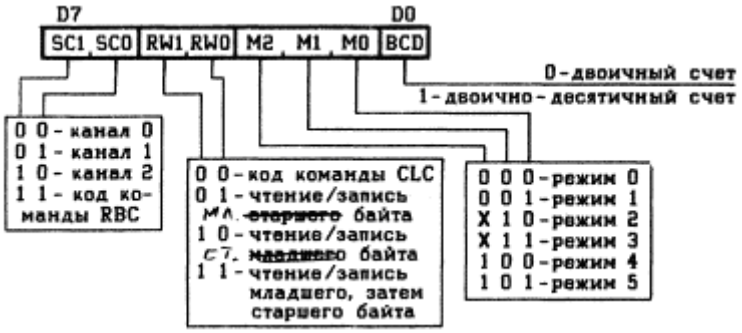


Рис.35. Формат управляющего слова

В формате УС можно выделить четыре функциональных поля: SC, RW, M, BCD, с помощью которых задаются основные параметры работы канала.

Поле **SC** (разряды D7, D6) определяет адрес регистра RSW,, конкретного канала. Если в этом поле содержится код 11, то загружаемая информация воспринимается ПТ как команда *Чтение состояния канала* (см. далее описание команды RBC).

Поле **RW** (разряды D5, D4) определяет размерность и порядок загрузки констант. Если в поле RW заданы коды 01 или 10, то размер константы определен соответственно старшим или младшим байтом. Если в поле RW задан код 11, то размер константы два байта; сначала загружается младший байт, затем старший. Если в поле RW задан код 00, то загружаемый байт воспринимается как команда *Защелка* (см. далее описание команды CLC).

Поле M (разряды D3 – D1) задает один из шести режимов работы канала:

* режим 0 (000) – прерывание от таймера;
* режим 1 (001) – программируемый ждущий мультивибратор;
* режим 2 (Xl0) – импульсный генератор частоты;

* режим 3 (Xll) – генератор импульсов со скважностью два;
* режим 4 (100) – программно-запускаемый одновибратор;
* режим 5 (101) – аппаратно-запускаемый одновибратор.

В режимах 2, 3 разряд D3 может принимать любое значение. (Подробно работа каналов в соответствующих режимах будет рассмотрена далее).

Поле **BCD** (разряд DO) определяет тип счета. При D0 = 0 константа задается в двоичном коде и может принимать значения в диапазоне 0 – 65 536. При D0 = 1 константа задается в двоично-десятичном коде в диапазоне 0 – 9999.

После загрузки УС необходимо загрузить в каналы константы пересчета. Константа пересчета загружается в ПТ также по командам вывода, но с адресом, формирующим на входах АО, А1 код соответствующего канала (00, 01, 10). Константа может быть задана байтом (старшим или младшим) или 16-разрядным словом (как это определено полем RW управляющего слова) и представлена двоичным или двоично-десятичным кодом (как определено полем BCD). Порядок загрузки каналов управляющими словами и константами строго определен. Возможны два варианта. Первый предполагает загрузку в любой последовательности сначала всех УС, затем констант пересчета. Второй предполагает загрузку управляющего слова для любого канала, а затем константы пересчета для этого же канала.

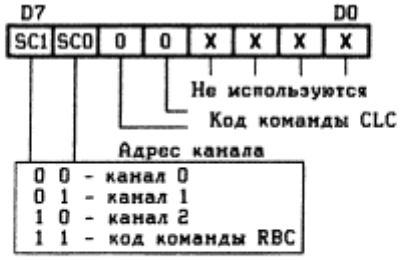
Общими и обязательными требованиями для загрузки УС и констант являются следующие: 1) загрузка УС должна опережать загрузку константы;

2) загрузка констант всегда должна выполняться до конца, как определено разрядами RW1, RWO в формате УС.

Состояние счетчика/таймера СЕ можно прочитать тремя способами.

Чтение по обычным командам ввода позволяет прочесть состояние счетчика таймера СЕ в любой момент времени. Выполняется с помощью обычных команд ввода с адресом, формирующим на входах АО, А1 код соответствующего канала. Необходимым условием для выполнения этой операции является остановка счета перед выполнением команд чтения, т. е. GATA=0. Операцию чтения необходимо выполнять до конца, т. е. нельзя прочитать только один байт СЕ, если константа задана 16-разрядным словом. Сначала считывается младший байт, затем старший.

Чтение по команде **CLC** *(Защелка).* Команда CLC позволяет прочитать состояние счетчика/таймера СЕ в любой момент времени без остановки счета. Для этого программист должен загрузить эту команду в ПТ в определенный момент времени. Команда загружается в ПТ так же, как УС, т. е. по команде вывода с адресом, формирующим на входах АО, А1 код 11. Формат команды CLC представлен на рис. 36. Разряды D7, D6 задают адрес защелкиваемого канала. Эти разряды не могут принимать значение 11, являющееся кодом команды RBC. Разряды D5, D4 являются кодом команды CLC и принимают значение 00. Разряды D3 – DO не участвуют в операции и могут принимать любое значение. Таким образом, существует три команды CLC, отличающиеся кодом в разрядах D7, D6:



CLC0 = 0000×××× защелкивается канал 0;
CLC1 = 0100×××× защелкивается канал 1;
CLC2 = 1000×××× защелкивается канал 2.

Рис.36. Формат команды CLC

После загрузки команды CLC выполняется операция чтения так же, как в первом способе, с теми же необходимыми условиями, кроме снятия сигнала GATA. Фактически команда CLC записывает состояние счетчика / таймера CE в буферный регистр OL, из которого информация считывается без нарушения продолжения счета. Информация из OL может быть считана в любое время. Необходимо помнить, что если операция чтения не выполнена или выполнена не до конца, то новая команда CLC не воспринимается. Прочитать состояние регистра OL можно только после выполнения хотя бы одного цикла в CE.

Команда **RBC** *(Чтение состояния канала)* позволяет в любой момент времени прочитать слово состояния канала (SW), т. е. содержимое регистра RS, а также выполнить защелку одного или нескольких каналов одновременно. Формат команды RBC представлен на рис. 37. Она загружается в ПТ так же, как УС, т. е. по команде вывода с адресом, формирующим код AOA1 = 11. Старшие разряды (D7D6 = 11) являются кодом, по которому ПТ распознает информацию на входе как команду RBC.

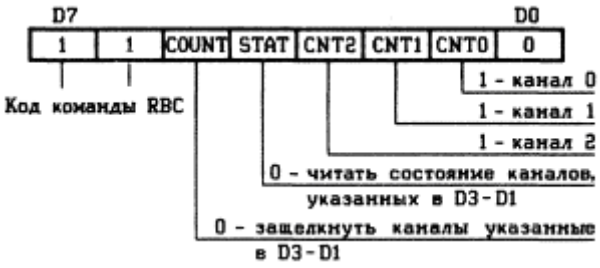


Рис. 37. Формат команды RBC

Команда RBC может выполнять две операции: 1) *Защелку,* что аналогично команде CLC, и 2) *Чтение регистра состояния канала.* Эти операции задаются независимо кодами D5 = 0, D4 = 0. Возможно совмещение операций. При D5D4 = 11 операций нет. Особенностью этой команды является возможность выполнения операций одновременно в нескольких каналах. Ус-

тановка кода D3D2D1 = 111 определяет операцию с соответствующим каналом.

Например, команда RBC = 11001110 одновременно защелкивает / перезаписывает состояние регистров CE всех трех каналов в собственные регистры OL, а также состояние каналов в RS и делает их доступными для чтения. Эта команда эквивалентна трем командам CLC. Как и при команде CLC, новое «защелкивание» возможно только после полного считывания содержимого OL.

Рис. 38. Слово-состояние

Слово-состояние канала **SW.** Каждый канал имеет регистр слова-состояния RS. Формат слова состояния показан на рис. 38, из которого видно, что в процессе работы канала изменяются только два старших разряда слова состояния: D7 и D6. Остальные разряды соответствуют разрядам ранее загруженного управляющего слова, что позволяет контролировать правильность его загрузки.

Разряд D7 слова состояния канала несет информацию о состоянии выхода канала OUT в момент выполнения команды RBC. Разряд D6 является флагом обновления регистра констант CR. Этот флаг особенно необходим для режимов 1 и 5, он позволяет определить, произошла ли загрузка константы из регистра CR в CE или нет. Напомним, что для этих режимов загрузка осуществляется аппаратно – по фронту сигнала GATA. Поэтому если флаг обновления установлен в нуль, то это означает, что ранее загруженная константа перезаписана из CR в CE и по ней осуществляется операция счета. В этом случае можно произвести загрузку новой константы, не нарушая предыдущего счета. По фронту следующего сигнала GATA начинается счет с новой константой.

Изменение состояния флага обновления при работе и программировании можно проиллюстрировать следующим примером:

при записи УС флаг FN =1;

при записи констант в CR FN = 1;

при загрузке константы в CE FN = 0.

Подключение ПТ к МПС осуществляется с помощью линий CS#, RD#, WR#, AO, A1 по общим правилам подключения интерфейсных БИС к системной шине МПС на базе МПК К1810. На рис. 39 показана схема подключения ПТ к шине МПС. Входы RD#, WR# ориентируют входной буфер

ПТ на прием или выдачу, поэтому подключаются к соответствующим выходам ЦП (I/OR, I/OW), на которых формируется сигнал низкого уровня при выполнении команд ввода – вывода. Вывод CS# является входом разрешения выбора ПТ.

Нулевой потенциал на нем разрешает работу входного буфера ПТ. Этот вход используется для адресации ПТ, он подключается либо к одной из свободных шин адреса (принцип раздельной дешифрации), либо к выходу дешифратора ВУ. Выводы А0, А1 являются входами адресов внутренних регистров ПТ, они подключаются к шинам адреса, не занятым в адресации ПТ, обычно к А0, А1 шины адреса МПС. Временная диаграмма сигналов CLK, WR#, GATA, OUT показана на рис. 40, а зависимость состояний сигналов на входах ПТ и выполняемых операций ПТ с ЦП – в табл. 22.

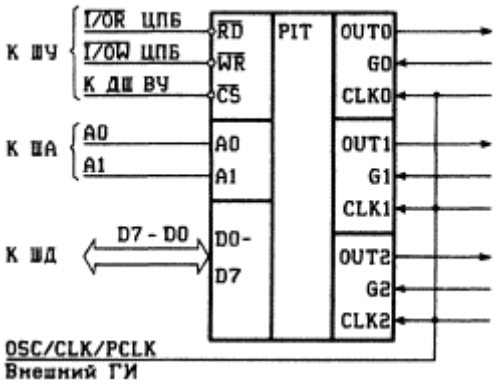


Рис. 39. Схема подключения ПТ к шине МПС

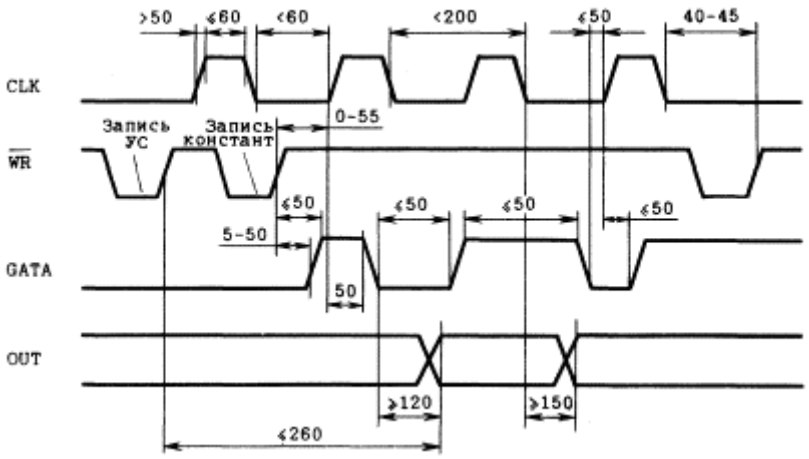


Рис. 40. Временная диаграмма сигналов CLK, WR#, GATA, OUT

Таблица 22

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A1	A0	Операция
0	1	0	0	0	Запись константы в канал 0
0	1	0	0	1	Запись константы в канал 1
0	1	0	1	0	Запись константы в канал 2
0	1	0	1	1	Запись УС. команд
0	0	1	0	0	Чтение состояния канала 0
0	0	1	0	1	Чтение состояния канала 1
0	0	1	1	0	Чтение состояния канала 2
0	0	1	1	1	Нет операции (z-состояние)
1	X	X	X	X	То же
0	1	1	X	X	>>

Функционирование. Как уже отмечалось, каналы ПТ независимо могут быть запрограммированы на работу в одном из шести режимов.

В режиме 0 – прерывание от таймера – низкий уровень сигнала на выводе OUT устанавливается сразу же после загрузки УС. Загрузка константы не оказывает влияния на этот выход. Счет разрешается положительным сигналом на входе GATA. Изменение состояния счетчика/таймера СЕ осуществляется по срезу сигнала CLK, причем по первому тактовому сигналу происходит загрузка СЕ константой из CR, и только второй тактовый сигнал принимает участие в счете. После отсчета загруженного числа устанавливается сигнал OUT = 1. Таким образом, сигнал OUT = 0 удерживается на время N+1 тактовых периодов, где N – загруженная константа.

Если во время счета снять сигнал GATA, то счет приостанавливается, содержимое счетчика / таймера сохраняется. Новый положительный сигнал на входе GATA вызывает продолжение счета без перезагрузки СЕ содержимым CR. Загрузка новой константы во время счета приводит: при записи младшего байта – к остановке текущего счета, а при записи старшего – к запуску нового цикла счета.

Контроль счетчика (выполнение команд CLC, RBC) в этом режиме возможен только после хотя бы одного цикла счета. На рис. 41 показана временная диаграмма работы ПТ в режиме 0.

В режиме 1 – программируемого ждущего мультивибратора – на выходе OUT формируется сигнал низкого уровня длительностью T = TclkN, где *Tclk* – период тактовых импульсов; N – константа. На выходе OUT по положительному фронту сигнала GATA устанавливается нулевой сигнал, который изменяется после окончания счета. Режим 1 является режимом с перезапуском. По каждому фронту сигнала на входе GATA регистр СЕ перезагружается содержимым CR. Это означает, что однажды загруженная константа участвует в счете всякий раз по фронту сигнала GATA, причем по фронту первого сигнала GATA флаг обновления устанавливается в нуль.

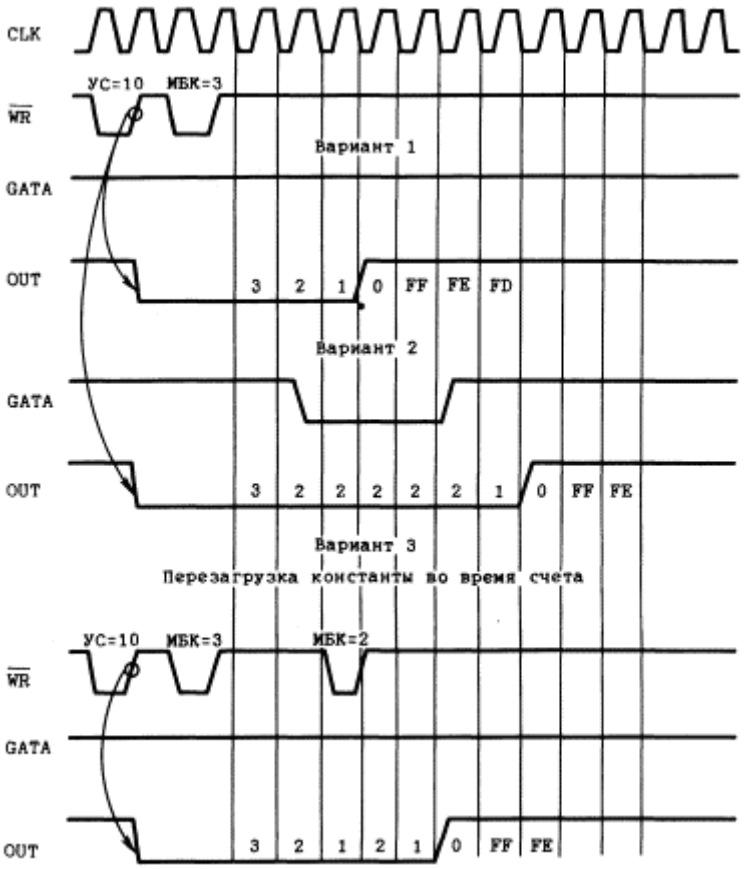


Рис. 41. Временная диаграмма работы ПТ в режиме 0

Если во время счета в ПТ загружается новая константа, то она устанавливает флаг обновления в единицу, но не влияет на текущий счет. Новый счет начинается только по фронту следующего сигнала GATA. Выполнение команд CLC и RBC возможно только после выполнения хотя бы одного цикла счета.

Временная диаграмма работы ПТ в режиме 1 показана на рис. 42. *В режиме 2 – импульсного генератора частоты* – канал работает как делитель входной частоты Fclk на N. Сразу же после загрузки УС на выходе OUT устанавливается единичный сигнал. При GATA = 1 на выходе OUT с частотой Fclk / N устанавливается нулевой сигнал на время одного периода CLK. Режим 2 является режимом с автозагрузкой, т. е. после окончания цикла счета СЕ автоматически перезагружается и счет повторяется. Перезагрузка канала новой константой не влияет на текущий счет, новый счет начинается по окончании предыдущего. При GATA = 0 на выходе OUT устанавливается напряжение высокого уровня и счет останавливается. При сигнале GATA = 1 счет продолжается, что позволяет синхронизировать работу канала с внешними событиями. Выполнение команд CLC и RBC возможно для этого режима после окончания двух циклов счета.

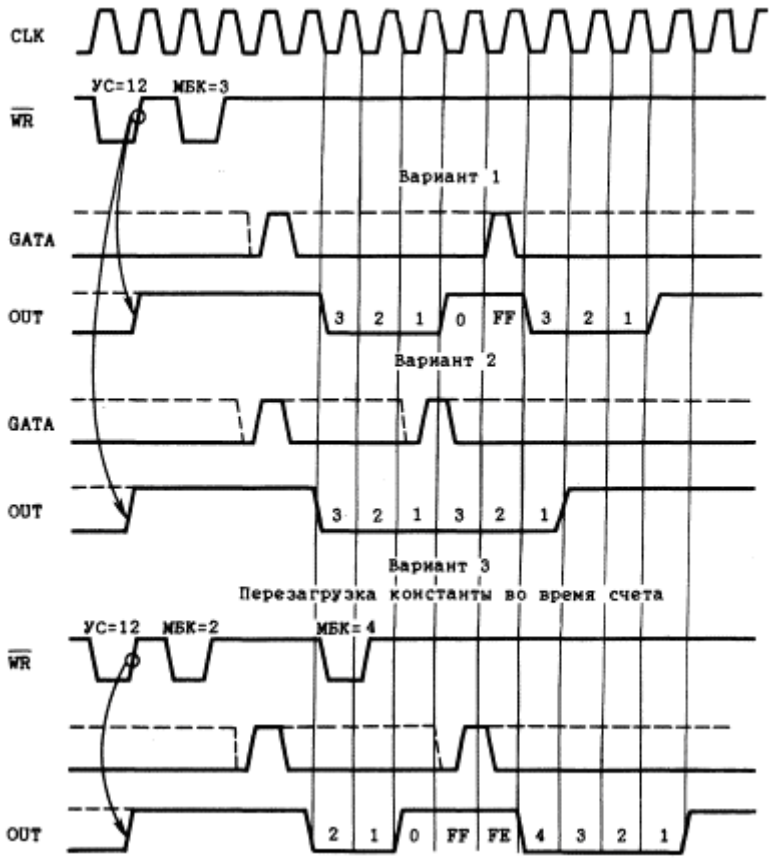


Рис.42. Временная диаграмма работы ПТ в режиме 1

Временная диаграмма работы ПТ в режиме 2 показана на рис. 43 для трех различных вариантов.

Режим 3 – генератора импульсов со скважностью два – аналогичен режиму 2, за тем исключением, что на выходе OUT формируются импульсы с длительностью полупериодов, равной (N/2)*Tclk при четных N; ((N+1)/2)*Tclk положительных и ((N-1)/2)*Tclk отрицательных полупериодов при нечетных .

Этот режим является режимом с автозагрузкой, т. е. перезагрузка CE константой из CR выполняется автоматически после окончания цикла счета. Перезагрузка константы во время счета не влияет на текущий счет, новый счет начинается по окончании предыдущего. Снятие сигнала GATA приостанавливает счет, установка его продолжает цикл счета. В этом режиме канал может работать только с константой больше трех. Выполнение команд CLC и RBC возможно только после двух циклов счета.

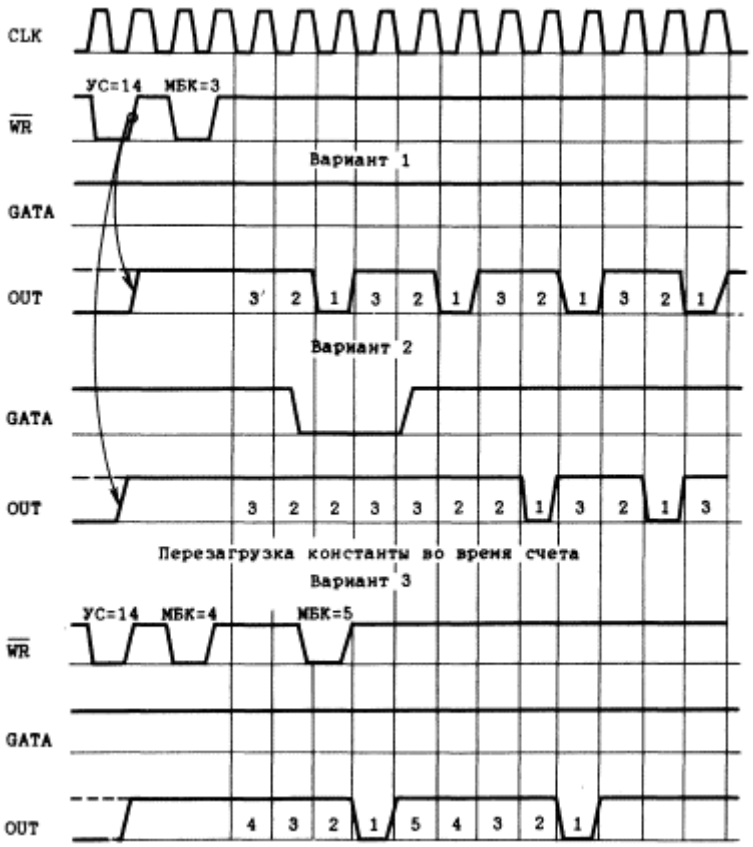


Рис.43. Временная диаграмма работы ПТ в режиме 2

Временная диаграмма работы ПТ в режиме 3 показана на рис. 44. *В режиме 4 – программно-запускаемого одновибратора* – по окончании отсчета числа, загруженного в счетчик/таймер, на выходе OUT устанавливается нулевой сигнал на время одного периода сигнала CLK. Высокий уровень сигнала на выходе OUT устанавливается сразу же после загрузки УС. Сигнал высокого уровня на входе GATA разрешает счет, причем первым тактовым сигналом происходит загрузка счетчика / таймера CE константой из CR, а второй тактовый сигнал начинает счет. Таким образом, сигнал длительностью, равной периоду тактовой частоты, устанавливается на выходе OUT через N+1 1 тактовых периодов. Если во время счета снимается сигнал GATA, то счет приостанавливается, текущее значение CE счетчика/таймера сохраняется. Новый положительный сигнал на GATA вызывает продолжение счета. Это режим одноразового выполнения функции. Загрузка новой константы во время счета приводит: при записи младшего байта к остановке текущего счета, а при записи старшего – к запуску нового цикла счета.

Выполнение команд CLC и RBC возможно только после окончания одного цикла счета.

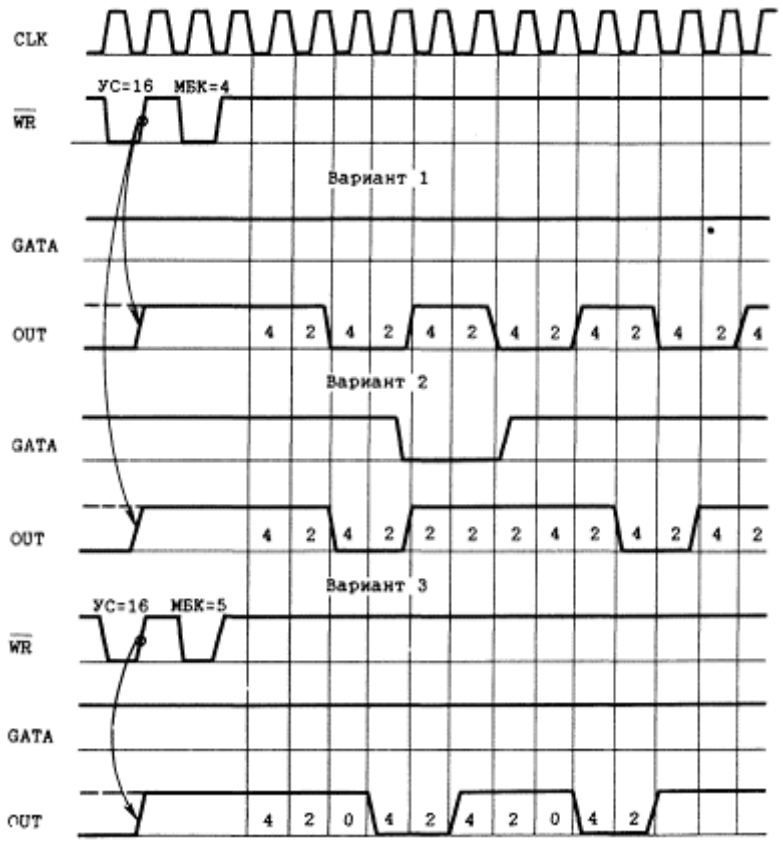


Рис.44. Временная диаграмма работы ПТ в режиме 3

Временная диаграмма работы ПТ в режиме 4 показана на рис.45. *Режим 5 – аппаратно-запускаемого одновибратора* – аналогичен режиму 4 по способу формирования сигнала на выходе OUT и режиму 1 по действию сигнала GATA.

На выходе OUT устанавливается сигнал нулевого уровня на время одного периода CLK после отсчета загруженной в CE константы. Загрузка в CE константы из CR осуществляется по фронту сигнала GATA. Из этого следует, что по фронту GATA происходит новая загрузка CE из CR, причем первый фронт GATA устанавливает флаг обновления в нуль. Если во время счета в канал загружается новая константа, то эта операция устанавливает флаг обновления в единицу, но не влияет на текущий счет. Новый счет начинается только по фронту следующего сигнала GATA. Выполнение команд CLC и RBC возможно только после выполнения хотя бы одного цикла счета.

На выходе OUT устанавливается сигнал нулевого уровня на время одного периода CLK после отсчета загруженной в CE константы. Загрузка в CE константы из CR осуществляется по фронту сигнала GATA. Из этого следует, что по фронту GATA происходит новая загрузка CE из CR, причем первый фронт GATA устанавливает флаг обновления в нуль. Если во время счета в канал загружается новая константа, то эта операция устанавливает флаг обновления в единицу, но не влияет на текущий счет. Новый счет начинается

только по фронту следующего сигнала GATA. Выполнение команд CLC и RBC возможно только после выполнения хотя бы одного цикла счета.

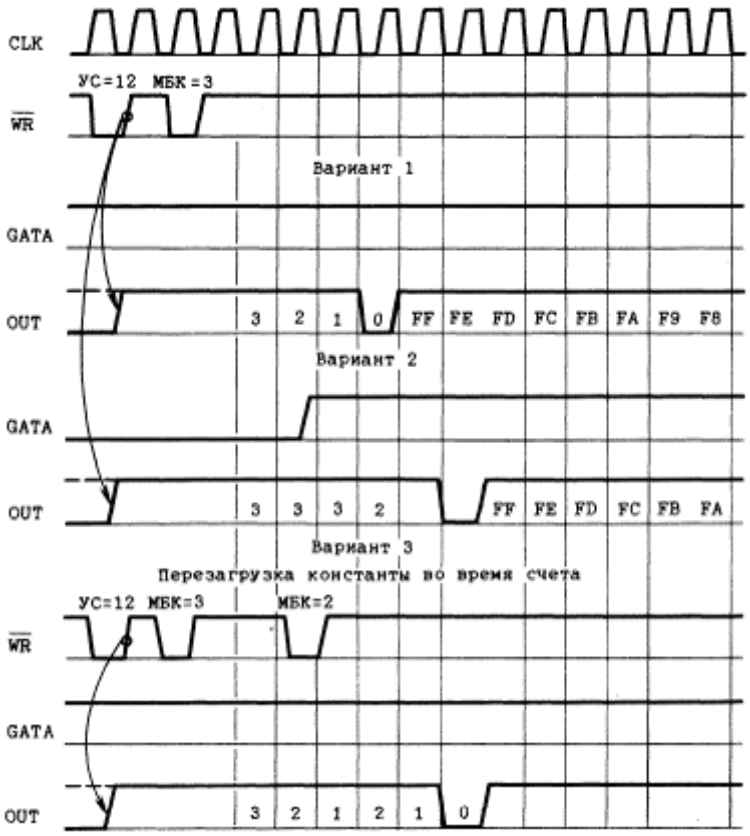


Рис.45. Временная диаграмма работы ПТ в режиме 4

Временная диаграмма работы ПТ в режиме 5 показана на рис.46. Из описания режимов ПТ видно, что таймер может реализовать все основные времязадающие функции, широко используемые в цифровой технике, с помощью которых выполняются как традиционные функции, связанные с управлением МПС, так и специфические, измерительные функции, например для измерения частоты. В заключение следует отметить, что при конструктивной совместимости таймеров К 580 ВИ 53 и К 1810 ВИ 54 последний кроме улучшенной частотной характеристики обладает более широкими функциональными возможностями, в особенности при реализации параллельных процессов.

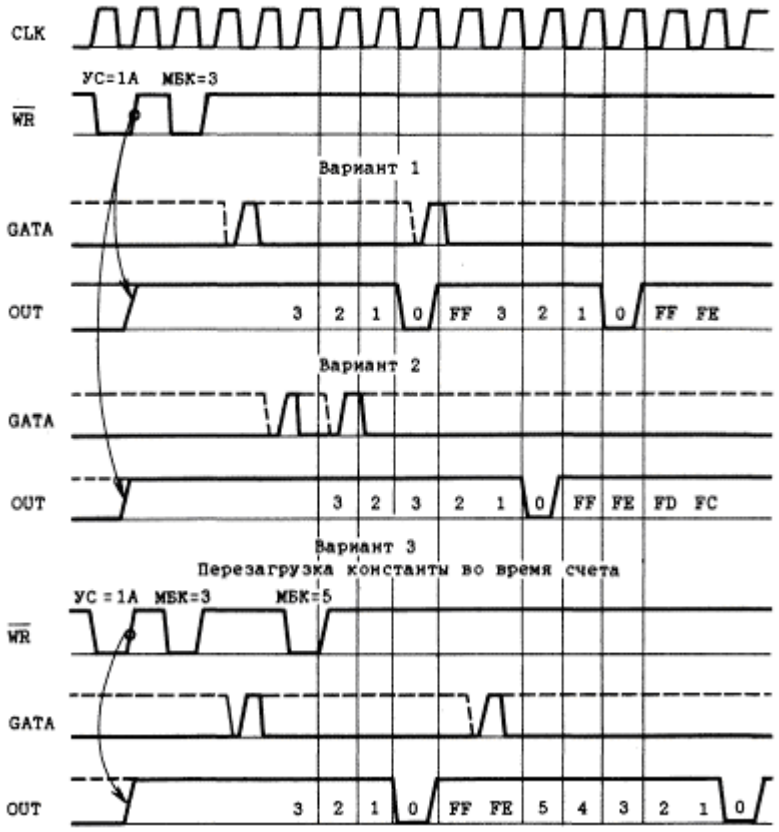


Рис.46. Временная диаграмма работы ПТ в режиме 5

Лекция № 15

5.6. Клавиатура и индикатор

5.6.1. Общие принципы управления клавиатурой и дисплеем

Консоли (пульты управления) небольших систем обычно реализуются как устройства ввода и вывода простых клавиатуры и индикатора. С помощью клавиатуры данные, адреса памяти и машинные коды вводятся в шестнадцатеричном формате. При выводе необходимые данные отображаются на светодиодных индикаторах.

Основные функции интерфейса, связывающие клавишные устройства ввода и вычислитель, заключаются в следующем:

опознавание факта замыкания ключа;

декодирование данных, соответствующей нажатой клавише.

Клавиатуру можно представить в виде столбцов и строк.

Рассмотрим принцип подключения клавиатуры на примере четырёх клавиш (рис.47).

В нормальном состоянии на вертикальных линиях поддерживается уровень логической 1. При нажатии клавиши нужную горизонтальную линию можно определить поочередной подачей логического 0 через порт А. Допустим, нажата клавиша А, тогда только на линии PB_2 будет состояние ло-

гического 0 (по PB_0 соответственно), на PB3 – логическая 1, т. е. поочередно подается «0» на горизонтальную линию и проверяется, где появится «0» на вертикальной линии. Считывая порт B, программно определяют нажатую клавишу. Данный процесс называется сканированием клавиатуры. С клавиатурой связаны две сложности:

важно не фиксировать дребезг контактов, для этого обычно через 20 мс производят проверку замыкания еще раз. Это программный подход к подавлению дребезга, он ведет к чрезмерным потерям времени. Подавление дребезга можно осуществить и аппаратно. Следующий шаг – преобразование полученной информации (номера строки и столбца) в более подходящий код (шестнадцатеричный или ASCI I), это делается программно;

второй проблемой является распознавание одновременно нажатых двух клавиш и более. Существуют клавиатуры, где можно нажать только одну клавишу (остальные игнорируются) и где можно нажимать несколько. Наиболее простое решение одновременного нажатия клавиш – сканирование всего массива и ввода информации о замыкании в порядке их обнаружения.

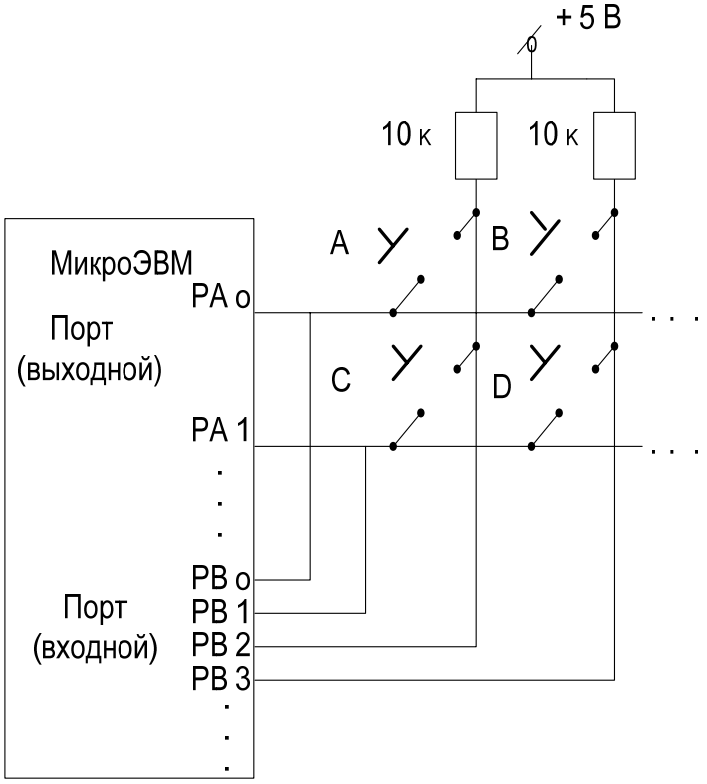


Рис. 47. Схема подключения клавиатуры 2 х 2 к микроЭВМ

Существует довольно много видов цифровых и буквенно-цифровых индикаторов. Шестнадцатеричные цифры обычно индицируются светодиодными семисегментными индикаторами. Цифры в семисегментном коде (рис. 2, а) подаются на входы a–g и ДР (десятичная точка). В зависимости от типа индикатора (общий анод или катод) активные сигналы имеют высокий или низкий уровень (рис. 48, а,б и в). Сегмент начинает светиться, когда соответ-

ствующий светодиод смещен в прямом направлении. Следовательно, для активизации набора индикаторов нужно вывести восьмиразрядный код, соответствующий цифре, плюс номер того индикатора, куда этот код должен попасть. Кроме того, необходимо обеспечить регенерацию свечения.

Часто используют многоразрядные индикаторы, образованные из восьми элементов. Все элементы подключены к двум 8-битным параллельным портам и работают в мультиплексном режиме. Каждый элемент включается на 1 мс, а после прохождения всех разрядов последовательность операций повторяется, начиная с первого. Так создается иллюзия непрерывно работающего многоразрядного индикатора.

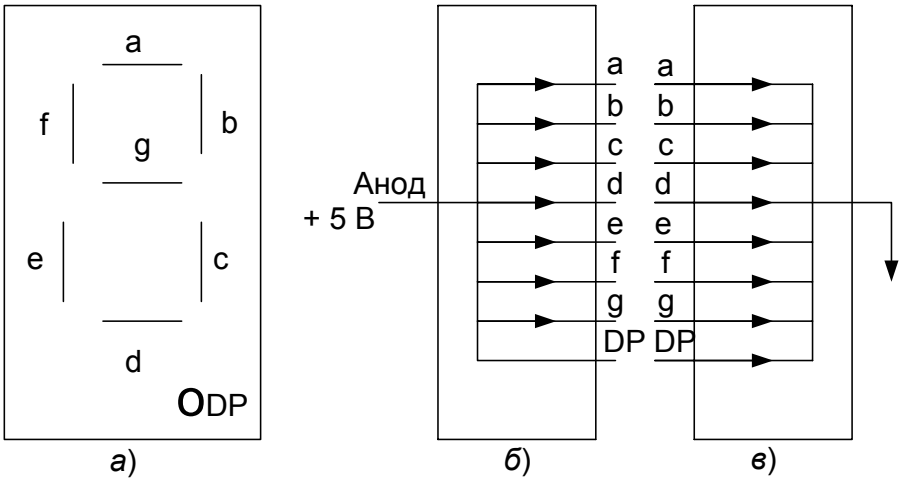


Рис.48. Семисегментный светодиодный индикатор (а), индикатор с общим анодом (б) и общим катодом (в)

5.6.1. Контроллер клавиатуры и индикации I 8279 (КР580ВВ79)

Контроллер предназначен для ввода данных в микропроцессор с клавиатуры и вывода из микропроцессора на индикаторный дисплей. Микросхема состоит из двух функционально автономных частей: клавиатурной и дисплейной. На рис. 49 приведена структурная схема контроллера.

Клавиатурная часть. Клавиатура может быть выполнена в виде 64-контактной клавишной матрицы или в виде набора сенсорных элементов или элементов на основе изменения магнитной проницаемости ферритов (ввод по стробу). Возможны два варианта работы с клавиатурой:

запрещены одновременные нажатия нескольких клавиш;

номера клавиш сканируются независимо друг от друга.

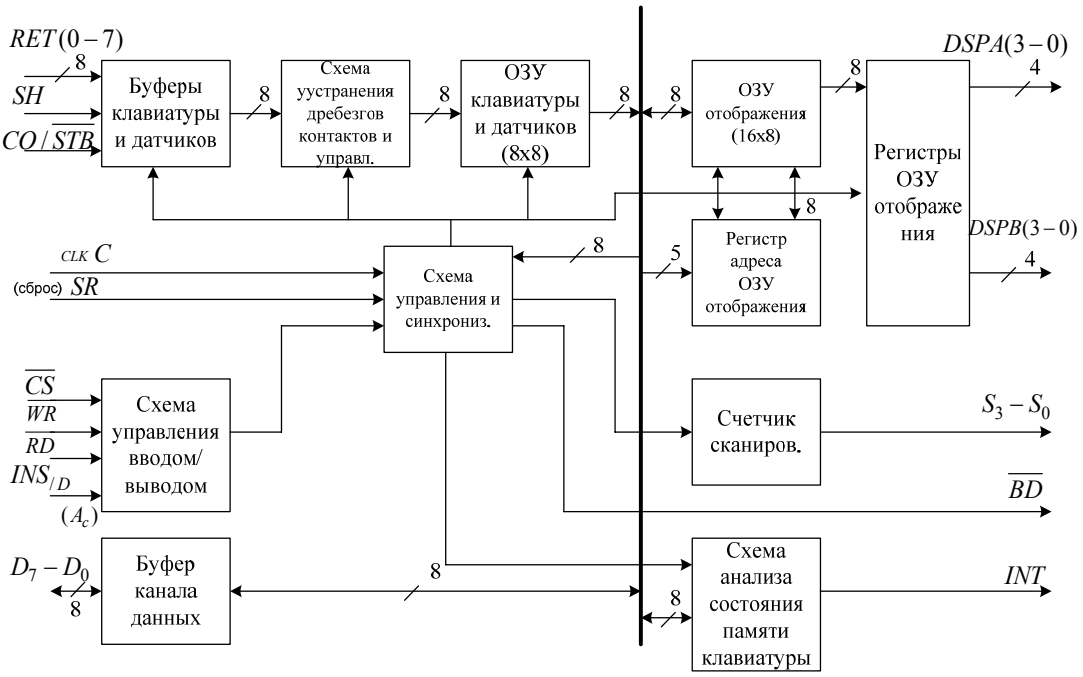


Рис. 49 Контроллер клавиатуры и индикации КР580ВВ79 (I 8279)

В микросхеме предусмотрено устранение дребезга контактов. Коды нажатых клавиш заносятся в ОЗУ клавиатуры емкостью 8 байт (8 х 8). ОЗУ организовано по принципу FIFO – «первым записан – первым прочитан». Появление первого же байта в ОЗУ приводит к возникновению на выходе INT сигнала запроса прерывания, поступающего на МП.

Для клавиатурной части возможны 3 режима работы:
режим сканирования контактной клавиатуры 1 матрица х 8 слов х 8 разрядов и 4 матрицы х 8слов х 8 разрядов;
режим сканирования сенсорной матрицы датчиков;
режим стробируемого ввода.

При работе в первом режиме, управляя клавиатурой, контроллер непрерывно сканирует каждую строку клавиатуры посредством выдачи адресов строк на линии S_0-S_3 и ввода сигналов с возвратных линий RET(0-7). При нажатой клавише соответствующая линия RET перейдет в низкий логический уровень. Итак:

RET(0 - 7) *– линии возврата, по ним поступают данные из клавиатуры. В режиме сканирования по линиям RET данные из клавиатуры поступают в контроллер для формирования 8-битного кодового слова, записываемого во внутреннюю память контроллера.*

S_0 - S_3 *– сигналы управления сканированием клавиатуры, по этим линиям выдаются адреса строк, чтобы можно было проверить каждую строку и отыскать нажатую клавишу. По этим же выводам идет информация об адресе (номере) индикатора в том случае, когда контроллер работает с индикаторами.*

Для выработки сигналов сканирования клавиатуры, матрицы датчиков и дисплея на линиях S_0-S_3 внутри микросхемы имеются схема управления и синхронизации имеются и счетчик сканирования. На вход схемы синхронизации подаются внешние тактовые импульсы CLK с периодом повторения $T_{CLK} \geq 0{,}4$ Мкс, а на выходе вырабатываются внутренние тактовые импульсы с периодом $T_{Тл} \geq 10$ Мкс. Коэффициент деления программного делителя частоты P = 2÷31.

Схемы синхронизации содержат цепочку счетчиков. Первый счетчик – это предварительный делитель тактовой частоты, который может быть запрограммирован на базовую частоту 1 МГц, что обеспечивает сканирование клавиатуры с периодом 5,1 Мкс. Прочие счетчики делят базовую частоту с тем, чтобы обеспечить необходимые времена сканирования отдельных клавиш, рядов клавишной матрицы, а также индикатора.

CLK – *вход синхронизации, частота импульсов на этом входе должна превышать 200 кГц.*

Счётчик сканирования может работать в двух режимах:

кодированного сканирования (т. е. с внешней дешифрацией), в этом случае происходит обычный двоичный счет, обеспечивающий выдачу на выходы сканирования S_0-S_3 двоичного кода последних четырех разрядов счетчика сканирования;

дешифрированного сканирования, при этом внутри микросхемы дешифрируются два младших разряда счетчика сканирования и обеспечивается выдача дешифрированных сигналов на выводы S_0-S_3, т. е. производится выборка одной из четырёх строк матрицы или одного из четырёх индикаторов.

В контроллере имеются входы CONTRL и SHIFT. Они предназначены для клавиатуры типа клавиатуры пишущей машинки, в которой имеются клавиши управления и переключения регистров.

SHIFT *– нижний/верхний регистр, т.е. это входной сигнал сдвига, используемый в режиме сканирования клавиатуры. Вход предназначен для кодирования номера матрицы клавиатуры (их, как уже говорилось, может быть 4), он «работает» в паре с другим входом: CO/STB (CONTR/STB). Этот сигнал воспринимается вместе с координатами нажатой клавиши и запоминается в буфере в виде одного бита.*

CO/STB (CONTRL/STB) *– «управляющий символ» или «строб клавиатуры». В режиме сканирования клавиатуры используется как выше описанный вывод SH, он также воспринимается вместе с координатами нажатой клавиши и запоминается в буфере в виде одного бита. В режиме «стробируемый ввод» по переходу сигнала CO/STB из нуля в единицу содержимое линий RET заносится в ОЗУ клавиатуры.*

В режиме сканирования клавиатуры при обнаружении нажатой клавиши из информации, поступившей с возвратных линий RET(0–7), в контроллере формируется 8-битное кодовое слово, где есть закодированные двоичным кодом позиции строки (3 бита) и столбца (3 бита), состояние клавиши

переключения регистров (SHIFT) и клавиши управления (CONTRL). Кодовое слово имеет формат, представленный на рис. 50.

Рис. 50. Формат кодового слова

В режиме сканирования сенсорной матрицы датчиков сигналы с возвратных линий RET запоминаются в строке памяти ОЗУ клавиатуры и датчиков (FIFO), соответствующей адресу сканирования. При этом потенциалы со входов SHIFT и CNTR не вводятся. Память FIFO хранит копию состояния датчиков. Этот режим удобен, когда информация от нескольких устройств вводится посредством опроса каждого устройства по линиям сканирования.

В режиме «стробируемый ввод» по перепаду сигнала на линии CO/STB из нуля в единицу содержимое линий RET заносится в ОЗУ клавиатуры и датчиков.

Во всех трёх случаях появление информации в ОЗУ приводит к выработке сигнала запроса прерывания INT.

INT *– сигнал запроса прерывания, поступающий на микропроцессор.*

Управление вводом – выводом осуществляется с помощью сигналов CS, A_0(INS/D), RD и WR. Они управляют считыванием и записью данных в различные внутренние регистры и буферы.

WR *– запись, подключается к соответствующему сигналу микропроцессорной системы при программировании микросхемы и передаче информации через двунаправленный канал данных* D_0-D_7.

RD *– чтение, подключается к соответствующему сигналу микропроцессорной системы при передаче информации через двунаправленный канал данных* D_0-D_7.

D_0-D_7 *– двунаправленная восьмиразрядная шина данных для связи контроллера с микропроцессором.*

CS *– выбор микросхемы, подключается к ША прямо или через дешифратор*

A_0 (INS/D) *– определяет характер информации, выдаваемой или запрашиваемой микропроцессором, т.е. это– вход «команда/данные». Если на этом входе* «1» *– выбирается регистр команды (запись) или регистр состояния (считывание); если на нём «0» – регистр данных. Вход подключается к младшему разряду ША.*

В таблице 23 приводится информация о совместном использованием выводов CS, RD, WR и A_0(INS/D).

Кроме указанных выводов в контроллере есть вход сигнала начальной установки (сброса) контроллера, имеющий разные варианты обозначений в справочниках.

SR (RES, CRL) *– сброс. Логическая единица на этом входе автоматически устанавливает в контроллере следующее:*
тип индикатора – 16-ть 8-разрядных символов, левый вход;
тип клавиатуры – сканируемая, запрет одновременного нажатия нескольких клавиш;
внутренний делитель тактовой частоты, автоматически устанавливается деление на 31.

Таблица 23

CS	RD	WR	INS/D A_0	Передача
0	1	0	0	ШД → в буфер регистра данных
0	1	0	1	ШД → в регистр управления
0	0	1	0	Буфер регистра данных → на ШД
0	0	1	1	Регистр состояний → на ШД
1	X	X	X	Высокоимпедансное состояние

Дисплейная часть. Индикаторная часть имеет своё ОЗУ 16х8 бит, которое также можно использовать как два ОЗУ 16х4 бита. Оно хранит воспроизводимые на индикаторе символы, загружается и опрашивается МП. Индикаторная часть работает в режиме сканирования индикаторов на светодиодах, жидких кристаллах или простых лампочках.

Регистры адреса ОЗУ отображения содержат:
адрес слова в ОЗУ, записываемого или опрашиваемого процессором;
адреса двух 4-битовых цифр, воспроизводимых на индикаторах в данный момент.

Адрес в регистр адреса записывается с помощью команды «Запись в индикаторное ОЗУ» (команда № 4). Команда может сделать так, чтобы адреса автоматически увеличивались или уменьшались после каждого считывания или записи.

Каждый адрес памяти ОЗУ отображения соответствует одному разряду индикатора, причем нулевой адрес относится к нулевому разряду.

Для индикации символов сначала выдается приказ записи в память индикатора, а затем выводятся в нее данные. После этого процессор освобождается от проблем, связанных с регенерацией, и контроллер начинает вывод на индикаторы. При этом ВВ79 по линиям S_0 –S_3 выдает адреса выбора разрядов, а по линиям DSPA(3-0) и DSPB(3-0) (в ряде справочников эти выводы обозначены как OUT A_0 - A_3 и OUT B_0-B_3) – выводимые символы.

В режиме ввода слева каждой позиции индикатора соответствует определенная строка в ОЗУ отображения. Нулевой адрес в ОЗУ определяет крайний левый символ. Ввод символов, начиная с нулевого адреса, вызывает построчное отображение информации слева направо.

Ввод справа со сдвигом обычно используется в калькуляторах. В этом случае ввод первого символа производится в крайнюю справа позицию дисплея. При вводе следующего символа все отображение сдвигается на одну позицию влево. Поскольку в этом режиме нет прямого соответствия между позицией отображаемого символа и адресом строки ОЗУ отображения, рекомендуется использовать последовательный ввод, начиная с нулевого адреса.

Информация на выходах DSPA (3-0) канала А соответствует разрядам $Д_7$-$Д_4$ канала данных, а на выходах DSPB (3-0) – разрядами $Д_3$-$Д_0$.

DSPA (3-0), DSPB (3-0) *– вывод на индикацию. Данные с этих линий используются для динамической индикации, они синхронизируются линиями* ST_0-ST_3.

Два 4-битовых порта (А и В) могут использоваться и быть погашены независимо друг от друга. Порты А и В могут рассматриваться как один 8-битовый порт. В последнем случае $DSPB_0$ соответствует разряду $Д_0$ на ОЗУ, т. е. это самый младший разряд индицируемого байта, $DSPA_3$ – самый старший, соответствует разряду D_7.

Следует обратить внимание, что индикатор управляется теми же сканирующими выходами S_0 - S_3, что и клавиатура. Поэтому в режиме с внутренней дешифрацией счетчика сканирования два младших бита дешифрируются внутри микросхемы и обеспечивают индикацию лишь первых 4-х знаков из содержащихся в индикаторном ОЗУ.

BD *– выход сигнала гашения отображения, имеющий длительность не менее 150 мкс. Используется для гашения индикации во время переключения знаков на индикаторе или при исполнении команды «гашение индикации».*

Состояние FIFO отображается в регистре состояния, его формат приведён на рис. 52. Биты 0, 1 и 2 показывают число находящихся в FIFO байт, а бит 3 говорит о заполнении FIFO. Биты 4 и 5 фиксируют антипереполнение (попытка считать из пустой памяти) и переполнение (попытка ввести символ в переполненную память). Бит 6 – показывает замыкание, когда контроллер находится в режиме матрицы датчиков. Бит 7 – говорит о доступности индикатора. Используя слово состояния контроллера, можно организовать работу с микропроцессором по опросу. Для этого командой IN считывается слово состояния и проверяется бит 3 в этом слове.

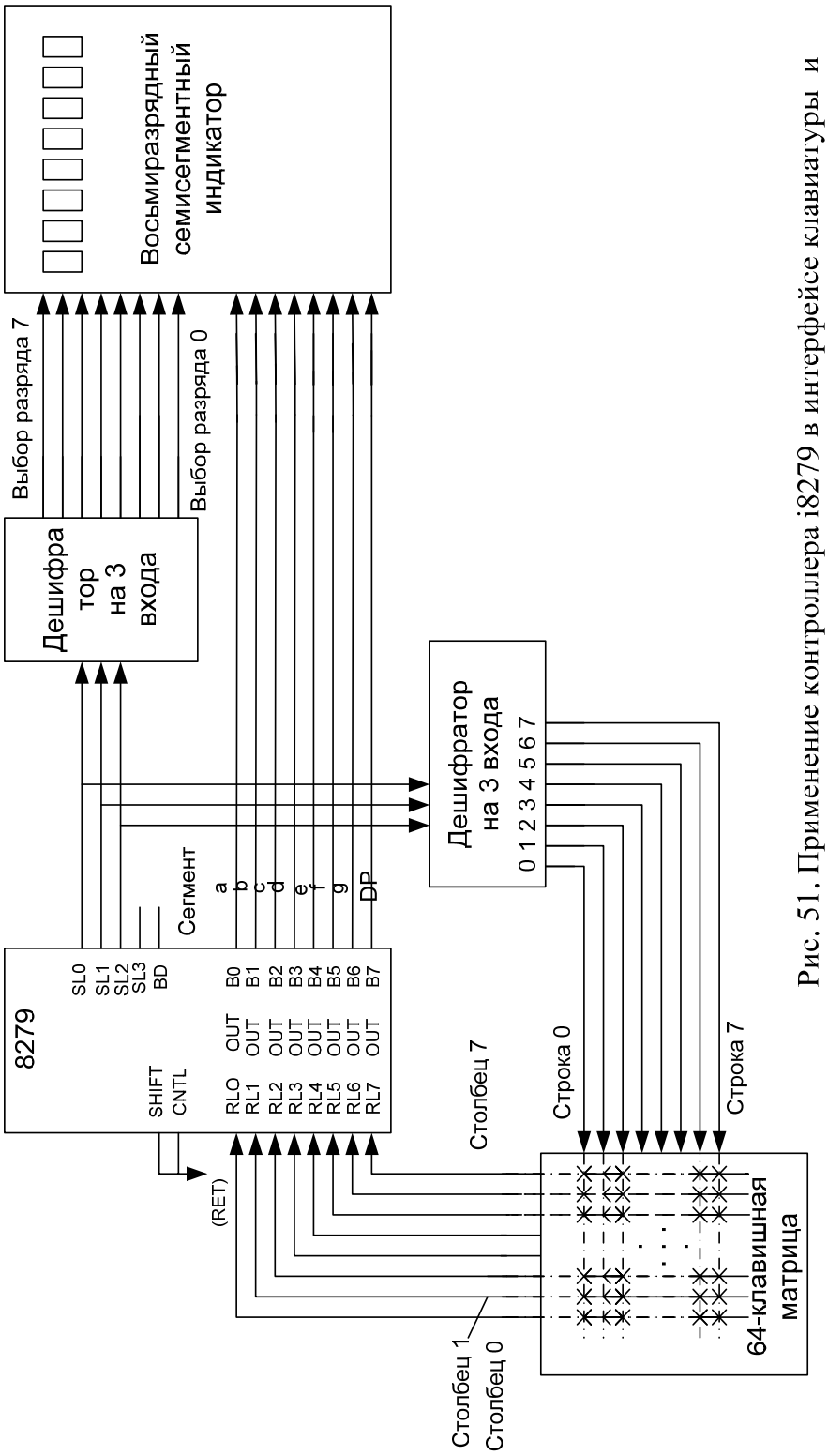


Рис. 51. Применение контроллера i8279 в интерфейсе клавиатуры и многоразрядного индикатора

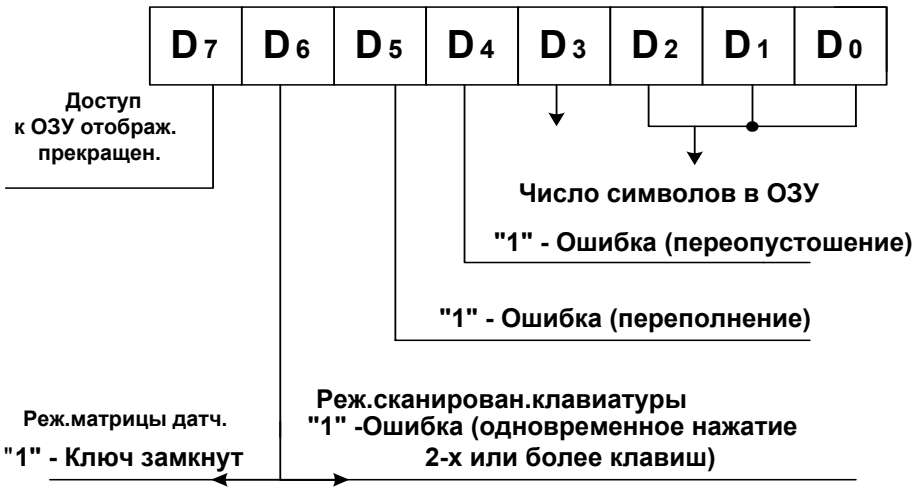


Рис.52. Слово состояния контроллера индикатора и клавиатуры

На рис. 51 показан один из вариантов подключения к контроллеру 64-клавишной клавиатуры и 8-разрядного семисегментного индикатора. Клавиатура и индикатор сканируются и регенерируются под управлением сигналов выбора S0-S2 / Выход S3 не подключен, так как индикатор имеет только восемь разрядов. Выходы обоих дешифраторов имеют активные сигналы низкого уровня. Один дешифратор выбирает строку клавиатуры, а другой разрешает один из 8-разрядных дисплеев.

Программирование контроллера. Схема управления контроллера представляет собой набор флажков и регистров, к которым обращаются командами.

Для программирования используются восемь команд, приведённых в конце раздела. Перед программированием в контроллер нужно подать сигнал сброса SR(RESET), этот аппаратный сигнал на ОЗУ клавиатуры и ОЗУ дисплея влияния не оказывает, память в исходное состояние устанавливается программно. Три старших бита в командах определяют ее тип (ее код), а смысл остальных зависит от кода.

Команда № 0 определяет режимы ввода с клавиатуры (D_0-D_2) и вывода на индикацию (D_3,D_4). В команде разряд D_0 определяет вид дешифрации: «0» – внешняя дешифрация (кодированное сканирование), «1» – внутренняя дешифрация (дешифрированное сканирование). Назначение всех остальных разрядов в команде достаточно понятно объяснено в таблице приложения.

Переход с одного режима на другой производится подачей команды без предварительного аппаратного или программного сброса.

Как уже ранее упоминалось, по аппаратному сбросу автоматически программируется кодирование клавиатуры с обнаружением двух нажатых клавиш и использование 16-разрядного дисплея со вводом слева.

Команда №1 – команда установки тактовой частоты. Она используется для согласования цикла синхронизации процессора с контроллером и обеспечивает требуемую скорость сканирования клавиатуры и дисплея. Поле D4-

D0 содержит двоичный код коэффициента деления P программируемого делителя частоты, который, как правило, выбирается так, чтобы частота внутренней синхронизации равнялась 100 кГц.

Команда № 2 – чтение памяти клавиатуры и датчиков. Она определяет, что будет вводиться байт из памяти FIFO. Если контроллер находится в режиме датчиков, команда показывает, какая строка считывается. Команда подается до ввода данных из FIFO. Биты D_0-D_2 и D_4 – нужны только для матрицы датчиков, а при сканировании всегда считывается байт, который был помещен в FIFO первым.

Команда № 3 – чтение индикаторного ОЗУ.

Команда № 4 – запись в память индикатора. Она показывает, что в буферный регистр данных будут помещаться данные в память индикатора. Команду необходимо выдать до того, как процессор будет выводить в контроллер индицируемые символы.

Команда № 5 – гашения и блокировки записи в индикаторное ОЗУ. Можно раздельно заблокировать запись 4-битовых цифр в порты A и B. Если пользователь хочет погасить индикатор, используются разряды D_0 и D_1 команды.

Команда № 6 – «Сброс» или «Очистка» – очищает содержимое индикаторного ОЗУ. Эта команда определяет значение символа, который потом будет использоваться как пробел. Разряд D_4 – разрешение очистки индикатора. Во время очистки (≈ 160 Мкс) в контроллер нельзя записывать данные. Все это время разряд D_7 слова состояния удерживается в значении «1». Когда доступ свободен, этот разряд автоматически сбрасывается в 0. По разряду D_0 производится полная очистка, при которой происходит сброс и перезапуск внутренних цепей синхронизации.

Команда № 7 – Команда сброса прерываний. Состояние разрядов D_0-D_3 – безразлично. В режиме сканирования сенсорной матрицы датчиков эта команда сбрасывает в «0» выход INT и разрешает дальнейшую запись информации в ОЗУ датчиков. В режиме независимого восприятия клавиш при D_4=1 включается специальный режим обнаружения ошибок. В этом режиме при нажатии нескольких клавиш блокируется запись символов во входное ОЗУ.

Рассмотрим пример программирования контроллера. Предположим, что он подключен к клавиатуре и многоразрядному индикатору в соответствии с рис. 51, имеет адреса 78h и 79h, выход запроса прерывания не используется, т.е. предполагается работа по опросу за счет чтения слова состояния контроллера. Вначале необходимо инициализировать контроллер, посылая в регистр управления приказ установки режима. Используем команду № 0. Запрограммируем контроллер в режим кодированного сканирования клавиатуры с 2-клавишной блокировкой и режим ввода слева 8-разрядного индикатора.

```
MOV AL, 0
OUT 79h, AL
```

Символы, сформированные нажатыми клавишами, можно считать через память FIFO.

Далее рассмотрим фрагмент программы, в котором применяется программный ввод-вывод (работа по опросу) для ввода 10 кодовых слов и запоминания их в массиве MASS (первый байт находится по старшему адресу).

```
        MOV SI, 0Ah
        MOV AL, 01000000B
        OUT 79h, AL
MET1: IN AL, 79h
        TEST AL,0Fh
        JZ MET1
        IN AL,78h
        MOV MASS[SI-1], AL
        DEC SI
        JNZ MET1
```

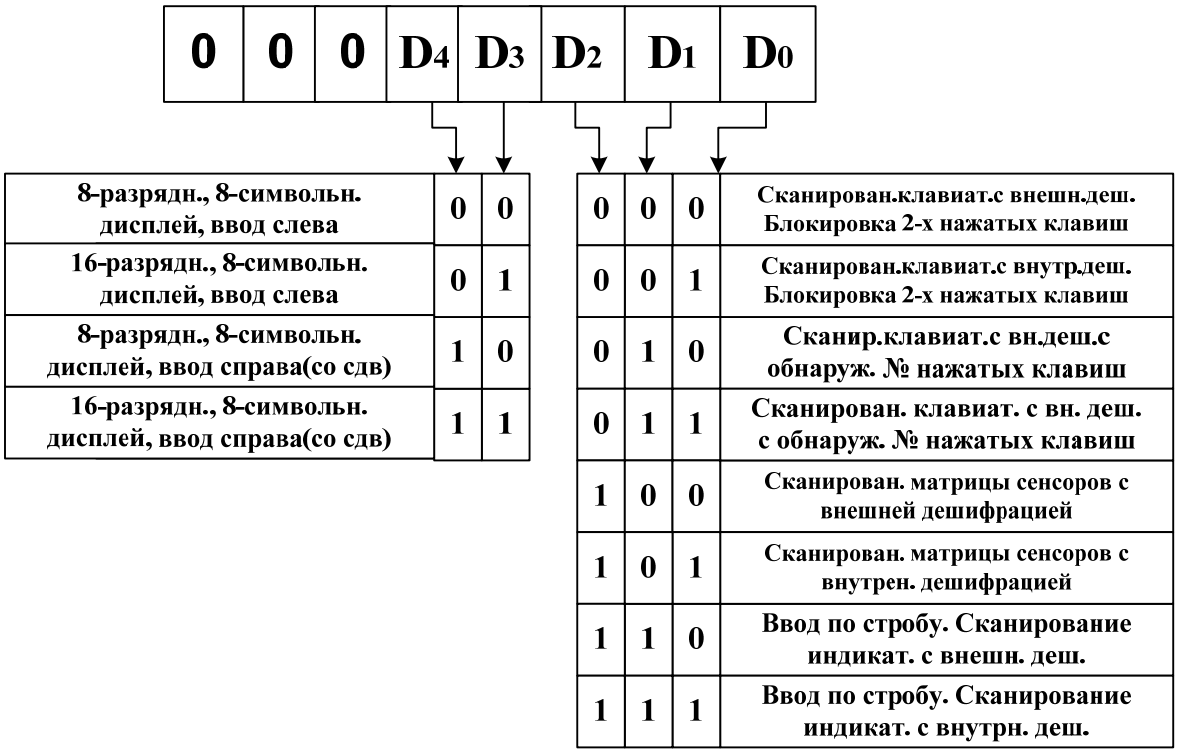
Первая команда инициализирует счётчик в SI, следующие две сообщают, что будет произведён ввод информации из FIFO. Следующие три команды, начиная с метки MET1, заставляют процессор ожидать готовности ввода, проверяя слово состояния контроллера, а находящиеся за ними две команды передают введённые данные в MASS. Последние две команды вызывают повторение последовательности до тех пор, пока не будут введены 10 символов.

Для индикации символов процессор сначала должен выдать приказ записи в память индикатора, а затем выводить в нее данные. Следующий фрагмент индицирует восемь цифр, которые хранятся в памяти начиная с SVET (младшая цифра хранится по меньшему адресу):

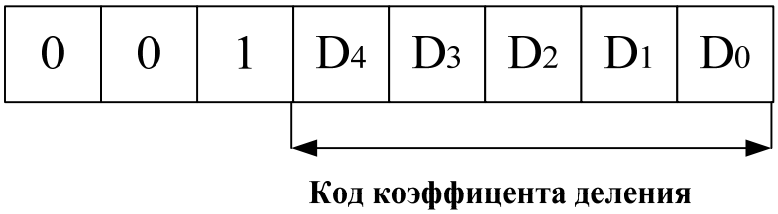
```
        MOV SI, 8
        MOV AL, 10010000B
        OUT 79h, AL
        AGAIN:  MOV AL, SVET[SI-1]
        OUT 78h, AL
        DEC SI
        JNZ AGAIN
```

Первая команда определяет счетчик в SI счетчик цифр, следующие две выводят приказ записи в память индикатора. Последние команды образуют цикл вывода в память индикатора.

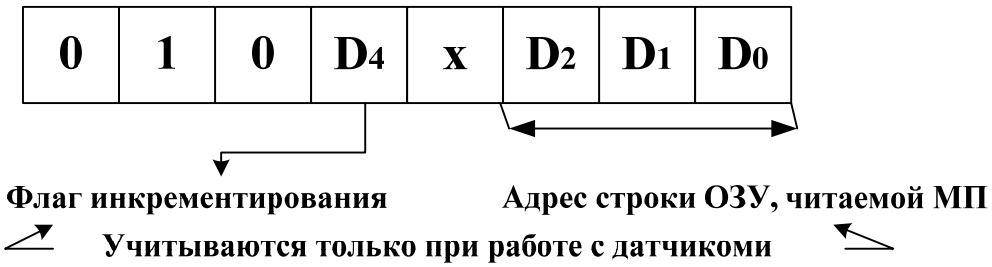
0. Команда установки режима клавиатуры – дисплея



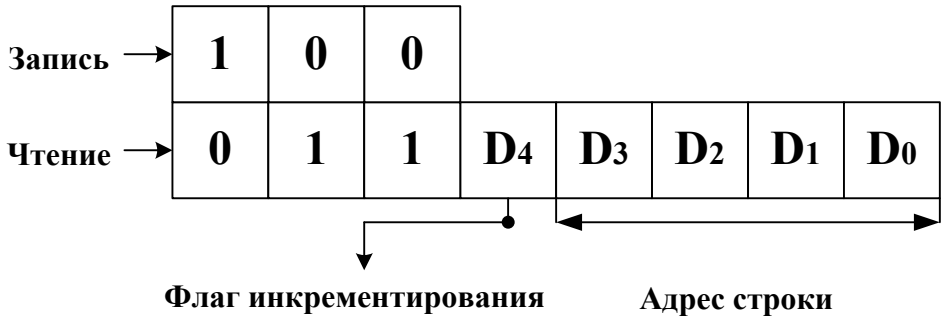
1. Команда установки тактовой частоты



2. Команда чтения ОЗУ клавиатуры и датчиков

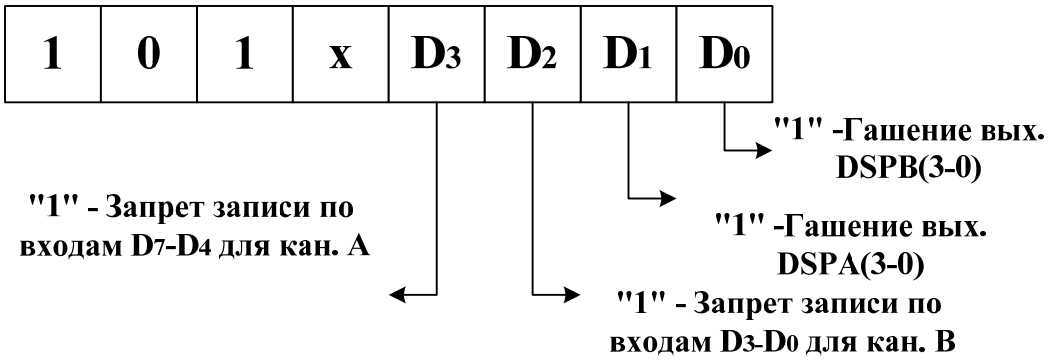


3–4. Команды чтения и записи индикаторного ОЗУ

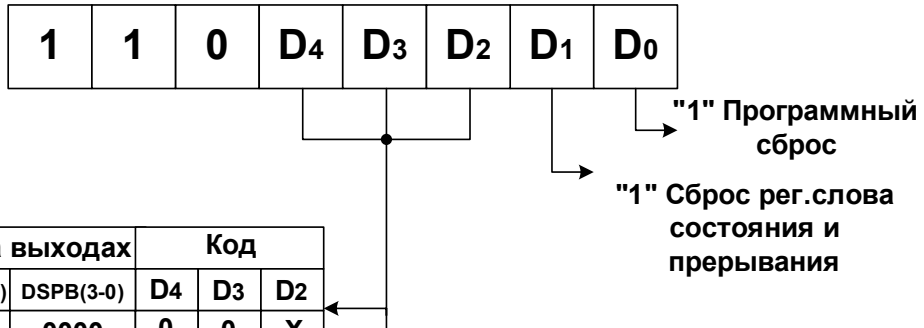


При D_4 =1 адрес увеличивается на 1

5. Команда гашения – запрета записи отображения



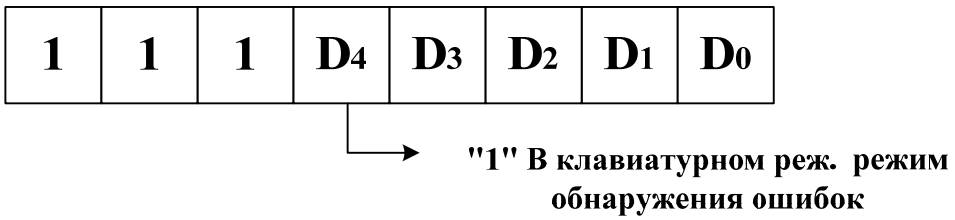
6. Команда «Сброс»



Код на выходах		Код		
DSPA(3-0)	DSPB(3-0)	D4	D3	D2
0000	0000	0	0	X
0010	0000	0	1	0
1111	1111	0	1	1

Код в ОЗУ отображ.		Код		
		D4	D3	D2
0000	0000	1	0	X
0010	0000	1	1	0
1111	1111	1	1	1

7. Команда «Сброс прерывания»



Лекция № 16

5.7. Контроллер прямого доступа к памяти I 8237 (К1810ВТ37)

Общие принципы организации ПДП. Режим ПДП является самым скоростным способом обмена, который реализуется с помощью специальных аппаратных средств — контроллеров ПДП без использования программного обеспечения. Для осуществления режима ПДП контроллер должен выполнить ряд последовательных операций:

1) принять запрос DREQ на ПДП от ВУ;
2) сформировать запрос HRQ на захват шин для ЦП,
3) принять сигнал HLDA, подтверждающий этот факт после того, как ЦП войдет в состояние захвата (ШД, ША, ШУ в z-состоянии);
4) сформировать сигнал DACK, сообщающий ВУ о начале выполнения циклов ПДП;
5) сформировать на ША адрес ячейки памяти, предназначенный для обмена;
6) выработать сигналы MR, IOW и MW, IOR, обеспечивающие управление обменом;
7) по окончании ПДП либо повторить цикл ПДП, изменив адрес, либо прекратить ПДП, сняв запросы на ПДП.

Циклы ПДП выполняются с последовательно расположенными ячейками памяти, поэтому контроллер ПДП должен иметь счетчик адреса ОЗУ. Число циклов ПДП определяется специальным счетчиком. Управление обменом осуществляется специальной логической схемой, формирующей в зависимости от типа обмена пары управляющих сигналов: MR, IOW (циклы чтения), MW, IOR (циклы записи).

Из изложенного следует, что контроллер ПДП по запросу должен взять на себя управление системными шинами и выполнять совмещенные циклы чтения / вывода или записи/ввода до тех пор, пока содержимое счетчика циклов ПДП не будет равно нулю. На рис. 53 показана структурная схема МПС с контроллером ПДП.

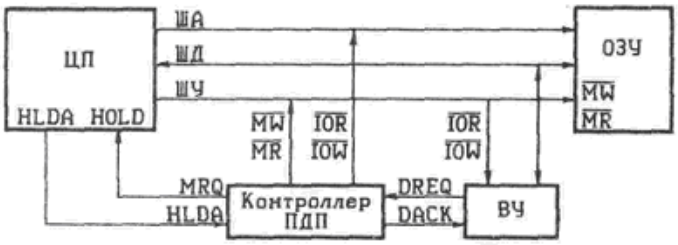


Рис. 53. Структурная схема МПС с контроллером ПДП

Контроллер ПДП К1810ВТ37 используется в составе МПС, выполненных на базе МПК К580, К1810, К1821, для реализации прямого доступа к памяти по четырем независимым каналам с положительным или отрицательным приращением адреса со скоростью до 1,6 Мбайт/с. КПДП позволяет реализовать передача память — память, имеет широкие возможности программного управления и каскадирования. Каждый канал может выполнять до 64К циклов ПДП и имеет возможность автоматической инициализации, т. е. повторения циклов ПДП с теми же параметрами. Структурная схема КПДП приведена на рис. 54.

Назначение выводов КПДП.

CLK – вход для подключения тактового генератора FcLк=3 МГц.

CS – выбор кристалла. CS=0 разрешает работу КПДП.

RESET – сброс Сигнал высокого уровня переводит КПДП в исходное состояние, устанавливая в нуль регистры команд, условий, временного хранения, а также устанавливая в единицу все разряды маски.

READY– готовность. Входной сигнал, используемый для синхронизации работы КПДП с медленнодействующими устройствами.

HLDA – подтверждение захвата. Входной сигнал, используемый ЦП для сообщения КПДП о возможности выполнения циклов ПДП.

DREQ3 – DREQ0 – входы запросов на ПДП от внешних устройств. По лярность запросов задается программно. Сигналы на этих входах должны удерживаться до прихода сигнала DACK. В исходном состоянии приоритет запросов естественный, DREQO имеет наивысший приоритет.

DB7 – DB0 – двунаправленная шина данных с буфером, имеющим z-состояние. В циклах ПДП на эти линии выдается восемь старших разрядов адресного кода, которые необходимо «защелкнуть» на внешнем регистре сигналом ADSTB. В режиме работы с ЦП по этим линиям осуществляется прием/передача данных.

I0R – чтение; как вход используется ЦП для чтения содержимого внутренних регистров КПДП; как выход в режиме ПДП разрешает выдачу данных из внешних устройств.

IOW – запись; как вход используется ЦП для загрузки данных в регистры КПДП; как выход в режиме ПДП разрешает запись данных в регистры внешних устройств.

EOP – окончание процесса. Вход / выход, используемый для указания окончания процесса передачи данных в режиме ПДП. Подавая на этот вход сигнал низкого уровня, можно прекратить передачу данных. После завершения передачи данных по одному из каналов на выходе устанавливается сигнал EOP = 0. По этому сигналу (внешнему или внутреннему) снимается запрос и обслуживание прекращается. Если установлен режим автоинициализации, то происходит загрузка рабочих регистров данного канала содержимым базовых регистров, а разряды регистра маски не меняются. В режимах без автоинициализации разряды маски и разряд ТС в слове-состоянии устанавливаются в соответствии с состоянием обслуженного канала. При передаче память – память вывод EOP ориентирован на выход, и по окончании счёта на этом выходе формируется сигнал. Если вывод EOP не используется, то он должен быть подключен через резистор к шине питания (+5 В) для предотвращения формирования ложных сигналов окончания процесса.

A3 – A0 – адресные входы/выходы. Используются как входные в режиме работы с ЦП и для адресации к каналам и регистрам каналов КПДП. В режиме ПДП являются выходами, по которым передаются четыре младших разряда адреса ОЗУ.

A7 – A4 – адресные выходы, на которые в режиме ПДП передаются соответствующие разряды адреса ОЗУ. В режиме работы с ЦП переходят в z-состояние.

HRQ – выход запроса захвата на управление системной шиной. Запрос на ПДП ЦП.

DACK3 – DACK0 – подтверждение ПДП. Выходные линии, на которые выдаются сообщения для ВУ о возможности выполнения циклов ПДП. Полярность сигнала задается программно. После сигнала RESET на выходах DACK устанавливается нуль.

AEN – разрешение адреса. AEN = 1 устанавливается на время выдачи восьми старших разрядов адреса ОЗУ на линии DB7 – DB0.

ADSTB – строб адреса. Выход, на котором формируется импульс (строб), осуществляющий запись старших разрядов (A15 – A8) адреса ОЗУ с шин DB7 – DB0 во внешний буферный регистр.

MEMR – чтение из памяти. Выход, используемый в режиме ПДП для управления операцией чтения из памяти.

MEMW – запись в память. Выход, используемый в режиме ПДП для управления операцией записи в память.

Ucc – шина питания (+5 В).

GND – общий.

Структура КПДП (рис. 54). Контроллер включает четыре канала, каждый из которых состоит из четырех 16-разрядных регистров.

Регистр текущего адреса **CAR** хранит текущий адрес ячейки памяти при выполнении цикла ПДП. После выполнения цикла ПДП содержимое этого регистра увеличивается или уменьшается на единицу. Оно может быть прочитано или загружено с помощью двух команд ввода – вывода. Содержи-

мое CAR может быть обновлено по сигналу EOP, если запрограммирован режим автоинициалиции.

Регистр циклов ПДП **CWR** хранит число слов, предназначенных для передачи. При загрузке этого регистра необходимо помнить, что загружаемая константа должна быть на единицу больше числа слов, необходимых для передачи. При выполнении циклов ПДП регистр работает в режиме вычитающего счетчика. Разряд TC регистра состояния устанавливается в единицу при переходе из нулевого состояния в состояние FFFFH. Чтение и запись содержимого регистра осуществляются двумя последовательно выполняемыми командами ввода – вывода. Содержимое CWR может быть обновлено при автоинициализации по сигналу EOP либо в регистре сохраняется значение FFFFH.

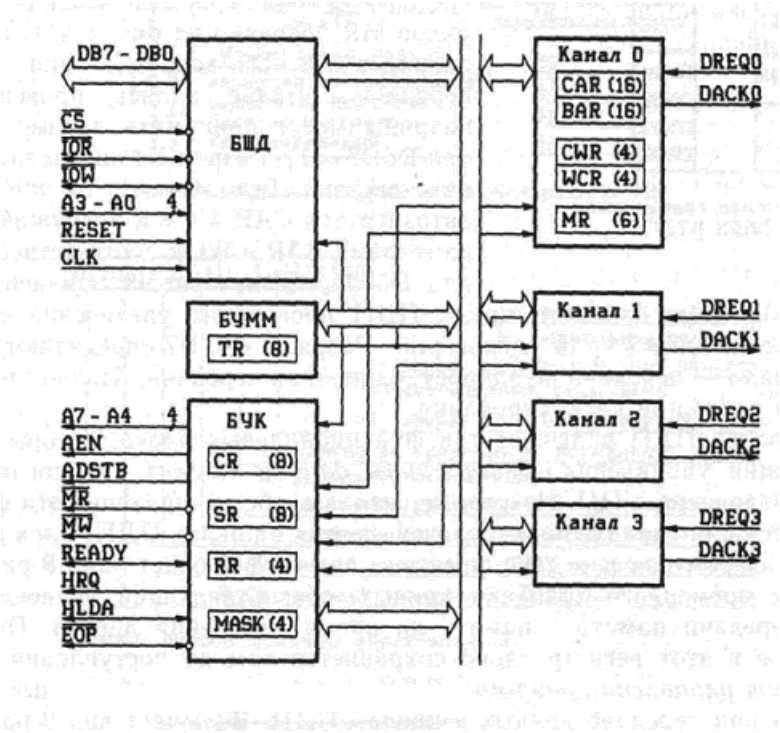


Рис. 54. Структурная схема КПДП

Регистр хранения базового адреса BAR и регистр хранения базового числа циклов ПДП **WCR** хранят базовые значения адреса и числа циклов ПДП, участвуют в автоинициализации. При начальной загрузке контроллера ПДП исходными параметрами происходит одновременная запись в регистры CAR, BAR, CWR и WCR. В процессе выполнения циклов ПДП содержимое BAR и WCR не изменяется. Прочитать состояние этих регистров невозможно.

Кроме того, каждый канал имеет 6-разрядный *регистр режима* **MR,** определяющий режим его работы. При загрузке этого регистра в младших разрядах D1, D0 указывается код номера канала. Назначение разрядов MR показано на рис. 55. С помощью разрядов D2, D3 задается один из типов передачи – чтение, запись, проверка. Эти разряды могут принимать любые значения при D6D7 =11. Разряд D4 определяет режим автозагрузки. Если D4 = 1,

то при условии автозагрузки CAR и CWR загружаются параметрами BAR и WCR соответственно. Разряд D5 определяет режим изменения CAR.

Если D5 = 0, после каждого цикла ПДП происходит увеличение содержимого CAR; если D5 = 1 – то уменьшение Разряды D6, D7 определяют режимы работы канала – передача по запросу, одиночная передача, блочная передача, контроллер в режиме каскадирования.

Контроллер ПДП включает три функциональных блока, которые выполняют функции управления. *Буфер шины данных* служит для согласования работы контроллера с ЦП. Некоторые сигналы, обеспечивающие эти функции, используются для управления передачей данных в циклах ПДП. *Блок управления контроллером* при передаче память – память включает один 8-разрядный регистр TR временного хранения данных, обеспечивающий хранение байта в цикле передачи память – память на время изменения адреса. Последнее загруженное в этот регистр слово сохраняется там до поступления сигнала RESET. *Блок управления режимом ПДП* вырабатывает необходимые сигналы управления при передаче данных в циклах ПДП. Включает два 8-разрядных и два 4-разрядных регистра.

Регистр команд **CR** определяет основные параметры работы канала. Загрузка CR осуществляется командой вывода от ЦП, а сброс – по сигналу RESET или команде общего сброса Назначение разрядов регистра показано на рис. 56. Разряды DO, D1 используются для задания режимов работы каналов 0 и 1 в режиме память – память. Разряд D2 инициализирует контроллер для выполнения ПДП, разряд D3 определяет режим выполнения циклов ПДП. Если D3 = 1, циклы ПДП выполняются с пропуском одного такта при изменении адреса в пределах младшего байта. Разряд D4 устанавливает режим приоритетов. Если D4 = 1, запросу обслуженного канала присваивается наинизший приоритет – это режим вращения приоритета. Разряд D5 устанавливает режим удлиненного цикла записи. Если D5 = 1, сигналы IOW и MEMW вырабатываются с двойной длительностью. Разрядами D6, D7 программируются уровни запросов на ПДП (DREQ) и сигналов подтверждения ПДП (DACK).

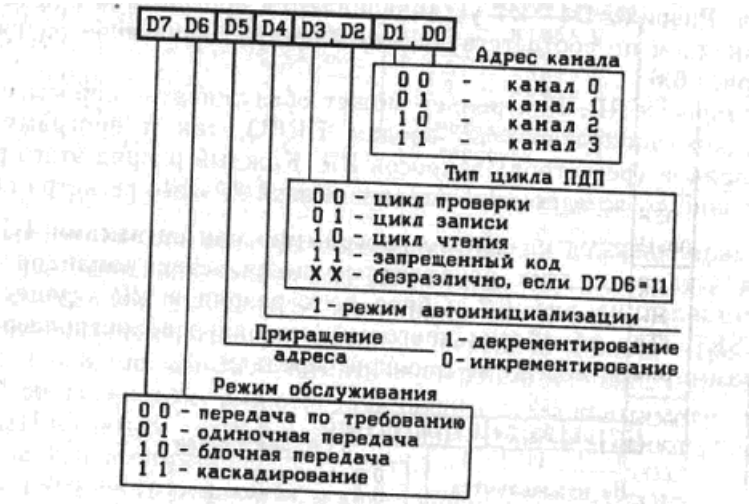


Рис. 55. Формат команды установки режима MR

Регистр условий **SR,** разряды D3-D0 которого устанавливаются аппаратно при возникновении сигнала TC, т. е. после окончания циклов ПДП или по внешнему сигналу EOP. Эти разряды сбрасываются (устанавливаются в нуль) сигналом RESET, а также после выполнения команды чтения содержимого этого регистра. Разряды D4–D7 устанавливаются программно при необходимости обслуживания по соответствующему каналу. Назначение разрядов SR показано на рис. 57.

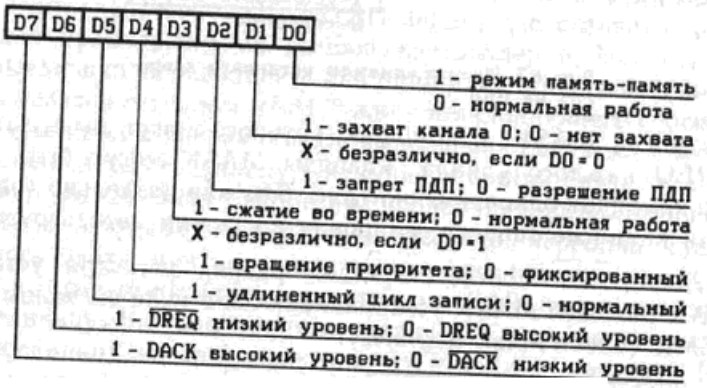


Рис. 56. Формат команды управления CR

Регистр запросов **RR.** Контроллер может обслуживать запросы на ПДП, формируемые как аппаратно – по входам DREQ, так и программно – по состоянию разрядов (регистров) запросов RR. Каждый разряд этого регистра соответствует запросу по одному из каналов. Разряды этого регистра не маскируются и устанавливаются раздельно программно или сигналами TC и EOP. Программная установка этих разрядов осуществляется командой, формат которой представлен на рис. 58. Сброс всех разрядов RR осуществляется

сигналом RESET. Для обработки программного запроса контроллер должен быть запрограммирован в режиме блочной передачи.

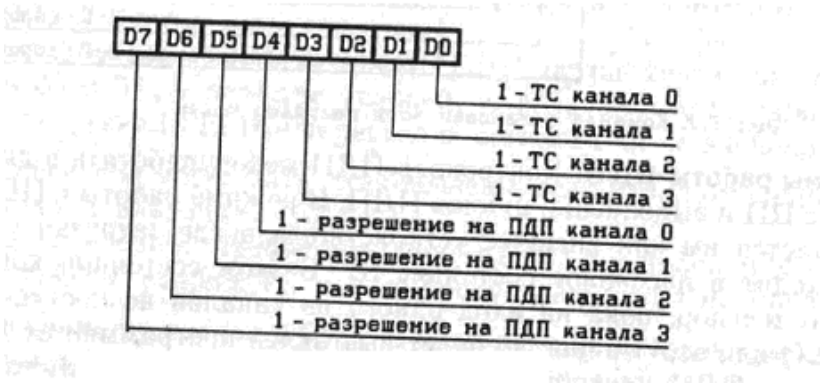


Рис. 57. Формат слова-состояния

Регистр запросов **RR.** Контроллер может обслуживать запросы на ПДП, формируемые как аппаратно – по входам DREQ, так и программно – по состоянию разрядов (регистров) запросов RR. Каждый разряд этого регистра соответствует запросу по одному из каналов. Разряды этого регистра не маскируются и устанавливаются раздельно программно или сигналами ТС и EOP. Программная установка этих разрядов осуществляется командой, формат которой представлен на рис. 58. Сброс всех разрядов RR осуществляется сигналом RESET. Для обработки программного запроса контроллер должен быть запрограммирован в режиме блочной передачи.

Рис. 58. Формат команды установки запросов

Регистр маски **MASK,** с помощью которого могут быть замаскированы сигналы DREQ каждого канала. Разряды MASK могут быть установлены специальной командой одновременно (рис. 59) или раздельно (рис. 60). Кроме того, если канал не запрограммирован на режим автозагрузки, после появления сигнала EOP соответствующий разряд регистра устанавливается в единицу. Все разряды MASK устанавливаются в нули сигналом RESET либо командой CMR (Clear Mask Register).

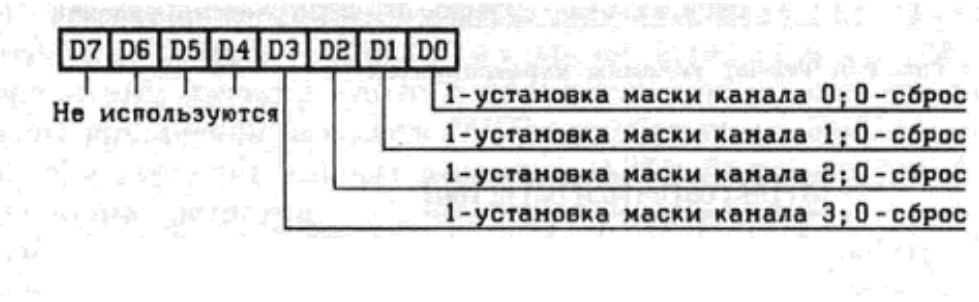


Рис. 59. Команда установки все разрядов маски

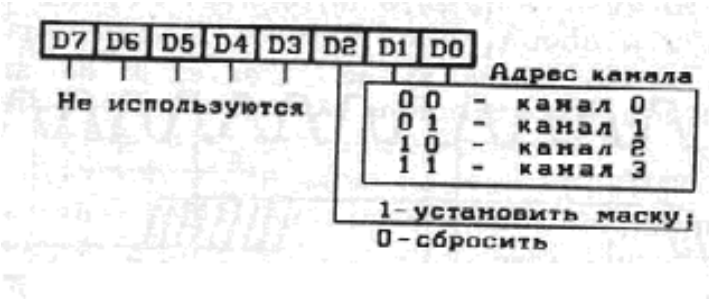


Рис. 60. Команда установки разряда маски

Лекция № 17

Режимы работы ПДП. Контроллер ПДП может работать в двух основных режимах: с ЦП и выполнения циклов ПДП. В режиме работы с ЦП контроллер воспринимается им как внешнее устройство, а после загрузки управляющих слов переходит в пассивное состояние S1. В этом состоянии контроллер находится до тех пор, пока на вход одного из каналов не поступит запрос на ПДП DREQ или этот запрос не будет выставлен программно от ЦП. Обнаружив запрос на ПДП, контроллер переходит в состояние SO и выставляет сигнал
запроса на захват системной шины HRQ, ожидая от ЦП сигнала подтверждения захвата HLDA. При получении сигнала HLDA контроллер начинает выполнять циклы ПДП.

Различают четыре рабочих состояния при выполнении этих циклов: S1– S4. Если при выполнении циклов ПДП на вход READY, подать нуль, контролёр между тактами S2/S3 и S4 выполняет такты ожидания SW. Состояние SW характеризуется активностью линий передачи данных. При передаче информации в режиме память – память необходимо выполнить два полных цикла чтения и записи, поэтому для передачи одного слова контроллер выполняет два цикла ПДП по четыре такта в каждом: S11 – S14 для чтения из памяти и S21 – S24 для записи в память.

Временная диаграмма работы контроллера в циклах ПДП представлена на рис. 61. В пассивном состоянии происходит опрос входов запросов на

ПДП и возможно взаимодействие с ЦП с помощью обычных команд ввода – вывода. Так как взаимодействие с ЦП КПДП чаще осуществляет словом из двух байтов, то для правильного их выбора контроллер использует внутренний триггер, указывающий на операцию с младшим или старшим байтом слова. Этот триггер сбрасывается сигналом RESET или командой общего сброса, указывая на операцию с младшим байтом. После выполнения операции с младшим байтом он устанавливается в единицу, указывая старший байт.

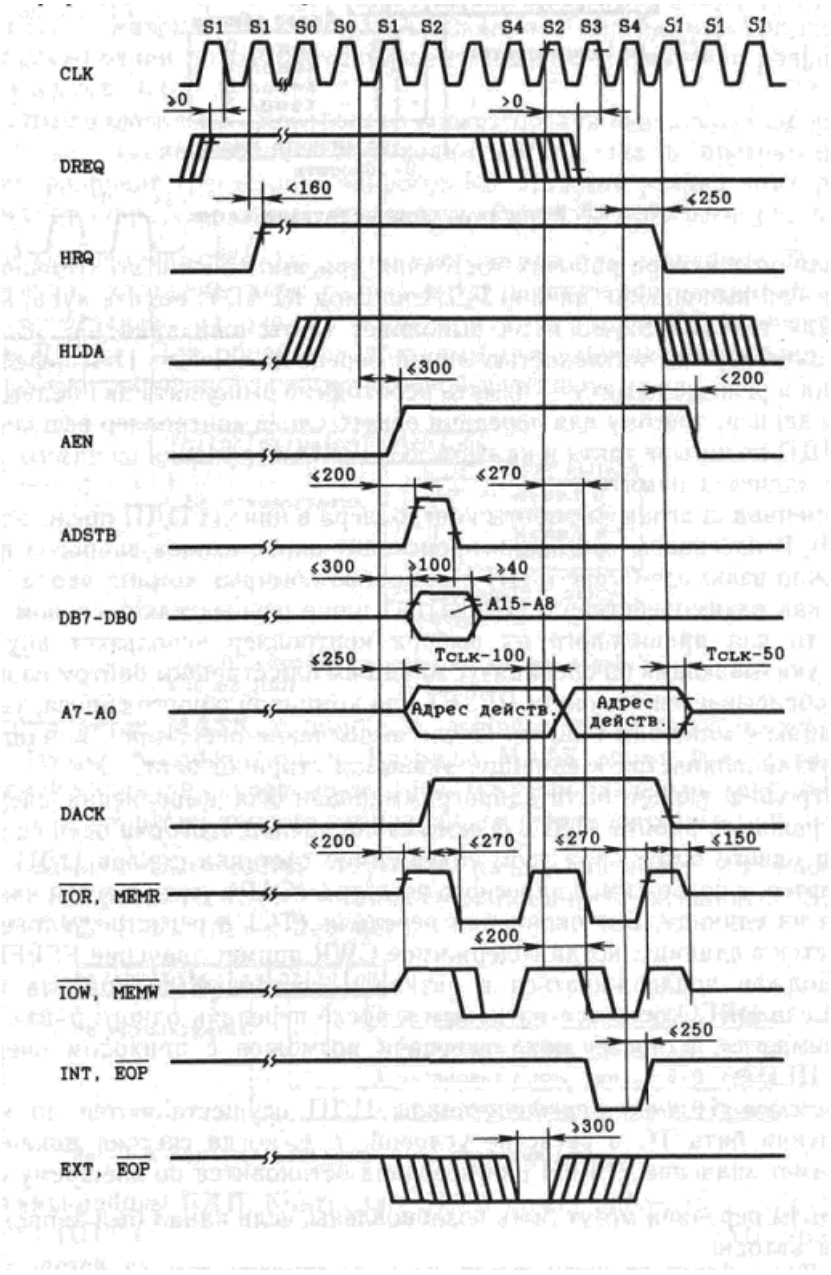


Рис. 61. Временные диаграммы работы КПДП

Контроллер может быть запрограммирован для выполнения следующих четырех режимов работы ПДП. *В режиме одиночной передачи* осуществляется передача одного байта, при этом содержимое счетчика циклов ПДП

(CWR) уменьшается, а содержимое адресного регистра (CAR) уменьшается или увеличивается на единицу. Бит окончания передачи (TC) в регистре условий устанавливается в единицу, когда содержимое CWR примет значение FFFFH. Вход DREQ должен поддерживаться в активном состоянии до прихода сигнала DACK. Если DREQ остается активным и после передачи одного байта, сигнал HRQ снимается, а Новый цикл передачи возможен с приходом очередного сигнала HLDA.

В режиме блочной передачи циклы ПДП осуществляются до момента установления бита TC в регистре условий, т. е. когда счетчик циклов ПДП CWR примет значение FFFFH или передача остановится по внешнему сигналу EOP. Циклы передачи могут быть возобновлены, если канал был запрограммирован на автоинициализацию.

В режиме передачи по требованию циклы ПДП продолжаются до тех пор, пока не установится разряд TC в регистре условий либо не придет сигнал EOP, либо не снимется сигнал DREQ. В этом режиме передача может осуществляться, пока внешнее устройство не закончит передачу информации. Автоинициализацию в этом режиме можно осуществлять после окончания передачи сигналом EOP, внешним или вырабатываемым по признаку TC. *Режим передачи память – память* позволяет осуществлять перемещение блоков информации в поле оперативной памяти. Для реализации этого режима используются параметры каналов 0 и 1. Передача инициализируется программно установкой DREQ в канале 0. После прихода сигнала HLDA = 1 контроллер за четыре такта считывает данные из ячейки памяти с адресом из регистра CAR канала 0 и записывает их в регистр временного хранения TR, затем за четыре такта записывает эти данные в ячейку памяти с адресом из CAR канала 1. Когда содержимое регистра циклов ПДП CWR примет значение FFFFH, установится разряд TC и передача закончится. Канал 0 может быть запрограммирован на передачу информации без изменения адреса, что позволяет заполнить ячейки блока ОЗУ константой. В *этом* режиме внешний сигнал EOP, воспринимаемый контроллером, используется при поиске нужных кодов в поле адресов ОЗУ. Временная диаграмма работы контроллера в этом режиме показана на рис. 62. Режим передачи память – память может быть инициализирован сигналом AEN без использования сигнала DACK.

В случае каскадирования выводы HRQ и HLDA дополнительной схемы подключаются к выводам DREQ и DACK основной схемы (рис. 63). В этом случае сигналы запросов на ПДП проходят через схемы приоритетов БИС КПДП более высокого уровня. При этом никакие другие сигналы основной схемы в формировании циклов ПДП не участвуют. Другие контроллеры могут быть подключены как к свободным входам запросов основной схемы, так и к входам подчиненной схемы.

Типы передачи ПДП. Во всех режимах ПДП возможны три основных типа передачи. *Запись данных* – осуществляется передача данных от внешнего устройства к ОЗУ. Контроллер в этом случае активизирует сигналы MEMW и IOR. *Чтение данных* – осуществляется передача данных от ОЗУ к

внешнему устройству, активизируются сигналы MEMR и IOW. В случае *проверки или псевдопередачи* контроллер выполняет действия такие же, как в цикле чтения / записи, но сигналы управления не вырабатываются. В этом случае сигнал READY не воспринимается. Кроме того, контроллер, может быть запрограммирован для выполнения дополнительных функций.

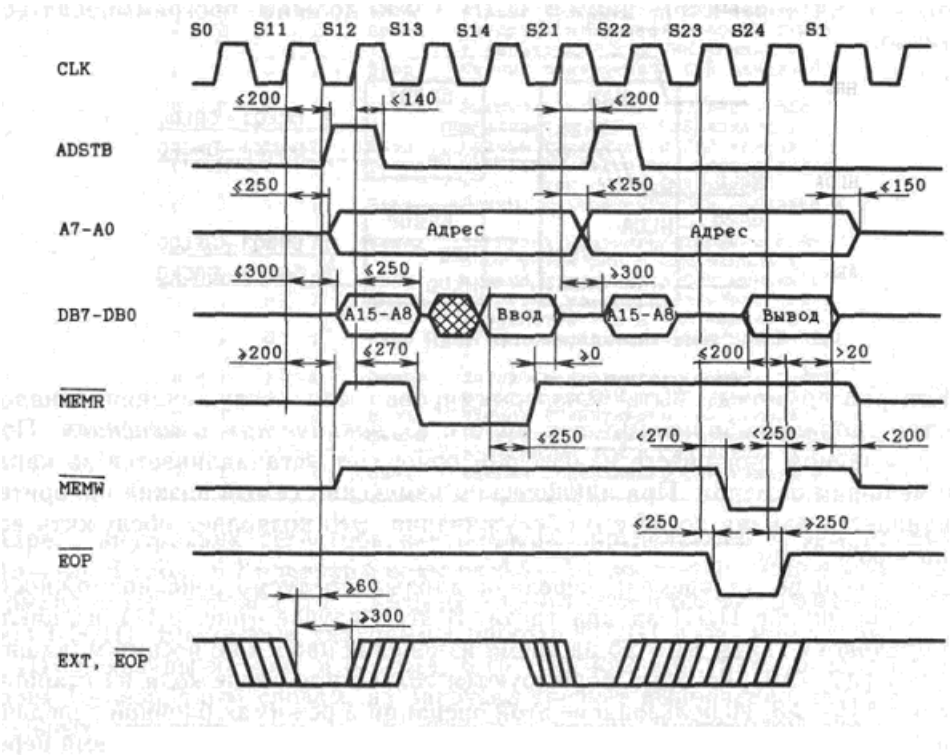


Рис. 62. Временная диаграмма работы КПДП в циклах память-память

Автоинициализация осуществляется, если установлен соответствующий разряд в регистре условий, и по сигналу EOP. При автоинициализации содержимое базовых регистров BAR и WCR загружается в регистры текущих значений CAR и CWR. Разряды маски при этом не меняются. После автоинициализации контроллер готов к работе и возобновляет действие с приходом очередного сигнала DREQ. Для автоинициализации обоих каналов в режиме память – память регистры циклов ПДП CWR должны программироваться идентично.

Контроллер может быть запрограммирован для обслуживания каналов с жестко *заданными приоритетами* либо с их *циклическим изменением.* При жестко заданном приоритете наивысший приоритет устанавливается за каналам с меньшим номером. При циклическом изменении самый низкий приоритет присваивается каналу после его обслуживания. Это позволяет обслужить все каналы поочередно.

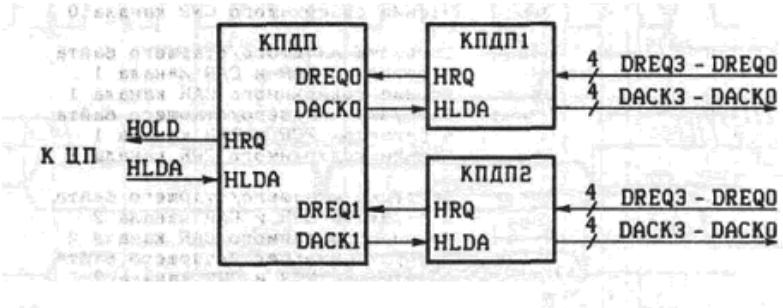


Рис. 63. Схема каскадирования КПДП

Для уменьшения времени передачи данных предусмотрена возможность выполнения циклов ПДП за два такта. В этом случае из цикла ПДП удаляются такты S1 и S3 на время изменения адреса по восьми младшим разрядам (A7–AO), которые формируются только при смене кода на старших разрядах A15–A8. Использование этой операции в режимах блочной передачи и передачи по требованию позволяет значительно сократить общее время передачи данных. Такая операция называется *сжатием во времени.*

Таблица 24

A3	A2	A1	A0	Команда	Операция
1	0	0	0	Ввод	Чтение регистра состояния
1	0	0	0	Вывод	Запись в регистр команд управления
1	0	0	1	То же	Запись в регистр запросов
1	0	1	0	>>	Установка всех разрядов маски
1	0	1	1	>>	Запись в регистр режима
1	1	0	0	>>	Установка режима ввода младшего байта
1	1	0	1	Ввод	Чтение регистра временного хранения
1	1	0	1	Вывод	Общий сброс
1	1	1	0	То же	Сброс всех разрядов маски
1	1	1	1	>>	Установка разряда маски

Программирование контроллера. Программирование контроллера осуществляется от ЦП командами ввода – вывода и возможно только в пассивном состоянии или при наличии на входе HLDA напряжения низкого уровня, если даже присутствует сигнал HRQ. Начальную инициализацию контроллера необходимо осуществить сразу же после включения напряжения питания по всем каналам, если даже они не используются, загружая команды и константы.

Таблица 25

A3	A2	A1	A0	Команда	Операция
0	0	0	0	Вывод	Загрузка младшего/старшего байта в регистры BAR и CAR канала 0
0	0	0	0	Ввод	Чтение содержимого CAR канала 0
0	0	0	1	Вывод	Загрузка младшего/старшего байта в регистры WCR и CWR канала 0
0	0	0	1	Ввод	Чтение содержимого CWR канала 0
0	0	1	0	Вывод	Загрузка младшего/старшего байта в регистры BAR и CAR канала 1
0	0	1	0	Ввод	Чтение содержимого CAR канала 1
0	0	1	1	Вывод	Загрузка младшего/старшего байта в регистры WCR и CWR канала 1
0	0	1	1	Ввод	Чтение содержимого CWR канала 1
0	1	0	0	Вывод	Загрузка младшего/старшего байта в регистры BAR и CAR канала 2
0	1	0	0	Ввод	Чтение содержимого CAR канала 2
0	1	0	1	Вывод	Загрузка младшего/старшего байта в регистры WCR и CWR канала 2
0	1	0	1	Ввод	Чтение содержимого CWR канала 2
0	1	1	0	Вывод	Загрузка младшего/старшего байта в регистры BAR и CAR канала 3
0	1	1	0	Ввод	Чтение содержимого CAR канала 3
0	1	1	1	Вывод	Загрузка младшего/старшего байта в регистры WCR и CWR канала 3
0	1	1	1	Ввод	Чтение содержимого CWR канала 3

Адреса внутренних регистров контроллера определяются кодом на выводах A3 – A0. В табл. 24 показаны коды на A3 – A0, соответствующие выполняемым командам ЦП, а в табл. 25 – коды на A3 –A0, соответствующие адресам регистров КПДП. Временные диаграммы работы КПДП в режиме взаимодействия с ЦП в цикле записи показаны на рис. 65. Так как константы всегда представлены 16-разрядным словом, их загрузка требует выполнения двух последовательных операций вывода с одинаковым кодом. Внутренний триггер управляет последовательностью ввода. Сначала загружается младший байт, затем старший.

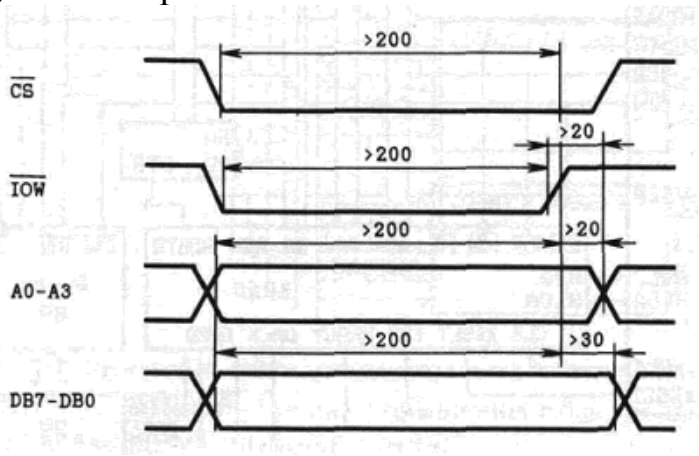


Рис. 65. Временная диаграмма работы КПДП в цикле записи

*Подключение контроллера к системной шине***.** Для уменьшения числа выводов на корпус БИС восемь старших разрядов адреса выдаются в такте S1

на выводы шины данных и должны быть «защелкнуты» на внешнем регистре БР, выходы которого подключаются к старшим разрядам шины адреса. Запись во внешний регистр осуществляется сигналом ADSTB. Линия AEN используется для того, чтобы разряды адреса оставались действующими на ША в течение трех тактовых периодов цикла ПДП. Линии А7 – АО подключаются непосредственно к ША. Сигналы MEMR, MEMW, IOR, IOW управляют в циклах ПДП соответственно ОЗУ и буфером внешнего устройства.

5.8. Арбитраж шин в многопроцессорных системах

По мере уменьшения отношения стоимость/производительность однокристальных процессоров становится более экономичным применять несколько процессоров вместо одного сложного многокристального. Кроме улучшения экономических показателей системы, мультипроцессорная конфигурация обеспечивает несколько положительных качеств, которых нет в однопроцессорной системе. Во-первых, несколько процессоров лучше приспосабливаются под требования конкретного применения, исключая расходы на ненужные возможности централизованной системы. Более того, модульность мультипроцессорной системы позволяет по мере необходимости вводить дополнительные процессоры. Во-вторых, в мультипроцессорной системе задачи разделяются между модулями, и при возникновении отказа проще найти и заменить неисправный процессор, чем отыскать неисправный элемент. В мультипроцессорной среде необходимо управлять обращениями к разделённому ресурсу, в ней невозможно непосредственно подключить два или более микропроцессора 8086 или 8088. В такой системе каждый микропроцессор имеет свою логику управления шиной, а арбитраж шины достигается путём расширения этой логики и введения общей для всех ведущих модулей внешней логики. Более подробно этот материал будет рассмотрен в последующих главах.

Лекция № 18

6. РЕАЛИЗАЦИЯ МУЛЬТИПРОЦЕССОРНЫХ СИСТЕМ

6.1. Организация мультипроцессорных систем в сильно связанных и слабо связанных конфигурациях

6.1.1. Общие сведения

Мультипроцессорная система содержит два или более процессоров, поскольку часто экономичнее применение нескольких процессоров вместо одного сложного. Можно отметить ещё ряд положительных факторов:

несколько процессоров лучше приспосабливаются под требования конкретного применения, исключаются расходы на ненужные возможности централизованной системы;

в мультипроцессорной системе задачи разделяются между модулями. При возникновении отказа проще и дешевле найти и заменить неисправный процессор, чем отыскивать и заменять отказавший элемент в сложной процессорной системе.

При проектировании мультипроцессорных систем приходится решать две задачи:

состязание за доступ к шине;

межпроцессорные взаимодействия.

В связи с тем, что память и устройство ввода-вывода по общей системной шине используют несколько процессоров, потребуется дополнительная логика управления шиной.

Для реализации мультипроцессорных систем используется максимальный режим работы МП, при этом различают три базовые конфигурации:
сопроцессор,
сильно связанная конфигурация,
слабо связанная конфигурация.

Первые две очень похожи друг на друга, структурная схема сопроцессорной и сильно связанной конфигурации представлена на рис. 66. в этом случае МП 8086/8088 является ведущим, а вспомогательный микропроцессор

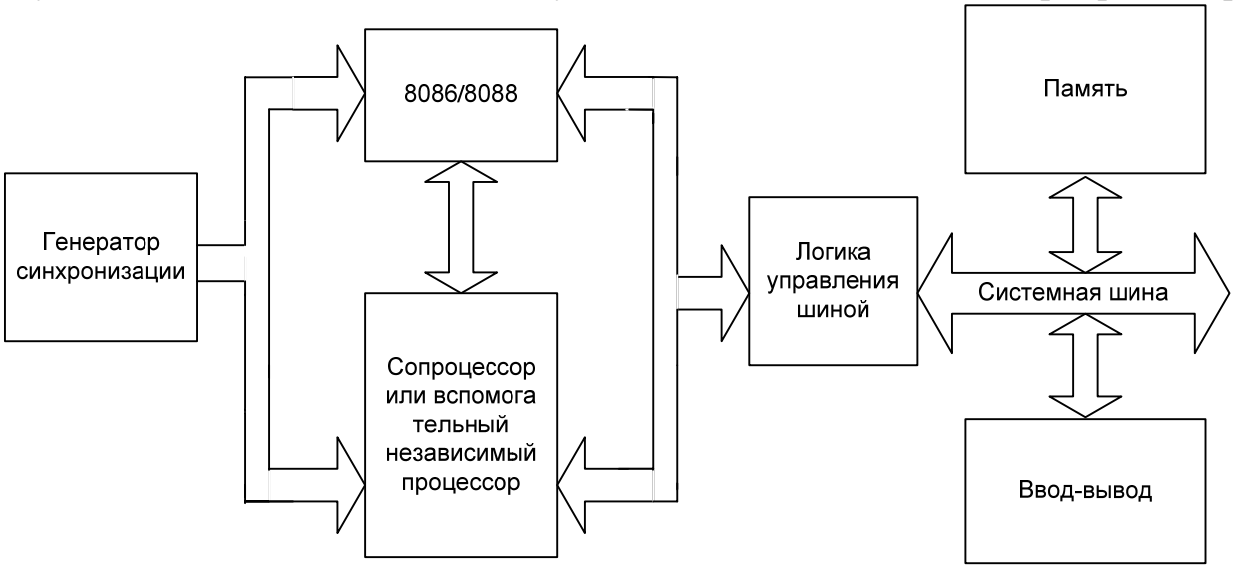


Рис. 66. Сильно связанная конфигурация

или сопроцессор – ведомым. Оба процессора разделяют не только всю подсистему памяти и ввода-вывода, но и логику управления шиной и генератор синхронизации. Управление доступом к шине осуществляет центральный процессор. В рассматриваемых конфигурациях не может быть двух микропроцессоров 8086/8088.

Слабо связанные конфигурации применяют в средних и больших системах, в них системные ресурсы разделяют несколько модулей, а проблему состязаний при доступе к шине должна решать логика управления системной шиной. Структурная схема слабо связанной конфигурации представлена на

рис 70. Как показано на этом рисунке, каждый потенциальный ведущий шины работает независимо, и прямые связи между ними отсутствуют. Межпроцессорное взаимодействие осуществляется через разделённые ресурсы. Кроме них у каждого модуля могут быть свои память и устройства ввода-вывода. Процессоры в отдельных модулях могут одновременно обращаться к своим локальным подсистемам по локальным шинам и выполнять независимо друг от друга выборки команд и обращения к локальным данным, что повышает степень параллельности обработки.

6.1.2. Сопроцессорные конфигурации

Для эффективного решения некоторых сложных задач вычислительных возможностей МП 8086/8088 недостаточно. Например, микропроцессоры 8086/8088 не имеют команд арифметики с плавающей точкой, а сопроцессор 8087 легко реализует такие вычисления. Далее подробно рассматривается структурная схема арифметического сопроцессора, его взаимодействие с центральным процессором и система команд, поэтому в этом разделе рассмотрим эти вопросы кратко, чтобы иметь общую картину работы МП I8086/8088 и I8087 в сопроцессорной конфигурации.

Система с арифметическим сопроцессором не требует никакой дополнительной логики, отличающейся от той, которая необходима в системе с максимальным режимом. Оба процессора выполняют команды из одной и той же программы. Взаимодействие между центральным МП и сопроцессором, когда команда выполняется сопроцессором, показано на рис 67. Центральный МП и сопроцессор выполняют свои команды из одной и той же программы. Если команда выполняется сопроцессором, центральный МП может подключиться для считывания требуемого операнда. Предназначенная для сопроцессора команда определяется появлением в программе одной из команд ESC. Мнемоника команд ESC имеет два формата, показанных на рис. 68, но любая команда сопроцессора содержит в первом байте код 11011.

Сопроцессор постоянно контролирует состояние очереди команд центрального процессора по линиям QS0, QS1, отслеживая комбинацию 01(т. е. «первый байт команды взят из очереди»). При совпадении этих двух факторов сопроцессор приступает к выполнению команды. Команда ESC одновременно дешифрируется сопроцессором и центральным процессором. В этой точке центральный МП может просто перейти к следующей команде или считать первое слово из памяти как операнд для сопроцессора, а затем перейти к следующей команде. Если центральный процессор считывает первое слово операнда, сопроцессор перехватывает слово данных и его 20-битный адрес. Когда операнд источник длиннее одного слова, сопроцессор получает остальные слова посредством запросов циклов шины по линии RQ/GT0 или RQ/GT1. если же определённый в команде ESC операнд является получателем, сопроцессор запоминает результат по перехваченному адресу. Сопроцессор выполняет действия, послав центральному МП на вход TEST сигнал

занятости (уровень логической единицы) и освободив шину. В это время центральный МП может выполнять свою программу. Такая параллельная работа продолжается до тех пор, пока центральному МП не понадобится результат текущей операции. При этом центральный МП должен выполнить команду WAIT и ожидать, пока сопроцессор не выдаст активный сигнал (уровень логической единицы) на вход TEST. Команда WAIT периодически проверяет вход TEST и, когда он становится активным, осуществляет передачу управления находящейся за ней команде.

К одному центральному МП допускается подключать два сопроцессора, один из них подсоединяется к линии RQ/GTO, а другой – к RQ/GT1.

При возникновении ошибки в процессе дешифрирования и выполнения команды ESC сопроцессор формирует запрос прерывания INT, который обычно подаётся в контроллер прерывания I8259A.

Когда к центральному МП подключены сопроцессор и независимый процессор ввода-вывода, который выбирает свои команды, сопроцессор должен определить, выбирается команда независимым процессором или центральным МП, иначе сопроцессор может ошибочно модифицировать свою

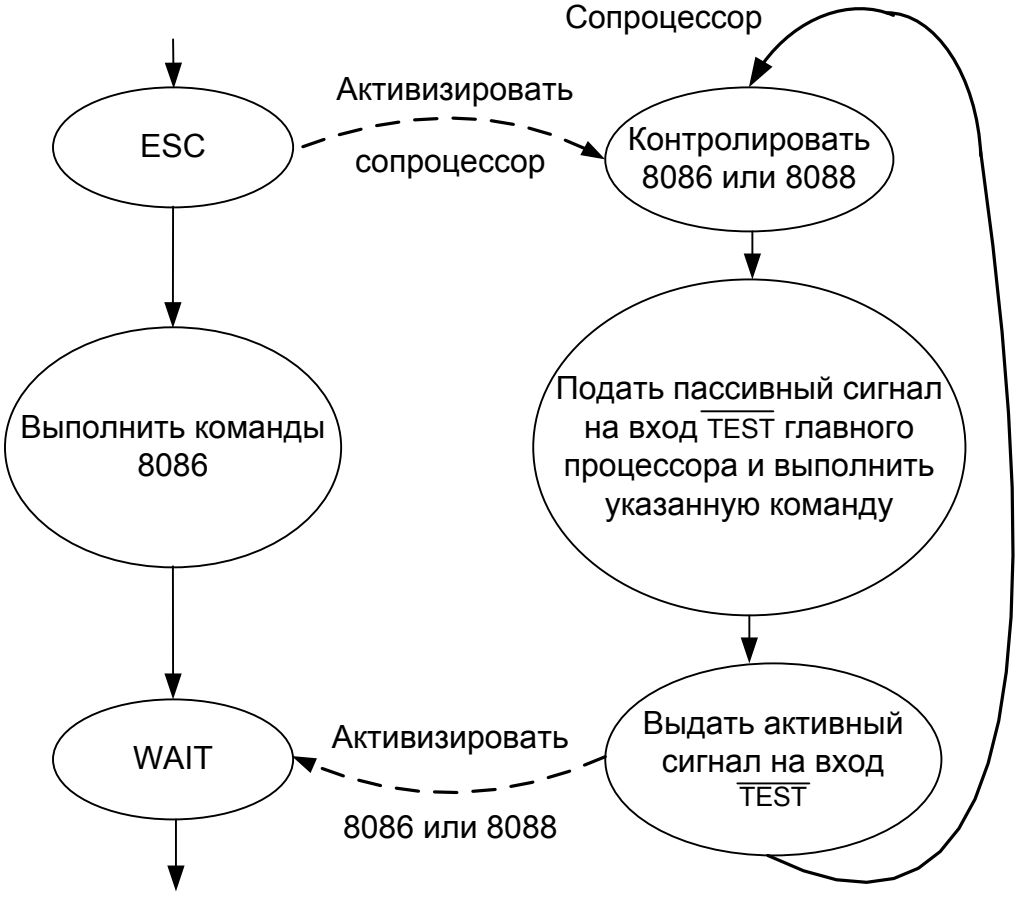


Рис. 67. Синхронизация микропроцессора 8086 с его сопроцессором

очередь команд. Для этого сопроцессор контролирует бит состояния ST6 - микропроцессоры I8086/8088 всегда выводят низкий уровень, а процессор ввода-вывода I8089 – высокий уровень.

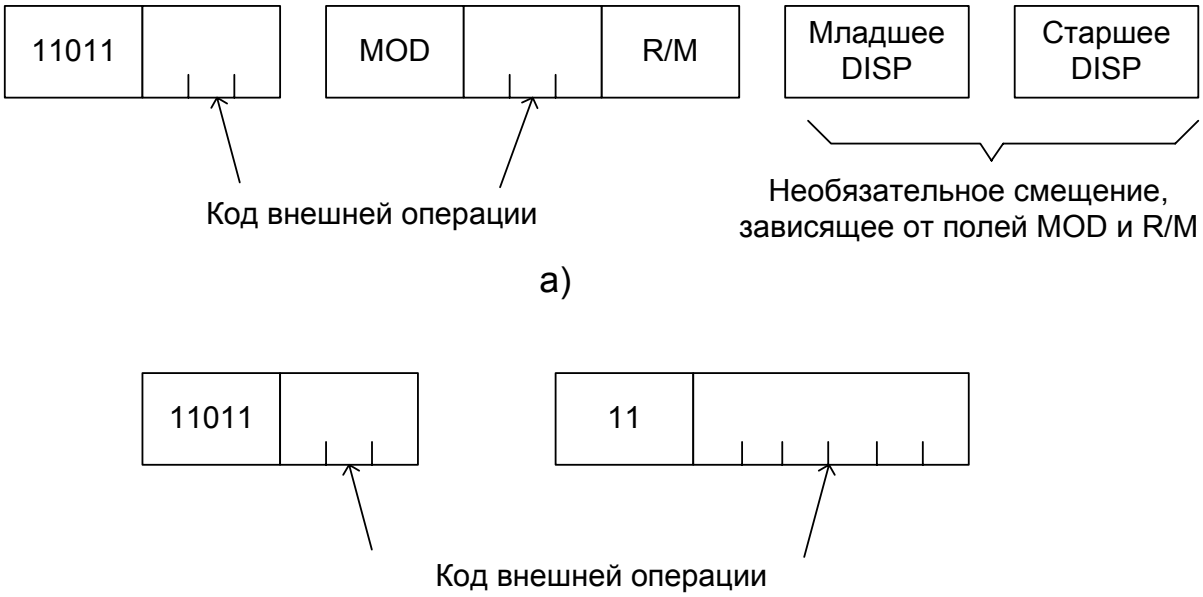


Рис.68. Форматы машинного кода команды ESC, когда операнд находится в памяти(а) и не находится (б).

6.1.3. Сильно связанные конфигурации

В сильно связанных конфигурациях в качестве независимого процессора выступает процессор ввода-вывода I8289. независимый процессор в отличие от сопроцессора выполняет свой командный поток. В этой конфигурации, так же как и при работе с сопроцессором, оба процессора разделяют генератор синхронизации и логику управления шиной.

Вместо специальных команд ESC и WAIT взаимодействие между центральным МП и независимым осуществляется через разделённое пространство памяти. На рис. 69 представлен алгоритм межпроцессорного взаимодействия через разделенную память. Центральный МП I8086/88 формирует сообщение в памяти и активизирует независимый процессор, посылая приказ в один из его портов. Затем процессор ввода-вывода обращается к разделённой памяти, получает оттуда свою задачу и выполняет её параллельно с центральным МП. О завершении задачи независимый процессор сообщает цен тральному МП с помощью бита состояния или запроса прерывания. При выполнении своей программы независимый процессор запрашивает шину по линии RQ/GT. Когда один процессор использует шину, другой переводит свои шины и выходы состояния в высокоимпедансное состояние. Так как микропроцессоры имеют две линии RQ/GT, то к центральному МП можно подключить два независимых процессора.

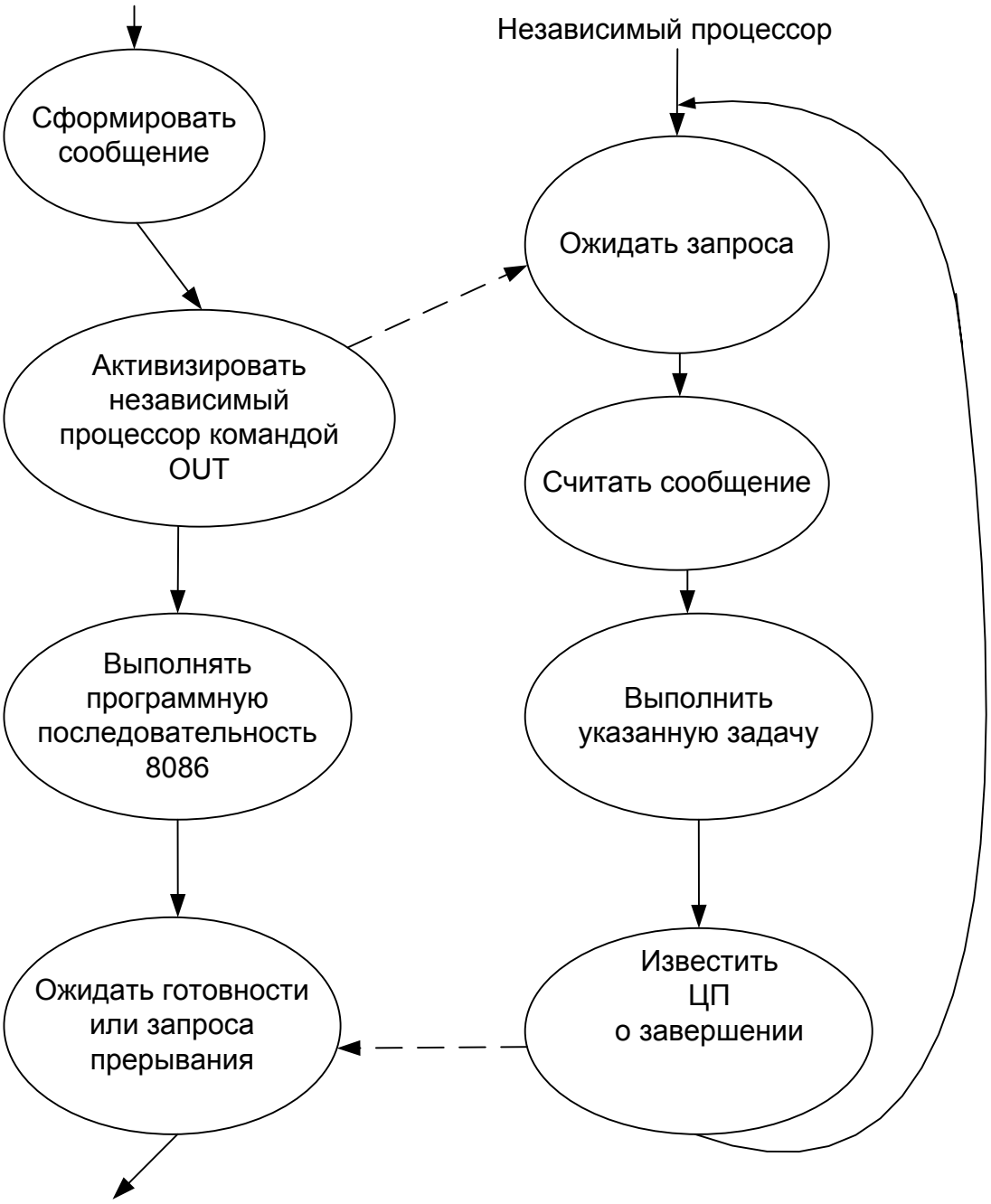


Рис.69. Межпроцессорные взаимодействия через разделённую память

Лекция № 19

6.1.4. Модульная организация в слабо связанных конфигурациях

В слабо связанных конфигурациях используется несколько «равносильных» МП I8086/88. Такая структура представлена на рис. 70. Все процессоры системы, находящиеся в процессорных модулях, взаимодействуют друг с другом и разделяют ресурсы через системную шину. Каждый центральный МП имеет свою логику управления шиной, а арбитраж шины дос-

тигается путём расширения этой логики и введения общей для всех ведущих модулей внешней логики. К каждому центральному МП можно подключить независимый процессор или сопроцессор. Слабо связанная конфигурация обладает рядом достоинств: повышенная пропускная способность системы, отдельные модули можно добавлять или удалять, не влияя на работоспособность всей системы, отказавший модуль можно легко заменить, поскольку каждый центральный МП имеет свою локальную шину, достигается высокая степень параллельной обработки, в слабо связанной конфигурации обязательным для каждого МП является наличие в модуле своего контроллера и арбитра шины. В любой момент времени системной шиной управляет только один модуль, поэтому необходима специальная схема арбитража. Одновременные запросы шины учитываются на приоритетной основе. При нескольких одновременных запросах учитываются приоритеты процессоров. Получив разрешение, процессор с наибольшим приоритетом реализует требуемый ему цикл шины и либо освобождает шину, либо ожидает приказа освободить её.

Применение арбитров шины позволяет реализовать или последовательный способ получения приоритета (приоритетная цепочка) или параллельный (независимое запрашивание). Способ приоритетной цепочки прост: если сигнал занятости системной шины пассивен, сигнал разрешения пассивно проходит через все ведущие модули до тех пор, пока не встречается первый модуль, запрашивающий доступ к шине. Этот модуль блокирует распространение сигнала разрешения шины, формируя сигнал занятости шины, и получает управление шиной. В этом случае приоритет определяется физическим размещением модулей. Недостатками такого способа являются большое время арбитража и меньшая надёжность, так как отказ одного модуля приводит к выходу из строя всей системы.

В способе независимых запросов приоритеты учитываются параллельно. Каждый модуль имеет отдельную пару линий запроса и разрешения шины со своим присвоенным приоритетом. Дешифратор приоритетов выбирает запрос с максимальным приоритетом и подаёт соответствующий сигнал разрешения шины. Арбитраж реализуется очень быстро и не зависит от числа модулей.

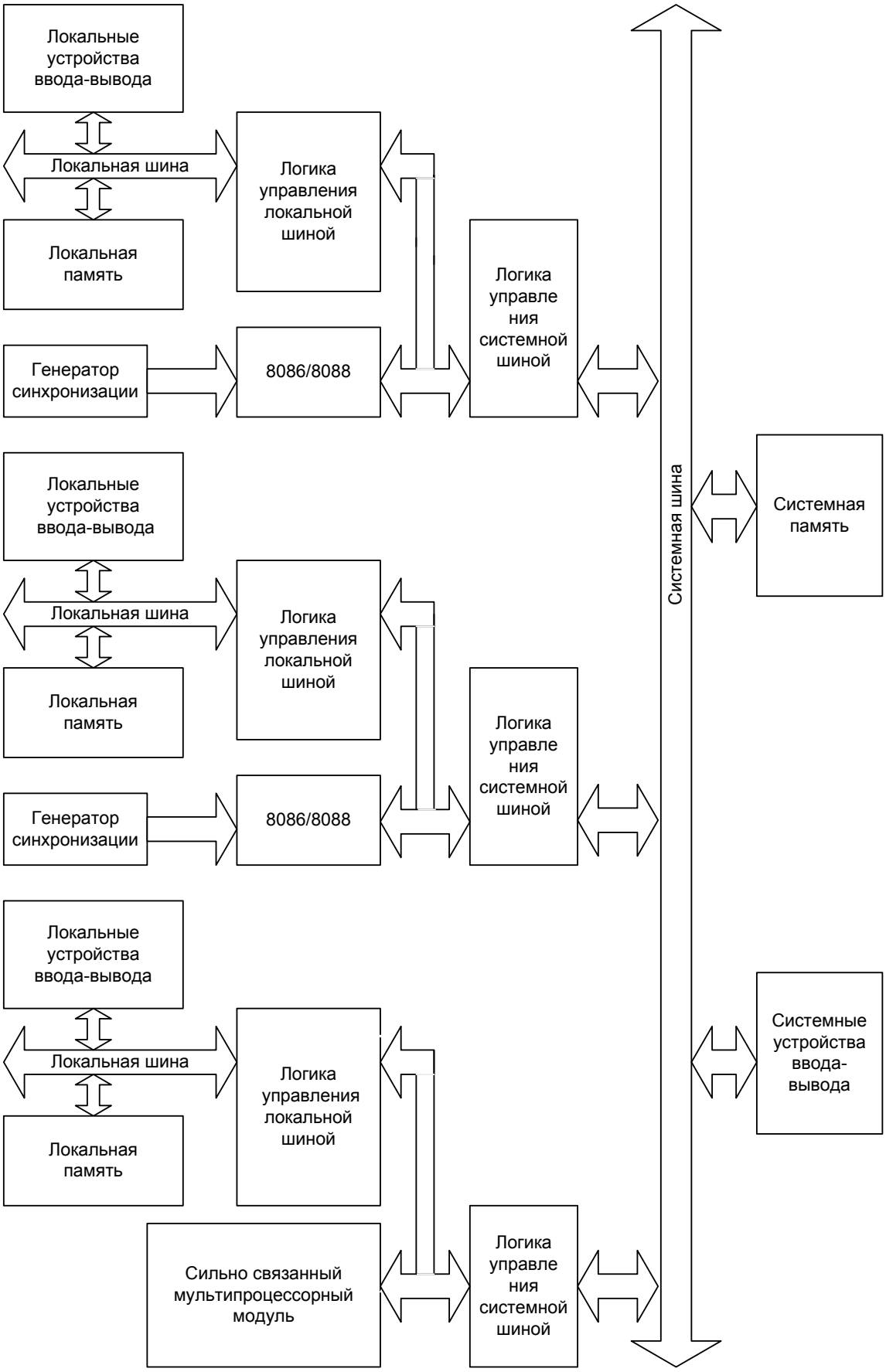


Рис.70. Слабо связанные конфигурации

Для реализации необходимого квитирования доступа к шине одного из нескольких микропроцессоров используется арбитр шины I8289 (К1810ВБ89). Арбитр шины предотвращает доступ к системной шине контроллера шины, формирователей шины данных и регистров адреса, переводя их выходные каскады в высокоимпедансное состояние, и заставляет МП перейти в состояние ожидания установкой сигнала RDY = 0. Микропроцессор остаётся в этом состоянии до тех пор, пока арбитр не разрешит доступ к системной шине, подключив к ней контроллер шины, формирователи шины данных и регистры адреса.

Структурная схема арбитра шины представлена на рис. 71.

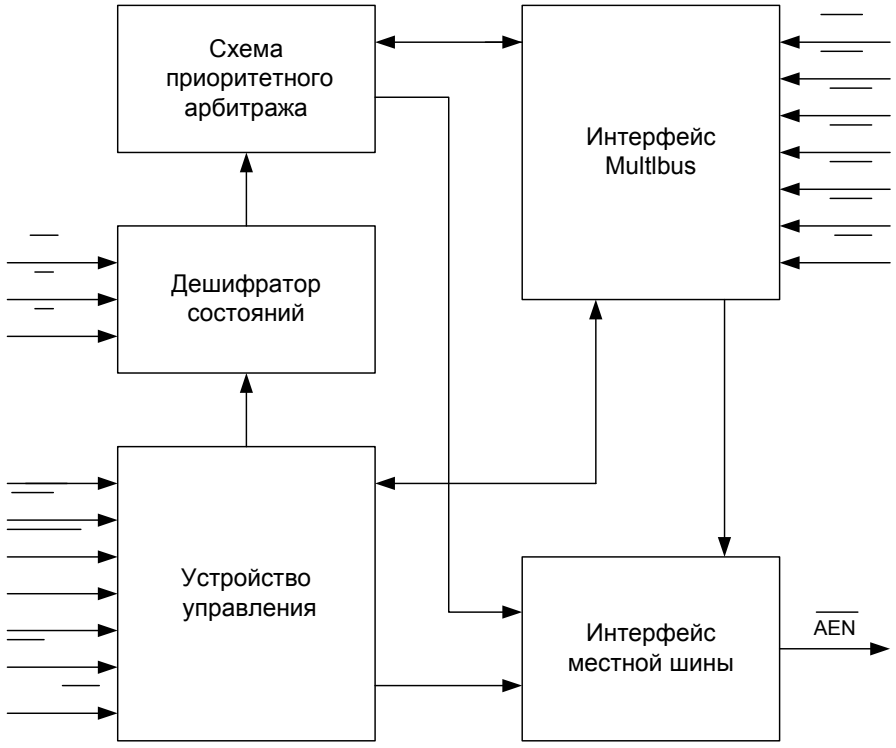


Рис. 71. Структурная схема арбитра шин

Рассмотрим кратко назначение узлов и основных выводов микросхемы. Дешифратор состояний, проанализировав код состояний МП, включает в работу схему приоритетного арбитража. Интерфейс местной шины формирует сигнал разрешения доступа к шине AEN для шинного интерфейса микропроцессорного модуля (контроллеру шины К1810ВГ88, регистрам-защёлкам адреса К1810ИР82/83 и шинным формирователямК1810ВА86/88). Интерфейс Multibus взаимодействует с другими арбитрами многопроцессорной системы и синхронизирует действия по захвату системной шины.

По линиям ST0, ST1, ST2 поступают сигналы состояния МП, по которым арбитр определяет для своего модуля, когда запрашивать или освобождать системную шину.

Для понимания способов учёта приоритетов ведущих шины достаточно рассмотреть следующие сигналы:

BREQ – запрос шины. Выходной сигнал, генерируемый арбитром для использования системной шины при параллельном способе организации приоритетов. Сигналы BREQ подаются в шифратор приоритетов, который формирует двоичный адрес активной линии BREQ с наибольшим приоритетом. Полученный адрес подаётся на дешифратор для выбора соответствующей линии BPRN, активный сигнал на которой возвращается в запрашивающий арбитр шины с наибольшим приоритетом.

BPRN – вход приоритета шины. Нулевой сигнал показывает арбитру, что он имеет приоритет над всеми другими арбитрами, запрашивающими системную шину. Этот арбитр разрешает своему МП доступ к системной шине, как только она будет доступна.

BUSY – вход / выход занятости шины. Выходной сигнал, равный нулю, выдаётся арбитром, получившем управление системной шиной. Он сообщает другим арбитрам, что системная шина занята. Для этого выводы BUSY (они имеют выходы с открытым коллектором) всех арбитров объединены в одну линию. Когда арбитр использовал шину, он снимает сигнал занятости BUSY (если на этом выходе устанавливается единица, шина доступна).

Способ последовательного учёта приоритетов реализуется путём организации приоритетной цепочки арбитров шины.

BPRO – выход приоритета шины. Данный сигнал используется при последовательном способе учёта приоритетов, когда выход BPRO некоторого арбитра подаётся на вход BPRN следующего арбитра с меньшим приоритетом. Если арбитр шины не запрашивает шину, он пропускает сигнал BPRN на выход BPRO.

Способ циклического учёта приоритетов аналогичен способу параллельного анализа приоритетов, но приоритеты арбитров периодически переназначаются.

CBRQ – вход/выход общего запроса шины.

Из рассмотренных способов учёта приоритетов предпочтение обычно отдаётся первому, так как он допускает использование большего числа арбитров шины и не требует сложных дополнительных схем. На рис. 72 и рис. 73 представлены структурные схема арбитража с последовательным и параллельным учётом приоритетов.

Следует акцентировать внимание ещё на два выбора арбитра: IOB и RESB. Эти два входа подключаются к постоянным уровням, показывая, какими ресурсами обладают отдельные модули. Сигнал на входе IOB, равный нулю, говорит о том, что в модуле есть своя периферийная шина ввода-вывода, часто называемая в литературе резидентной шиной ввода-вывода. Если на входе RESB установить единицу, данный модуль работает с рези

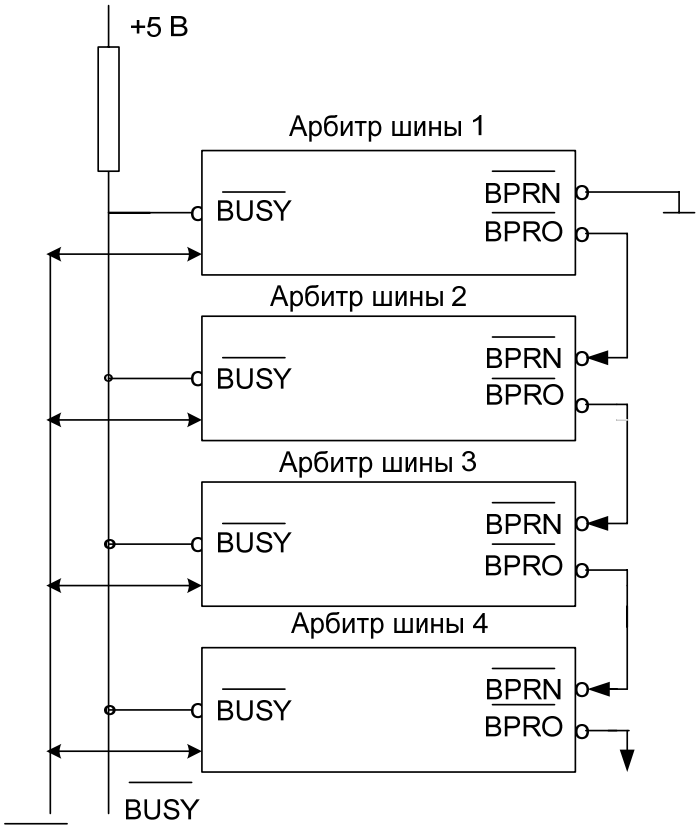


Рис. 72. Последовательный учёт приоритетов

дентной шиной памяти. В этом случае системная шина запрашивается или освобождается как функция входного сигнала SYSB/RESB. Четыре комбинации сигналов на этих входах позволяют получить соответствующее число конфигураций, в которых процессор может обращаться к локальным устройствам ввода-вывода и памяти:

режим работы с системной шиной (RESB=0, IOB=1);

режим работы с системной шиной памяти и резидентной шиной ввода-вывода (RESB = 0, IOB = 0);

режим с системной шиной и резидентной шиной (RESB=1, IOB=1);

режим с системной шиной памяти, резидентной шиной и резидентной шиной ввода-вывода (RESB = 1, IOB = 0).

Для полноты картины обозначим смысл сигнала AEN. Его назначением является разрешение доступа к системной шине. Сигнал AEN = 0 переводит формирователи ША и ШД, а также тактовый генератор МП в активное состояние и позволяет ему управлять системной шиной. На рис. 19 приведён пример схемы подключения арбитра шины к резидентской шине ввода-

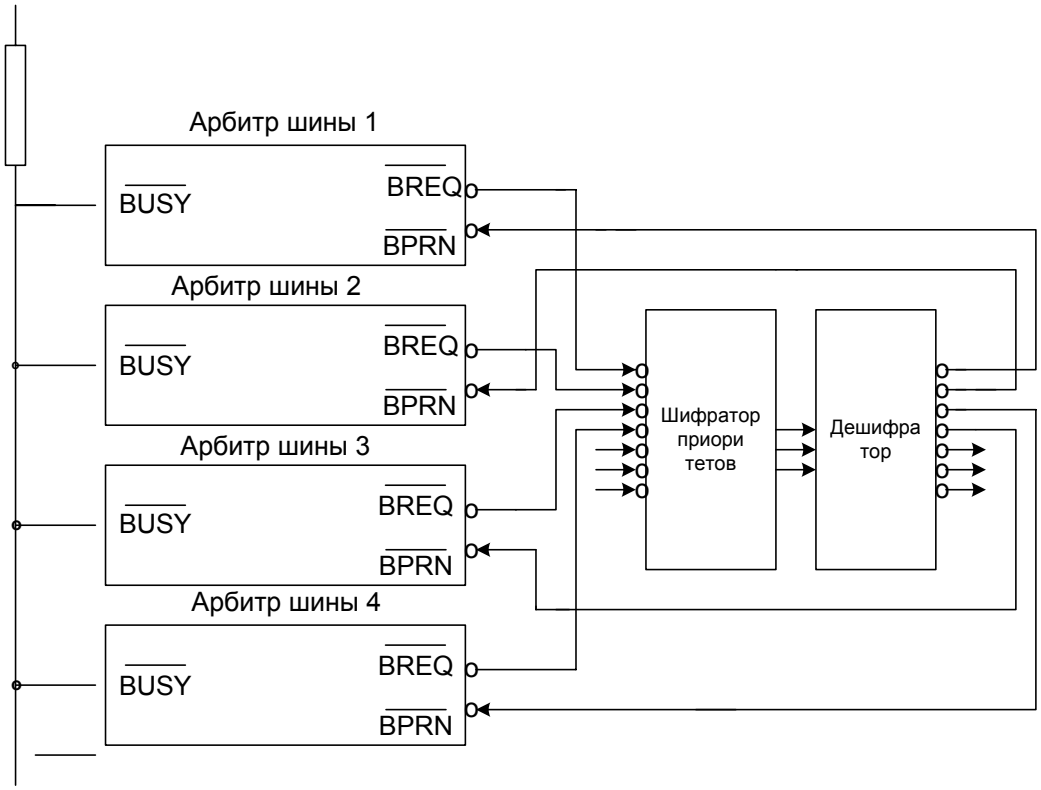


Рис. 73. Параллельный учёт приоритетов

вывода, резидентская шина памяти отсутствует, так как вход RESB подсоединён к нулю.

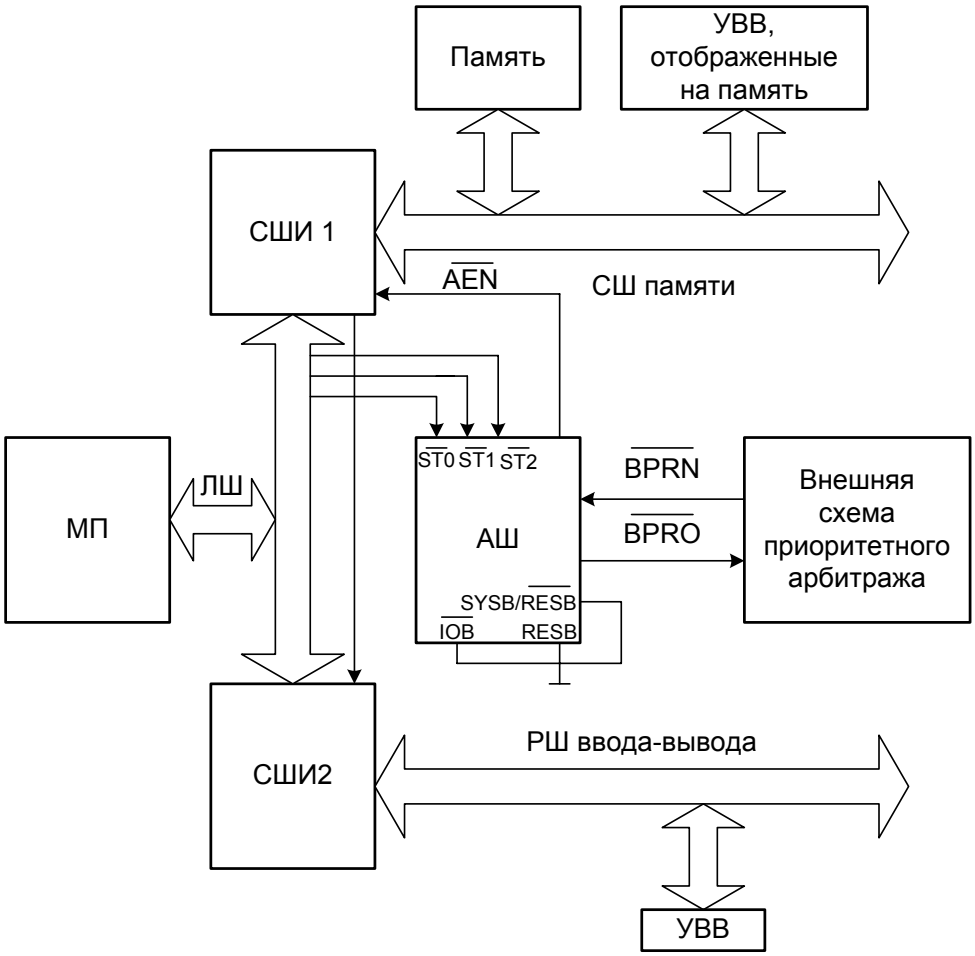


Рис. 74. Схема подключения АШ к резидентной шине

*6.2. Процессор ввода – вывода I 8089 (К1810 ВМ89)

Процессор ВМ89 имеет 40 выводов; напряжение питания – 5 В; мощность P≥2,5 Вт; синхронизируется частотой 1-5 МГц. Кроме передач с помощью контроллеров ПДП, подготовку и саму передачу данных осуществляет ЦП, но даже в режиме ПДП центральный процессор должен подготовить контроллер и следить за завершением каждой операции ПДП.

Процессор ввода – вывода (ПВВ) 8089 специально предназначен для эффективного управления вводом – выводом, он должен выбирать и выполнять свои команды, но кроме ввода – вывода, смогут выполняться арифметические и логические операции, переходы, поиск и преобразование.

ПВВ используется совместно с ВМ86/88, он может работать параллельно с центральным процессором одновременно по двум каналам ввода – вывода, каждый из которых обеспечивает скорость передачи информации до 1,25 Мбайт/с при f=5 МГц. Центральный процессор взаимодействует с ВМ89 посредством управляющих блоков в памяти. Он готовит управляющие блоки, которые описывают подлежащую выполнению задачу, а затем направляет задачу ВМ89. ПВВ считывает управляющие блоки канальной программы, которая написана в командах ПВВ.

На рисунке 75 представлена структурная схема процессора ввода – вывода.

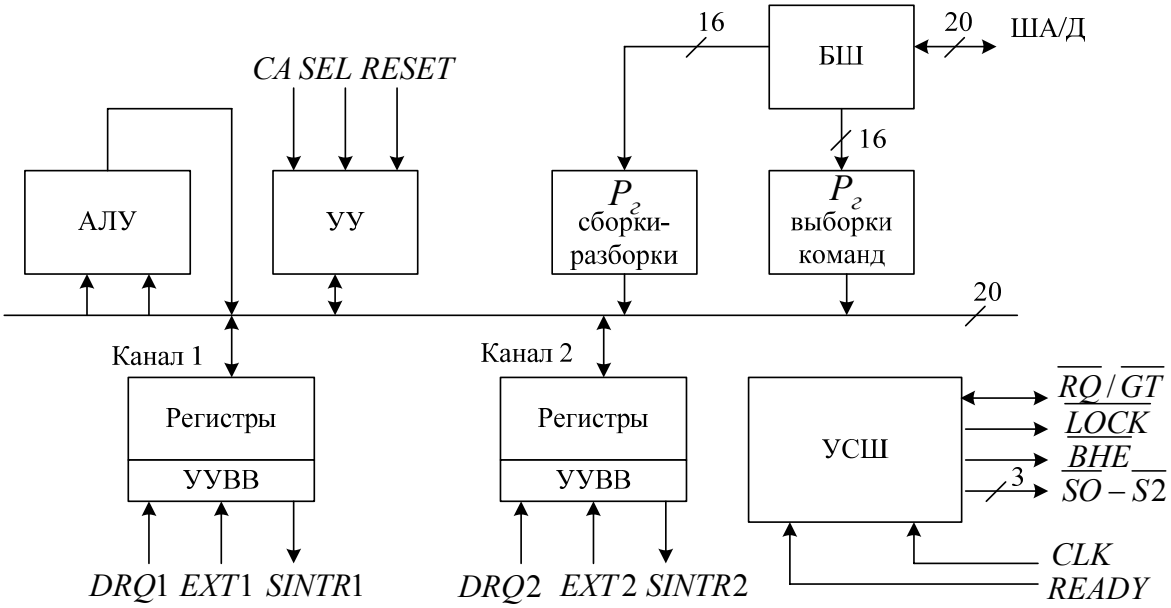


Рис. 75. Схема процессора ввода – вывода

Каждый из двух одинаковых каналов ввода – вывода содержит пять 20-ти битовых, четыре 16-ти битовых и один 4-х битовый регистр. Взаимодействие каналов осуществляется под управлением встроенной логики приоритетов. Контроллер имеет 16-битовую мультиплексируемую ШД и 20- разрядную ША. Оба канала разделяют логику управления и АЛУ. Каждый из каналов может осуществлять высокоскоростные пересылки в режиме ПДП с одновременным преобразованием пересылаемых данных. Что значит «одновременные преобразования»? Из схемы видно, что в состав входит АЛУ, оно может:

а) осуществлять арифметические операции над 8 и 16 двоичными числами.

б) логические операции И, ИЛИ, НЕ.

в) может анализировать условия окончания передачи.

г) может с помощью регистра сборки – разборки принимать из 1-байтного источника, а передавать их в 2-байтный получатель и наоборот.

Общее устройство управления УУ обеспечивает начальную установку сопроцессора. После сброса ВМ86 подготавливает в памяти блоки, позволяющие инициализировать сопроцессор и выдать ему задачи через память.

Таблица 26

S_2	S_1	S_0	Сигналы состояния микропроцессора
0	0	0	Выборка команды из адресного пространства ввода-вывода
0	0	1	Чтение данных из адресного пространства ввода-вывода
0	1	0	Запись данных в адресное пространство ввода-вывода
0	1	1	Не используется
1	0	0	Выборка команды из системного пространства адресов
1	0	1	Чтение данных из системного пространства адресов
1	1	0	Запись данных в системное пространство адресов
1	1	1	Пассивное пространство

Назначение выводов.

AD_0 – AD15 – ША/ШД. Функции шины задаются сигналами состояния $S_0 \div S_2$.

A16/S3, A17/S4, A18/S5, A19/S6 – четыре старших разряда адреса и сигналов состояния.

Сигналы адресов формируются в течение первой части цикла шины (T1), а затем работают сигналы состояний.

S6=S5=1 – ПДП-пересылка

S4=0 – ПДП-пересылка

S4=1 – цикл шины без ПДП

S3=0 – работает канал 1

S3=1 – работает канал 2.

После сброса эти шины находятся в третьем состоянии.

$S_0 \div S_2$ – выходы для кодирования состояния ВМ89, назначение этих выводов приведены в таблице 26. С их помощью контроллер шины и арбитр формируют команды управления памятью и устройствами ввода – вывода. Сигналы формируются в такте Т4 передающего цикла, определяя начало нового цикла. В такте Т3 текущего цикла переходят в пассивное состояние.

BHE – выходной сигнал разрешения старшего байта ШД. При сбросе и отсутствии обращений в 3-ем состоянии (может не фиксироваться в регистре адреса, т.к. не мультиплексирован с другим сигналом).

LOCK – выходной сигнал, говорящий о занятости системной шины в многопроцессорных системах. Подается на аналогичный вход арбитра шины, запрещая доступ к шине другим процессорам. Сигнал формируется или командой TSL, или установкой соответствующего разряда регистра управления.

RQ/GT –вход-выход сигнала, запрос-предоставление шин. Сигнал необходим для арбитража шины между сопроцессором и процессором или между двумя сопроцессорами.

CLK – вход тактовых импульсов (с ГФ84)

READY –входной сигнал готовности о возможности пересылки данных (синхронизируется через ГФ84)
CA – входной сигнал запроса готовности канала. Используется при инициализации СПВВ и при определении задания каналами. Также по срезу этого сигнала опрашивается состояние входа SEL. Этот сигнал можно назвать сигналом внимания для СП.
SEL – входной сигнал выбора канала. По первому после сброса сигналу CA определяется, ведущий или ведомый данный сопроцессор и далее запускается последовательность инициализации. Следующим сигналом CA сигнал SEL определяет номер канала (1 или 2),которому поступит сообщение от ЦП. На вход SEL обычно подключается линия адреса A_0.
RESET – начальная установка, переводящая СП в пассивное состояние до получения сигнала запроса готовности канала.
DRQ1, DRQ2 – входы запросов ПДП от внешних устройств (для 1 или 2 канала).
EXT1, EXT2 – входящие сигналы окончания прямого доступа, если канал запрограммирован на окончание по внешнему сигналу.
SINTR1, SINTR2 – выходные сигналы запросов прерывания для МП.
Размещение СПВВ в адресном пространстве.

СПВВ может работать в сильно связанной (локальной) конфигурации или в слабо связанной (дистанционной, удалённой) конфигурации.

СПВВ может обращаться к памяти и ВУ, размещёнными в системном пространстве ёмкостью 1 Мбайт или в пространстве ввода/вывода ёмкостью 64 Кбайт (рис. 76).

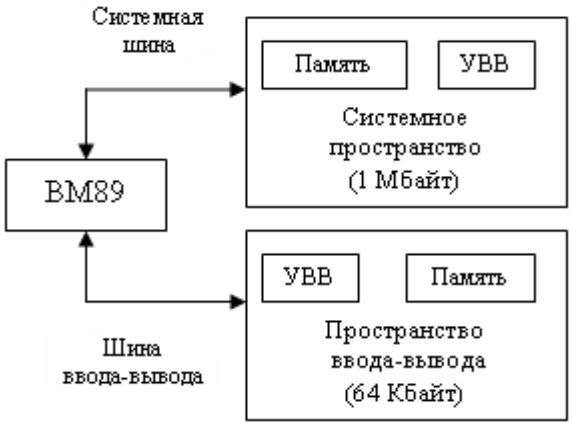


Рис.76. Два пространства адресов процессоров ввода-вывода

Различие между этими двумя шинами состоит в том, что СШ управляется сигналами: чтения и записи в память, то есть УВВ входит в адресное пространство памяти, а ШВВ – сигналами чтения и записи внешних устройств.

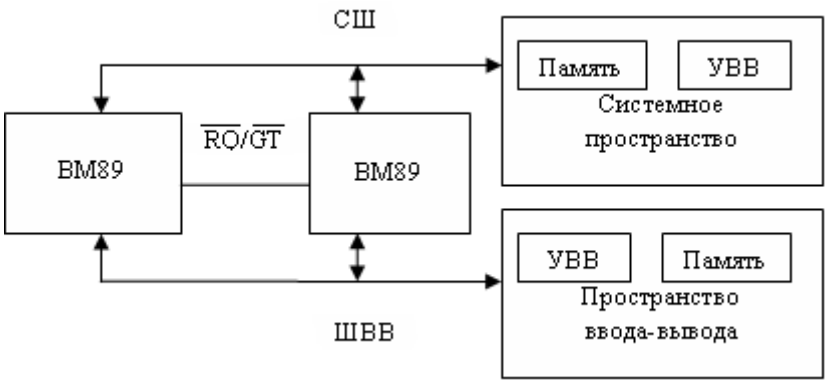
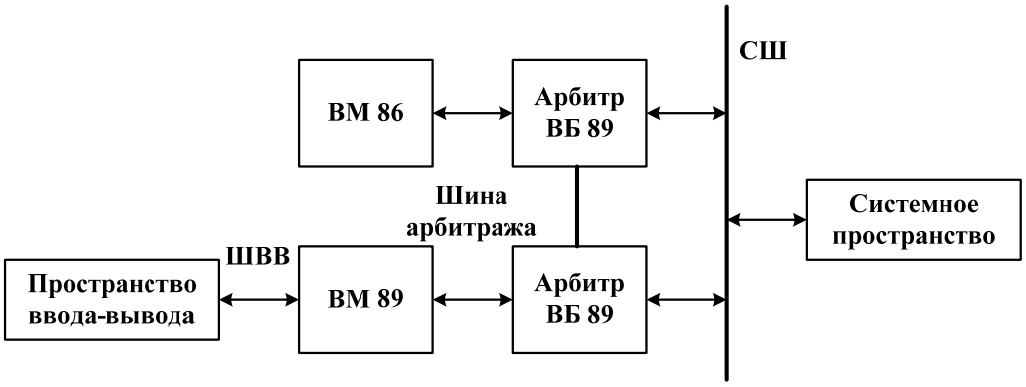


Рис. 77. Использование СШ и ШВВ в местной конфигурации

Таким образом, устройства ввода-вывода, размещённые в системном пространстве, оказываются отображёнными на память (реагируют на 20-битовые адреса, закреплённые за ними по командам чтения и записи в память), а память, размещённая в пространстве ввода-вывода, – отображённой на внешние устройства(адресуемой 16-битовыми адресами и реагирующей на команды чтения и записи во внешние устройства).



б)

Рис.78. Использование СШ и ШВВ в удалённой конфигурации

Рассмотренные СШ и ШВВ работают по-разному в зависимости от конфигурации, в которой используется СПВВ.

Структура СПВВ и его функционирование.

Оба канала сопроцессора (рис. 75) программируются и работают независимо друг от друга, но имеют общую логику управления и АЛУ.

Процессор может выполнять передачи ПДП со множеством вариантов. Направления могут быть: память-память, память-ввод/вывод, ввод/вывод-память, ввод/вывод-ввод/вывод.

Если данные передаются из 8-битного источника в 16-битный получатель, процессор «собирает» их в одно 16-битовое слово и передаёт на шину за один цикл, увеличивая скорость передачи. Или обратная ситуация, этим занимаются регистры сборки-разборки.

Кроме того, между циклами считывания и запоминания в процессоре можно сравнить или преобразовать данные с помощью АЛУ и регистров.

Операцию ПДП можно окончить по внешнему запросу, по соответствию или несоответствию при маскированном сравнении или же при достижении счётчиком нуля. Такие варианты работы определяются содержимым регистра управления канала СС.

Как уже говорилось выше, для того, чтобы СПВВ мог начать работать, он должен быть инициализирован. Для этого ЦП готовит необходимое сообщение в память, чтобы произвести начальную инициализацию, а затем выдаёт команды каналам.

Сообщение для СПВВ формируется ЦП в виде четырёх блоков.

В этих блоках сообщается физическая ширина системной шины, ширина использования шины ввода/вывода, начальный адрес программы канала и видимая служебная информация.

Названия этих блоков:

- блоки параметров (2);
- управляющие блоки (2);
- блок системной конфигурации;
- блок указателя системной конфигурации.

Итак, инициализация ПВВ готовится ЦП и записывается в память. Запускается инициализация путём выдачи сигнала СА. *Сигнал по линии СА заставляет ПВВ выполнять следующее:*

1. из фиксированной ячейки $FFFF6_h$ ввести слово BUSY, определяющее ширину системной шины;
2. загрузить адрес блока системной конфигурации SCB из ячеек $FFFF8_h$ и $FFFFA_h$;
3. ввести слово, определяющее режим и ширину шины ввода/вывода (SOC);
4. загрузить адрес управляющего блока (из SCB+2 и SCB+4) в регистр CCP;
5. установить BUSY в CB+1 на 00 (ширина шины – 8 б.) ;
6. !ожидать от ЦП следующего сигнала СА, инициирующего выполнение задачи.

Если в системе несколько ПВВ, то ЦП инициализирует их по очереди.

Затем идёт инициализация отдельных каналов, следующим будет сигнал СА. Сигнал СА появляется в принципе при выполнении любой команды, связанной с выставлением адреса, обычно это команда OUT. Следующий сигнал СА заставляет СПВВ анализировать сигнал SEL, если SEL=0 , значит 16-разрядная системная шина, если SEL=1, то 8-разрядная. Затем декодируется управляющее слово и начинает выполняться действие в соответствии с кодами полей этого слова.

Использование СПВВ в местной (сильно связанной) и удалённой (слабо связанной) конфигурации.

В сильно связанной конфигурации ЦП и ПВВ совместно используют общую шину, при этом ПВВ – ведомый. ПВВ запрашивает шину по линии $\overline{RQ}/\overline{GT}$. Когда

работает один процессор, другой переводит шины и линии S_0-S_2 в третье состояние. Шина управления формируется системным контроллером ВГ88, ША – регистрами типа ИР82, ШД – приёмопередатчиком ВА86. Для синхронизации используется общий генератор. Для формирования сигнала готовности СА используется дешифратор адреса A_1-A_{15}, срабатывающий по команде OUT ADRR, где указывается адрес, выделенный для ВМ89.

Значение сигнала SEL формируется по адресу линии A_0. сигналы запросов прерываний с выходов SINTR1, SINTR2 ВМ89 подаются на вход INTR ВМ86 через контроллер прерываний (на рис. не показан).

Сигналы запросов прямого доступа от ВУ, участвующих в ПДП – пересылках, подаются на входы DRQ1, DRQ2 процессора ВМ89 (если >2, то можно объединить по схеме *или*). Если окончанием ПДП управляет ВУ, то оно сообщает об этом по сигналам EXT1 и EXT2.

Так как у ВМ86 две линии RQ/GT, то к нему можно подключить два ПВВ.

В слабо связанной (удалённой, дистанционной) конфигурации ВМ89 пользуется СШ, являющейся общей для всех процессорных модулей.

Каждый процессор должен иметь свой арбитр шины, связанный с другими арбитрами по шине арбитража. Если программа процессора ввода/вывода размещена только в локальной памяти, а в системной памяти располагается только один из четырёх управляющих блоков SCPB, SCB, CP и PB, осуществляется действительно полная параллельная работа процессоров.

Система команд процессора ввода/вывода имеет 53 команды, команды классифицируются на следующие группы:

1. общие передачи данных;
2. 8 и 16-битные арифметические операции;
3. 8 или 16-битные логические операции;
4. команды загрузки и запоминания указателей;
5. условные и безусловные переходы и вызовы подпрограмм;
6. операции манипуляции битами и проверки;
7. команды управления процессором.

Лекция № 20

7. АРИФМЕТИЧЕСКИЙ СОПРОЦЕССОР

7.1. Архитектура арифметического сопроцессора

7.1.1. Общие сведения

Существующие современные однокристальные микропроцессоры с длиной слова 16 и 32 бита оперируют практически с двумя простыми форматами численных данных: знаковыми и беззнаковыми целыми двоичными числами, хотя с определенными ограничениями допускают десятичные целые числа. Вычислительные возможности их ограничены только арифметическими операциями. Реализация расширенных форматов данных и систем

команд в однокристальных микропроцессорах ведет к их чрезмерному усложнению и трудности в программировании. Программная реализация сложных команд на микропрограммном уровне ведет к значительному снижению быстродействия микропроцессора.

В сложившейся ситуации наибольшее распространение получил принцип специализации, который применяется в процессорах средних и больших компьютеров. Суть его заключается в разработке вспомогательных процессорных модулей со своими системами команд, ориентированных на конкретные прикладные области: процессор арифметики с плавающей точкой, процессор символьных данных, процессор ввода-вывода и др. Такие вспомогательные процессоры работают под общим управлением центрального (главного) процессора и обычно разделяют с ним основную память. Специализация позволяет достичь очень высокого быстродействия.

В сопроцессорной конфигурации вспомогательный процессор (сопроцессор) подключается к системной шине параллельно центральному процессору. Сопроцессор не имеет своей отдельной программы и не может считывать команды из памяти, но может обращаться к памяти для записи и считывания данных, запрашивая для этого шину центрального процессора. Кроме того, сопроцессор контролирует системную шину и может «перехватить» адреса и данные при обращении к памяти центрального процессора. Таким образом, программа оказывается смесью команд центрального процессора и сопроцессора, причем считывание команд из памяти осуществляет только центральный процессор.

Первым из сопроцессоров, предназначенных для работы с центральным процессором К1810ВМ86 (I8086), появился процессор числовых данных, или арифметический сопроцессор К1810ВМ87 (I8087). Он рассчитан на применение в системах с интенсивной численной обработкой, в которых:

- численные данные изменяются в очень широком диапазоне;
- при реализации алгоритмов возникают очень большие и малые промежуточные данные;
- требуется высокая точность вычисления;
- необходима производительность, превышающая возможности центрального процессора.

Наряду с традиционными арифметическими командами, работающими как целыми, так и числами с плавающей точкой, арифметический сопроцессор имеет команды таких сложных операций, как извлечение квадратного корня, вычисление тригонометрических и обратных тригонометрических функций, возведение в степень, логарифмирование и др.

7.1.2. Архитектура сопроцессора. Структурная схема. Взаимодействие блоков

Структурную схему сопроцессора удобно представить в виде программной модели. В программную (регистровую) модель любого процессора

включаются только те регистры, которые доступны программисту на уровне машинных команд. Арифметический сопроцессор имеет общую стековую организацию. Выбор такой организации обусловлен несколькими обстоятельствами. Одно из них заключается в том, что в математических расчетах результат текущей операции часто может заместить один или оба исходных операнда и является операндом следующей команды. Стековая организация позволяет в этих случаях применять так называемые безадресные команды небольшой длины (неявная адресация), сокращая, таким образом, число обращений к памяти и, следовательно, повышая быстродействие.

Основу программной модели сопроцессора, показанной на рис. 79, образует регистровый стек из восьми 80-битовых регистров R0–R7. В этих (арифметических) регистрах хранятся числа, представленные в так называемом временном вещественном формате. В любой момент времени трехбитовое поле ST в слове состояния определяет регистр, являющийся текущей вершиной стека (Stack Top) и обозначается ST(0) или просто ST. При операции включения в стек осуществляется декремент поля ST и загружаются адресуемые данные в новую вершину стека. При операции извлечения из стека в получатель, которым чаще всего является память, передается содержимое текущей вершины стека, а затем производится инкремент поля ST.

В организации регистрового стека сопроцессора имеется несколько отличий от классического стека, который в большинстве современных процессоров аппаратно реализуется в памяти.

Во-первых, стек имеет круговую (кольцевую) организацию. Необходимо помнить, что максимальное число включений в стек без промежуточных извлечений равно восьми, а девятое включение перезапишет элемент, помещенный в стек первым (на самом деле сопроцессор зафиксирует здесь особый случай недействительной операции и, если прерывание замаскировано или запрещено, тогда произойдет перезапись, иначе в сопроцессор загрузится NAN).

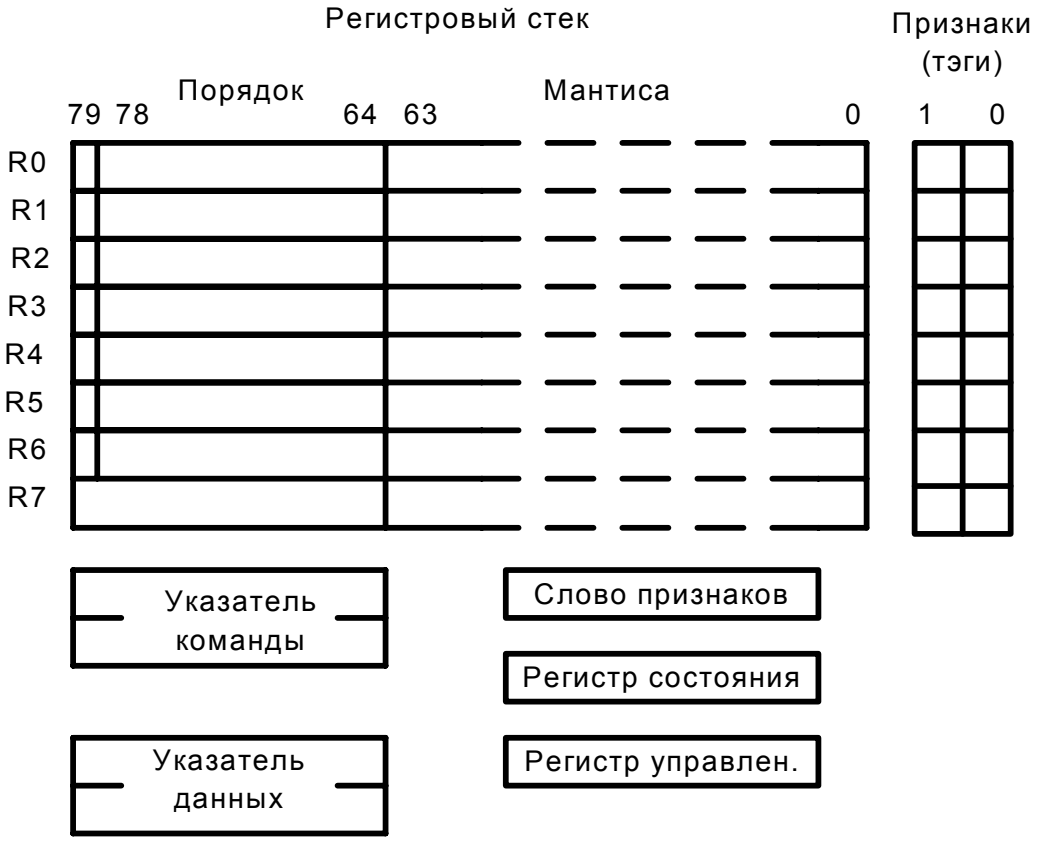


Рис. 79. Программная модель сопроцессора

Во-вторых, в командах сопроцессора допускают явное или неявное обращение к регистрам стека с модификацией или без модификации поля ST. Так в некоторых операциях операндом служит содержимое вершины стека.

В-третьих, сопроцессор имеет команды, в которых не изменяется содержимое вершины стека (в процессоре стек автоматически модифицируется при любом обращении к нему).

С каждым регистром стека ассоциируется двухбитовый тэг (признак), совокупность которых образует слово тэгов. Тэг регистра R0 находится в младших битах этого слова, а тэг регистра R7 – в старших. Тэг фиксирует наличие в регистре действительного числа (конечное ненулевое число) – код 00, истинного нуля – код 01, специального числа (денормализованное число, не-число или бесконечность) – код 10 и отсутствие данных – код 11. Попытка команды извлечь число из пустого регистра фиксируется как особый случай недействительной операции. Кроме того, попытка загрузить число в непустой регистр также вызывает регистрацию особого случая.

Программист может использовать слово тэгов для интерпретации содержимого регистров только после передачи этого слова в память.

Остальными регистрами в программной модели сопроцессора являются регистр управления, регистр состояния, два регистра указателя команды и два регистра указателя данных.

Регистр состояния. В слове состояния (рис. 80) отражается текущее состояние сопроцессора после выполнения последней команды.

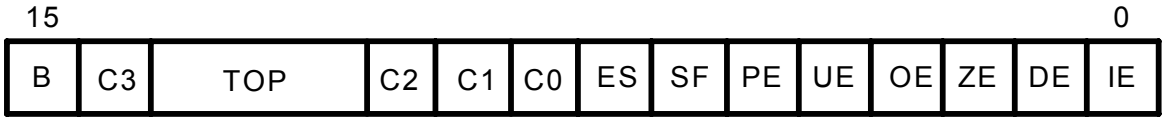


Рис. 80. Слово состояния сопроцессора

Шесть младших битов показывают исключительные ситуации:

IE – недействительная операция;

DE – денормализованный операнд;

ZE – ошибка деления на ноль;

OE – ошибка переполнения. Возникает в случае выхода порядка числа за максимально допустимый диапазон;

UE – ошибка антипереполнения. Возникает, когда результат слишком мал;

PE – ошибка точности. Устанавливается, когда процессору приходится округлять результат из-за того, что его точное представление невозможно (например при делении 1 на 3).

SF – ошибка работы стека сопроцессора. Устанавливается в единицу, если возникает одна из трех исключительных ситуаций – PE, UE или IE. В частности, его установка информирует о попытке записи в заполненный стек, или при попытке чтения из пустого стека.

ES – суммарная ошибка работы сопроцессора. Бит устанавливается в единицу, если установится в единицу хотя бы один из шести битов (IE, DE, ZE, OE, UE, PE).

C0-C3 – коды условия. Отражают результат выполнения последней команды сопроцессора (таблица 27).

Таблица 27

C3	C2	C1	C0	Описание
0	0	x	0	ST > источника (src) или нуля
0	0	x	1	ST < источника (src) или нуля
1	0	x	0	ST = источнику (src) или нулю
1	1	x	0	Не сравнимы

TOP – трехбитовое поле, показывающее номер регистра, который является вершиной стека.

Регистр управления. Слово управления (рис. 81) определяет для сопроцессора один из вариантов обработки численных данных.

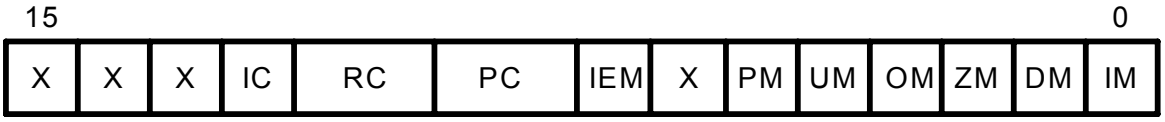


Рис. 81. Слово управления сопроцессора

Для каждого варианта программист задает маскирование особых случаев, точность вычислений, способ округления и интерпретацию бесконечности. Слово управления загружается из памяти с помощью команды FLDCW.

Шесть младших битов слова управления представляют собой индивидуальные маски особых случаев. Если любой из этих битов установлен в 1, то возникновение соответствующего особого случая, которое фиксируется в шести младших битах слова состояния, не будет вызывать прерывание центрального процессора (прерывание запрещено или замаскировано), а если бит содержит – 0, тогда генерируется соответствующее прерывание центрального процессора.

IEM – маска управления прерывания, которая разрешает (IEM = 0) или запрещает (IEM = 1) прерывание центрального процессора.

PC – биты управления точностью вычисления 00 – 24 бита, 10 – 53 бита, 11 – 64 бита. По умолчанию вводится режим с максимальной точностью.

RC – биты управления округлением (таблица 28). Необходимость операции округления может возникнуть, например, при вычислении дробной или целой части числа. В этом случае целесообразно выбирать режим усечения. Для того чтобы выяснить характер округления, обозначим:

m – значение в ST(0) или результат работы некоторой команды;

a и b – наиболее близкие значения к m, причем a < m < b.

Таблица 28

RC	Режим	Принимаемый результат
00	Округления к ближайшему	Значение m округляется к ближайшему числу a или b
01	Округления вниз	a
10	Округления вверх	b
11	Усечение	Производится отбрасывание дробной части

IC – бит управления бесконечностью. Определяет одну из двух моделей интерпретации бесконечности: проективную (IC=0) или аффинную (IC = 1). По умолчанию вводится проективный режим, в котором сопроцессор обрабатывает два специальных значения «плюс бесконечность» и «минус бесконечность» как одно и то же значение «бесконечность», не имеющее знака. В аффинном режиме сопроцессор допускает значения бесконечности со знаком плюс или минус.

Информация, содержащаяся в регистрах указателя команды и операнда (совместно их называют указателями особого случая), предназначена только

для процедур обработки особых случаев. Когда сопроцессор выполняет численную команду, он автоматически сохраняет в этих регистрах физический адрес команды сопроцессора, физический адрес операнда, если он присутствует в команде.

Лекция № 21

7.1.3 Программирование сопроцессора

В систему команд арифметического сопроцессора К1810ВМ87 входят 69 базовых команд, которые можно разделить на шесть групп: команды передачи данных, арифметические команды, команды сравнения, команды трансцендентных операций, команды загрузки констант и команды управления сопроцессором.

Типичная команда сопроцессора воспринимает один или два операнда. Операндами наиболее часто служит содержимое регистров сопроцессора, но может привлекаться и содержимое ячеек памяти (с регистрами процессора арифметический сопроцессор не работает).

В арифметическом сопроцессоре формирование команды происходит следующим образом:

F I C,

где F – обозначение команды арифметического сопроцессора; I – формат операнда, I – целочисленный формат (integer), для вещественных чисел буква I отсутствует; C – команда.

Например, FILD – загрузка целого числа в арифметический сопроцессор (LD – load), для вещественного числа команда загрузки – FLD.

Расширенные форматы данных арифметического сопроцессора требуют введения в ассемблер специальных директив определения данных. Резервирования памяти для переменных и констант сопроцессора осуществляют следующие директивы:

DW – определить слово (16 битов);

DD – определить двойное слова (32 бита);

DQ – определить счетверенное слово (64 бита);

DT – определить 10 байтов.

Основные характеристики численных данных приведены в табл. 29.

Таблица 29

Формат	Диапазон	Точность	Особенность
Целое слово	10^{4}	16 битов	Дополнительный код
Короткое целое	10^{9}	32 бита	Дополнительный код
Длинное целое	10^{19}	64 бита	Дополнительный код
Упакованное десятичное	10^{18}	18 цифр	Прямой код
Короткое вещественное	$10^{\pm 38}$	24 бита	Неявный бит F_0
Длинное вещественное	$10^{\pm 308}$	54 бита	Неявный бит F_0
Временное вещественное	$10^{\pm 4932}$	64 бита	Явный бит F_0

Необходимо помнить, что с операндами размером 1 байт арифметический сопроцессор не работает. Для представления чисел в вещественном формате, как правило, используют директивы DD и DQ.

*7.2. Система команд сопроцессора

Команды передачи данных. Команды этой группы производят передачу данных между регистрами стека, а также между вершиной стека и памятью. Одной командой число из памяти, представленное в любом формате сопроцессора, преобразуется во временный вещественный формат (10 байтов) и загружается (включается) в стек. Аналогичным образом, но в обратном порядке осуществляется передача числа в память.

Команды загрузки. В арифметическом сопроцессоре существует три команды загрузки следующего вида:

FLD	src	загрузка вещественного числа (src – источник);
FILD	src	загрузка двоичного целого числа;
FBLD	src	загрузка десятичного целого числа.

Команды загрузки осуществляют декремент указателя стека и передачу в новую вершину стека содержимое источника, т. е. производят включения в стек. Например, даны три числа:

```
A    DQ   5.672
B    DW   342
C    DT   10
```

Программа выглядит следующим образом:

```
FLD     A
FILD    B
FLD     ST
FBLD    C
```

После выполнения программы состояние сопроцессора будет:

ST(0)	10
ST(1)	342
ST(2)	342
ST(3)	5.672

В команде FLD источником может быть один из регистров стека или память, а в командах FILD и FBLD – только память, причем команда FBLD поддерживает только формат DT. Команды FILD и FBLD являются точными, т. е. в них ошибки округления отсутствуют.

Для загрузки данных из массива чаще всего используют косвенную адресацию. Например, если определен массив данных

```
A    DQ   5.7649, 6.0, – 4.98E-6
```

загрузка данных в сопроцессор выглядит следующим образом:

```
LEA  SI, A
```

```
FLD  QWORD PTR [SI]
FLD  QWORD PTR [SI+8]
FLD  QWORD PTR [SI+16]
```

Состояние сопроцессора будет следующим:

ST(0)	5.7649
ST(1)	6.0
ST(2)	– 4.98E-6

Примечание. Во всех приводимых примерах при отсутствии явной спецификации типа операнда (QWORD PTR) предполагается, что он определен соответствующей директивой.

Команды запоминания. В арифметическом сопроцессоре существуют две команды запоминания:

FST запоминание вещественного числа (dst – приемник);
FIST запоминание двоичного целого числа.

Эти команды производят передачу содержимого вершины стека в память без изменения указателя стека ST.

В команде FST получателем может быть регистр стека или вещественная переменная в памяти (только короткая или длинная форма представления чисел, т. е. DW или DQ).

В команде FIST получателем является переменная в памяти, имеющая формат короткого целого или целого слова (DW или DD). Поскольку все числа в сопроцессоре хранятся в вещественном формате, при запоминании данных с помощью команды FIST округление производится в соответствии с управляющим словом сопроцессора (за исключением, если запоминается число, загруженное до этого с помощью команды FILD).

Примеры ассемблерных команд запоминания:

```
FST   ST(5)              ; Передает ST(5) в ST(0).
FST   QWORD PTR [SI]     ; Передает вещественное число в память.
FISTP QWORD PTR [SI]     ; Передать целое слово в память.
```

Рассмотренные команды не допускают получатель в форматах длинного целого (8 байтов), временного вещественного (10 байтов) и упакованного десятичного (10 байтов).

Команды запоминания с извлечением из стека. Три команды, помимо передачи данных из ST(0) в получатель, осуществляют извлечения из стека, а регистр, бывший вершиной стека, отмечается как пустой и производится инкремент указателя стека:

FSTP запоминание вещественного числа (dst – приемник);
FISTP запоминание двоичного целого числа.
FBSTP запоминание десятичного целого числа.

Действие этих команд эквивалентно действию команд загрузки с той лишь разницей, что данные из вершины стека удаляются. Примеры рассмотренных команд на ассемблере:

```
FSTP ST                  ; Извлечение из стека.
```

```
FISTP DWORD PTR [SI]      ; Передает короткое целое в память.
FBSTP MEAN                ; Передать десятичное целое в память.
```

Команды обмена. Команда обмена содержимого регистров

FXCH dst ST(0) ↔ dst

обменивает содержимое получателя ST(i) и вершины стека ST(0). При пустом поле операнд обменивает содержимое регистров ST(0) и ST(1). Примеры команды обмена:

```
FXCH ST(5)                ; Обменивает содержимое ST(5) и ST(0).
FXCH                      ; Обменивает содержимое ST(1) и ST(0).
```

Арифметические команды. Основные арифметические команды, используемые в сопроцессоре, ориентированы на разработку эффективных алгоритмов, что позволяет программисту минимизировать число обращений к памяти и оптимально использовать регистровый стек. Все арифметические команды, работающие с регистрами, выполняются только в вещественном формате.

Команды сложения. Операция сложения реализуется командами со следующими формами:

FADD	//src/dst, src	сложение вещественных чисел;
FADDP	dst, src	сложение вещественных чисел с извлечением из стека;
FIADD	src	сложение вершины стека с целым числом.

Команда FADD ST,ST производит удвоение вершины стека. Примеры использования команд сложения:

```
FADD ST(3),ST             ; ST(3) = ST(3) + ST(0).
FADD QWORD PTR [SI]       ; ST(0) = ST(0) + содержимое ячейки памяти
                          ; (вещественное число).
FIADD WORD PTR A          ; ST(0) = ST(0) + ячейка памяти (целое число).
FADD                      ; ST(0) = ST(0) + ST(1) (с выталкиванием).
FADDP ST(1), ST(2)        ; ST(0) = ST(1) + ST(2) (с выталкиванием).
FADD ST(2)                ; ST(0) = ST(0) + ST(2).
```

Команды вычитания. Вычитание dst = dst – src осуществляют следующие команды:

FSUB	//src/dst, src	вычитание вещественных чисел;
FSUBP	dst, src	вычитание вещественных чисел с извлечением из стека;
FISUB	src	вычитание из вершины стека целого числа.

Для вычитания dst = src – dst предназначены команды:

FSUBR	//src/dst, src	вычитание вещественных чисел;
FSUBRP	dst, src	вычитание вещественных чисел с извлечением из стека.

Примеры команд вычитания:

```
FSUB ST(3),ST             ; ST(3) = ST(3) – ST(0).
FSUB QWORD PTR [SI]       ; ST(0) = ST(0) – содержимое ячейки памяти
```

```
                                ; (вещественное число).
FISUB WORD PTR A        ; ST(0) = ST(0) – ячейка памяти (целое число).
FSUB                    ; ST(0) = ST(1) – ST(0) (с выталкиванием).
FSUBP ST(1), ST         ; ST(0) = ST(1) – ST(0) (с выталкиванием).
FSUB ST(2)              ; ST(0) = ST(0) – ST(2).
```

Команды умножения. Операция умножения реализуется следующими командами:

```
FMUL    //src/dst, src    умножение вещественных чисел;
FMULP   dst, src          умножение вещественных чисел с извлечением
                          из стека;
FMUL                      ST(0) = ST(0) * ST(1);
FIMUL   src               умножение вершины стека на целое число.
```

Примеры команд умножения:

```
FMUL ST(3),ST           ; ST(3) = ST(3) * ST(0).
FMUL QWORD PTR [SI]     ; ST(0) = ST(0) * содержимое ячейки памяти
                        ; (вещественное число).
FIMUL WORD PTR A        ; ST(0) = ST(0) * ячейка памяти (целое число).
FMUL                    ; ST(0) = ST(0) * ST(1) (с выталкиванием).
FMULP ST(1), ST(2)      ; ST(0) = ST(1) * ST(2) (с выталкиванием).
FMUL ST(2)              ; ST(0) = ST(0) * ST(2).
```

*Команды деления***.** Для выполнения операции деления предусмотрены команды:

```
FDIV    //src/dst, src    деление вещественных чисел;
FDIVP   dst, src          деление вещественных чисел с извлечением из
                          стека;
FIDIV   src               деление вершины стека на целое число.
```

Для деления dst = src / dst предназначены команды:

```
FDIVR   //src/dst, src    деление вещественных чисел;
FDIVRP  dst, src          деление вещественных чисел с извлечением из
                          стека.
```

Примеры команд деления:

```
FDIV ST(3),ST           ; ST(3) = ST(3) / ST(0).
FDIV QWORD PTR [SI]     ; ST(0) = ST(0) / содержимое ячейки памяти
                        ; (вещественное число).
FIDIV WORD PTR A        ; ST(0) = ST(0) / ячейка памяти (целое число).
FDIV                    ; ST(0) = ST(1) / ST(0) (с выталкиванием).
FDIVP ST(1), ST         ; ST(0) = ST(1) / ST(0) (с выталкиванием).
FDIV ST(2)              ; ST(0) = ST(0) / ST(2).
```

Дополнительные команды. К арифметическим командам также относятся семь дополнительных команд, имеющих безоперандную форму, т. е. работают с вершиной стека.

Команда FSQRT – извлечение квадратного корня, заменяет число, находящееся в вершине стека, значением квадратного корня. При использова-

нии этой команды необходимо помнить об области определения операнда (содержимое вершины стека должно быть больше или равно нулю).

Команда FSCALE – масштабирование интерпретирует содержимое регистра ST(1) как целое двоичное число и прибавляет его к смещенному порядку числа, находящегося в вершине стека:

FSCALE ; $ST(0) = ST(0) * 2^{ST(1)}$

Таким образом, команда FSCALE осуществляет быстрое умножение (когда $ST(1) > 0$) или деление (когда $ST(1) < 0$) содержимого вершины стека на целую степень 2. В этой команде предполагается, что масштабный коэффициент в ST(1) является целым числом в диапазоне $-2^{15} < ST(1) < 2^{15}$. Если он не является целым числом, но находится в указанном диапазоне и больше по абсолютному значению 1, в команде принимается ближайшее целое, меньшее по абсолютному значению исходного масштабного коэффициента (усечение дробной части).

Команда FPREM вычисляет частичный остаток от деления числа, находящегося в вершине стека ST(0), на следующий элемент стека ST(1) и загружает результат в ST(0):

FPREM ; $ST(0) = ST(0) – (q * ST(1))$,

где q – целое число. Другими словами, содержимое ST(1) выступает модулем в операции деления. Например, если $ST(1) = 2$, тогда получаем деление по модулю 2. Знак остатка имеет знак исходного делимого, т. е. содержимого ST(0).

Команда FRNDINT осуществляет округление числа, находящегося в вершине стека ST(0), до целого. Режим округления показывает поля RC в слове управления сопроцессора.

Команда FXTRACT выделяет компоненты числа с плавающей точкой, преобразует число, находящееся в вершине стека ST(0), в два числа, представляющих собой фактическое значение его порядка и мантиссы. Выделенный порядок заменяет исходный операнд в вершине стека, а мантисса включается в стек (с декрементом указателя стека). Например, если в ST(0) находилось число 5.766754, то после выполнения команды FXTRACT ST(0) = =1.4416885, а $ST(1) = 2$. Соответственно $5.766754 = 1.4416885*2^2$.

Две последние команды выполняют элементарные операции нахождения абсолютного значения и изменения знака числа, которое содержится в вершине стека:

```
FABS        ; Абсолютное значение числа.
FCHS        ; Изменение знака числа.
```

Команды сравнения. Команды данной группы предназначены для анализа числа в вершине стека (иногда по отношению к другому числу) и формирования кода условия в слове состояния сопроцессора. Проверить образованный код можно только центральным процессором.

Команда сравнения вещественных чисел имеет форму FCOM src и осуществляет сравнение содержимого вершины стека ST(0) и источника src. Источником может быть регистр стека или вещественное число в памяти (в

формате короткого или длинного вещественного). Если поле операнда пустое, производится сравнение ST(0) и ST(1). Операнды считаются несравнимы (C3, C1 = 11), когда хотя бы один из них имеет специальное значение (проективная бесконечность или не-число). Например, имеем число A = – 0.43 (A DQ – 4.3e-1), необходимо определить знак числа.

```
FLDZ                    ; Загрузка нуля в стек.
FLD  QWORD PTR A        ; Загрузка числа A в стек.
FCOM                    ; Сравнения чисел.
FSTSW AX                ; Чтение слова состояния в AX.
SAHF                    ; Запись содержимого регистра AH в
                        ; регистр флагов центрального процессора.
```

До выполнения программы состояния регистра флагов SW = 0000H, после выполнения программы SW = 3100H (C3 – C0 = 0001). После выполнения команды SAHF устанавливается флаг C регистра флагов (CF = 1), далее можно использовать команду условного перехода JC.

Команда FCOMP src сравнения и извлечения из стека действует аналогично команде FCOM, но дополнительно осуществляет извлечение из стека. Если в предыдущем примере вместо команды FCOM использовать FCOMP, тогда в стеке останется одно число, а регистр флагов будет равен SW = 3900H (TOP = 111).

При выполнении команды FICOM содержимое источника (память) интерпретируется как целое слово или короткое целое, преобразуется во временный вещественный формат и сравнивается с ST(0). Команда FICOMP src производит те же действия и дополнительно реализует извлечение из стека.

Команда FTST производит проверку числа, находящегося в вершине стека, посредством сравнения его с нулем. Результат фиксируется в слове состояния сопроцессора. Рассматривая предыдущий пример, можно написать программу:

```
FLD  QWORD PTR A        ; Загрузка числа A в стек.
FTST                    ; Сравнения числа с нулем.
FSTSW AX                ; Чтение слово состояния в AX.
SAHF                    ; Запись содержимого регистра AH в
                        ; регистр флагов центрального процессора.
```

После выполнения программы SW = 3900H (C3 – C0 = 0001).

Команды трансцендентных функций. Команды данной группы выполняют базовые вычисления, относящиеся к тригонометрическим, обратным тригонометрическим, логарифмическим и показательным функциям. Операнды команд находятся в одном или двух верхних регистрах стека и результат также возвращается в стек.

Предполагается, что операнды являются нормализованными числами и находятся в допустимом для каждой из команд диапазоне. Ответственность за удовлетворение этих требований возлагается на программиста.

Команда FPTAN – вычисление частичного тангенса, как результат формируется два числа X и Y, отношение которых дает тангенс угла α в виде

$\mathrm{tg}\,\alpha = \mathrm{Y}/\ \mathrm{X}$. Значение угла α должно находится в вершине стека ST(0) и быть в диапазоне от нуля до $\pi/4$ ($0 < \alpha < \pi/4$).

Примечание. Для арифметических сопроцессоров I80387 и более поздних модификаций команда FPTAN заменяет содержимое ST(0) на tg (ST(0)) и затем включает 1.0 в стек, т. е. результат выполнения функции находится в ST(1). Содержимое ST(0) должно быть выражено в радианах и находится в диапазоне от -2^{63} до 2^{63}. Если число находится вне допустимого диапазона, тогда флаг C 2 устанавливается в 1 и содержимое ST(0) не изменяется.

Команда FPATAN – вычисление частичного арктангенса формирует результат $\alpha = \mathrm{arctg}(Y/X)$, причем значение X берется из вершины стека ST(0), а значение Y – из регистра ST(1). Значения исходных операндов должны удовлетворять требованию $0 < Y < X < \infty$.

Примечание. Для арифметических сопроцессоров I80387 и более поздних модификаций для команды FPATAN содержимое ST(0) и ST(1) должно находится в диапазоне от -2^{63} до 2^{63}. Результат имеет тот же знак, что и операнд в ST(1), и величину меньше π.

Если обозначить через z аргумент обратной функции, через X и Y значения в ST(0) и ST(1) до выполнения команды FPATAN, то получение в ST(0) значения осуществляется по следующим формулам:

$$\arcsin(z) = arctg\left(\frac{z}{\sqrt{(1-z)(1+z)}}\right) = arctg\left(Y/X\right);$$

$$\arccos(z) = 2\,arctg\left(\sqrt{\frac{1-z}{1+z}}\right) = 2arctg\left(Y/X\right);$$

$$arctg(z) = arctg\left(z/1\right) = arctg\left(Y/X\right),\ arcctg(z) = arctg\left(1/z\right) = arctg\left(Y/X\right);$$

$$arc\sec(z) = arctg\left(\frac{1}{\sqrt{(z-1)(z+1)}}\right) = arctg\left(Y/X\right);$$

$$\arccos\mathrm{ec}(\mathrm{z}) = 2\,\mathrm{arctg}\left(\sqrt{\frac{\mathrm{z}-1}{\mathrm{z}+1}}\right) = 2\mathrm{arctg}\left(\mathrm{Y}/\mathrm{X}\right).$$

Например, программа для вычисления arcsin(z):

```
FLD QWORD PTR z        ; Загрузка в ST(0) аргумента
FLD1                   ; Загрузка 1
FSUB QWORD PTR z       ; Вычисление 1–z
FPATAN                 ; Вычисление arcsin(z)
```

Результат находится в ST(0).

Команда FSIN заменяет содержимое ST(0) на sin(ST(0)). Для арифметических сопроцессоров I80387 и более поздней модификации аргумент выражен в радианах и должен удовлетворять диапазону от -2^{63} до 2^{63}.

Команда FCOS производит вычисление косинуса угла. Данная команда работает аналогично команде FSIN.

Команда FSINCOS вычисляет значение синуса и косинуса угла. Результатом выполнения будет в ST(0) косинуса угла, а в ST(1) – синуса.

Команда F2XM1 вычисляет значение функции $Y = 2^x - 1$. Значение ST(0) должно находится в диапазоне $-1 < ST < 1$. Используя команду F2XM1 можно осуществить возведение в степень X любых чисел, пользуясь формулами:

$$10^x = 2^{x \log_2 10}, \quad e^x = 2^{x \log_2 e}, \quad Y^x = 2^{x \log_2 Y}.$$

Необходимые для таких вычислений константы $\log_2 10$ и $\log_2 e$ встроены в сопроцессор.

Команда FYL2X предназначена для вычисления значений функции $Z=Y\log_2 X$. Аргумент X находится в вершине стека, а аргумент Y – в ST(1). Диапазон изменения чисел $0 < X < \infty$, $-\infty < Y < \infty$. С помощью данной команды удобно вычислять логарифмы по любому основанию с применением тождества $\log_n X = \log_n 2 * \log_2 X$.

Команда FYL2XP1 вычисляет значение функции $Z = Y \log_2(X+1)$. Аргумент X берется из вершины стека, а Y – из ST(1). Значение X должно находится в диапазоне $-\left(1-\frac{\sqrt{2}}{2}\right) \le X \le \sqrt{2}-1$, а Y в диапазоне $-\infty < Y < \infty$.

Команды загрузки констант. Простые команды загрузки наиболее часто встречающихся в вычислениях констант приведены в табл. 30. Загрузка осуществляется путем включения константы в стек, т. е. декремента указателя стека, и передачи в новую вершину стека значения константы, представленной во временном вещественном формате.

Таблица 30

Мнемоника	Операция
FLDZ	Загрузка в стек +0.0
FLD1	Загрузить в стек +1.0
FLDPI	Загрузить в стек π
FLDL2T	Загрузить $\log_2 10$
FLDL2E	Загрузить $\log_2 e$
FLDLG2	Загрузить $\log_{10} 2$
FLDLN2	Загрузить $\log_e 2$

Команды управления сопроцессором. С помощью команд управления сопроцессором можно задать режим работы сопроцессора, а также проанализировать результаты команд сравнения и проверки.

Команда FINIT инициализации сопроцессора функционально эквивалентна сигналу сброса CLR, но она не влияет на синхронизацию выборки команд центрального процессора и сопроцессора. Состояние сопроцессора после команды инициализации приведено в табл. 31.

Таблица 31

Поле	Состояние	Интерпретация
Слово управления		
Управление бесконечностью Управление точностью Управление округлением Маска разрешения прерываний	0 11 00 1	Проективная 64 бита Округление к ближайшему Прерывания запрещены
Слово состояния		
Код условия Указатель стека Запрос прерывания Флажки особых случаев	XXXX 000 0 000000	Не определен Начало стека Отсутствует
Слово тэгов		
Тэги	11	Пустые регистры

Команда инициализации чаще всего используется в начале программы и при необходимости очистки стека. Так же для очистки стека часто используется команда FNINIT, которая быстрее выполняется по сравнению с командой FINIT.

Основное назначение команды FLDCW src заключается в загрузке в регистр управления сопроцессора нового содержимого из источника src (им должно быть целое число из памяти) с целью установки или изменения режима работы, например режима управления округлением.

Команды FSTCW dst и FSTSW dst осуществляют запоминание текущих слов управления и состояния соответственно в ячейке памяти, определяемой получателем. Команда FSTSW dst допускает использование в качестве приемника слова состояния сопроцессора регистр AX центрального процессора. Данная команда часто используется для реализации условных переходов по результатам команд сопроцессора, а также вызова процедур обработки особых случаев в таких системах, где не применяются прерывания. После команды чтения состояния необходимо использовать команду WAIT, чтобы состояние сопроцессора было передано до продолжения программы.

Команды FDECSTP и FINCSTP производят декремент и инкремент стека соответственно.

С помощью команд FDISI и FENI программист управляет запрещением (маскированием) и разрешением прерываний от сопроцессора. Обе команды соответствующим образом воздействуют на бит IEM маски разрешения прерываний в слове управления сопроцессора.

Функция команды FCLEX сброса особых случаев заключается в том, чтобы сбросить в нуль флажки особых случаев.

Лекция № 21

7.3. Специальные числовые значения и особые случаи

Для расширения вычислительных возможностей сопроцессора в нем наряду с обычными целыми и вещественными числами предусмотрены кодирование и обработка нескольких специальных значений. К специальным значениям относятся вещественные числа с нарушением нормализации, нули и псевдонули, бесконечности, не-числа и неопределенности.

7.3.1. Вещественные числа с нарушением нормализации

Сопроцессор хранит ненулевые вещественные числа в нормализованной форме, т. е. старший бит мантиссы, являющийся разрядом целой части, содержит единицу. Отсутствие старших нулей в мантиссе, имеющей в каждом формате фиксированную длину, позволяет сохранить максимум значащих цифр. Например, число, хранящееся в сопроцессоре 2.359256745865683E–3 имеет больше значащих цифр, чем число 0.235925674586568E–4. Обозначим минимально представимое число δ, тогда для представления чисел в диапазоне 0 – δ приходится нарушать нормализацию, т. е. допускать числа, мантисса которых содержит один или несколько старших нулей. Такие числа возникают, когда результат операции слишком мал для представления в нормализованной форме. Например, при хранении в сопроцессоре во временном вещественном формате числа 1.375543754123429E–4934 произойдет нарушение нормализации, т. к. это число будет представлено в стеке в виде 0.013755437541234E–4932. Наличие старших нулей в мантиссе расширяет диапазон в области малых чисел, но за счет некоторой потери точности, т. к. количество значащих цифр мантиссы уменьшается.

Денормализованные числа

Денормализованное число возникает в результате реакции сопроцессора на замаскированный особый случай антипереполнения (исчезновение порядка). Антипереполнение появляется, когда абсолютное значение вещественного числа становится слишком малым для представления в формате получателя, т. е. истинный порядок нормализованного результата слишком отрицателен. Например, получение истинного –102 вызывает антипереполнение, когда получатель имеет формат короткого вещественного (минимальное число в этом формате 10^{-38}). В форматах длинного и временного веществено-го истинный порядок – 102 не приводит к антипереполнению.

Во многих компьютерах реакцией на антипереполнение является возвращение нулевого результата, т. е. все числа меньше δ, считаются нулем (так называемый машинный нуль). Когда особый случай антипереполнения не замаскирован, сопроцессор генерирует сигнал прерывания INT и обработка специального случая возлагается на подпрограмму обслуживания преры-

вания. Если особый случай замаскирован, потеря значимости происходит постепенно, при этом результат денормализуется до тех пор, пока смещенный порядок не станет нулевым. Денормализация осуществляется путем инкремента порядка и сдвиг мантиссы вправо с введением нулей в ее старшие разряды.

В большинстве случаев денормализация встречается крайне редко, т. к. очень малыми, как правило, бывают промежуточные данные, а не конечные результаты, представляемые короткими или длинными вещественными числами. Если использовать для промежуточных результатов временный вещественный формат, его огромный диапазон делает антипереполнение маловероятным.

7.3.3. Ненормализованные числа

Ненормализованным числом является результат операции с привлечением денормализованного операнда и оказывается следствием маскированной реакции сопроцессора на антипереполнение. Ненормализованные числа существуют только во временном вещественном формате; их смещенный порядок может иметь любое ненулевое значение, бит целой части мантиссы равен нулю, а остальные биты мантиссы произвольны (но не все нули). Ненормализованное число в регистре сопроцессора отмечается тэгом действительного (разрешенного) числа.

Ненормализованные числа позволяют продолжать арифметические операции после антипереполнения, сохраняя при этом свое отличительное свойство как чисел с уменьшенной значимостью. Но если ненормализованное число оказывает несущественное воздействие на вычисление с нормализованными числами, результат будет нормализованным. Например, сложение небольшого ненормализованного числа с большим нормализованным числом дает нормализованный результат.

7.3.4. Нули и псевдонули

Число нуль (или истинный нуль) в десятичном и вещественном форматах может иметь положительный или отрицательный знак, но знак двоичного целого нуля всегда положителен. В вычислениях положительные и отрицательные нули интерпретируются одинаково. При загрузке истинного нуля в регистр (FLDZ) его содержимое отмечается специальным тэгом.

Во временном вещественном формате предусмотрен специальный класс значений, называемых псевдонулями. Псевдонуль – это ненормализованное число с нулевой мантиссой и ненулевым смещенным порядком (истинный нуль имеет нулевой смещенный порядок). Псевдонуль может получиться при умножении двух ненормализованных операндов, в которых суммарное число старших нулевых битов мантиссы сомножителей больше 64 ($0.0*10^{-4932}$).

Псевдонули-операнды обрабатываются как ненормализованные числа, за исключением следующих команд и операций, в которых они аналогичны истинным нулям:

- команды сравнения и проверки;
- команда FRNDINT округления до целого;
- операции деления, в которых делимое является истинным нулем или псевдонулем, а делитель – псевдонуль.

При сложении и вычитании псевдонуля и истинного нуля или второго псевдонуля псевдонули ведут себя как ненормализованные числа.

Лекция № 22

7.3.5. Бесконечности

В вещественных форматах предусмотрено представление специальных значений, называемых бесконечностями. Они кодируются с максимальным смещенным порядком 11…11 и мантиссой 1 00…0. При загрузке бесконечности в регистр его содержимое отмечается тегом специального значения.

Таблица 32

Истинный результат		Режим округления	Получаемый результат
Нормализация	Знак		
Нормализован	+	Вверх	Плюс бесконечность
Нормализован	+	Вниз	Наибольшее положительное число
Нормализован	–	Вверх	Наименьшее отрицательное число
Нормализован	–	Вниз	Минус бесконечность
Не нормализован	+	Вверх	Плюс бесконечность
Не нормализован	+	Вниз	Наибольший порядок, мантисса результата
Не нормализован	–	Вверх	То же
Не нормализован	–	Вниз	Минус бесконечность

Сопроцессор может образовать бесконечность как результат маскированной реакции на особые случаи переполнения или деления на ноль. При округлении вверх или вниз маскированная реакция может сформировать вместо бесконечности наибольшее допустимое число, представленное в формате получателя (табл. 32).

Поведение бесконечностей как операндов команд сопроцессора зависит от содержимого поля управления бесконечностью в слове управления. При задании проективной модели бесконечность имеет одно беззнаковое представление, поэтому бесконечность нельзя сравнивать ни с каким значением, кроме бесконечности. В аффинном режиме учитываются знаки бесконечностей и сравнение оказывается возможным. Обозначаются бесконечности как –INF и +INF.

7.3.6. Не-числа

Не-число является элементом класса специальных значений, которые существуют только в вещественных форматах. Не-число имеет любой знак, максимальный смещенный порядок и любую мантиссу (кроме мантиссы, зарезервированной для бесконечности 100….00). Наличие не-числа в регистре отмечается тэгом специального значения.

Сопроцессор формирует особое не-число, называемое вещественной неопределенностью, в результате маскированной реакции на особый случай недействительной операции. Например, при попытке выполнить какую-либо арифметическую операцию с пустым регистром. Если этот особый случай замаскирован, сопроцессор возвращает как результат не-число, а если оба операнда являются не-числа, результатом будет не-число с большей мантиссой. Таким образом, полученное однажды не-число распространяется в вычислениях и выдается как окончательный результат. Обозначаются не-числа как –NAN и +NAN.

8.МИКРОПРОЦЕССОРЫ КЛАССА PENTIUM

8.1. Основные особенности архитектуры и программирования процессоров I80186 и I80286

Выше подробно рассматривалась архитектура микропроцессоров I8086/8088. Посвятим несколько слов дальнейшему развитию этой ветви, кратко рассмотрев основные архитектурные особенности микропроцессоров серии Intel. Все вычислительные устройства имеют некоторые общие и индивидуальные свойства архитектуры. Большинство типов ЭВМ построено на фон-неймановских правилах, основанных на следующем:

принципе хранимой программы, согласно которому программа и данные находятся в одном адресном пространстве;

линейное пространство памяти – совокупность ячеек памяти, которым последовательно присваиваются адреса;

последовательное выполнение программ, которое может быть изменено только командами условного и безусловного перехода и командами вызова подпрограмм.

Процессоры I80186/I88 не представляют нового поколения архитектуры. У них разрядность шины адреса – 20 бит, шины данных – 16 бит (у 80188 – 8 бит). Эти процессоры имеют встроенные контроллеры прерываний, прямого доступа к памяти, трехканальный таймер и генератор синхронизации. В них сокращено число тактов, необходимых для выполнения некоторых команд. Есть модификации со встроенными последовательными портами.

Для расширения вычислительных возможностей процессоров 8086/8088, 80186/80188 фирмой INTEL был разработан сопроцессор 8087, получивший название NPX (Numeric Processor Extension). Его применение

добавляет 68 мнемоник, включающих арифметические, тригонометрические, экспоненциальные и логарифмические инструкции.

Процессор 80286 выпущен в 1982 году и представляет второе поколение 16-разрядных процессоров. Его существенным отличием является механизм управления адресацией памяти, обеспечивающий поддержку виртуальной памяти, а также средства для переключения задач (Task switching). К командам процессора 8086 добавляется несколько новых команд. Процессор может работать в двух режимах:

9086 Real Address Mode – реальный режим, полностью совместимый с 8086;

Protected Virtual Address Mode – защищенный режим виртуальной адресации. В этом режиме процессор позволяет адресовать до 16 Мб физической памяти, через которые при страничной адресации могут отображаться до 1 Гб виртуальной памяти каждой задачи.

За счет архитектуры МП 80286 с тактовой частотой 12,5 МГц работает более чем в 6 раз быстрее, чем 8086 с тактовой частотой 5 МГц. Параллельно разработан числовой процессор 80287, программно совместимый с 8087. Шина адреса разрядностью 24 бита позволяет адресовать 16 Мб физической памяти, но в реальном режиме доступен только 1 Мб, начинающийся с младших адресов.

Регистры 80286 в реальном режиме практически совпадают с регистрами 8086, есть некоторые изменения в регистре флагов. Защищенный режим еще не достаточно совершенен. Как и 8086/8088, процессор 80286 может обслуживать до 256 типов прерываний, они делятся на аппаратные, программные и исключения. Два первых типа прерываний аналогичны уже известным, исключения случаются при появлении особых условий при выполнении операций (в 8086/8088 аналогом исключений были внутренние прерывания процессора).

8.2. Реальный и защищенный режим

Изучив принципы работы микропроцессоров I8086 и I8088, нет необходимости пояснять особенности этого режима. До недавнего времени это был единственный режим, в котором функционировала операционная система MS-DOS. Для неё был разработан большой объём программного обеспечения, поэтому фирма Intel во всех модернизациях своего микропроцессора поддерживает этот режим. Его основные характеристики следующие :

пространство оперативной памяти делится на сегменты по 64 Кбайта. Сегменты в памяти могут перекрываться;

страничное преобразование адреса запрещено, то есть физический адрес равен линейному и формируется как сумма двух составляющих: 16-разрядного эффективного адреса и 20-разрядного результата сдвига содержимого конкретного сегментного регистра на 4 разряда влево;

максимальное значение физического адреса равно 0FF FFFh, то есть 1 Мбайт, но, фактически, в реальном режиме адресуется на 64 Кбайт больше, что следует из следующего вычисления:

FFFF0 – максимальное значение сегментной части адреса, сдвинутое на 4 разряда влево;

+

0FFFF – максимальное значение смещения;

10FFEF = 1 114 096 байт – максимальный физический адрес в реальном режиме.

Защищенный режим (Protected Mode) предназначен для обеспечения независимости выполнения нескольких задач, т.е. ресурсы одной задачи защищены от воздействия другой задачи. Основным защищаемым ресурсом является память, в которой хранятся коды, данные и различные таблицы. Защита памяти основана на использование сегментации.

Сегмент, как известно – это блок адресного пространства памяти определенного назначения. Максимальный размер сегмента для процессоров 8086 и 80286 составляет 64 Кб, в 32-разрядных процессорах он отодвинулся до 4 Гб. Сегменты памяти выдаются задачами операционной системы, но в реальном режиме любая может переопределить значение сегментных регистров, задающих положение сегмента в памяти, и «залезть» в чужую область данных или кода. В защищенном режиме сегменты тоже распределяются операционной системой, но прикладная программа сможет использовать только разрешенные для нее сегменты памяти, выбирая их с помощью **селекторов из таблиц дескрипторов** сегментов. Появляются отличия в определении сегментных регистров. Сегментные регистры CS, DS, SS и ES хранят не сами базовые адреса сегментов, а селекторы, по которым из таблицы, хранящейся в ОЗУ, извлекаются дескрипторы сегментов. Появились два новых термина, которые в дальнейшем используются во всех типах процессоров и требуют пояснения.

Селектор – это 16-разрядный указатель, загружаемый в сегментной регистр. С его помощью выбирается определенный дескриптор из таблицы дескрипторов, хранящихся в ОЗУ. Селектору имеют три поля:

TI (3р) – индикатор использования одной из таблиц дескрипторов, локальной или глобальной;

RPL (0-2р) – запрашиваемый уровень привилегий. Используется механизмом защиты системы;

NDEX (4-15р) – номер дескриптора в таблице.

Процессор умножает индекс восемь (это длина дескриптора) и добавляет результат к базовому адресу таблицы.

Дескрипторы хранятся в таблицах дескрипторов. Под каждой дескриптор отводится четыре смежных слова (8 байт). В дескрипторе указываются базовый адрес сегмента, размер сегмента, уровень защиты, размер операндов, тип сегмента и т.д. Дескрипторы 16 и 32-разрядных процессоров отличаются разрядностью поля базового адреса (24 бита для 80286 и 32 бита для

80386 процессоров) и трактовкой поля лимита (т.е. длины сегмента), которое должно обеспечивать размер сегмента до 64 Кб или 4 Гб соответственно. Два старших байта у дескрипторов 80286 всегда нулевые (заранее зарезервированы для старших процессоров).

Защищенные режим предоставляет средства **переключения задач.** Состояние каждой задачи (значение всех связанных с ней регистров процессора) может быть сохранено в специальном сегменте состояния задач (*TSS*), на который указывает селектор в регистре задачи. При переключении задач достаточно загрузить новый селектор в регистр задачи, и состояние предыдущей задачи автоматически сохранится в сегменте состояния задачи (*TSS*).

8.3. Основные особенности архитектуры и программирования 32-разрядных процессоров

Первым 32-разрядным процессором был Intel 386, выпущенный в 1985 году. Он имел 32-разрядные раздельные шины адреса и данных. Позже, в 1988 году, был разработан вариант процессора Intel 386 с 16-разрядной шиной данных и 24-разрядной шиной адреса, а полноразрядный вариант получил название Intel 386 DX. Как и в случае с 8088, это было сделано с целью удешевления компьютера, собранного на этом процессоре, хотя производительность компьютера снижалась приблизительно в два раза по сравнению с DX на той же тактовой частоте. Программные модели этих процессоров одинаковы.

В 1990 году появился процессор Intel 386 со средствами управления энергопотреблением, разработанными специально для портативных компьютеров. В комплекте с ним выпускался набор периферии. По меркам сегодняшнего дня такие процессоры трудно считать высокопроизводительными, хотя на этом процессоре «вышел в люди» сначала оболочка. А потом *ОС MS WINDOWS*, существенно изменив облик персональных компьютеров.

В 32-разрядных процессорах обеспечена программная совместимость со своими 16-разрядными предшественниками, но в них преодолено очень жесткое ограничение на длину сегмента памяти – 64 Кб. В защищенном режиме оно отодвинулось до предела физически реализуемой памяти – 4 Гб. Эти процессоры имеют поддержку виртуальной памяти объемом 64 Тб, встроенный блок управления памятью поддерживает механизм сегментации и страничной трансляции адресов. Процессоры обеспечивают четырехуровневую систему защиты памяти и устройств ввода-вывода (система привилегий), переключение задач.

Процессоры могут работать в двух режимах: реальном и защищенном. В реальном режиме возможна адресация до 1 Мб физической памяти, т.е. он работает как быстрый 8086, в защищенном режиме, как уже говорилось, до 4 Гб физической памяти и до 16 Тб виртуальной памяти каждой из 4-ех задач.

Весовым дополнением является режим виртуального процессора 8086 (Virtual 8086 Mode). Этот режим служит особым состоянием защищенного режима, в котором процессор функционирует как 8086. На одном процессоре в таком режиме могут одновременно исполняться несколько задач с изолированными друг от друга реальными ресурсами. Вычисление физического адресного пространства памяти производится механизмами сегментации и трансляции страниц.

В архитектуру процессоров введены средства отладки и тестирования.

Существенным недостатком восьми и 16-разрядных микропроцессоров было их невысокое быстродействие, объясняемое не только низкой тактовой частотой, но и несовершенством архитектуры. Существенную роль в повышении быстродействия сыграла **конвейеризация** вычислений. Наиболее общий прием конвейеризации заключается в том, что во время выполнения предыдущей команды выбирается и декодируется следующая. Для 32-разрядных процессоров характерна суперскалярная архитектура. Скалярным называют процессор с единственным конвейером. Конвейер – это специальное устройство, реализующее такой метод обработки команд в микропроцессоре, при котором выполнение команды разбивается на несколько этапов, при этом одновременно может выполняться ряд команд. Все процессоры Intel до 486 включительно относятся к скалярным, 1486 имеет пятиступенчатый конвейер, включающий пять этапов:

выборку команды;

декодирование;

вычисление адреса операндов;

выполнение операции;

запись результатов.

При использовании конвейера в МП на разной стадии выполнения могут находиться пять команд. Суперскалярный (superscalar) процессор имеет более одного конвейера, способных обрабатывать команды параллельно. Pentium является двухпотоковым процессоров (имеет два конвейера), Pentium Pro трехпотоковым, т.е. он может выполнять параллельно три выбранные команды.

8.4. Кэш-память

Следует остановиться на таком понятии, как **кэширование** памяти. Большое внимание на производительность вычислителей фон-неймановской архитектуры оказывают время доступа к памяти и ее пропускная способность. Кэширование – это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в кэш-памяти. Кэш-память представляет собой быстродействующее запоминающее устройство, размещенное в одном кристалле с процессором или же внешнее по отношению к процессору. Она служит высокоскоростным буфером между процессором и относительно медленной памятью. В процессоре I80386 использовалось только внешнее кэширование, организованное на дорогих микросхемах

SRAM. Идея кэш-памяти основана на предвосхищении наиболее вероятного использования процессором информации из основной памяти путем копирования ее в кэш-память. Следовательно, возникают понятия «кэш-попадание» и «кэш-промах». Эффективность кэш-попаданий выражается отношением успешных обращений к общему количеству обращений к кэш-памяти.

Предсказание адреса следующего обращения в память было бы невозможно, если бы программы обращались к памяти абсолютно произвольным образом. На самом же деле программы имеют тенденцию обращаться к памяти по адресу, близкому к адресу предыдущего обращения. Этот принцип называется **локальностью программы** или **локальностью ссылок.** Даже небольшая кэш-память объемом 64 Кб обеспечивает коэффициент попадания до 90%, что оправдывает расходы. В таблицах приведенынекоторые данные по размерам и эффективностью обращения кэш-памяти, а также сделано сравнение обращений.

В таблице 33 приведено соотношение размера кэш-памяти с её эффективностью.

Таблица 33

РАЗМЕР КЭШ-ПАМЯТИ	ПОПАДАНИЯ	ВЫИГРЫШ
Нет КЭШ, DRAM с двумя T_w	0%	0%
16к	81%	35%
32к	86%	38%
64к	88%	39%
128к	89%	39%
Нет КЭШ, SRAM без T_w	100%	47%

В таблице 34 приведены результаты сравнения обращений.

Таблица 34

ДЕЙСТВИЕ	ТАКТЫ
Некэшированное считывание:	
Выдача адреса в память	1
Ожидание RAM	2
Данные возвращаются	1
Всего:	4т.(2 T_w)
Кэшированное считывание (попадание):	

Выдача адреса в КЭШ-память Проверка КЭШ, выдача данных Всего:	 1 1 2т.(нет T_w)
Кэшированное считывание (промах): Выдача адреса в КЭШ-память Проверка КЭШ, нет соответствия Выдача адреса в память Ожидание RAM Данные возвращаются Всего:	 1 1 1 2 1 6т.(4 T_w)

Внутреннее кэширование обращений к памяти применяется в процессорах, начиная с I80486. В нем содержится один блок встроенной кэш-памяти размером 8 Кб, который используется для кэширования и кодов, и данных. Pentium имеет два блока кэш-памяти по 8 Кб, один для кода, другой для данных. Процессор становится более интеллектуальным. Pentium Pro также имеет две ступени встроенной кэш-памяти. Первый уровень кэширования состоит из четырехканального наборно-ассоциативного кэша инструкций и двухканального наборно-ассоциативного КЭШа данных. Длина строки КЭШа – 32 байта. Вторичный кэш, также смонтированный на процессоре, скрывает многие промахи первичного. В случае промаха на обоих уровнях минимальная задержка доставки данных из DRAM составляет 11-14 тактов одновременный доступ по записи и чтению, если эти запросы относятся к разным банкам кэш-памяти.

8.5. Архитектура микропроцессоров семейства Pentium

Процессоры Pentium представляют пятое поколение процессоров формы Intel. По архитектуре и системе команд они совместимы с 32-разрядными процессорами, но имеют 64-битную шину данных, из-за чего их иногда ошибочно называют 64-разрядными. По сравнению с предыдущими поколениями, процессоры Pentium имеют следующие отличия:

суперскалярная архитектура: процессор имеет два параллельно работающих конвейера обработки команд (U-конвейер с полным набором инструкций и V-конвейер с несколько ограниченным набором). Благодаря этому процессор способен одновременно выполнять две команды (при специальном режиме компиляции программного обеспечения);

применение технологии динамического предсказания ветвлений вместе с внутренним КЭШем команд объемом 8 Кб обеспечивает максимальную загрузку конвейеров;

внешняя шина данных ради повышения производительности имеет разрядность 64 бит, что требует соответствующей организации памяти;

встроенный сопроцессор за счет улучшения архитектуры в 2-10 раз превосходит сопроцессор 486 по производительности;

введено несколько инструкций. В том числе распознавание семейства и модели CPU;

введены средства управления энергопотреблением

Обратимся к структурной схеме микропроцессора, представленной на рис.82 .

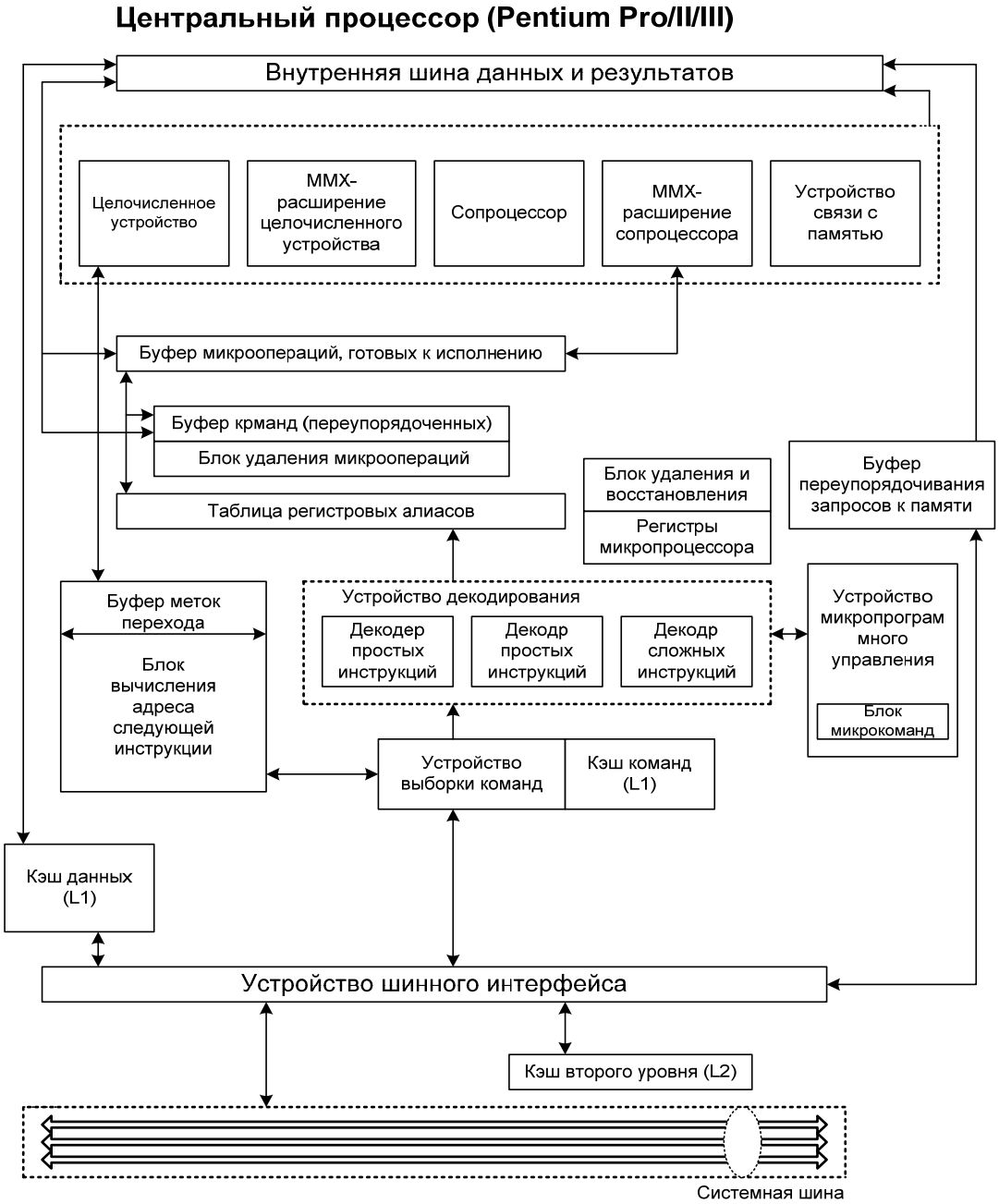


Рис. 82. Структурная схема микропроцессора семейства P6 (Pentium Pro/II/III)

В структурную схему входят:

подсистема памяти;

блок выборки/декодирования;

буфер переупорядоченных команд;

устройство диспетчеризации/исполнения.

Подсистема памяти состоит из КЭШа первого уровня L_1 (команд и данных), кэша второго уровня, устройства шинного интерфейса и системной шины, устройства связи с памятью и буфера переупорядочивания запросов к памяти. Устройство шинного интерфейса обращается к оперативной памяти системы через 64-разрядную внешнюю шину. Во время обслуживания одного запроса формируется ряд других, обрабатываемых в порядке поступления. Любой внутренний запрос процессора на обращение к памяти направляется во внутренний кэш. Если адресуемая область представлена в кэш-памяти (случай попадания – cache hit), запрос на чтение обслуживается только кэш-памятью, не выходя на внешнюю шину. Запрос на запись изменяет строку кэш, и в зависимости от политики записи либо сразу выходит на внешнюю шину (при сквозной записи), либо несколько позже (при использовании алгоритма обратной записи). При записи в кэш-память данных процессор использует один из следующих способов модификации данных:

сквозная запись, при которой запись в строку кэш обновляет кэш-память и основную;

обратная запись, при которой модифицируется только строка в кэш-памяти. Этот способ записи приводит к сокращению за счет исключения ненужной записи в основную память. Перезапись модифицированной строки из кэш-памяти в основную производится посредством обратной записи, производимой при удалении строки из кэш-памяти.

В случае промаха (Cache Miss) запрос на запись направляется только на внешнюю шину, а запрос на чтение обслуживается сложнее. Если этот запрос относится к кэшируемой области памятью, то выполняется цикл заполнения целой строки КЭШа (т.е. для Pentium – 32 байта) из оперативной памяти самым быстрым способом (64-битными передачами).

Запросы на данные из памяти в исполнительном устройстве обеспечивается с помощью устройства связи с памятью и буфера переупорядочивания запросов к памяти. Буфер отслеживает все запросы данных; если таковые отсутствуют в кэш-памяти данных первого уровня (L_1), инициируется обращение к кэшу второго уровня (L_2). Если в кэше L_2 операндов не оказалось, буфер переупорядочивания запросов к памяти заставляет устройство шинного интерфейса сформировать запрос к оперативной памяти.

Блок выборки/декодирования состоит из устройства выборки команд, буфера предсказаний переходов, декодера инструкций, блока микропрограммного управления и таблицы регистровых алиасов. Поток команд из КЭШа команд (L_1) декодируется в последовательность микроопераций при помощи трех параллельно работающих декодеров. Микрооперации выполняются пятью исполнительными устройствами, работающими параллельно. Порядком исполнения микроопераций занимается блок микропрограммного

управления. Декодер команд может формировать до шести микроопераций за такт. После декодирования команд порядок выполнения микроопераций трудно предсказать. Поэтому могут возникнуть трудности с использованием регистров. Эта проблема называется проблемой ложных взаимозависимостей и реализуется механизмом переименования регистров. Для этого в вычислительных процессах используется дополнительный набор из 40 внутренних регистров. Информация о действительных именах регистров процессора иих внутренних именах (номерах универсальных регистров) помещается в таблицу регистровых алиасов. С помощью этой таблицы микрооперации готовятся к неупорядоченному выполнению.

Буфер переупорядоченных команд содержит команды, переупорядоченные для оптимальной загрузки конвейера. Он представляет собой массив памяти, выполненный в виде 40 регистров. В них находятся микрокоманды, ожидающие своей очереди на исполнение и уже частично выполненные, но из-за переупорядочивания исполнения команд не до конца.

Устройство диспетчеризации/исполнения содержит буфер микроопераций, готовых к исполнению два исполнительных устройства для целочисленных операций, два – для чисел с плавающий точкой и устройство связи с памятью (это деление достаточно условно). Такая организация позволяет за один такт выполнить пять микроопераций. Устройство диспетчеризации/исполнения может выбирать микрооперации из буфера переупорядоченных команд в любом порядке. Порядок исполнения микроопераций определяет специальный буфер, называемый буфером команд, готовых к исполнению. Микрооперации, готовые к работе, отправляются в соответствующие исполнительные устройства. Результаты исполнения возвращаются в буфер переупорядоченных команд, где будут храниться до тех пор, пока не будут удалены устройством удаления и восстановления.

Два целочисленных исполнительных устройства могут параллельно обрабатывать две целочисленные микрооперации. Исполнительные устройства с плавающей точкой обрабатывают вещественные числа.

Устройство связи с памятью управляет загрузкой и сохранением данных для микроопераций. Последним в работу схемы выполнения команд исходной программы вступает блок удаления и восстановления, восстанавливающий исходную последовательность команд. Он постоянно проверяет буфер переупорядоченных команд для обнаружения полностью выполненных микроопераций. Такие микрооперации удаляются из буфера и восстанавливаются в исходном порядке. При этом результаты операций записываются в реальные регистры микропроцессора и в оперативную память.

На этом пока завершим обсуждение общих вопросов, связанных с архитектурой. При разработке программ каждый программист получает в свое распоряжение определенный ресурс микропроцессора, необходимый для выполнения и хранения в памяти команд, данных и служебной информации. Набор этих ресурсов представляет программную модель микропроцессора. На рис. 83 представлена программная модель микропроцессора Pentium Ш.

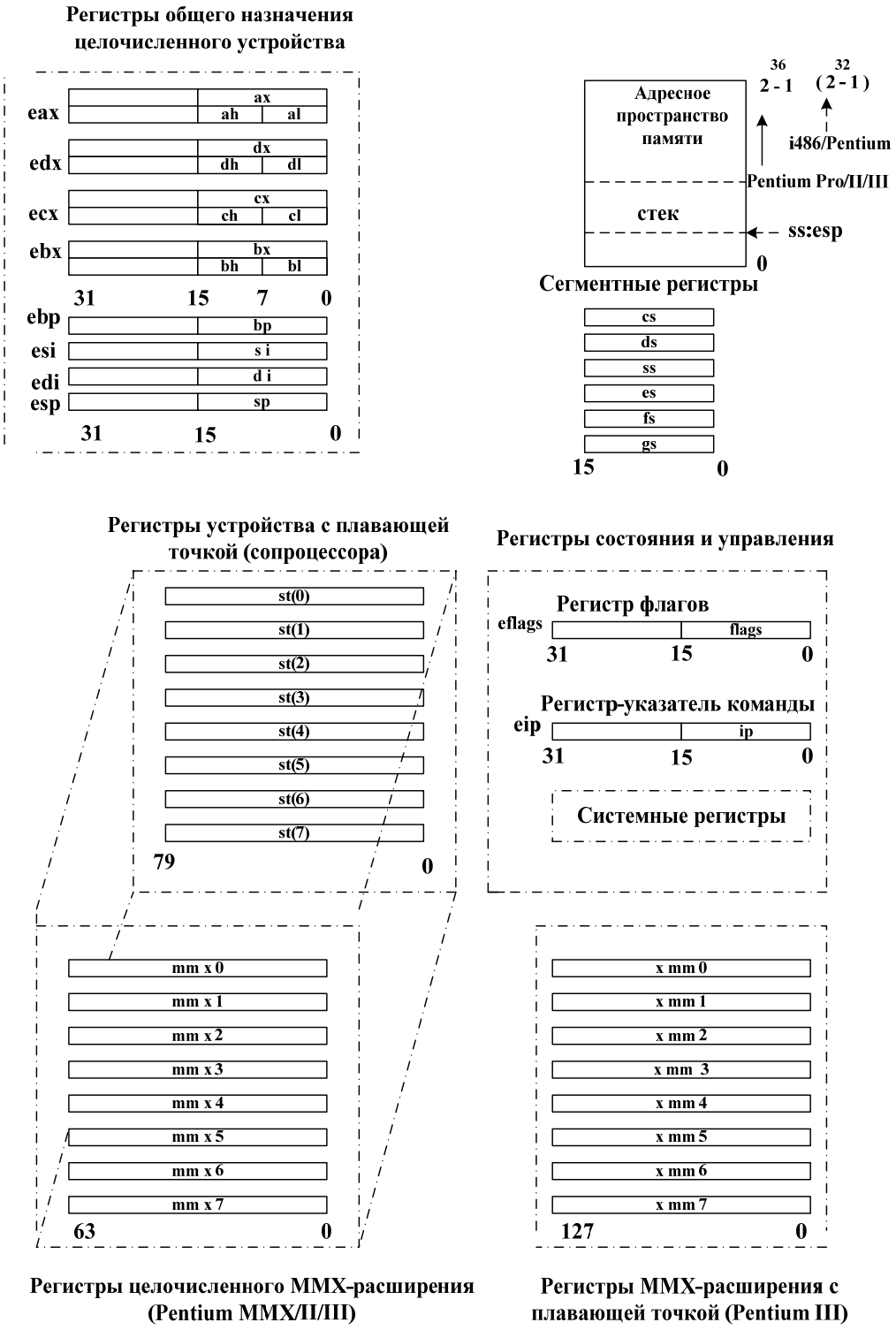


Рис. 83. Программная модель микропроцессора Pentium III

Следует отметить, что возросшая производительность процессора Pentium требует и соответствующей организации системы на его основе. Компания Intel разработала и поставляет все необходимые для этого наборы микросхем. Прежде всего, для согласования скорости с динамической основной памятью необходима кэш-память второго уровня. Контроллер кэш-

памяти 82496 и микросхемы статической памяти 82491 обеспечивают построение такой кэш-памяти объемом 256 Кбайт и работу процессора без тактов ожидания. Для эффективной организации систем Intel разработала стандарт на высокопроизводительную локальную шину PCI. Выпускаются наборы микросхем для построения мощных компьютеров на ее основе.

В настоящее время компания Intel разработала новый процессор, продолжающий архитектурную линию x86. Этот процессор получил название P6. Число транзисторов в новом кристалле составляет около 5 миллионов, что обеспечивает повышение производительности до уровня 200 MIPS (66 МГц Pentium имеет производительность 112 MIPS). Для достижения такой производительности использованы технические решения, широко применяющиеся при построении RISC-процессоров:

- выполнение команд не в предписанной программой последовательности, что устраняет во многих случаях приостановку конвейеров из-за ожидания операндов операций;
- использование методики переименования регистров, позволяющей увеличивать эффективный размер регистрового файла (малое количество регистров - одно из самых узких мест архитектуры x86);
- расширение суперскалярных возможностей по отношению к процессору Pentium, в котором обеспечивается одновременная выдача только двух команд с достаточно жесткими ограничениями на их комбинации.

Кроме того, в борьбу за новое поколение процессоров x86 включились компании, ранее занимавшиеся изготовлением Intel-совместимых процессоров. Это компании Advanced Micro Devices (AMD), Cyrix Corp и NexGen. С точки зрения микроархитектуры наиболее близок к Pentium процессор M1 компании Cyrix. Также как и Pentium он имеет два конвейера и может выполнять до двух команд в одном такте. Однако в процессоре M1 число случаев, когда операции могут выполняться попарно, значительно увеличено. Кроме того, в нем применяется методика обходов и ускорения пересылки данных, позволяющая устранить приостановку конвейеров во многих ситуациях, с которыми не справляется Pentium. Процессор содержит 32 физических регистра (вместо 8 логических, предусмотренных архитектурой x 86) и применяет методику переименования регистров для устранения зависимостей по данным. Как и Pentium, процессор M1 для прогнозирования направления перехода использует буфер целевых адресов перехода емкостью 256 элементов, но кроме того поддерживает специальный стек возвратов, отслеживающий вызовы процедур и последующие возвраты.

Процессоры K5 компании AMD и Nx586 компании NexGen используют в корне другой подход. Основа их процессоров – очень быстрое RISC-ядро, выполняющее высокорегулярные операции в суперскалярном режиме. Внутренние форматы команд (ROP у компании AMD и RISC86 у компании NexGen) соответствуют традиционным системам команд RISC-процессоров. Все команды имеют одинаковую длину и кодируются в регулярном формате. Обращения к памяти выполняются специальными командами загрузки и за-

писи. Как известно, архитектура x 86 имеет очень сложную для декодирования систему команд. В процессорах K5 и N x 586 осуществляется аппаратная трансляция команд x86 в команды внутреннего формата, что дает лучшие условия для распараллеливания вычислений. В процессоре K5 имеются 40, а в процессоре N x 586 22 физических регистра, которые реализуют методику переименования. В процессоре K5 информация, необходимая для прогнозирования направления перехода, записывается прямо в кэш команд и хранится вместе с каждой строкой кэш-памяти. В процессоре Nx586 для этих целей используется кэш-память адресов переходов на 96 элементов.

Таким образом, компания Intel больше не обладает монополией на методы конструирования высокопроизводительных процессоров x86, и можно ожидать появления новых процессоров, не только не уступающих, но и возможно превосходящих по производительности процессоры компании, стоявшей у истоков этой архитектуры.

*8.6. Модели памяти: плоская, страничная, сегментная

Проектировщиком систем не базе микропроцессора 80386 предоставлен широкий выбор типов памяти. Для каждого типа архитектуры памяти характерен соответствующий режим работы процессора: защищенный с полной сегментацией памяти, расслоенной памяти с подкачкой станиц по требованию и расслоенной памяти с традиционной адресацией.

Микропроцессор 80386 поддерживает 16 000 сегментов различного объёма. Базовый адрес сегмента находится в таблице дескрипторов., и его положение в таблице определяется 14 старшими разрядами сегментного регистра. Размер каждого сегмента может достигать 4 Гбайт, это позволяет реализовать управление виртуальной памятью микропроцессора 80386 емкостью до 64 Тбайт.

Виртуальная память. Наличие механизма виртуальной памяти делает возможным использование программ и структур данных, размеры которых превышают действительный объём физической памяти. Например, микропроцессор 80386 обладает огромным потенциалом: емкость его виртуальной памяти равна 64 Тбайт.

Для реализации механизма виртуальной памяти определяется логическое адресное пространство. Если размер этого пространства превышает размер физического адресного пространства, то вместо ОЗУ используются внешние накопительные устройства. Логический адрес преобразуется в действительный физический аппаратным способом. Логическое адресное пространство разделяется на размещаемые блоки (сегменты или страницы), которыми и осуществляется обмен между оперативной памятью и вспомогательным запоминающим устройством. Так как все эти действия управляются средствами операционной системы, то процессы перекачки кодов и данных остаются скрытыми от прикладной программы.

Сегментация памяти. Если виртуальная память разбивается на сегменты, то каждому модулю программных кодов и данных может быть присвоен свой собственный логический сегмент памяти. Это будет способствовать достаточно простой реализации механизмов, обеспечивающих защиту отдельных модулей, разделение информации между сегментами, а также совместную или раздельную их обработку. в микропроцессоре 80386 допускаются сегменты размером до 4 Гбайт, причем под каждую задачу может выделяться 16 000 сегментов.

Страничная организация памяти. Виртуальная память может быть поделена также на страницы. В отличие от сегментов, для которых допускаются переменные размеры и размещение в оперативной памяти, наиболее приемлемое для программных модулей, страницы имеют фиксированный размер 4 Кбайт и жесткую привязку к адресам памяти. Страничная организация памяти придает алгоритмам перекачки данных в процедурах размещения, запоминания и поиска более рациональную форму благодаря равномерному распределению блоков памяти в адресном пространстве. В любой программе можно объединить основные принципы каждого из рассмотренных способов управления памятью, если, допустим, логическое адресное пространство разделить на сегменты, а для управления физической памятью применить методы страничной организации.

Линейная плоская адресация. Метод линейной адресации является наиболее простым. Согласно этому методу в микропроцессоре блокируется механизм виртуальной памяти, а все команды, осуществляющие операции над кодами программ, данными и действия со стеком, включаются в один сегмент. Размеры этого сегмента могут достигать объёма всего пространства физической памяти, адресуемой микропроцессором (т.е. до 4 Гбайт). Метод линейной адресации не требует никаких преобразований адресов с помощью диспетчера памяти, поскольку любой логический адрес равен действительному физическому адресу.

9. СЕКЦИОНИРОВАННЫЙ МИКРОПРОЦЕССОРНЫЙ КОМПЛЕКТ БИС К1804

Возможности однокристальных микропроцессоров ограничены аппаратурными ресурсами кристалла. С помощью секционированных микропроцессорных комплектов увеличивают разрядность обрабатываемых слов, наращивая количество секций. Микросхемы микропроцессорного комплекта К1804 предназначены для построения микропроцессорных устройств с разрядно-модульной организацией. Большинство БИС серии К1804 – 4-х разрядные микропроцессорные секции, используемые либо для обработки четырех разрядов данных, либо для выполнения определенного набора управляющих функций. Секционность БИС комплекта обеспечивает возможность построения на их основе микропроцессорных устройств с разрядностью, кратной четырем при параллельном включении БИС. Управление работой

отдельных БИС комплекта осуществляется программно, что позволяет реализовать практически любой набор.

Комплект БИС можно разделить на две группы: БИС для построения операционной части микропроцессора и БИС для построения управляющей части микропроцессора. В первую группу входят: микропроцессорные секции К1804ВС1 и К1804ВС2, схема переноса К1804ВР1, 4-разрядный параллельный регистр К1804ИР1 и схема управления состоянием и сдвигами К1804ВР2. Вторую группу составляют: секции управления адресом микрокоманды К1804ВУ1 и К1804ВУ2, схема управления следующим адресом К1804ВУ3 и схема управления последовательностью микрокоманд К1804ВУ4.

Рассмотрим назначение и структурные схемы некоторых БИС этой серии.

9.1. Микропроцессорная секция К1804ВС1

Микропроцессорная секция К1804ВС1 предназначена для построения процессоров с длиной слова, кратной четырем. Она рассчитана на микропрограммное управление, имеет двухвходовое АЛУ, двухпортовую регистровую память 16 х 4, рабочий регистр ***Q*** , тристабильную выходную шину Y_{3-0} и две пары двунаправленных тристабильных линий сдвига.

Структурная схема секции показана на рис. 84. Арифметическо-логическое устройство выполняет восемь арифметических и логических операций над двумя операндами, подаваемыми на входы ***R*** и ***S*** через селектор источников данных, и формирует четыре признака результата: перенос из старшего разряда ***C4***- «переполнение ***OVR***, знак ***F3*** и признак нулевого результата ***Z***. Источниками данных (операндов) могут быть: входная шина данных, содержимое одного или двух внутренних регистров, содержимое регистра ***Q*** или нулевая константа. Выбор операндов и требуемой операции АЛУ производится с помощью разрядов микрокоманд ***I0, I1, I2*** и ***I3, I4, I5*** соответственно. Источники операндов и перечень операций АЛУ приведены в табл. 34 (***C0*** – вход переноса).

Результат операций выдается на выходную шину данных через селектор данных, управляемый разрядами ***I6, I7, I8*** микрокоманды (табл. 35).

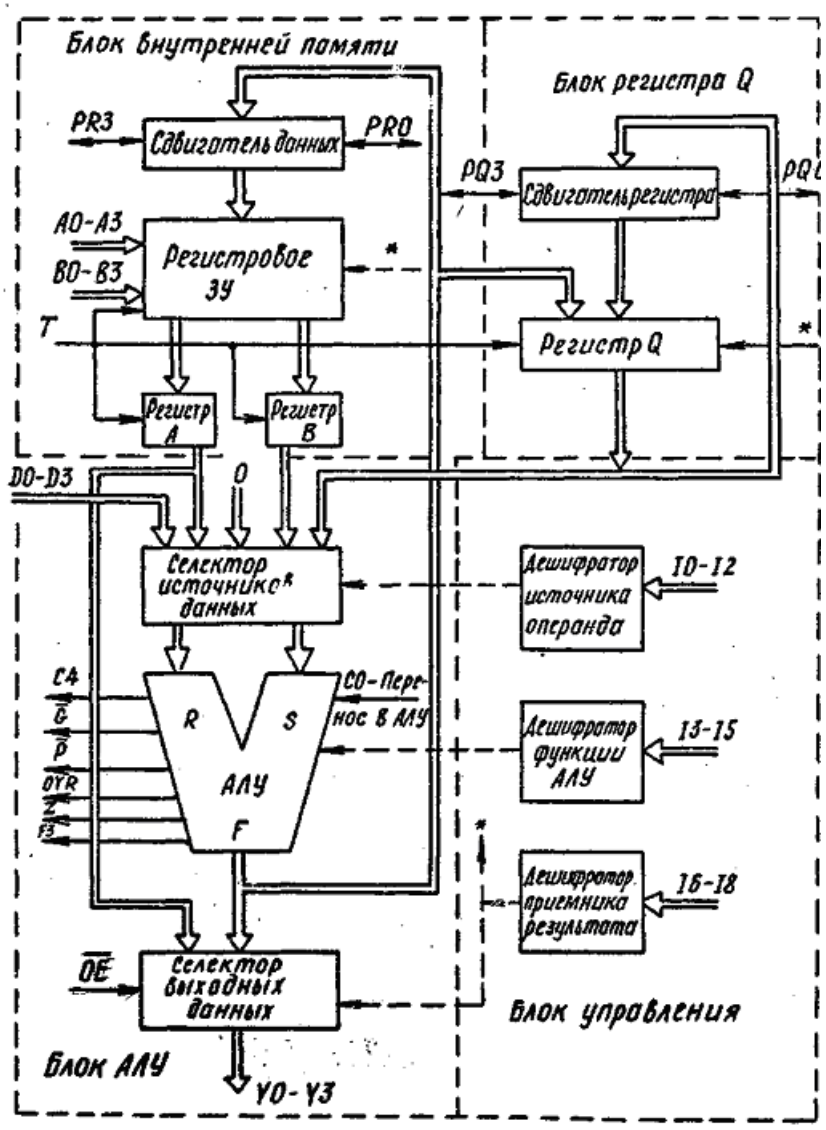


Рис. 84. Структурная схема микропроцессорной секции К1804ВС1

Таблица 34

Источники операндов АЛУ					Перечень операций АЛУ			
Микрокоманда			Источники		Микрокоманда			Функция АЛУ
I2	I1	I0	R	S	I5	I4	I3	
0	0	0	A	Q	0	0	0	R+S+CO
0	0	1	A	B	0	0	1	S-R-1+CO
0	1	0	0	Q	0	1	0	R-S-1+CO
0	1	1	O	B	0	1	1	RVS
1	0	0	O	A	1	0	0	RΛS
1	0	1	D	A	1	0	1	$\overline{R}AS$
1	1	0	D	Q	1	1	0	R⊕S
1	1	1	D	O	1	1	1	$\overline{R \oplus S}$

Здесь "x" обозначает неопределенное состояние. Для шины V подразумевается, что сигнал разрешения выхода $\bar{O}\bar{E}$ равен нулю. Сигналы (C4) и входы (CO) переносов АЛУ обеспечивают возможность комплектования БИС в блоки с разрядностью, кратной четырем. Ускоренный перенос при комплектовании БИС организуется при помощи внешних схем с использованием сигналов $\bar{P}$ и $\bar{G}$.

Таблица 35

Микро команда			Функция памяти		Функция регистра Q		Выход Y	Сдвигатель АЛУ		Сдвигатель Q	
I8	I7	I6	Сдвиг	Загрузка	Сдвиг	Загрузка		PR_3	PR_O	PR_3	PR_0
0	0	0	Нет	Нет	Нет	F→Q	F	x	x	x	x
0	0	1	Нет	Нет	Нет	Нет	F	x	x	x	x
0	1	0	Нет	F → B	Нет	Нет	A	x	x	x	x
0	1	1	Нет	F → B	Нет	Нет	F	x	x	x	x
1	0	0	Вправо	F/2→ B	Вправо	Q/2→Q	F	Вхз	F_0	Bx_3	Q_0
1	0	1	Вправо	F/2→ B	Нет	Нет	F	Вхз	F_0	x	Q_0
1	1	0	Влево	2F → B	Влево	2Q→Q	F	F3	Bx_0	Q_3	Bx_0
1	1	1	Влево	2F →• B	Нет	Нет	F	F3	Bx_0	Q_3	x

Блок внутренней памяти содержит 16 четырехразрядных регистров и имеет два независимых канала адресации – шины адреса регистра внутренней памяти ***A*** и внутренней памяти ***B***. Причем запись в регистры возможна только при адресации по шине ***B***. Информация поступает в регистры с выхода АЛУ через сдвигатель данных, который обеспечивает три режима записи: без сдвига, со сдвигом на один разряд влево и со сдвигом на один разряд вправо. При выполнении операции сдвига в сдвигателе данных формируется сигнал переноса ***PR3*** или ***PRO***.

Информация с выхода АЛУ может быть записана и в дополнительный рабочий регистр ***Q*** . Содержимое регистра ***Q*** можно также сдвинуть на один разряд влево или вправо.

На рис.85 приведены обозначение и нумерация выводов БИС К1804ВС1. Назначение выводов микросхемы показано в табл.36.

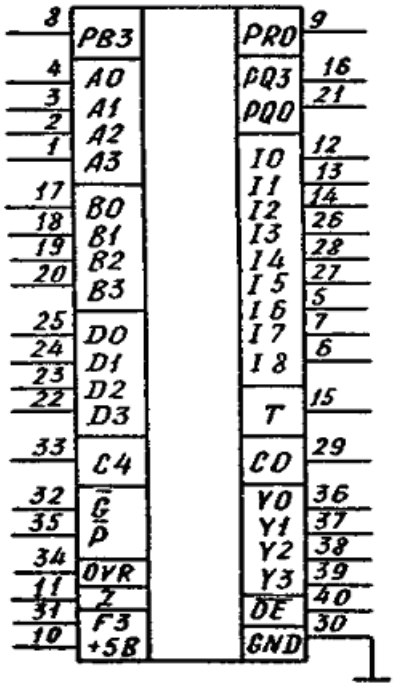


Рис. 85. Обозначение и нумерация выводов БИС К1804ВС1

Таблица 36

Вывод	Назначение
A0, A1 A2, A3 B0, B1	Входные линии адресов регистров внутренней памяти А и В
I0, I1 I2, I3 I4, I5 16, 17	Входные линии микрокоманды, определяющей действия секции в текущем микроцикле
D3 - D0	Входные линии данных от внешних источников
Y3 - Y0	Выходная шина секции
$\bar{O}\bar{E}$	Сигнал разрешения выхода. При высоком уровне буферы выходной шины находятся в высоко-импедансном состоянии, а при низком – выводят содержимое выходной шины АЛУ или регистра А
T	Вход тактирующего сигнала
CO	Вход переноса АЛУ
C4	Выход переноса АЛУ, используется только в случае последовательного переноса, при организации ускоренного переноса не используется
OVR	Выходной сигнал переполнения АЛУ
F3	Значение старшего бита результата АЛУ. В дополнительном коде совпадает со знаком
Z	Выходной сигнал о получении нулевого результата АЛУ (F - 0). Выход типа "открытый коллектор"
$\bar{P}$, $\bar{G}$	Выходы распространения и генерирования переноса, предназначенные для схем ускоренного переноса

PR3, PRO	Двунаправленные тристабильные линии сдвигателя на выходе АЛУ (или на входе пямяти)
PQ3, PQO	Двунаправленные тристабильные линии сдвигателя регистра

Для каскадирования секций микросхем К1804ВС1 в 16-разрядный процессор (рис. 86) применяется схема ускоренного переноса 1804ВР1, изображенная на рис.87.

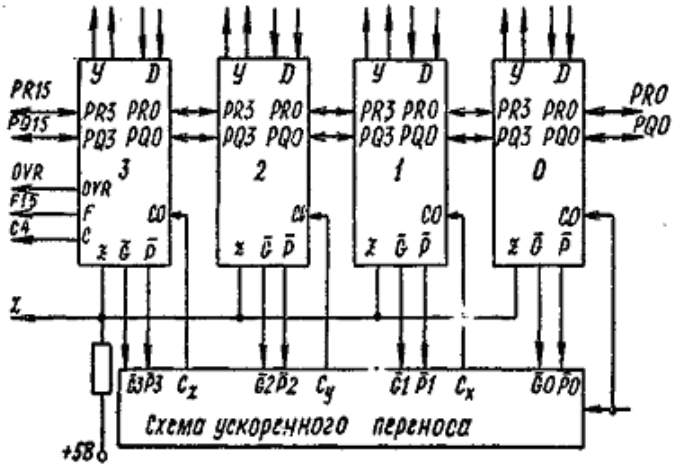


Рис. 86. Каскадирования секций микросхем К1804ВС1 в 16-разрядный процессор

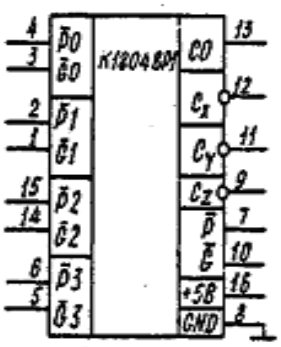


Рис. 87. Схема ускоренного переноса 1804ВР1

Обычно сигналы ***I, A, B, $\bar{O}\bar{E}$*** и ***T*** подаются параллельно во все секции, а большинство сигналов состояний берется из старшей серии.

Микропроцессорная секция К1804ВС1 может быть разделена на четыре блока: блок внутренней памяти, блок рабочего регистра ***Q*** , блок АЛУ и блок управления.

Рассмотрим их построение и функционирование.

Блок внутренней памяти включает в себя регистровое запоминающее устройство, содержащее 16 четырехразрядных регистров. Адреса регистров представляются 4-разрядными кодовыми комбинациями. Два адресных входа **АО-А3** и **ВО-В3** регистрового запоминающего устройства, на которые информация поступает из микрокоманды, определяют адреса любой пары реги-

стров, содержимое которых принимается регистрами **A** и **B.** Далее эти регистры служат источниками операндов, над которыми выполняются операции.

Запись в регистровое запоминающее устройство (**ЗУ**) в каждом тактовом периоде может производиться лишь в один из регистров, адрес которого задается шиной **B0-B3**. Записываемые данные при этом подаются с выхода АЛУ через сдвигатель данных. Они могут передаваться без сдвига либо со сдвигом на один разряд влево или вправо. Таким образом, за один тактовый период из регистрового ЗУ может быть выдано содержимое двух регистров; над ними в АЛУ выполнена некоторая операция, и полученный результат может быть сдвинут вправо или влево и вновь записан в регистр регистрового ЗУ. Выводы ***PR0*** и ***PR3*** в зависимости от направления сдвига служат входом или выходом, через которые производятся запись значения в освобождающийся при сдвиге разряд и выдача содержимого выдвигаемого разряда.

Чтение из регистров регистрового ЗУ, адресуемых шинами **A3-A0** и **B3-B0**, происходит при высоком уровне тактового сигнала. Его вход отключен и не реагирует на поступающую информацию. При низком уровне тактового сигнала входы регистров **A** и **B** отключаются и регистры хранят принятую информацию. При этом в регистровое ЗУ производится запись информации через сдвигатель данных по адресу **B0-B3**.

Таким образом, чтение, и запись информации в регистровом ЗУ разнесены во времени.

Блок рабочего регистра содержит одиночный 4-разрядный регистр ***Q***, построенный на триггерах ***D***-типа. Содержимое регистра постоянно передается в АЛУ. Запись в регистр может производиться на положительном фронте тактовых импульсов. Данные на вход регистра передаются через узел сдвигателя регистра ***Q*** , который передает записываемые в регистр данные без сдвига либо со сдвигом влево или вправо на один разряд. На вход регистра ***Q*** может передаваться результат операции с выхода АЛУ или содержимое самого регистра ***Q***. Последнее обеспечивает возможность выполнения сдвига содержимого регистра ***Q***, производимого параллельно с операцией в АЛУ.

Блок АЛУ включает в себя АЛУ, которое имеет два 4-разрядных входа ***R*** и ***S*** Данные на эти входы поступают с выхода селектора источников данных. Кроме этих входов АЛУ имеет вход для подачи переноса ***C0***.

На вход ***R*** АЛУ селектор источников данных коммутирует выход регистра A или внешнюю шину данных DO ~ D3 либо передает на этот вход нулевое значение. На вход ***S*** селектор источников данных коммутирует или выход регистра A, или выход регистра B, или выход регистра ***Q*** , или нулевое значение.

Результат операции с выхода АЛУ подается на сдвигатель данных, регистр Q и селектор выходных данных. Последний коммутирует в выходную шину данных содержимое регистра A или выход АЛУ. Он построен на элементах с тремя состояниями и управляется сигналом $\overline{OE}$.

Блок управления предназначен для преобразования содержимого кода операции ***I0-I8*** микрокоманды в систему управляющих сигналов, под дейст-

вием которых в узлах микросхемы выполняются микрооперации.

Таким образом, для управления рассмотренными процессами требуется микрокоманда, типичный формат которой имеет вид, приведенный на рис. 88, а микроцикл изображен на рис.89.

Рис. 88. Формат микрокоманды

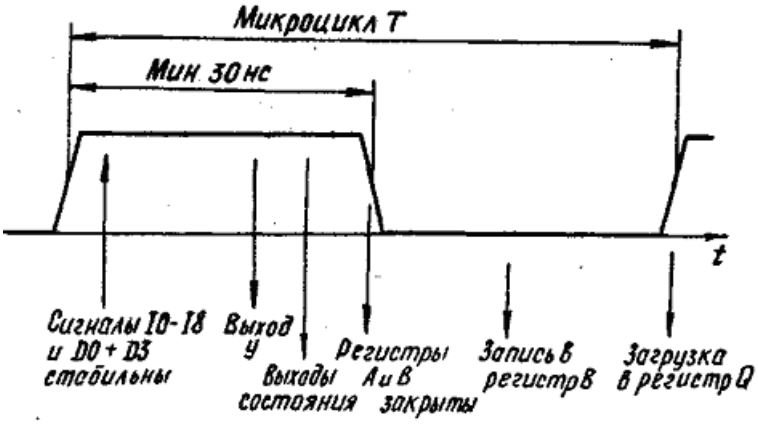


Рис. 89. Микроцикл

Микропроцессорная секция К1804ВС2 состоит из тех же функциональных узлов, что и БИС К1804ВС1, однако функциональные возможности БИС К1804ВС2 шире.

Лекция № 25

9.2. Объединение микропроцессорных секций в операционном устройстве

Требуемая разрядность операционного устройства обеспечивается объединением некоторого числа микропроцессорных секций. Каждая секция хранит и обрабатывает 4-разрядную группу данных; если используется ***n*** микропроцессорных секций, то разрядность операционного устройства равна ***4n*** . На рис. 90 показано объединение четырех секций в 16-разрядном операционном устройстве.

Одной из задач, которые приходится решать при объединении секции, является обеспечение малого времени задержки переноса, поступающего на вход СО секций. Один из возможных способов построения цепи передачи переносов при объединении микропроцессорных секций — последовательный при котором выход ***C4*** секции подключается к входу ***C0*** следующей, более старшей секции. При этом на вход ***C0*** каждой секции сигнал переноса

поступает с задержкой, с которой проходят переносы через все предыдущие секции. Эта задержка в цепи от входа ***C0*** до выхода ***C4*** в одной секции составляет 20 нс. При большем числе объединяемых функций задержка существенно отразится на быстродействии операционного устройства. Уменьшение задержки в формировании и подаче переносов на входы ***C0*** микропроцессорной секции обеспечивает применение схемы ускоренного переноса К1804ВР1. Информация, необходимая для формирования переносов в данной микросхеме, выдаваемых, на входы ***C0*** секций, подается в виде сигналов с выходов $\bar{P}$ и $\bar{G}$ секций.

В операционном устройстве, состоящем из четырех секций, при последовательной передаче переносов время распространения сигнала от входов ***A3 — A0*** и ***B3 — B0*** до выхода ***C4*** переноса в первой секции составляет 70 нс, далее задержка во второй и третьей секциях 20х2=40 нс; таким образом, искомая задержка 110 нс.

В схеме с ускоренным переносом время распространения сигнала от входов ***A3 — A0*** и ***B3 — B0*** до выходов $\bar{P}$ и $\bar{G}$ составляет 59 нс, задержка в схеме ускоренного переноса 5 нс. Таким образом, общая задержка равна 64 нс.

Другая задача, решаемая при объединении микропроцессорных секций, состоит в построении цепей передачи переносов при выполнении операций сдвигов. Если производится сдвиг вправо, то выдвигаемое из секций на выходы ***PR0*** и PQO содержимое младших разрядов должно передаваться на входы ***PR3*** и ***PQ3*** следующих младших секций для ввода их в освобождающиеся при сдвиге старшие разряды регистров. При сдвиге влево из секций на выходы ***PR3*** и ***PQ3*** выдвигается содержимое старших разрядов, оно должно вдвигаться через входы ***PR0*** и ***PQ3*** в освобождающиеся при сдвиге младшие разряды следующих старших микропроцессорных секции. Таким образом, при объединении секций необходимо обеспечить соединение выводов ***PR0*** и ***PQ3*** младшей секции с выводами ***PR0*** и ***PQ3*** следующей старшей секции.

Третья задача, решаемая при построении операционного устройства, – формирование слова состояния (признаков, предназначенных для выполнения условных переходов). При объединении секций нужно объединить выходы ***Z*** секций и подключить через регистр к источнику питания. Такая объединенная цепь служит выходом признака нуля. В качестве остальных выводов признаков используются выходы признаков из старшей микропроцессорной секции.

Выходы признаков остальных секций остаются неиспользованными. Слово состояния операционного устройства формируется объединенным выходом признака нуля, выходами признаков старшей секции, выходами ***PR0*** и ***PQ0*** младшей секции и выходом ***PR3*** старшей секции. В зависимости от решаемой задачи разрядность слова состояния и схема его формирования могут быть различными. Они определяются проектировщиком и решаются с применением мультиплексоров.

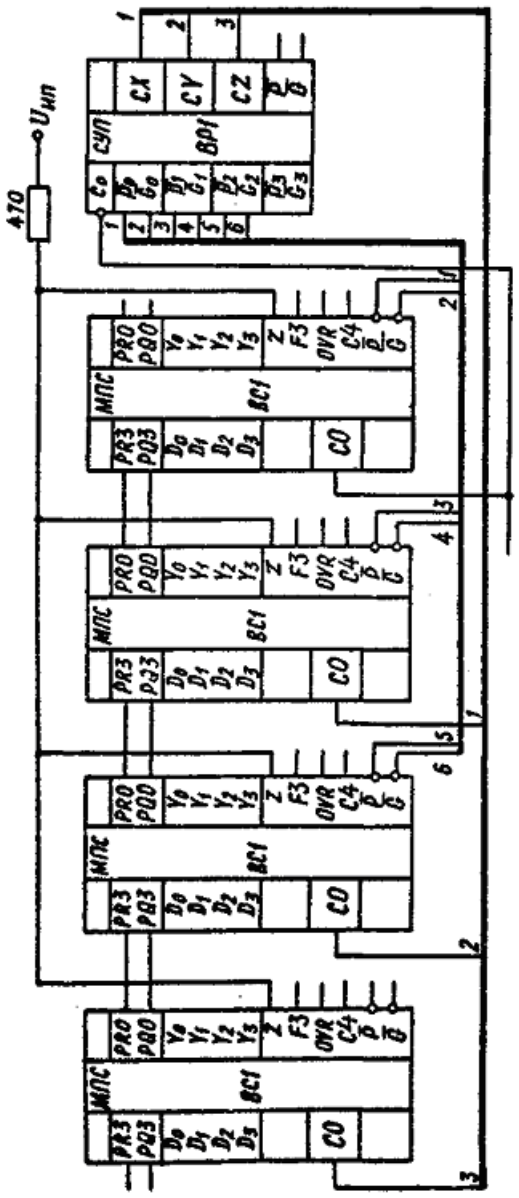


Рис. 90. объединение четырех секций в 16-разрядном операционном устройстве.

9.3. Схема управления состоянием и сдвигами К1804ВР2

БИС К1804ВР2 предназначена для выполнения различных операций по обслуживанию АЛУ:

формирования сигналов входного переноса для микропроцессорных секций и схем ускоренного переноса;

организации арифметических, логических и циклических сдвигов чисел одинарной и двойной длины (всего 32 варианта сдвигов);

выполнения различных операций с содержимым двух внутренних регистров состояния ***M*** и ***N***;

проверки за один такт одного из 16 различных условий, поступающих с выходов регистров состояния ***M*** и ***N*** либо из микропроцессорной секции.

В БИС К1804ВР2 можно выделить следующие функциональные блоки: блок обработки признаков, блок проверки условия, блок управления переносом и блок управления сдвигами. Детальное изучение этой микросхемы в нашу задачу не входит.

9.4. Построение управляющего устройства

9.4.1. Секции управления адресом микрокоманды К1804ВУ1 и К1804ВУ2

Четырехразрядные секции К1804ВУ1 и К1804ВУ2 используются для построения блоков микропрограммного управления различных цифровых устройств. Основной функцией этих БИС является формирование адреса микрокоманды под воздействием внешних управляющих сигналов. В БИС К1804ВУ1 реализованы три способа адресации микрокоманд: последовательная по счетчику микрокоманд, выборка адреса из одного из внутренних или внешних источников адреса, модификация младших разрядов адреса микрокоманды. В БИС К1804ВУ2 реализованы только два первых способа адресации. Обе БИС обеспечивают шесть различных управляющих конструкций в алгоритмах микропрограммного управления: безусловный переход, условный переход по одному или нескольким направлениям, цикл проверки условия, повторение предыдущего адреса микрокоманды, условный переход к микропрограмме и возврат по условию, безусловный переход к программе и возврат.

Реализация перечисленных способов адресации и управляющих конструкций обеспечивается с помощью блока выбора адреса, регистра адреса, счетчика микрокоманд и стека (рис.93).

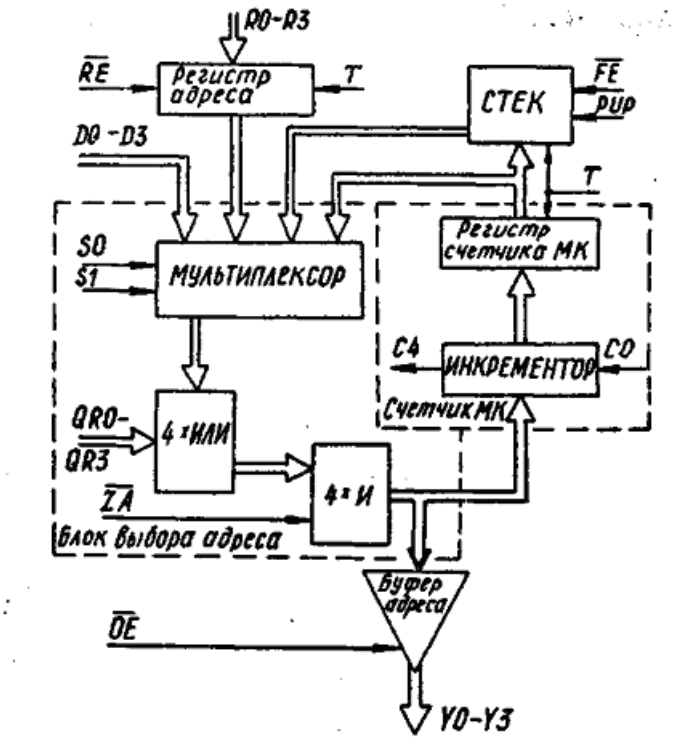


Рис. 93. Секция управления адресом микрокоманды К1804ВУ1

В микросхеме предусматривается четыре источника адреса, каждый из которых может выдать 4-разрядный двоичный адрес: счетчик микрокоманд, регистр адреса, стек и входная шина адреса D. Блок выбора адреса содержит мультиплексор, с помощью которого в зависимости от управляющих сигналов SO и S1 выбирается один из источников адреса (см. табл.36).

Таблица 36

S1	S0	Источник адреса
0	0	Счетчик МК
0	1	Регистр адреса
1	0	Стек
1	1	Шина адреса

Выбранный адрес может быть модифицирован с помощью маски, подаваемой по шине QKO-QR3 . Кроме того, блок выбора адреса имеет вход, который используется для установки на его выходе нулевого значения адреса, обеспечиваемого при подаче на этот вход нулевого сигнала.

Рассмотрим подробнее узлы, служащие источниками адреса.

Счетчик микрокоманд состоит из 4-разрядного регистра, в который при положительном фронте тактовых импульсов заносится значение, имеющееся на выходе блока выбора адреса. Инкрементор может увеличить этот адрес на единицу при ***C0*** = 1. В инкременторе имеется выход переноса ***C4*** . При объединении микросхем схем управления адресом микрокоманды выходная цепь переноса ***C4*** подключается к входной цепи переноса следующей (старшей) секции. Таким образом, при подаче на вход младшей секции логической еди-

ницы в начале каждого тактового периода в счетчик микрокоманд заносится значение адреса, увеличенное на единицу по сравнению со значением адреса в предыдущем тактовом периоде. Так формируется адрес микрокоманды, если не нарушается естественный порядок следования адресов, т.е. в отсутствие условных и безусловных переходов. Адрес микрокоманды, записанный в регистре счетчика микрокоманд, передается либо в стек, либо снова в блок выбора адреса.

Регистр адреса – 4-разрядный регистр, информация в который может приниматься по 4-разрядной шине ***R3-R0***. Вход $\overline{RE}$ является управляющим. На этот вход подается сигнал разрешения записи в регистр адреса. При $\overline{RE}$ = 0 на положительном фронте тактового импульса информация, поступающая по шине ***R3-R0***, принимается в регистр адреса.

Стек содержит накопитель из четырех 4-разрядных регистров и 2-разрядного указателя, стека, хранящего адрес входа в накопитель. Работой стека управляют сигналы $\overline{FE}$ и ***PUP*** . Сигнал $\overline{FE}$ служит сигналом разрешения изменения содержимого указателя стека, ***PUP*** – сигналом, определяющим направление изменения содержимого указателя стека (при ***PUP*** = 0 – уменьшение, при **PUP** = 1 – увеличение содержимого указателя стека).

Пусть регистры накопителя CT_0, CT_1 CT_2, CT_3 хранят соответственно адреса ***A, B , C, D*** . В дальнейшем под регистром CT_0 будем понимать регистр накопителя, адресуемый указателем стека. Рассмотрим процессы в стеке при различных комбинациях управляющих сигналов $\overline{FE}$ и ***PUP***.

Пусть в текущий такт поступает сигнал $\overline{FE}$=1, значение сигнала ***PUP*** безразлично. Значение $\overline{FE}$=1 задает режим чтения без изменения содержимого указателя стека. При этом в текущем такте из стека на вход блока выбора адреса поступает содержимое регистра CT_0 . При переходе к следующему такту размещение информации в регистрах накопителя остается прежним.

Если в текущем такте подается комбинация управляющих сигналов $\overline{FE}$=0 и ***PUP*** = 0, то устанавливается так называемый режим выталкивания из стека. В этом случае в текущем такте на вход блока выбора адреса выдается адрес A, хранившийся в регистре CT_0; при переходе к следующему такту происходит перемещение информации в регистрах.

При подаче в текущем такте комбинации сигналов $\overline{FE}$=0 и ***PUP***=1 устанавливается так называемый режим записи, при котором в текущем такте на вход блока выбора адреса выдается содержимое регистра CT_0 (адрес A), а при переходе к следующему такту происходит перемещение информации в регистрах накопителя в обратном направлении и в регистр CT_0 принимается содержимое счетчика микрокоманд.

Стек используется при обращении к подпрограммам. При переходе к подпрограмме адрес ее первой микрокоманды выдается на выход схемы управления адресом микрокоманды из регистра адреса либо с шины ***D***. Стек устанавливается в режим записи, и при переходе к следующему такту в регистр CT_0 накопителя стека принимается содержимое счетчика микрокоманд,

соответствующее адресу очередной микрокоманды, на которой было установлено выполнение главной программы. После окончания выполнения подпрограммы выдается адрес из стека и происходит возврат в главную программу.

Блок выбора адреса кроме входов, предназначенных для приема содержимого четырех рассмотренных выше источников адреса, имеет входы маски QR3 ~ Q$0, которые используются для модификации адреса: может быть установлена единица в любом разряде адреса путем подачи ее в соответствующий разряд шины ***QR3-QR0***.

Адрес с выхода блока выбора адреса передается на выход микросхемы через буфер адреса, который построен на злементах с тремя состояниями, управляемых сигналом $\overline{OE}$. При $\overline{OE}$ = 0 буфер адреса открыт, при $\overline{OE}$=1 он отключает микросхему от высшей шины адреса.

9.4.2. Схема управления следующим адресом К1804ВУ3 и схема управления последовательностью микрокоманд К1804ВУ4

БИС К1804ВУ3 предназначена для преобразования закодированного поля микрокоманды в набор управляющих сигналов для различных узлов блока микропрограммного управления адресом микрокоманды К1804ВУ1 и К1804ВУ2, счетчик циклов, регистр микрокоманд и т.д. Она представляет собой преобразователь, выполненный на ПЗУ емкостью 32 х 8 разрядных слова. Входными сигналами являются разряды ***I0-I3*** микрокоманды и код условия ветвления ***TST*** , указанный в микрокоманде (рис. 94). На восьми управляемых выходах формируется набор управляющих сигналов, соответствующих значению кода микрокоманды (***I0-I3***) или условию ветвления (***TST***).

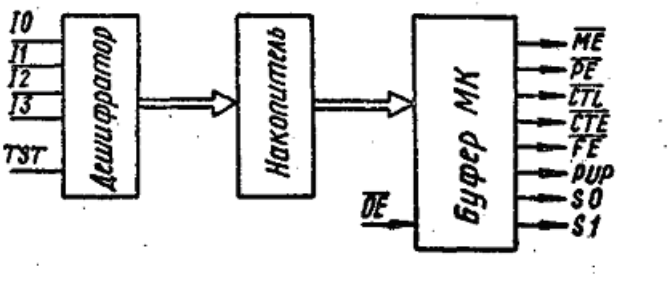


Рис. 94. БИС К1804ВУ3

При каждой комбинации значений входных сигналов ***I0-I3*** и ***TST*** дешифратор осуществляет чтение содержимого определенной ячейки накопителя и через буфер микрокоманды, построенный на элементах с тремя состояниями. Управление буфером производится сигналом $\overline{OE}$ При $\overline{OE}$=1 он устанавливается в выключенное состояние, а при $\overline{OE}$=0 на выходе микросхемы появляются управляющие сигналы.

БИС К1804ВУ4 используется для построения блоков микропрограммного управления различных цифровых устройств и предназначена для фор-

мирования адресов микрокоманд под воздействием внешних управляющих сигналов. Она обеспечивает:

адресацию 4096 ячеек микропрограммной памяти;

получение адреса микрокоманды из четырех источников (внутреннего регистра адреса / счетчика, счетчика микрокоманд, адресной шины, стека);

реализацию 16 инструкций управления последовательностью выполнения микрокоманд;

управление работой трех внешних устройств (регистров), подключенных к адресной шине.

9.5. Элементы программирования секционированного микропроцессора серии К1804

Для эффективного применение комплекта К1804 необходимо знать принципы микропрограммного управления в ЭВМ. В микропрограммной ЭВМ для выполнения различных операций используется однородная последовательность микрокоманд. Выполнение машинной команды интерпретируется набором микрокоманд, образующих микропрограмму. Элементарные функции, реализуемые при выполнении микрокоманды, инициируются микрокомандами. Обычно микрокоманда выполняет две главные функции: определение и управление всеми микрооперациями, определение и управление адресом – следующей микрокоманды. Первая функция связана с выбором операндов для АЛУ, заданием операции АЛУ, выбором получателя результата АЛУ, управлением переносом, сдвигом, прерываниями, вводом и выводом данных и т.д. Вторая функция выбирает источник адреса следующей микрокоманды и иногда явно определяет этот адрес.

В блоке микропрограммного управления (БМУ) имеется микропрограммная память **M x N**. Диапазон адресов составляет от 0 до $N-1$. Каждое слово (микрокоманда) состоит из M бит, разделенных на поля различной длины. Определение полей называется форматом микрокоманды. В типичной ЭВМ микрокоманда содержит следующие поля: 1 – общего назначения, 2 – адрес перехода (адрес микропрограммной памяти), 3 – функция управления адресом следующей микрокоманды, 4 – управление прерыванием, 5 – управление выбором синхронизации, 6 – управление переносом, 7 – управление источниками операндов АЛУ, 8 – управление функцией АЛУ, 9 – управление получателем результата АЛУ.

После определения формата микрокоманд необходимо обеспечить последовательное выполнение микрокоманд во времени. В простейшем случае для этого потребуется счетчик микропрограммного адреса, или счетчик микрокоманд, длина которого $K=\log_2 N$. Для перехода к следующей микрокоманде в каждом такте синхронизации выполняется инкремент счетчика микрокоманд (увеличение содержимого счетчика на единицу), он адресует следующую по порядку микрокоманду.

Чтобы БМУ был более гибким, в нем необходимо реализовать функцию безусловного перехода ***JUMP*** . Наиболее просто для адреса перехода отвести специальное поле микрокоманды и управлять выбором между операторами безусловного перехода ***JUMP*** и последовательного выполнения микрокоманд ***CONT*** с помощью одного бита в микрокоманде: ***SEL*** = 0 соответствует оператору ***CONT, SEL = 1*** – оператору ***JUMP*** (при этом разрешается загрузка адреса перехода в счетчик микрокоманд).

Для реализации операций условного перехода ***JUMP CJ*** в микрокоманде необходимо предусмотреть двухбитное поле управления выбором загрузки ***SEL***. Когда оно содержит 00, сигнал загрузки пассивен (реализуется оператор ***CONT***). При ***SEL*** = 01 управление загрузкой адреса памяти зависит от значения первого условия. Если оно равно логическому нулю, то в счетчике микрокоманд проводится инкремент, а если равно логической единице, то в счетчик микрокоманд загружается адрес памяти. При ***SEL***=10 реализуется функция условного перехода аналогично описанному выше в зависимости от второго условия. Если ***SEL***=11, реализуется операция безусловного перехода.

Для увеличения производительности микропрограммной ЭВМ выполнение текущей микрокоманды совмещают с выборкой из микропрограммной памяти следующей микрокоманды. Это называется конвейеризацией. Конвейеризация обеспечивается конвейерным регистром, который содержит текущую команду, выполняемую ЭВМ. Одновременно с выполнением этой команды в микропрограммную память подается адрес следующей микрокоманды, производится ее считывание, и сигналы следующей микрокоманды устанавливаются на входах конвейерного регистра. На адрес следующей микрокоманды влияют значения условий, полученных при выполнении предыдущей микрокоманды.

Как и при программировании на уровне машинных команд, в микропрограммировании возможно использование подпрограмм. При переходе к подпрограмме необходимо запомнить адрес, к которому подпрограмма должна вернуться после завершения своих действий (адрес возврата). Для этого применяется стек, т. е. группа (файл) регистров, которая работает таким образом, что последний включенный в нее адрес возврата исключается первым.

Последний включенный в стек адрес возврата (вершина стека ***TOS***) адресует специальный регистр, называемый указателем стека ***SP***. Все стековые операции сопровождаются автоматической модификацией ***SP***. При включении сначала выполняется декремент ***SP***, затем в адресуемый им регистр загружается адрес возврата (этот регистр становится новой ***TOS***), а при исключении содержимое TOS передается в нижний получатель, после чего производится инкремент ***SP***.

Блок микропрограммного управления имеет стек и связанный с ним указатель стека ***SP*** Для управления стеком обходимы сигнал разрешения стека $\overline{FE}$, низкий уровень которого разрешает выполнение стековой операции, а высокий запрещает работу стека, и сигнал $PUP(PUSH/\overline{POP})$, высокий уровень

которого определяет операцию включения, а низкий – операцию исключения с соответствующей модификацией ***SP***.

Поле управления следующим адресом ***SEL*** может быть расширено да трех бит, что обеспечивает восемь функций переходов. Это трехбитное поле, и вход кода условия CC образуют четыре бита управления адресом для стека и мультиплексора A. Этот адрес определяет выходной код, представляющий собой четыре управляющих сигнала: два сигнала ***Y0*** и ***Y1*** подаются на управляющие входы BO и S1 мультиплексора A, а два других – ***Y2*** и ***Y3*** – управляют работой стека (сигналы $\overline{FE}$ и ***PUP***). В результате определяются восемь функций переходов (табл. 37).

Таблица 37

Функция	Входы				Выходы			
	A3	A2	A1	A0 = CC	S1	S0	$\overline{FE}$	PUP
Продолжение (CONT)	0	0	0	X	0	0	I	X
Безусловный переход	0	0	I	X	0	I	I	X
Условный переход (CJ)	0	I	0	0	0	0	I	X
	0	I	0	I	0	I	I	X
Включение в стек (PUSH)	0	I	I	X	0	0	0	I
Переход к подпрограмме (JSR)	I	0	0	X	0	I	0	I
Условный переход к подпрограмме (CJSP)	I	0	I	0	0	0	I	X
	I	0	I	I	0	I	0	I
Возврат из подпрограммы (RIN)	I	I	0	X	I	0	0	0
Проверка конца цикла	I	I	I	0	I	0	I	X
(LOOP)	I	I	I	I	0	0	0	0

Лекция № 26

10. Надежность работы микропроцессорного вычислителя. Заключение.

10.1. Методы повышения надёжности работы вычислительных устройств. Ошибки, неисправности, дефекты

В жизненном цикле микропроцессорной системы, как любой дискретной системы, выделяются три стадии: проектирование, изготовление и эксплуатация. Каждая из стадий подразделяется на несколько фаз, для которых существуют вероятности возникновения конструктивных или физических неисправностей, приводящих систему в неработоспособное состояние. Поэтому на каждой фазе необходимы процедуры тестового контроля, направленные на обнаружение и локализацию неисправностей. Процедура тестового контроля может быть определена как проведение экспериментов с "черным ящиком". Дискретная система любой сложности или часть такой системы может рассматриваться как "черный ящик" с множеством входов и выхо-

дов. Правильность функционирования этого "черного ящика" должна устанавливаться путем подачи входных сигналов и наблюдения ответных выходных сигналов системы. В тех случаях, когда поведение "черного ящика" отличается от нормального, характеризуемого его спецификацией или представлениями человека, говорят о наличии ошибки. Ошибка вызывается некоторой неисправностью, представляющей собой некорректное состояние внутри "черного ящика". Неисправности классифицируют в соответствии с их причинами: физическая, если причиной ее служат либо дефекты элементов, либо физическое воздействие окружающей среды; субъективная (внесенная, нефизическая), если ее причиной служат ошибки проектирования, неправильный монтаж элементов, грубые ошибки оператора. Физические неисправности - непредусмотренные, нежелательные изменения значения одной или нескольких логических переменных в системе. Субъективные неисправности - конкретные проявления недостатков программного и аппаратного обеспечения и неправильных действий оператора, имеющих место при выполнении дискретной системой предписанных спецификацией действий.

Под субъективными неисправностями подразумеваются неисправности нефизические, вызванные недостатками различных схем, конструкций, программ, средств эксплуатации - компиляторов, ассемблеров, программ автоматизации проектирования, инструкции по эксплуатации, процедур и средств контроля и т. д. Субъективные неисправности делят на проектные и интерактивные.

Проектные неисправности вызваны недостатками, вносимыми в систему на различных стадиях реализации исходного задания при структурном проектировании, разработке алгоритмов, написании программ, трансляции в машинный код, детальном логическом и техническом проектировании, а также при последующих модификациях аппаратного и программного обеспечения. Интерактивные неисправности возникают, когда в процессе работы, технического обслуживания или отработки системы оператор вводит в нее через интерфейс человек-машина ложную информацию, не соответствующую текущему состоянию системы. Как правило, это происходит в результате непонимания инструкции для оператора или вследствие неточностей ввода информации.

Ошибка – проявление неисправности (физической или субъективной). В зависимости от уровня иерархической структуры системы термин "ошибка" может иметь различный смысл. Так, для дискретного устройства он означает появление неверных двоичных сигналов ("0" вместо "1" и "1" вместо "0"); для программы ошибка означает отклонение поведения программы от заданного, приводящее к выдаче неверных результатов.

Следует четко разграничивать понятия "ошибка" и "неисправность". Неисправность может приводить или не приводить к ошибке в зависимости от состояния системы. В то же время возникновение ошибки обязательно говорит о существовании какой-то неисправности. Одна и та же ошибка может быть вызвана множеством неисправностей, а одна неисправность может слу-

жить причиной целого ряда ошибок. Например, если триггер, предназначенный для хранения кода переполнения разрядной сетки ЭВМ, вследствие неисправности все время находится в состоянии "0", то ошибки из-за неисправности не будет до тех пор, пока в процессе вычислений реально не возникнет арифметическое переполнение, при котором триггер останется в состоянии "0" вместо перехода в состояние "1". Однако даже и в этом случае такая ошибка процессора не обязательно приведет к ошибке на программном уровне, если в программе условие переполнения не проверяется и, следовательно, ни с какой стороны не влияет на ее дальнейшее поведение.
Дефекты – физические изменения параметров компонентов системы, выходящие за допустимые пределы. Их называют сбоями, если они носят временный характер, и отказами, если они постоянны.

10.2. Тестовый контроль

Дефект не может быть обнаружен до тех пор, пока не будут созданы условия для возникновения из-за него неисправности, для того чтобы сделать неисправность наблюдаемой. Метод испытаний должен позволить генерировать тесты, ставящие испытуемый объект в условия, при которых моделируемые неисправности проявляли бы себя в виде обнаруживаемых ошибок. Если испытуемый объект предназначен для эксплуатации, то при обнаружении ошибки необходимо произвести локализацию неисправности с целью ее устранения.

О правильности функционирования микропроцессорной системы на уровне "черного ящика" с полностью неизвестной внутренней структурой можно говорить лишь тогда, когда произведены ее испытания, в ходе которых реализованы все возможные комбинации входных воздействий, и в каждом случае проверена корректность ответных реакций. Однако исчерпывающее тестирование имеет практический смысл лишь для простейших элементов систем. Следствием этого является тот факт, что ошибки проектирования встречаются при эксплуатации, и для достаточно сложных систем нельзя утверждать об их отсутствии на любой стадии жизни системы. В основе почти всех методов испытаний лежит та или иная гипотетическая модель неисправностей, первоисточником которой служат неисправности, встречающиеся в практике. В соответствии с моделью в рамках каждого метода предпринимаются попытки создания тестовых наборов, которые могли бы обеспечить удовлетворительное выявление моделируемых неисправностей. Любой метод тестирования хорош ровно настолько, насколько правильна лежащая в его основе модель неисправности.

Важным моментом является правильный выбор соотношения между степенью общности модели, стоимостью и степенью сложности формирования и прогона тестов, ориентированных на моделируемые неисправности. Чем конкретнее модель, тем легче создать для нее систему тестов, но тем выше вероятность того, что неисправность останется незамеченной. Если же

модель неисправностей излишне общая, то из-за комбинаторного возрастания числа необходимых тестовых наборов и/или времени вычислений, требуемого для работы алгоритмов формирования тестов, она станет непрактичной и пригодной только для несложных систем.

10.3. Структура автоматической системы контроля и диагностирования. Обнаружение ошибки и диагностика неисправности

Диагностика неисправности – процесс определения причины появления ошибки по результатам тестирования. Отладка – процесс обнаружения ошибок и определение источников их появления по результатам тестирования при проектировании микропроцессорных систем. Средствами отладки являются приборы, комплексы и программы.

Точность, с которой тот или иной тест локализует неисправности, называется его разрешающей способностью. Требуемая разрешающая способность определяется конкретными целями испытаний. Например, при испытаниях аппаратуры в процессе эксплуатации для ее ремонта часто необходимо установить, в каком сменном блоке изделия имеется неисправность. В заводских условиях желательно осуществлять диагностику неисправности вплоть до уровня наименьшего заменяемого элемента, чтобы минимизировать стоимость ремонта. В лабораторных условиях в процессе отладки опытного образца необходимо определять природу неисправности (физического или нефизического происхождения). В случае возникновения и проявления дефекта требуется локализовать место неисправности с точностью до заменяемого элемента, а при проявлении субъективной неисправности – с точностью до уровня представления (программного, схемного, логического и т. д.), на котором была внесена неисправность, и места.

Так как процесс проектирования микропроцессорной системы содержит неформализуемые этапы, то отладка системы предполагает участие человека.

Свойство контролепригодности системы.

Успех отладки зависит от того, как спроектирована система, предусмотрены ли свойства, делающие её удобной для отладки, а также от средств, используемых при отладке. Для проведения отладки проектируемая микропроцессорная система должна обладать свойствами управляемости, наблюдаемости, предсказуемости.

Управляемость – свойство системы, при котором ее поведение поддается управлению, т. е. имеется возможность остановить функционирование системы в определенном состоянии, и затем снова ее запустить. Наблюдаемость – свойство системы, позволяющее проследить за поведением системы, сменой ее внутренних состояний. Предсказуемость – свойство системы, позволяющее установить систему в состояние, из которого все последующие состояния могут быть предсказаны.

10.4. Основные тенденции развития вычислительных устройств

Будущее микропроцессорной техники связано сегодня с двумя новыми направлениями - нанотехнологиями и квантовыми вычислительными системами. Эти пока еще главным образом теоретические исследования касаются использования в качестве компонентов логических схем молекул и даже субатомных частиц: основой для вычислений должны служить не электрические цепи, как сейчас, а положение отдельных атомов или направление вращения электронов. Если "микроскопические" компьютеры будут созданы, то они обойдут современные машины по многим параметрам.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Зубков С.В. ASSEMBLER для DOS, Windous и UNIX. СПб: Питер, 2004
2. Юров В. Ассемблер: Учебник. СПб.: Питер, 2000.
3. Юров В. 3. ASSEMBLER. Практикум. СПб.: Питер, 2001.
4. Ю-Чжен-Лю, Гибсон, Г. Г. Микропроцессоры семейства 8086/8088. М.: «Радио и связь», 1987.
5.Микропроцессорный комплект К1810. Структура, программирование, применение / Под ред. Ю.М. Казаринова. М.: Высш.шк., 1990.
6. Казаринов, Ю. М. Номоконов, В. Н., Филиппов, Ф. В. Применение микропроцессоров и микроЭВМ в радиотехнических система. М.: «Высшая школа», 1998.
7. Микропроцессоры и микропроцессорные комплекты интегральных схем / Под ред. В.А.Шахнова Т.1 – М.: Радио и связь,1998
8. Микропроцессоры / Под ред. А.В.Преснухина. Кн.1– М.: Высш.шк., 1999
9. С.С.Булгаков, В.М.Мещеряков. Проектирование цифровых систем на комплектах микропрограммируемых БИС– М.: Радио и связь, 2000

ОГЛАВЛЕНИЕ

НЛНССИНП COmPUTER SCIENCE

Э. ТАНЕНБАУМ

АРХИТЕКТУРА КОМПЬЮТЕРА

4-Е ИЗДАНИЕ

С^ПТЕР ®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2003

Краткое содержание

Содержание

Сюзанне, Барбаре, Марвину, Брэму и памяти моей дорогой я

Об авторе

Эндрю С. Таненбаум получил степень бакалавра естественных наук в Массачусетском технологическом институте и степень доктора в Университете Калифорнии в Беркли. В настоящее время является профессором Амстердамского университета в Нидерландах, где возглавляет группу разработчиков компьютерных систем. Он также возглавляет факультет вычислительной техники (межвузовскую аспирантуру, в которой исследуются и разрабатываются системы параллельной обработки, распределенные системы и системы формирования изображения). Тем не менее он всеми силами старается не превратиться в бюрократа.

В прошлом он занимался компиляторами, операционными системами, сетями и локальными распределенными системами. Его настоящее исследование связано с разработкой глобальных распределенных систем, которые включают в себя миллионы пользователей. Результатом этих исследовательских проектов стали 85 статей, опубликованных в разных журналах, выступления на конференциях и 5 книг.

Профессор Таненбаум разрабатывает программное обеспечение. Он является главным разработчиком пакета «Amsterdam Compiler Kit» (набора инструментальных средств для написания портативных компиляторов), а также разработчиком системы MINIX (клона UNIX для студенческих лабораторий программирования). Вместе со своими учениками и программистами он участвовал в разработке системы Amoeba (это распределенная система с высокой производительностью на основе микроядра). Системы MINIX и Amoeba находятся в свободном доступе в Интернете.

Его аспиранты достигли больших высот после получения ученых степеней. Он очень гордится ими.

Профессор Таненбаум — член Ассоциации по вычислительной технике, член Института инженеров по электротехнике и электронике (IEEE), член Королевской голландской академии науки и искусства. В 1994 году получил премию от Ассоциации по вычислительной технике как выдающийся педагог. В 1997 году награжден премией от Специальной группы по образованию в области вычислительной техники (Ассоциации по вычислительной технике) за вклад в образование в области вычислительной техники. Его имя включено в справочник *«Кто есть кто» («Who's Who in the World»)*. Его домашнюю страничку в Интернете можно найти по адресу http://www.cs.vu.nl/~ast/.

Предисловие

В основе первых трех изданий книги лежит идея о том, что компьютер можно рассматривать как иерархию уровней, каждый из которых выполняет какую-либо определенную функцию. Это фундаментальное утверждение сейчас столь же правомерно, как и в момент выхода в свет первого издания, поэтому мы по-прежнему берем его за основу, на этот раз уже в четвертом издании. Как и в первых трех изданиях, в этой книге мы подробно описываем цифровой логический уровень, уровень архитектуры команд, уровень операционной системы и уровень языка ассемблера (хотя мы изменили некоторые названия, чтобы следовать современным установившимся обычаям).

В целом структура книги осталась прежней, но в четвертое издание внесены некоторые изменения, что объясняется стремительным развитием компьютерной промышленности. Например, все программы, которые в предыдущих изданиях были написаны на языке Pascal, в четвертом издании переписаны на язык Java, чтобы продемонстрировать популярность языка Java в настоящее время. Кроме того, в качестве примеров в книге рассматриваются более современные машины (Intel Pentium II, Sun UltraSPARC II и Sun picojava II).

Мультипроцессоры и компьютеры параллельного действия получили широкое распространение, поэтому материал, связанный с архитектурами параллельного действия, был полностью переделан и значительно расширен. В этой книге мы затрагиваем широкий диапазон тем от мультипроцессоров до кластеров рабочих станций.

С годами книга увеличилась в объеме (хотя не так сильно, как другие популярные компьютерные издания). Это неизбежно, поскольку происходит постоянное развитие и о предмете становится известно все больше и больше. Поэтому если книга используется в целях обучения, нужно иметь в виду, что этот курс может занять более длительное время, чем раньше. Возможный вариант — изучать первые три главы, часть четвертой главы (до раздела о разработке микроархитектурного уровня включительно) и пятую главу в качестве минимума, а оставшееся время на ваше усмотрение потратить на шестую, седьмую, восьмую главы и вторую часть четвертой главы.

В четвертое издание внесены следующие изменения В главе 1 по-прежнему излагается история развития архитектуры компьютеров, но мы расширили ряд рассматриваемых машин. В главе вводятся три основных примера: Pentium II, UltraSPARC II и picojava II.

Материал второй главы обновлен и переработан. В ней мы рассматриваем современные устройства ввода-вывода: диски RAID, CD-R, DVD, цветные принтеры и т. п.

Глава 3 (цифровой логический уровень) претерпела некоторые изменения — теперь в ней рассматриваются компьютерные шины и современные устройства ввода-вывода. Главное изменение — это новый материал о шинах (в частности, PCI и USB). Три новых примера описываются на уровне микросхем.

Глава 4 (теперь она называется «Микроархитектурный уровень») была полностью переписана. Идея использовать пример микропрограммируемой машины для демонстрации работы тракта данных была сохранена, но в качестве примера взят сокращенный вариант JVM. В соответствии с этим была изменена микроархитектура. В главе продемонстрированы возможные компромиссы с точки зрения стоимости и производительности. В последнем примере, Mic-4, используется конвейер из семи стадий. Этот пример наглядно демонстрирует основные принципы работы современных компьютеров (например, Pentium II). К главе добавлен новый раздел о способах увеличения производительности, в котором рассматриваются новые технологии (кэширование, прогнозирование переходов, исполнение с изменением последовательности, спекулятивное выполнение и предикация). Новые примеры машин рассматриваются на микроархитектурном уровне.

В главе 5 (теперь она называется «Уровень архитектуры команд») рассказывается о так называемом машинном языке. В качестве основных примеров здесь используются Pentium II, UltraSPARC II и JVM.

Глава 6 (уровень операционной системы) содержит примеры операционных систем для Pentium II (Windows NT) и UltraSPARC II (UNIX). Первая операционная система сравнительно новая. Она содержит множество особенностей, которые стоит изучить. Система UNIX все еще используется во многих университетах и компаниях, и, кроме того, она довольно проста, поэтому тоже заслуживает нашего внимания.

В главе 7 (уровень языка ассемблера) появились примеры для тех машин, которые мы рассматриваем в этой книге. Кроме того, добавлен новый материал о динамическом связывании.

Глава 8 (архитектура компьютеров параллельного действия) полностью изменена. В ней подробно описываются мультипроцессоры (UMA, NUMA и COMA) и мультикомпьютеры (MPP и COW).

Пересмотрен список литературы. Большинство работ, на которые мы ссылаемся в этой книге, опубликованы после выхода третьего издания. Двоичные числа и числа с плавающей точкой не сильно изменились за последнее время, поэтому приложения вошли в четвертое издание почти без изменений.

Наконец, многие проблемы, изложенные в третьем издании, были пересмотрены и к ним добавлены новые.

Существует web-сайт этой книги. В Интернете имеются файлы PostScript для всех иллюстраций. Их можно получить и распечатать. Кроме того, там можно найти симулятор и другие инструментальные программные средства. Универсальный адрес ресурса: http://www.cs.vu.nl/~ast/sco4/

Симулятор и программные средства написаны Реем Онтко (Ray Ontko). Автор выражает признательность Рею за эти чрезвычайно полезные программы.

Автор искренне благодарит всех, кто читал рукопись данной книги и высказал ценные замечания и предложения или оказал какую-либо помощь, в частности Генри Бола (Henri Bal), Алана Чарльзуорта (Alan Charlesworth), Куроша Гарахор-

лоо (Koorosh Gharachorloo), Маркуса Гонкалвеса (Marcus Goncalves), Карена Панетту Ленц (Karen Panetta Lentz), Тимоти Мэттсона (Timothy Mattson), Гарлана Мак-Гана (Harlan McGhan), Майлза Мердокка (Miles Murdocca), Кэвина Нормойла (Kevin Normoyle), Майка О'Коннора (Mike O'Connor), Митсунори Огихара (Mitsunori Ogihara), Рея Онтко (Ray Ontko), Аске Плаата (Aske Plaat), Вильяма Потвина (William Potvin II), Нагарайана Прабхакарана (Nagarajan Prabhakaran), Джеймса Г. Пагсли (James H. Pugsley), Рональда Н. Шредера (RonaldN. Schroeder), Райана Шумейкера (Ryan Shoemaker), Чарльза Силио-мл. (Charles Silio, Jr.) и Дейла Скрина (Dale Skrien). Мои ученики Адриаан Бон (Adriaan Bon), Лаура де Вриес (Laura de Vries), Дольф Лот (Dolf Loth), Гуидо ван Нордент (Guido van't Noordende) помогали мне в работе над текстом, за что им большое спасибо.

Особую благодарность выражаю Джиму Гудману Qim Goodman) за его вклад в создание этой книги (в частности, четвертой и пятой глав). Идея использовать JVM принадлежит именно ему, и микроархитектура для реализации JVM тоже его. Он же предложил множество новаторских идей. Книга значительно улучшилась благодаря его содействию.

Наконец, я хотел бы поблагодарить Сюзанну за терпеливое отношение ко мне, несмотря на то, что я долгие часы проводил за своим Pentium'oM. С моей точки зрения, Pentium — более усовершенствованная машина, чем мой старый IBM-386, но с ее точки зрения никакой разницы нет. Я также хочу поблагодарить Барбару и Марвина за то, что они такие замечательные дети, а также Брэма за то, что он вел себя тихо, когда я писал эту книгу.

ЭндрюС. Таненбаум

Глава 1
Предисловие

Цифровой компьютер — это машина, которая может решать задачи, выполняя данные ей команды. Последовательность команд, описывающих решение определенной задачи, называется **программой.** Электронные схемы каждого компьютера могут распознавать и выполнять ограниченный набор простых команд. Все программы перед выполнением должны быть превращены в последовательность таких команд, которые обычно не сложнее чем:

- сложить 2 числа;
- проверить, не является ли число нулем;
- скопировать кусок данных из одной части памяти компьютера в другую.

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Такой язык называется **машинным языком.** Разработчик при создании нового компьютера должен решать, какие команды включить в машинный язык этого компьютера. Это зависит от назначения компьютера, от того, какие задачи он должен выполнять. Обычно стараются сделать машинные команды как можно проще, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику. Так как большинство машинных языков очень примитивны, использовать их трудно и утомительно.

Это простое наблюдение с течением времени привело к построению ряда уровней абстракций, каждая из которых надстраивается над абстракцией более низкого уровня. Именно таким образом можно преодолеть сложности при общении с компьютером. Мы называем этот подход **многоуровневой компьютерной организацией.** Так мы и назвали эту книгу. В следующем разделе мы расскажем, что понимаем под этим термином. Затем мы расскажем об истории развития этой проблемы и положении дел в настоящий момент, а также рассмотрим некоторые важные примеры.

Многоуровневая компьютерная организация

Как мы уже сказали, существует огромная разница между тем, что удобно для людей, и тем, что удобно для компьютеров. Люди хотят сделать X, но компьютеры могут сделать только Y. Из-за этого возникают проблемы. Цель данной книги — объяснить, как можно решать эти проблемы.

Языки, уровни и виртуальные машины

Проблему можно решить двумя способами. Оба эти способа включают в себя разработку новых команд, которые более удобны для человека, чем встроенные машинные команды. Эти новые команды в совокупности формируют язык, который мы будем называть Я 1. Встроенные машинные команды тоже формируют язык, и мы будем называть его Я 0. Компьютер может выполнять только программы, написанные на его машинном языке Я 0. Упомянутые два способа решения проблемы различаются тем, каким образом компьютер будет выполнять программы, написанные на языке Я 1.

Первый способ выполнения программы, написанной на языке Я 1, — замена каждой команды на эквивалентный набор команд в языке Я 0. В этом случае компьютер выполняет новую программу, написанную на языке Я 0, вместо старой программы, написанной на Я 1. Эта технология называется **трансляцией.**

Второй способ — написание программы на языке Я 0, которая берет программы, написанные на языке Я 1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка Я 0. Эта технология не требует составления новой программы на Я 0. Она называется **интерпретацией,** а программа, которая осуществляет интерпретацию, называется **интерпретатором.**

Трансляция и интерпретация сходны. При применении обоих методов компьютер в конечном итоге выполняет набор команд на языке Я 0, эквивалентных командам Я 1. Различие лишь в том, что при трансляции вся программа Я 1 переделывается в программу Я 0, программа Я 1 отбрасывается, а новая программа на Я 0 загружается в память компьютера и затем выполняется.

При интерпретации каждая команда программы на Я 1 перекодируется в Я 0 и сразу же выполняется. В отличие от трансляции, здесь не создается новая программа на Я 0, а происходит последовательная перекодировка и выполнение команд. Оба эти метода, а также их комбинация широко используются.

Обычно гораздо проще представить себе существование гипотетического компьютера или **виртуальной машины,** для которой машинным языком является язык Я 1, чем думать о трансляции и интерпретации. Назовем такую виртуальную машину М 1, а виртуальную машину с языком Я 0 — М 0. Если бы такую машину М 1 можно было бы сконструировать без больших денежных затрат, язык Я 0, да и машина, которая выполняет программы на языке Я 0, были бы не нужны. Можно было бы просто писать программы на языке Я 1, а компьютер сразу бы их выполнял. Даже если виртуальная машина слишком дорога или ее очень трудно сконструировать, люди все же могут писать программы для нее. Эти программы могут транслироваться или интерпретироваться программой, написанной на языке Я 0, которая сама могла бы выполняться фактически существующим компьютером. Другими словами, можно писать программы для виртуальных машин, как будто они действительно существуют.

Чтобы трансляция и интерпретация были целесообразными, языки Я 0 и Я 1 не должны сильно различаться. Это значит, что язык Я 1 хотя и лучше, чем Я 0, но все же далек от идеала. Возможно, это несколько обескураживает в свете первоначальной цели создания языка Я 1 — освободить программиста от бремени написа-

ния программ на языке, который более удобен для компьютера, чем для человека. Однако ситуация не так безнадежна.

Очевидное решение этой проблемы — создание еще одного набора команд, которые в большей степени ориентированы на человека и в меньшей степени на компьютер, чем Я 1. Этот третий набор команд также формирует язык, который мы будем называть Я 2, а соответствующую виртуальную машину — М 2. Человек может писать программы на языке Я 2, как будто виртуальная машина с машинным *ЯЗЫКОМЯ2действительносуществует. Такиепрограммымогутилитранслироваться*наязык*Я1,иливыполнятьсяинтерпретатором,написаннымнаязыкеЯ1.*

Изобретение целого ряда языков, каждый из которых более удобен для человека, чем предыдущий, может продолжаться до тех пор, пока мы не дойдем до подходящего нам языка. Каждый такой язык использует своего предшественника как основу, поэтому мы можем рассматривать компьютер в виде ряда уровней, как показано на рис. 1.1. Язык, находящийся в самом низу иерархической структуры — самый примитивный, а находящийся на самом верху — самый сложный.

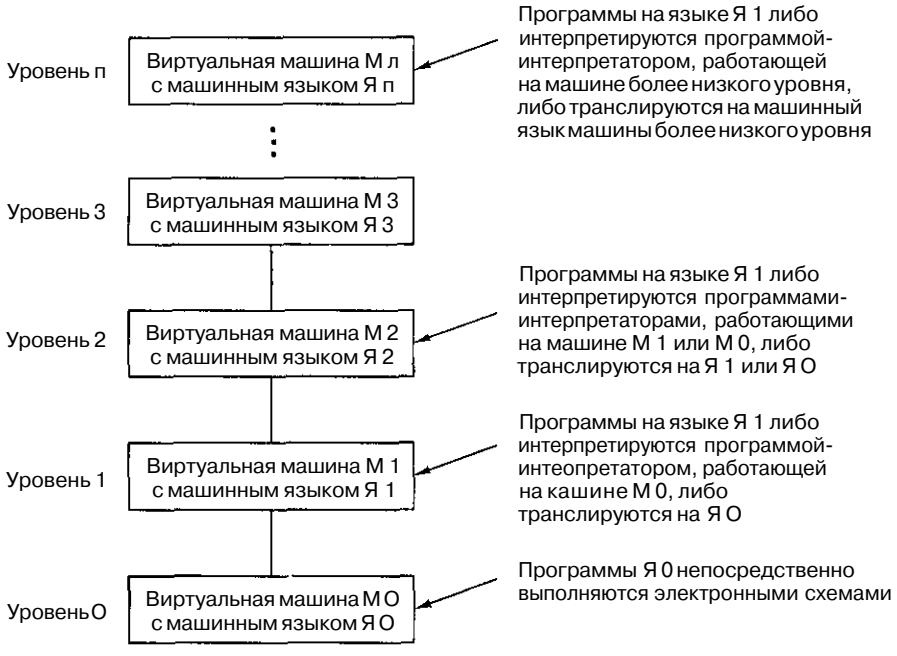


Рис. 1.1. Многоуровневая машина

Между языком и виртуальной машиной существует важная зависимость. У каждой машины есть какой-то определенный машинный язык, состоящий из всех команд, которые эта машина может выполнять. В сущности, машина определяет язык. Сходным образом язык определяет машину, которая может выполнять все программы, написанные на этом языке. Машину, задающуюся определенным языком, очень сложно и дорого сконструировать из электронных схем, но мы можем представить себе такую машину. Компьютер с машинным языком C++ или COBOL был бы слишком сложным, но его можно было бы сконструировать, если учиты-

вать высокий уровень современных технологий. Однако существуют веские причины не создавать такой компьютер: это слишком сложно по сравнению с другими техническими приемами.

Компьютер с n уровнями можно рассматривать как n разных виртуальных машин, у каждой из которых есть свой машинный язык. Термины «уровень» и «виртуальная машина» мы будем использовать как синонимы. Только программы, написанные на Я 0, могут выполняться компьютером без применения трансляции и интерпретации. Программы, написанные на Я 1, Я 2,..., Я n, должны проходить через интерпретатор более низкого уровня или транслироваться на язык, соответствующий более низкому уровню.

Человеку, который пишет программы для виртуальной машины уровня n, не обязательно знать о трансляторах и интерпретаторах более низких уровней. Машина выполнит эти программы, и не важно, будут ли они выполняться шаг за шагом интерпретатором или их будет выполнять сама машина. В обоих случаях результат один и тот же: программа будет выполнена.

Большинство программистов, использующих машину уровня n, интересуются только самым верхним уровнем, то есть уровнем, который меньше всего сходен с машинным языком. Однако те, кто хочет понять, как в действительности работает компьютер, должны изучить все уровни. Те, кто проектирует новые компьютеры или новые уровни (то есть новые виртуальные машины), также должны быть знакомы со всеми уровнями. Понятия и технические приемы конструирования машин как системы уровней, а также детали уровней составляют главный предмет этой книги.

Современные многоуровневые машины

Большинство современных компьютеров состоит из двух и более уровней. Существуют машины даже с шестью уровнями (рис. 1.2). Уровень 0 — аппаратное обеспечение машины. Его электронные схемы выполняют программы, написанные на языке уровня 1. Ради полноты нужно упомянуть о существовании еще одного уровня, расположенного ниже уровня 0. Этот уровень не показан на рис. 1.2, так как он попадает в сферу электронной техники и, следовательно, не рассматривается в этой книге. Он называется **уровнем физических устройств.** На этом уровне находятся транзисторы, которые являются примитивами для разработчиков компьютеров. Объяснять, как работают транзисторы, — задача физики.

На самом нижнем уровне, **цифровом логическом уровне,** объекты называются **вентилями.** Хотя вентили состоят из аналоговых компонентов, таких как транзисторы, они могут быть точно смоделированы как цифровые средства. У каждого вентиля есть одно или несколько цифровых входных данных (сигналов, представляющих 0 или 1). Вентиль вычисляет простые функции этих сигналов, такие как И или ИЛИ. Каждый вентиль формируется из нескольких транзисторов. Несколько вентилей формируют 1 бит памяти, который может содержать 0 или 1. Биты памяти, объединенные в группы, например, по 16,32 или 64, формируют регистры. Каждый регистр может содержать одно двоичное число до определенного предела. Из вентилей также может состоять сам компьютер. Подробно вентили и цифровой логический уровень мы рассмотрим в главе 3.

Рис. 1.2. Компьютер с шестью уровнями. Способ поддержки каждого уровня указан под ним. В скобках указывается название поддерживающей программы

Следующий уровень — **микроархитектурный уровень.** На этом уровне можно видеть совокупности 8 или 32 регистров, которые формируют локальную память и схему, называемую **АЛУ (арифметико-логическое устройство).** АЛУ выполняет простые арифметические операции. Регистры вместе с АЛУ формируют **тракт данных,** по которому поступают данные. Основная операция тракта данных состоит в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-либо операцию, например сложения, а результат помещается в один из этих регистров.

На некоторых машинах работа тракта данных контролируется особой программой, которая называется **микропрограммой.** На других машинах тракт данных контролируется аппаратными средствами. В предыдущих изданиях книги мы назвали этот уровень «уровнем микропрограммирования», потому что раньше он почти всегда был интерпретатором программного обеспечения. Поскольку сейчас тракт данных обычно контролируется аппаратным обеспечением, мы изменили название, чтобы точнее отразить смысл.

На машинах, где тракт данных контролируется программным обеспечением, микропрограмма — это интерпретатор для команд на уровне 2. Микропрограмма вызывает команды из памяти и выполняет их одну за другой, используя при этом тракт данных. Например, для того чтобы выполнить команду ADD, эта команда вызывается из памяти, ее операнды помещаются в регистры, АЛУ вычисляет сумму, а затем результат переправляется обратно. На компьютере с аппаратным контролем тракта данных происходит такая же процедура, но при этом нет программы, которая контролирует интерпретацию команд уровня 2.

Второй уровень мы будем называть уровнем архитектуры системы команд. Каждый производитель публикует руководство для компьютеров, которые он продает, под названием «Руководство по машинному языку» или «Принципы работы компьютера Western Wombat Model 100X» и т. п. Такие руководства содержат информацию именно об этом уровне. Когда они описывают набор машинных команд, они в действительности описывают команды, которые выполняются микропрограммой-интерпретатором или аппаратным обеспечением. Если производитель поставляет два интерпретатора для одной машины, он должен издать два руководства по машинному языку, отдельно для каждого интерпретатора.

Следующий уровень обычно гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд (команды, имеющиеся на одном из уровней, вполне могут находиться на других уровнях). У этого уровня есть некоторые дополнительные особенности: набор новых команд, другая организация памяти, способность выполнять две и более программ одновременно и некоторые другие. При построении третьего уровня возможно больше вариантов, чем при построении первого и второго.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным. Мы будем называть этот уровень **уровнем операционной системы.**

Между третьим и четвертым уровнями есть существенная разница. Нижние три уровня конструируются не для того, чтобы с ними работал обычный программист. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются так называемыми **системными программистами,** которые специализируются на разработке и построении новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — способ, которым поддерживаются более высокие уровни. Уровни 2 и 3 обычно интерпретируются, а уровни 4, 5 и выше обычно, хотя и не всегда, поддерживаются транслятором.

Другое различие между уровнями 1,2,3 и уровнями 4,5 и выше — особенность языка. Машинные языки уровней 1,2 и 3 — цифровые. Программы, написанные на этих языках, состоят из длинных рядов цифр, которые удобны для компьютеров, но совершенно неудобны для людей. Начиная с четвертого уровня, языки содержат слова и сокращения, понятные человеку.

Четвертый уровень представляет собой символическую форму одного из языков более низкого уровня. На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык уровня 1, 2 или 3, а затем интерпретируются соответствующей виртуальной или фактически существующей машиной. Программа, которая выполняет трансляцию, называется **ассемблером.**

Пятый уровень обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются **языками высокого уровня.** Существуют сотни языков высокого уровня. Наиболее известные среди них — BASIC, C, C++, Java, LISP и Prolog. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются **компиляторами.** Отметим, что иногда также используется метод интерпретации. Например, программы на языке Java обычно интерпретируются.

В некоторых случаях пятый уровень состоит из интерпретатора для такой сферы приложения, как символическая математика. Он обеспечивает данные и операции для решения задач в этой сфере в терминах, понятных людям, сведущим в символической математике.

Вывод: компьютер проектируется как иерархическая структура уровней, каждый из которых надстраивается над предыдущим. Каждый уровень представляет собой определенную абстракцию с различными объектами и операциями. Рассматривая компьютер подобным образом, мы можем не принимать во внимание ненужные нам детали и свести сложный предмет к более простому для понимания.

Набор типов данных, операций и особенностей каждого уровня называется архитектурой. Архитектура связана с аспектами, которые видны программисту. Например, сведения о том, сколько памяти можно использовать при написании программы, — часть архитектуры. А аспекты разработки (например, какая технология используется при создании памяти) не являются частью архитектуры. Изучение того, как разрабатываются те части компьютерной системы, которые видны программистам, называется изучением **компьютерной архитектуры.** Термины «компьютерная архитектура» и «компьютерная организация» означают в сущности одно и то же.

Развитие многоуровневых машин

В этом разделе мы кратко изложим историю развития многоуровневых машин, покажем, как число и природа уровней менялись с годами. Программы, написанные на машинном языке (уровень 1), могут сразу выполняться электронными схемами компьютера (уровень 0), без применения интерпретаторов и трансляторов. Эти электронные схемы вместе с памятью и средствами ввода-вывода формируют **аппаратное обеспечение.** Аппаратное обеспечение состоит из осязаемых объектов — интегральных схем, печатных плат, кабелей, источников электропитания, запоминающих устройств и принтеров. Абстрактные понятия, алгоритмы и команды не относятся к аппаратному обеспечению.

Программное обеспечение, напротив, состоит из алгоритмов (подробных последовательностей команд, которые описывают, как решить задачу) и их компьютерных представлений, то есть программ. Программы могут храниться на жестком диске, гибком диске, компакт-диске или других носителях, но в сущности программное обеспечение — это набор команд, составляющих программы, а не физические носители, на которых эти программы записаны.

В самых первых компьютерах граница между аппаратным и программным обеспечением была очевидна. Со временем, однако, произошло значительное размывание этой границы, в первую очередь благодаря тому, что в процессе развития

компьютеров уровни добавлялись, убирались и сливались. В настоящее время очень сложно отделить их друг от друга. В действительности центральная тема этой книги может быть выражена так: *аппаратное и программное обеспечение логически эквивалентны*.

Любая операция, выполняемая программным обеспечением, может быть встроена в аппаратное обеспечение (желательно после того, как она осознана). Карен Панетта Ленц говорил; «Аппаратное обеспечение — это всего лишь окаменевшее программное обеспечение». Конечно, обратное тоже верно: любая команда, выполняемая аппаратным обеспечением, может быть смоделирована в программном обеспечении. Решение разделить функции аппаратного и программного обеспечения основано на таких факторах, как стоимость, скорость, надежность, а также частота ожидаемых изменений. Существует несколько жестких правил, сводящихся к тому, что X должен быть в аппаратном обеспечении, а Y должен программироваться. Эти решения изменяются в зависимости от тенденций в развитии компьютерных технологий.

Изобретение микропрограммирования

У первых цифровых компьютеров в 1940-х годах было только 2 уровня: уровень архитектуры набора команд, на котором осуществлялось программирование, и цифровой логический уровень, который выполнял программы. Схемы цифрового логического уровня были сложны для производства и понимания и ненадежны.

В 1951 году Морис Уилкс, исследователь Кембриджскс-о университета, предложил идею разработки трехуровневого компьютера для того, чтобы упростить аппаратное обеспечение [158]. Эта машина должна была иметь встроенный неизменяемый интерпретатор (микропрограмму), функция которого заключалась в выполнении программ посредством интерпретации. Так как аппаратное обеспечение должно было теперь вместо программ уровня архитектуры команд выполнять только микропрограммы с ограниченным набором команд, требовалось меньшее количество электронных схем. Поскольку электронные схемы тогда делались из электронных ламп, такое упрощение должно было сократить количество ламп и, следовательно, увеличить надежность.

В 50-е годы было построено несколько трехуровневых машин. В 60-х годах число таких машин значительно увеличилось. К 70-м годам идея о том, что написанная программа сначала должна интерпретироваться микропрограммой, а не выполняться непосредственно электроникой, стала преобладающей.

Изобретение операционной системы

В те времена, когда компьютеры только появились, принципы работы с ними сильно отличались от современных. Одним компьютером пользовалось большое количество людей. Рядом с машиной лежал листок бумаги, и если программист хотел запустить свою программу, он записывался на какое-то определенное время, скажем, на среду с 3 часов ночи до 5 утра (многие программисты любили работать в тишине). В назначенное время программист направлялся в комнату, где стояла машина, с пачкой перфокарт, которые тогда служили средством ввода, и карандашом. Каждая перфокарта содержала 80 колонок; на ней в определенных местах

пробивались отверстия. Войдя в комнату, программист вежливо просил предыдущего программиста освободить место и приступал к работе.

Если он хотел запустить программу на языке FORTRAN, ему необходимо было пройти следующие этапы:

1. Он подходил к шкафу, где находилась библиотека программ, брал большую зеленую стопку перфокарт с надписью «Компилятор FORTRAN», помещал их в считывающее устройство и нажимал кнопку «Пуск».
2. Затем он помещал стопку карточек со своей программой, написанной на языке FORTRAN, в считывающее устройство и нажимал кнопку «Продолжить». Программа считывалась.
3. Когда компьютер прекращал работу, программист считывал свою программу во второй раз. Некоторые компиляторы требовали только одного считывания перфокарт, но в большинстве случаев необходимо было производить эту процедуру несколько раз. Каждый раз нужно было считывать большую стопку перфокарт.
4. В конце концов трансляция завершалась. Программист часто становился очень нервным, потому что если компилятор находил ошибку в программе, ему приходилось исправлять ее и начинать процесс ввода программы заново. Если ошибок не было, компилятор выдавал программу на машинном языке на перфокартах.
5. Тогда программист помещал эту программу на машинном языке в устройство считывания вместе с пачкой перфокарт из библиотеки подпрограмм и загружал обе эти программы.
6. Начиналось выполнение программы. В большинстве случаев она не работала или неожиданно останавливалась в середине. Обычно в этом случае программист начинал дергать переключатели на пульте и смотрел на лампочки. В случае удачи он находил и исправлял ошибку, подходил к шкафу, в котором лежал большой зеленый компилятор FORTRAN, и начинал все заново. В случае неудачи он делал распечатку содержания памяти, что называлось **разгрузкой оперативного запоминающего устройства,** и брал эту распечатку домой для изучения.

Это процедура была обычной на протяжении многих лет. Программистам приходилось учиться, как работать с машиной и что нужно делать, если она выходила из строя, что происходило довольно часто. Машина постоянно простаивала без работы, пока люди носили перфокарты по комнате или ломали головы над тем, почему программа не работает.

В 60-е годы человек попытался сократить количество потерянного времени, автоматизировав работу оператора. Программа под названием **«операционная система»** теперь содержалась в компьютере все время. Программист приносил пачку перфокарт со специальной программой, которая выполнялась операционной системой. На рисунке 1.3 показана модель пачки перфокарт для первой широко распространенной операционной системы FMS (FORTRAN Monitor System) к компьютеру IBM-709.

Рис. 1.3. Схема работы с операционной системой FMS

Операционная система считывала перфокарту *JOB и использовала содержащуюся на ней информацию для учета системных ресурсов (звездочка ставилась, чтобы отличать перфокарты с программой контроля от перфокарт с данными). Затем операционная система считывала перфокарту *FORTRAN, которая представляла собой команду для загрузки компилятора FORTRAN с магнитной ленты. После этого компилятор считывал и компилировал программу, написанную на языке FORTRAN. Как только компилятор заканчивал работу, операционная система считывала перфокарту *DATA — команду по выполнению транслированной программы с использованием перфокарт данных.

Операционная система была придумана для того, чтобы автоматизировать работу оператора (отсюда и название), но это не единственное ее назначение. Создание операционной системы было первым шагом в развитии новой виртуальной машины. Перфокарту *FORTRAN можно рассматривать как виртуальную команду к компилятору, а перфокарту *DATA — как виртуальную команду для выполнения программы. И хотя этот уровень состоял всего из двух команд, он был первым этапом в развитии виртуальных машин.

В последующие годы операционные системы все больше и больше усложнялись. К уровню архитектуры команд добавлялись новые команды, приспособления и особенности, и в итоге сформировался новый уровень. Некоторые команды нового уровня были идентичны командам предыдущего, но другие (в частности команды ввода-вывода) полностью отличались. Эти новые команды тогда назывались **макросами операционной системы** или **вызовами супервизора.** Сейчас обычно используется термин **«системный вызов».**

Первые операционные системы считывали пачки перфокарт и распечатывали результат на принтере. Такая организация вычислений называлась **пакетным**

режимом. Чтобы получить результат, обычно нужно было ждать несколько часов. При таких условиях было трудно развивать программное обеспечение.

В начале 60-х годов исследователи Массачусетского технологического института (МТИ) разработали операционную систему, которая давала возможность работать с компьютером сразу нескольким программистам. В этой системе к центральному компьютеру через телефонные линии подсоединялись отдаленные терминалы. Таким образом, центральный процессор разделялся между большим количеством пользователей. Программист мог напечатать свою программу и получить результаты почти сразу прямо в офисе, гараже или где бы то ни было еще (там, где находился терминал). Эти системы назывались (и сейчас называются) **системами с разделением времени.**

Хотя нас интересуют только те части операционной системы, которые интерпретируют команды третьего уровня, необходимо понимать, что это не единственная функция операционных систем.

Перемещение функциональности системы на уровень микрокода

С 1970 года, когда микропрограммирование стало обычным, у производителей появилась возможность вводить новые машинные команды путем расширения микропрограммы, то есть с помощью программирования. Это открытие привело к виртуальному взрыву в производстве программ машинных команд, поскольку производители начали конкурировать друг с другом, стараясь выпустить лучшие программы. Эти команды не представляли особой ценности, поскольку те же задачи можно было легко решить, используя уже существующие программы, но обычно они работали немного быстрее. Например, во многих компьютерах использовалась команда INC (INCrement), которая прибавляла к числу единицу. Тогда уже существовала общая команда сложения ADD, и не было необходимости вводить новую команду, прибавляющую к числу единицу. Тем не менее команда INC работала немного быстрее, чем команда ADD, поэтому ее также включили в набор команд.

Многие программы были добавлены в микропрограмму по той же причине. Среди них можно назвать:

1. Команды для умножения и деления целых чисел.
2. Команды для арифметических действий над числами с плавающей точкой.
3. Команды для вызова и прекращения действия процедур.
4. Команды для ускорения циклов.
5. Команды для работы со строкой символов.

Как только производители поняли, что добавлять новые команды очень легко, они начали думать, какие дополнительные технические характеристики можно добавить к микропрограмме. Приведем несколько примеров:

1. Ускорение работы с массивами (индексная и косвенная адресация).
2. Перемещение программы из одного раздела памяти в другой после запуска программы (настройка).

3. Системы прерывания, которые дают сигнал процессору, как только закончена операция ввода или вывода
4. Способность приостановить одну программу и начать другую, используя небольшое число команд (переключение процесса).

В дальнейшем дополнительные команды и технические характеристики вводились также для ускорения работы компьютеров.

Устранение микропрограммирования

В 60-х-70-х годах количество микропрограмм сильно увеличилось. Однако они работали все медленнее и медленнее, поскольку требовали большого объема памяти. В конце концов исследователи осознали, что с устранением микропрограммы резко сократится количество команд и компьютеры станут работать быстрее. Таким образом, компьютеры вернулись к тому состоянию, в котором они находились до изобретения микропрограммирования.

Мы рассмотрели развитие компьютеров, чтобы показать, что граница между аппаратным и программным обеспечением постоянно перемещается. Сегодняшнее программное обеспечение может быть завтрашним аппаратным обеспечением и наоборот. Так же обстоит дело и с уровнями — между ними нет четких границ. Для программиста не важно, как на самом деле выполняется команда (за исключением, может быть, скорости выполнения). Программист, работающий на уровне архитектуры системы, может использовать команду умножения, как будто это команда аппаратного обеспечения, и даже не задумываться об этом. То, что для одного человека — программное обеспечение, для другого — аппаратное. Мы еще вернемся к этим вопросам ниже.

Развитие компьютерной архитектуры

В период развития компьютерных технологий были разработаны сотни разных компьютеров. Многие из них давно забыты, но некоторые сильно повлияли на современные идеи. В этом разделе мы дадим краткий обзор некоторых ключевых исторических моментов, чтобы лучше понять, каким образом разработчики дошли до создания современных компьютеров. Мы рассмотрим только основные моменты развития, оставив многие подробности за скобками.

Компьютеры, которые мы будем рассматривать, представлены в табл. 1.1.

Нулевое поколение — механические компьютеры (1642-1945)

Первым человеком, создавшим счетную машину, был французский ученый Блез Паскаль (1623-1662), в честь которого назван один из языков программирования. Паскаль сконструировал эту машину в 1642 году, когда ему было всего 19 лет, для своего отца, сборщика налогов. Она была механическая: с шестеренками и ручным приводом. Счетная машина Паскаля могла выполнять только операции сложения и вычитания.

Тридцать лет спустя великий немецкий математик Готфрид Вильгельм Лейбниц (1646-1716) построил другую механическую машину, которая кроме сложения и вычитания могла выполнять операции умножения и деления. В сущности, Лейбниц три века назад создал подобие карманного калькулятора с четырьмя функциями.

Еще через 150 лет профессор математики Кембриджского университета Чарльз Бэббидж (1792-1871), изобретатель спидометра, разработал и сконструировал **разностную машину.** Эта механическая машина, которая, как и машина Паскаля, могла только складывать и вычитать, подсчитывала таблицы чисел для морской навигации. В машину был заложен только один алгоритм — метод конечных разностей с использованием полиномов. У этой машины был довольно интересный способ вывода информации: результаты выдавливались стальным штампом на медной дощечке, что предвосхитило более поздние средства ввода-вывода — перфокарты и компакт-диски.

Хотя это устройство работало довольно неплохо, Бэббиджу вскоре наскучила машина, выполнявшая только один алгоритм. Он потратил очень много времени, большую часть своего семейного состояния и еще 17000 фунтов, выделенных правительством, на разработку **аналитической машины.** У аналитической машины было 4 компонента: запоминающее устройство (память), вычислительное устройство, устройство ввода (для считывания перфокарт), устройство вывода (перфоратор и печатающее устройство). Память состояла из 1000 слов по 50 десятичных разрядов, каждое из которых содержало переменные и результаты. Вычислительное устройство принимало операнды из памяти, затем выполняло операции сложения, вычитания, умножения или деления и возвращало полученный результат обратно в память. Как и разностная машина, это устройство было механическим.

Преимущество аналитической машины заключалось в том, что она могла выполнять разные задачи. Она считывала команды с перфокарт и выполняла их. Некоторые команды приказывали машине взять 2 числа из памяти, перенести их в вычислительное устройство, произвести над ними операцию (например, сложить) и отправить результат обратно в запоминающее устройство. Другие команды проверяли число, а иногда совершали операцию перехода в зависимости от того, положительное оно или отрицательное. Если в считывающее устройство вводились перфокарты с другой программой, то машина выполняла другой набор операций. А разностная машина могла осуществлять только один алгоритм.

Поскольку эта аналитическая машина программировалась на ассемблере, ей было необходимо программное обеспечение. Чтобы создать это программное обеспечение, Бэббидж нанял молодую женщину — Аду Августу ЛовлеЙс, дочь знаменитого британского поэта Байрона. Ада Ловлейс была первым в мире программистом. В ее честь назван современный язык программирования Ada.

К несчастью, Бэббидж никогда не отлаживал компьютер. Ему нужны были тысячи и тысячи шестеренок, сделанных с такой точностью, которая была невозможна в XIX веке. Но идеи Бэббиджа опередили его эпоху, и даже сегодня большинство современных компьютеров по строению сходны с аналитической машиной. Поэтому справедливо будет сказать, что Бэббидж был дедушкой современного цифрового компьютера.

Таблица 1.1. Основные этапы развития компьютеров

Год выпуска	Название компьютера	Создатель	Примечания
1834	Аналитическая машина	Бэббидж	Первая попытка построить цифровой компьютер
1936	Z1	Зус	Первая релейная вычислительная машина
1943	COLOSSUS	Британское правительство	Первый электронный компьютер
1944	Mark!	Айкен	Первый американский многоцелевой компьютер
1946	ENIAC I	Экерт/ Моушли	Отсюда начинается история современных компьютеров
1949	EDSAC	Уилкс	Первый компьютер с программами, хранящимися в памяти
1951	Whirlwind I	МТИ	Первый компьютер реального времени
1952	IAS	Фон Нейман	Этот проект используется в большинстве современных компьютеров
1960	**PDP-1**	DEC	Первый мини-компьютер (продано 50 экземпляров)
1961	**1401**	**IBM**	Очень популярный маленький компьютер
1962	**7094**	**IBM**	Очень популярная небольшая вычислительная машина
1963	B5000	Burroughs	Первая машина, разработанная для языка высокого уровня
1964	**360**	**IBM**	Первое семейство компьютеров
1964	6600	CDC	Первый суперкомпьютер для научных расчетов
1965	PDP-8	DEC	Первый мини-компьютер массового потребления (продано 50 000 экземпляров)
1970	PDP-11	DEC	Эти мини-компьютеры доминировали на компьютерном рынке в 70-е годы.
1974	**8080**	Intel	Первый универсальный 8-битный компьютер на микросхеме
1974	CRAY-1	Cray	Первый векторный супер-компьютер
1978	VAX	DEC	Первый 32-битный суперминикомпьютер
1981	IBM PC	IBM	Началась эра современных персональных компьютеров
1985	MIPS	MIPS	Первый компьютер RISC
1987	SPARC	Sun	Первая рабочая станция RISC на основе процессора SPARC
1990	RS6000	IBM	Первый суперскалярный компьютер

В конце 30-х годов XX века немец Конрад Зус сконструировал несколько автоматических счетных машин с использованием электромагнитных реле. Ему не удалось получить денежные средства от правительства на свои разработки, потому что началась война. Зус ничего не знал о работе Бэббиджа, и его машины были уничтожены во время бомбежки Берлина в 1944 году, поэтому его работа никак не повлияла на будущее развитие компьютерной техники. Однако он был одним из пионеров в этой области.

Немного позже счетные машины были сконструированы в Америке. Машина Атанасова была чрезвычайно развитой для того времени. В ней использовалась бинарная арифметика и информационные емкости, которые периодически обновлялись, чтобы избежать уничтожения данных. Современная динамическая память (ОЗУ) работает точно по такому же принципу. К несчастью, эта машина так и не стала действующей. В каком-то смысле Атанасов был похож на Бэббиджа: мечтатель, которого не устраивали технологии своего времени.

Компьютер Стибитса действительно работал, хотя и был примитивнее, чем машина Атанасова. Стибитс продемонстрировал работу своей машины на конференции в Дартмутском колледже в 1940 году. На этой конференции присутствовал Джон Моушли, ничем не знаменитый профессор физики из университета Пенсильвании. Позднее он стал очень известным в области компьютерных разработок.

Пока Зус, Стибитс и Атанасов разрабатывали автоматические счетные машины, молодой Говард Айкен с трудом проектировал ручные счетные машины как часть своего философского исследования в Гарварде. После окончания исследования Айкен осознал важность автоматических вычислений. Он пошел в библиотеку, узнал о работе Бэббиджа и решил создать из реле такой же компьютер, который Бэббиджу не удалось создать из зубчатых колес.

Работа над первым компьютером Айкена «Mark I» была закончена в 1944 году. Компьютер содержал 72 слова по 23 десятичных разряда каждое и мог выполнить любую команду за 6 секунд. На устройствах ввода-вывода использовалась перфолента. К тому времени, как Айкен закончил работу над компьютером «Mark II», релейные компьютеры уже устарели. Началась эра электроники.

Первое поколение — электронные лампы (1945-1955)

Стимулом к созданию электронного компьютера стала Вторая мировая война. В начале войны германские подводные лодки разрушали британские корабли. Германские адмиралы посылали на подводные лодки по радио команды, а англичане могли перехватывать эти команды. Проблема заключалась в том, что эти радиопослания были закодированы с помощью прибора под названием **ENIGMA,** предшественник которого был спроектирован изобретателем-дилетантом и бывшим президентом США Томасом Джефферсоном.

В начале войны англичанам удалось приобрести ENIGMA у поляков, которые, в свою очередь, украли его у немцев. Однако чтобы расшифровать закодированное послание, требовалось огромное количество вычислений, и их нужно было произвести сразу после того, как радиопослание было перехвачено. Поэтому британское правительство основало секретную лабораторию для создания электронного компьютера под названием **COLOSSUS.** В создании этой машины принимал участие знаменитый британский математик Алан Тьюринг. COLOSSUS работал уже в 1943 году, но так как британское правительство полностью контролировало этот проект и рассматривало его как военную тайну на протяжении 30 лет, COLOSSUS не мог служить основой дальнейшего развития компьютеров. Мы упомянули его только потому, что это был первый в мире электронный цифровой компьютер.

Вторая мировая война повлияла и на развитие компьютерной техники в США. Армии нужны были таблицы стрельбы, которые использовались при нацеливании тяжелой артиллерии. Сотни женщин нанимались для высчитывания этих таблиц на ручных счетных машинах (считалось, что женщины более аккуратны при расчетах, чем мужчины). Тем не менее этот процесс требовал много времени, и часто случались ошибки.

Джон Моушли, который был знаком с работами Атанасова и Стибитса, понимал, что армия заинтересована в создании механических счетных машин. Он потребовал от армии финансирования работ по созданию электронного компьютера. Требование было удовлетворено в 1943 году, и Моушли со своим студентом, Дж. Преспером Экертом, начали конструировать электронный компьютер, который они назвали **ENIAC** (Electronic Numerical Integrator and Computer — электронный цифровой интегратор и калькулятор). Он состоял из 18 000 электровакуумных ламп и 1500 реле. ENIAC весил 30 тонн и потреблял 140 киловатт электроэнергии. У машины было 20 регистров, каждый из которых мог содержать 10-разрядное десятичное число. (Десятичный регистр — это память очень маленького объема, которая может вмещать число до какого-либо определенного максимального количества разрядов, что-то вроде одометра, который запоминает километраж пройденного автомобилем пути.) В ENIAC было установлено 6000 многоканальных переключателей и множество кабелей было протянуто к розеткам.

Работа над машиной была закончена в 1946 году, когда она уже была не нужна. Но поскольку война закончилась, Моушли и Экерту позволили организовать школу, где они рассказывали о своей работе коллегам-ученым. С этой школы началось развитие интереса к созданию больших цифровых компьютеров.

После появления школы и другие исследователи взялись за конструирование электронных вычислительных машин. Первым рабочим компьютером был EDSAC (1949 год). Эту машину сконструировал Морис Уилкс в Кембриджском университете. Далее JOHNIAC — в корпорации Rand, ILLIAC — в Университете Иллинойса, MANIAC — в лаборатории Лос-Аламоса и WEIZAC — в Институте Вайцмана в Израиле.

Экерт и Моушли вскоре начали работу над машиной **EDVAC** (Electronic Discrete Variable Computer — электронная дискретная параметрическая машина). К несчастью, этот проект закрылся, когда они ушли из университета, чтобы основать компьютерную корпорацию в Филадельфии (Силиконовой долины тогда еще не было). После ряда слияний эта компания превратилась в Unisys Corporation.

Экерт и Моушли хотели получить патент на изобретение цифровой вычислительной машины. После нескольких лет судебной тяжбы было вынесено решение, что патент недействителен, так как цифровую вычислительную машину изобрел Атанасов, хотя он и не запатентовал свое изобретение.

В то время как Экерт и Моушли работали над машиной EDVAC, один из участников проекта ENIAC, Джон фон Нейман, поехал в Институт специальных исследований в Принстоне, чтобы сконструировать свою собственную версию EDVAC, машину **IAS**[1]. Фон Нейман был гением в тех же областях, что и Леонардо да Винчи. Он знал много языков, был специалистом в физике и математике и обладал феноменальной памятью; он помнил все, что когда-либо слышал, видел или читал.

[1] Сокр. от Immediate Address Storadge — память с прямой адресацией. — *Примеч. перев.*

Он мог дословно процитировать по памяти тексты книг, которые читал несколько лет назад. Когда фон Нейман стал интересоваться вычислительными машинами, он уже был самым знаменитым математиком в мире.

Фон Нейман вскоре осознал, что создание компьютеров с большим количеством переключателей и кабелей требует длительного времени и очень утомительно. Он пришел к мысли, что программа должна быть представлена в памяти компьютера в цифровой форме, вместе с данными. Он также отметил, что десятичная арифметика, используемая в машине ENIAC, где каждый разряд представлялся 10 электронными лампами (1 включена и 9 выключены), должна быть заменена бинарной арифметикой.

Основной проект, который он описал вначале, известен сейчас как **фон-неймановская вычислительная машина. Он** был использован в EDSAC, первой машине с программой в памяти, и даже сейчас, более чем полвека спустя, является основой большинства современных цифровых компьютеров. Этот замысел и машина IAS оказали очень большое влияние на дальнейшее развитие компьютерной техники, поэтому стоит кратко описать его. Схема архитектуры этой машины дана на рис. 1.4.

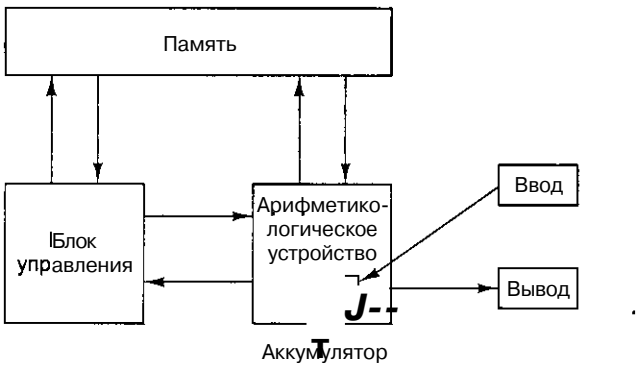


Рис. 1.4. Схема фон-неймановской вычислительной машины

Машина фон Неймана состояла из пяти основных частей: памяти, арифметико-логического устройства, устройства управления, а также устройств ввода-вывода. Память включала 4096 слов, каждое слово содержало 40 битов, бит — это 0 или 1. Каждое слово содержало или 2 команды по 20 битов, или целое число со знаком на 40 битов. 8 битов указывали на тип команды, а остальные 12 битов определяли одно из 4096 слов.

Внутри арифметико-логического устройства находился особый внутренний регистр в 40 битов, так называемый аккумулятор. Типичная команда добавляла слово из памяти к аккумулятору или сохраняла содержимое аккумулятора в памяти. Эта машина не выполняла арифметические операции с плавающей точкой, так как фон Нейман понимал, что любой сведущий математик был способен держать плавающую запятую в голове.

Примерно в то же время, когда фон Нейман работал над машиной IAS, исследователи МТИ разрабатывали свой компьютер Whirlwind I. В отличие от IAS, ENIAC и других машин того же типа со словами большой длины, машина Whirlwind I содержала слова по 16 битов и была предназначена для работы в реальном времени.

Этот проект привел к изобретению памяти на магнитном сердечнике (изобретатель Джей Форрестер), а затем и первого серийного мини-компьютера.

В то время IBM была маленькой компанией, производившей перфокарты и механические машины для их сортировки. Хотя фирма IBM частично финансировала проект Айкена, она не интересовалась компьютерами и только в 1953 году построила компьютер IBM-701, через много лет после того, как компания Экерта и Моушли со своим компьютером UNIVAC стала номером один на компьютерном рынке.

В IBM-701 было 2048 слов по 36 битов, каждое слово содержало две команды. Он стал первым компьютером, лидирующим на рынке в течение десяти лет. Через три года появился IBM-704, у которого было 4 Кбайт памяти на магнитных сердечниках, команды по 36 битов и процессор с плавающей точкой. В 1958 году компания IBM начала работу над последним компьютером на электронных лампах, IBM-709, который по сути представлял собой усложненную версию IBM-704.

Второе поколение — транзисторы (1955-1965)

Транзистор был изобретен сотрудниками лаборатории Bell Laboratories Джоном Бардином, Уолтером Браттейном и Уильямом Шокли, за что в 1956 году они получили Нобелевскую премию в области физики. В течение десяти лет транзисторы произвели революцию в производстве компьютеров, и к концу 50-х годов компьютеры на вакуумных лампах устарели. Первый компьютер на транзисторах был построен в лаборатории МТИ. Он содержал слова из 16 битов, как и Whirlwind I. Компьютер назывался TX-0 (Transistorized experimental computer 0 — экспериментальная транзисторная вычислительная машина 0) и предназначался только для тестирования машины TX-2.

Машина TX-2 не имела большого значения, но один из инженеров из этой лаборатории, Кеннет Ольсен, в 1957 году основал компанию DEC (Digital Equipment Corporation — корпорация по производству цифровой аппаратуры), чтобы производить серийную машину, сходную с TX-0. Эта машина, PDP-1, появилась только через четыре года главным образом потому, что капиталисты, финансирующие DEC, считали производство компьютеров невыгодным. Поэтому компания DEC продавала в основном небольшие электронные платы.

PDP-1 появился только в 1961 году. У него было 4 Кбайт слов по 18 битов и время цикла 5 микросекунд. Этот параметр был в два раза меньше, чем у IBM-7090, транзисторного аналога IBM-709. PDP-1 был самым быстрым компьютером в мире в то время. PDP-1 стоил $120000, а IBM-7090 стоил миллионы. Компания DEC продала десятки компьютеров PDP-1, и так появилась компьютерная промышленность.

Одну из первых машин модели PDP-1 отдали в МТИ, где она сразу привлекла внимание некоторых молодых исследователей, подающих большие надежды. Одним из нововведений PDP-1 был дисплей с размером 512 на 512 пикселов, на котором можно было рисовать точки. Вскоре студенты МТИ составили специальную программу для PDP-1, чтобы играть в «Войну миров» — первую в мире компьютерную игру.

Через несколько лет DEC разработал модель PDP-8, 12-битный компьютер. PDP-8 стоил гораздо дешевле, чем PDP-1 ($16000). Главное нововведение — одна шина (Omnibus) (рис. 1.5). Шина — это набор параллельно соединенных проводов

для связи компонентов компьютера. Это нововведение сильно отличало PDP-8 от IAS. Такая структура с тех пор стала использоваться во всех компьютерах. Компания DEC продала 50 000 компьютеров модели PDP-8 и стала лидером на рынке мини-компьютеров.

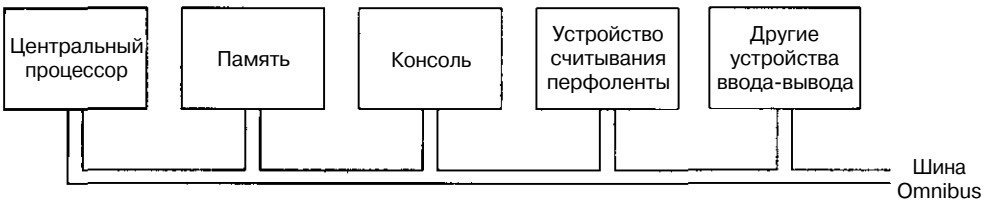


Рис. 1.5. Шина компьютера PDP-8

Как уже было сказано, с изобретением транзисторов компания IBM построила транзисторную версию IBM-709 — IBM-7090, а позднее — IBM-7094. У нее время цикла составляло 2 микросекунды, а память состояла из 32 К слов по 16 битов. IBM-7090 и IBM-7094 были последними компьютерами типа ENIAC, но они широко использовались для научных расчетов в 60-х годах прошлого века.

Компания IBM также выпускала компьютеры IBM-1401 для коммерческих расчетов. Эта машина могла считывать и записывать магнитные ленты и перфокарты и распечатывать результат так же быстро, как и IBM-7094, но при этом стоила дешевле. Для научных вычислений она не подходила, но зато была очень удобна для ведения деловых записей.

У IBM-1401 не было регистров и фиксированной длины слова. Память содержала 4 Кбайт по 8 битов (4 Кбайт). Каждый байт содержал символ в 6 битов, административный бит и бит для указания конца слова. У команды MOVE, например, есть исходный адрес и адрес пункта назначения. Эта команда перемещает байты из первого адреса во второй, пока бит конца слова не примет значение 1.

В 1964 году компания CDC (Control Data Corporation) выпустила машину 6600, которая работала почти на порядок быстрее, чем IBM-7094. Этот компьютер для сложных расчетов пользовался большой популярностью, и компания CDC пошла в гору. Секрет столь высокой скорости работы заключался в том, что внутри ЦПУ (центрального процессора) находилась машина с высокой степенью параллелизма. У нее было несколько функциональных устройств для сложения, умножения и деления, и все они могли работать одновременно. Для того чтобы машина быстро работала, нужно было составить хорошую программу, но приложив некоторые усилия, можно было сделать так, чтобы машина выполняла 10 команд одновременно.

Внутри машины 6600 было встроено несколько маленьких компьютеров. ЦПУ, таким образом, производило только подсчет чисел, а остальные функции (управление работой машины, а также ввод и вывод информации) выполняли маленькие компьютеры. Некоторые принципы устройства 6600 используются и в современных компьютерах.

Разработчик компьютера 6600 Сеймур Крей был легендарной личностью, как и фон Нейман. Он посвятил всю свою жизнь созданию очень мощных компьютеров, которые сейчас называют суперкомпьютерами. Среди них можно назвать CDC-6600, CDC-7600 и Cray-1. Сеймур Крей также является автором известного

«алгоритма покупки автомобилей»: вы идете в магазин, ближайший к вашему дому, показываете на машину, ближайшую к двери, и говорите: «Я беру эту». Этот алгоритм позволяет тратить минимум времени на не очень важные дела (покупку автомобилей) и оставляет большую часть времени на важные (разработку суперкомпьютеров).

Следует упомянуть еще один компьютер — Burroughs B5000. Разработчики машин PDP-1, ШМ-7094 и CDC-6600 занимались только аппаратным обеспечением, стараясь снизить его стоимость (DEC) или заставить работать быстрее (IBM и CDC). Программное обеспечение не менялось. Производители B5000 пошли другим путем. Они разработали машину с намерением программировать ее на языке Algol 60 (предшественнике языка Pascal), сконструировав аппаратное обеспечение так, чтобы упростить задачу компилятора. Так появилась идея, что программное обеспечение также нужно учитывать при разработке компьютера. Но вскоре эта идея была забыта.

Третье поколение — интегральные схемы (1965-1980)

Изобретение кремниевой интегральной схемы в 1958 году (изобретатель — Роберт Нойс) дало возможность помещать десятки транзисторов на одну небольшую микросхему. Компьютеры на интегральных схемах были меньшего размера, работали быстрее и стоили дешевле, чем их предшественники на транзисторах. Ниже описаны наиболее значительные из них.

К 1964 году компания IBM лидировала на компьютерном рынке, но существовала одна большая проблема: компьютеры IBM-7094 и IBM-1401, которые она выпускала, были несовместимы друг с другом. Один из них предназначался для сложных расчетов, в нем использовалась двоичная арифметика на регистрах по 36 битов, а во втором использовалась десятичная система счисления и слова разной длины. У многих покупателей были оба компьютера, и им не нравилось, что они совершенно несовместимы.

Когда пришло время заменить эти две серии компьютеров, компания IBM сделала решительный шаг. Она выпустила серию компьютеров на транзисторах, System/360, которые были предназначены и для научных, и для коммерческих расчетов. System/360 содержала много нововведений. Это было целое семейство компьютеров с одним и тем же языком (ассемблером). Каждая новая модель была больше по размеру и по мощности, чем предыдущая. Компания могла заменить IBM-1401 на IBM-360 (модель 30), а IBM-7094 - на IBM-360 (модель 75). Модель 75 была больше по размеру, работала быстрее и стоила дороже, но программы, написанные для одной из них, могли использоваться для другой. На практике программы, написанные для маленькой модели, выполнялись большой моделью без особых затруднений. Но в случае переноса программного обеспечения с большой машины на маленькую могло не хватить памяти. И все же создание такой серии компьютеров было большим достижением. Идея создания семейств компьютеров вскоре стала очень популярной, и в течение нескольких лет большинство компьютерных компаний выпустило целые серии сходных машин с разной сто-

имостью и функциями. В табл. 1.2 показаны некоторые параметры первых моделей из семейства IBM-360.0 других моделях этого семейства мы расскажем ниже.

Таблица 1.2. Первые модели серии IBM-360

Параметры	Модель 30	Модель 40	Модель 50	Модель 60
Относительная производительность	1	3,5	10	21
Время цикла, не	1000	625	500	250
Максимальный объем памяти, Кбайт	64	256	256	512
Количество байтов, вызываемых из памяти за 1 цикл	1	2	4	16
Максимальное количество каналов данных	3	3	4	6

Еще одно нововведение в IBM-360 — мультипрограммирование. В памяти компьютера могло находиться одновременно несколько программ, и пока одна программа ждала, когда закончится процесс ввода-вывода, другая выполнялась.

IBM-360 была первой машиной, которая могла полностью имитировать работу других компьютеров. Маленькие модели могли имитировать IBM-1401, а большие — IBM-7094, поэтому программисты могли оставлять свои старые программы без изменений и использовать их в работе с IBM-360. Некоторые модели IBM-360 выполняли программы, написанные для IBM-1401, гораздо быстрее, чем сама IBM-1401, поэтому не было никакого смысла в переделывании программ.

Компьютеры серии IBM-360 могли имитировать работу других компьютеров, потому что они создавались с использованием микропрограммирования. Нужно было всего лишь написать три микропрограммы: одну — для системы команд IBM-360, другую — для системы команд IBM-1401 и третью — для системы команд IBM-7094. Требование совместимости было одной из главных причин использования микропрограммирования.

IMB-360 удалось разрешить дилемму между двоичной и десятичной системой: у этого компьютера было 16 регистров по 32 бита для бинарной арифметики, но память состояла из байтов, как у IBM-1401. В ней использовались такие же команды для перемещения записей разного размера из одной части памяти в другую, как и в IBM-1401.

Объем памяти у IBM-360 составлял 2^{24} байтов (16 Мбайт). В те времена такой объем памяти казался огромным. Серия IBM-360 позднее сменилась серией IBM-370, затем IBM-4300, IBM-3080, IBM-3090. У всех этих компьютеров была сходная архитектура. К середине 80-х годов 16 Мбайт памяти стало недостаточно, и компании IBM пришлось частично отказаться от совместимости, чтобы перейти на систему адресов в 32 бита, необходимую для памяти объемом в 2^n байтов.

Можно было бы предположить, что поскольку у машин были слова в 32 бита и регистры, у них вполне могли бы быть и адреса в 32 бита. Но в то время никто не мог даже представить себе компьютер с объемом памяти 16 Мбайт. Обвинять IBM в отсутствии предвидения — все равно что обвинять современных производителей персональных компьютеров в том, что адреса в них всего по 32 бита. Возможно, через несколько лет объем памяти компьютеров будет составлять намного больше 4 Гбайт, и тогда адресов в 32 бита будет недостаточно.

Мир мини-компьютеров сделал большой шаг вперед в третьем поколении вместе с производством серии компьютеров PDP-11, последователей PDP-8со

словами по 16 битов. Во многих отношениях PDP-11 был младшим братом IBM-360, а PDP-1 - младшим братом IBM-7094. И у IBM-360, и у PDP-11 были регистры, слова, память с байтами, и в обеих сериях были компьютеры разной стоимости и с разными функциями. PDP-1 широко использовался, особенно в университетах, и компания DEC продолжала лидировать среди производителей миникомпьютеров.

Четвертое поколение — сверхбольшие интегральные схемы (1980-?)

Появление **сверхбольших интегральных** схем **(СБИС)** в 80-х годах позволило помещать на одну плату сначала десятки тысяч, затем сотни тысяч и, наконец, миллионы транзисторов. Это привело к созданию компьютеров меньшего размера и с более высокой скоростью работы. До появления PDP-1 компьютеры были настолько большие и дорогостоящие, что компаниям и университетам приходилось иметь специальные отделы **(вычислительные центры).** К 80-м годам цены упали так сильно, что возможность приобретать компьютеры появилась не только у организаций, но и у отдельных людей. Началась эра персональных компьютеров.

Персональные компьютеры использовались совсем для других целей. Они применялись для обработки слов, а также для различных диалоговых прикладных программ, с которыми большие компьютеры не могли работать.

Первые персональные компьютеры продавались в виде комплектов. Каждый комплект содержал печатную плату, набор интегральных схем, обычно включающий схему Intel 8080, несколько кабелей, источник питания и иногда 8-дюймовый дисковод. Сложить из этих частей компьютер покупатель должен был сам. Программное обеспечение к компьютеру не прилагалось. Покупателю приходилось самому писать программное обеспечение. Позднее появилась операционная система CP/M, написанная Гари Килдаллом для Intel 8080. Эта действующая операционная система помещалась на дискету, она включала в себя систему управления файлами и интерпретатор для выполнения пользовательских команд, которые набирались с клавиатуры.

Еще один персональный компьютер, Apple (а позднее и Apple II), был разработан Стивом Джобсом и Стивом Возняком. Он стал чрезвычайно популярен среди отдельных покупателей, а также широко использовался в школах, и это сделало компанию Apple серьезным конкурентом IBM.

Наблюдая за тем, чем занимаются другие компании, компания IBM, лидирующая тогда на компьютерном рынке, тоже решила заняться производством персональных компьютеров. Но вместо того чтобы конструировать компьютер с нуля, что заняло бы слишком много времени, компания IBM предоставила одному из своих работников, Филипу Эстриджу, большую сумму денег, приказала ему отправиться куда-нибудь подальше от вмешивающихся во все бюрократов главного управления компании, находящегося в Нью-Йорке, и не возвращаться, пока не будет сконструирован действующий персональный компьютер. Эстридж открыл предприятие достаточно далеко от главного управления компании (во Флориде), взял Intel 8088 в качестве центрального процессора и создал персональный компьютер из серийных компонентов. Этот компьютер (IBM PC) появился в 1981 году **и** стал самым покупаемым компьютером в истории.

Но компания IBM сделала одну вещь, о которой она позже пожалела. Вместо того чтобы держать проект машины в секрете (или по крайней мере оградить себя патентами), как она обычно делала, компания опубликовала полные проекты, включая все электронные схемы, в книге стоимостью $49. Эта книга была опубликована для того, чтобы другие компании могли производить сменные платы для IBM PC, что повысило бы совместимость и популярность этого компьютера. К несчастью для IBM, как только проект IBM PC стал широко известен, поскольку все составные части компьютера можно было легко приобрести, многие компании начали делать клоны PC и часто продавали их гораздо дешевле, чем IBM. Так началось бурное производство персональных компьютеров.

Хотя некоторые компании (такие как Commodore, Apple, Amiga, Atari) производили персональные компьютеры с использованием своих процессоров, а не Intel, потенциал производства IBM PC был настолько велик, что другим компаниям приходилось пробиваться с трудом. Выжить удалось только некоторым из них, и то потому, что они специализировались в узких областях, например в производстве рабочих станций или суперкомпьютеров.

Первая версия IBM PC была оснащена операционной системой MS-DOS, которую выпускала тогда еще крошечная корпорация Microsoft. IBM и Microsoft совместно разработали последовавшую за MS-DOS операционную систему OS/2, характерной чертой которой был графический интерфейс, сходный с интерфейсом Apple Macintosh. Между тем компания Microsoft также разработала собственную операционную систему Windows, которая работала на основе MS-DOS, на случай, если OS/2 не будет иметь спроса. OS/2 действительно не пользовалась спросом, а Microsoft успешно продолжала выпускать операционную систему Windows, что послужило причиной грандиозного раздора между IBM и Microsoft. Легенда о том, как крошечная компания Intel и компания Microsoft, которая была еще меньше, чем Intel, умудрились свергнуть IBM, одну из самых крупных, самых богатых и самых влиятельных корпораций в мировой истории, подробно излагается в бизнес-школах по всему миру.

В середине 80-х годов на смену CISC[1] пришел RISC[2]. Команды RISC были проще и работали гораздо быстрее. В 90-х годах появились суперскалярные процессоры, которые могли выполнять много команд одновременно, часто не в том порядке, в котором они появляются в программе. Мы введем понятия RISC, CISC и суперскалярного процессора в главе 2 и обсудим их подробно.

Типы компьютеров

В предыдущем разделе мы кратко изложили историю компьютерных систем. В этом разделе мы расскажем о положении дел в настоящий момент и сделаем некоторые предположения на будущее. Хотя наиболее известны персональные компьютеры, в наши дни существуют и другие типы машин, поэтому стоит кратко рассказать о них.

[1] Complex instruction set computer — компьютер на микропроцессоре с полным набором команд. — *Примеч. перев.*

[2] Reduced instruction set computer — компьютер с сокращенным набором команд. — *Примеч. перев.*

Технологические и экономические аспекты

Компьютерная промышленность двигается вперед как никакая другая. Главная движущая сила — способность производителей помещать с каждым годом все больше и больше транзисторов на микросхему. Чем больше транзисторов (крошечных электронных переключателей), тем больше объем памяти и мощнее процессоры.

Степень технологического прогресса можно наблюдать, используя **закон Мура,** названный в честь одного из основателей и главы компании Intel Гордона Мура, который открыл его в 1965 году. Когда Мур готовил доклад для промышленной группы, то заметил, что каждое новое поколение микросхем появляется через три года после предыдущего. Поскольку у каждого нового поколения компьютеров было в 4 раза больше памяти, чем у предыдущего, он понял, что число транзисторов на микросхеме возрастает на постоянную величину, и, таким образом, этот рост можно предсказывать на годы вперед. Закон Мура гласит, что число транзисторов на одной микросхеме удваивается каждые 18 месяцев, то есть увеличивается на 60% каждый год. Размеры микросхем и даты их производства, показанные на рис. 1.6, подтверждают, что закон Мура до сих пор действует.

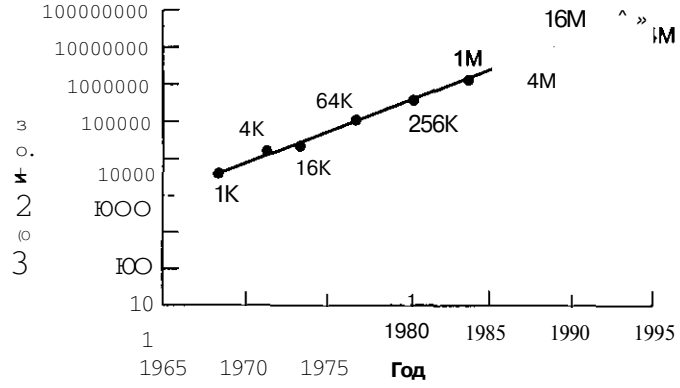


Рис. 1.6. Закон Мура предсказывает, что количество транзисторов на одной микросхеме увеличивается на 60% каждый год. Точки на графике — размер памяти в битах

Конечно, закон Мура — это вообще не закон, а просто эмпирическое наблюдение о том, с какой скоростью физики и инженеры-технологи развивают компьютерные технологии, и предсказание, что с такой скоростью они будут работать и в будущем. Многие специалисты считают, что закон Мура действует и в XXI веке, возможно, до 2020 года. Вероятно, транзисторы скоро будут состоять всего лишь из нескольких атомов, хотя достижения квантовой компьютерной техники, может быть, позволят использовать для размещения 1 бита спин одного электрона.

Закон Мура связан с тем, что некоторые экономисты называют **эффективным циклом.** Достижения в компьютерных технологиях (увеличение количества транзисторов на одной микросхеме) приводят к продукции лучшего качества и более низким ценам. Низкие цены ведут к появлению новых прикладных программ (никому не приходило в голову разрабатывать компьютерные игры, когда каждый компьютер стоил $10 млн). Новые прикладные программы приводят к возникновению новых компьютерных рынков и новых компаний. Существование всех этих

компаний ведет к конкуренции между ними, которая, в свою очередь, создает экономический спрос на лучшие технологии. Круг замыкается.

Еще один фактор развития компьютерных технологий — первый натановский закон программного обеспечения, названный в честь Натана Мирвольда, главного администратора компании Microsoft Этот закон гласит: «Программное обеспечение — это газ. Оно распространяется и полностью заполняет резервуар, в котором находится» В 80-е годы электронная обработка текстов осуществлялась программой troff (программа troff использовалась при создании этой книги). Troff занимает несколько десятков килобайтов памяти. Современные электронные редакторы занимают десятки мегабайтов. В будущем, несомненно, они будут занимать десятки гигабайтов. Программное обеспечение продолжает развиваться и создает постоянный спрос на процессоры, работающие с более высокой скоростью, на больший объем памяти, на большую производительность устройств ввода-вывода.

С каждым годом происходит стремительное увеличение количества транзисторов на одной микросхеме. Отметим, что достижения в развитии других частей компьютера столь же велики. Например, у IBM PC/XT, появившегося в 1982 году, объем жесткого диска составлял всего 10 Мбайт, гораздо меньше, чем у большинства современных настольных компьютеров. Подсчитать, насколько быстро происходит совершенствование жесткого диска, гораздо сложнее, поскольку тут есть несколько параметров (объем, скорость передачи данных, цена и т. д), но измерение любого из этих параметров покажет, что показатели возрастают, по крайней мере, на 50% в год.

Крупные достижения наблюдаются также и в сфере телекоммуникаций и создания сетей. Меньше чем за два десятилетия мы пришли от модемов, передающих информацию со скоростью 300 бит/с, к аналоговым модемам, работающим со скоростью 56 Кбит/с, телефонным линиям ISDN, где скорость передачи информации 2x64 Кбит/с, оптико-волоконным сетям, где скорость уже больше чем 1 Гбит/с. Оптико-волоконные трансатлантические телефонные кабели (например, TAT-12/13) стоят около $700 млн, действуют в течение 10 лет и могут передавать 300 000 звонков одновременно, поэтому стоимость 10-минутной межконтинентальной связи составляет менее 1 цента Лабораторные исследования подтвердили, что возможны системы связи, работающие со скоростью 1 терабит/с (10^{12} бит/с) на расстоянии более 100 км без усилителей, Едва ли нужно упоминать здесь о развитии сети Интернет.

Широкий спектр компьютеров

Ричард Хамминг, бывший исследователь из Bell Laboratories, заметил, что количественное изменение величины на порядок ведет к качественному изменению. Например, гоночная машина, которая может ездить со скоростью 1000 км/ч по пустыне Невада, коренным образом отличается от обычной машины, которая ездит со скоростью 100 км/ч по шоссе Точно так же небоскреб в 100 этажей несопоставим с десятиэтажным многоквартирным домом А если речь идет о компьютерах, то тут за три десятилетия количественные показатели увеличились не в 10, а в 1 000 000 раз.

Развивать компьютерные технологии можно двумя путями: или создавать компьютеры все большей и большей мощности при постоянной цене, или выпускать один и тот же компьютер, с каждым годом снижая цену. Компьютерная промышленность использует оба эти пути, создавая широкий спектр разнообразных компьютеров. Очень приблизительная классификация современных компьютеров представлена в табл. 1.3.

Таблица 1.3. Типы современных компьютеров. Указанные цены приблизительны

Тип	Цена ($)	Сфера применения
«Одноразовые» компьютеры	1	Поздравительные открытки
Встроенные компьютеры	10	Часы, машины, различные приборы
Игровые компьютеры	100	Домашние компьютерные игры
Персональные компьютеры	1000	Настольные и портативные компьютеры
Серверы	10 000	Сетевые серверы
Рабочие станции	100 000	Мини-суперкомпьютеры
Большие компьютеры	1 000 000	Обработка пакетных данных в банке
Суперкомпьютеры	10 000 000	Предсказание погоды на длительный срок

В самой верхней строчке находятся микросхемы, которые приклеиваются на внутреннюю сторону поздравительных открыток для проигрывания мелодий «Happy Birthday», свадебного марша или чего-нибудь подобного. Автор идеи еще не придумал открытки с соболезнованиями, которые играют похоронный марш, но поскольку он выпустил эту идею в потребительскую сферу, вскоре можно будет ожидать появления и таких открыток. Тот, кто воспитывался на компьютерах стоимостью в миллионы долларов, воспринимает такие доступные всем компьютеры примерно так же, как доступный всем самолет. Тем не менее такие компьютеры, вне всяких сомнений, должны существовать (а как насчет говорящих мешков для мусора, которые просят вас не выбрасывать алюминиевые банки?).

Вторая строчка — компьютеры, которые помещаются внутрь телефонов, телевизоров, микроволновых печей, CD-плейеров, игрушек, кукол и т. п. Через несколько лет во всех электрических приборах будут находиться встроенные компьютеры, количество которых будет измеряться в миллиардах. Такие компьютеры состоят из процессора, памяти менее 1 Мбайт и устройств ввода-вывода, и все это на одной маленькой микросхеме, которая стоит всего несколько долларов.

Следующая строка — игровые компьютеры. Это обычные компьютеры с особой графикой, но с ограниченным программным обеспечением и почти полным отсутствием открытости, то есть возможности перепрограммирования. Примерно равны им по стоимости электронные записные книжки и прочие карманные компьютеры, а также сетевые компьютеры и web-терминалы. Все они содержат процессор, несколько мегабайтов памяти, какой-либо дисплей (может быть, даже телевизионный) и больше ничего. Поэтому они такие дешевые.

Далее идут персональные компьютеры. Именно они ассоциируются у большинства людей со словом «компьютер». Персональные компьютеры бывают двух видов: настольные и ноутбуки. Они обычно содержат несколько мегабайтов памяти, жесткий диск с данными на несколько гигабайтов, CD-ROM, модем, звуковую карту

и другие периферийные устройства. Они снабжены сложными операционными системами, имеют возможность наращивания, при работе с ними используется широкий спектр программного обеспечения. Компьютеры с процессором Intel обычно называются «персональными компьютерами», а компьютеры с другими процессорами — «рабочими станциями», хотя особой разницы между ними нет.

Персональные компьютеры и рабочие станции часто используются в качестве сетевых серверов как для локальных сетей (обычно в пределах одной организации), так и для Интернета. У этих компьютеров обычно один или несколько процессоров, несколько гигабайтов памяти и много Гбайт на диске. Такие компьютеры способны работать в сети с очень высокой скоростью. Некоторые из них могут обрабатывать тысячи поступающих сообщений одновременно.

Помимо небольших серверов с несколькими процессорами существуют системы, которые называются **сетями рабочих станций (NOW — Networks of Workstations)** или **кластерами рабочих станций (COW — Clusters of Workstations).** Они состоят из обычных персональных компьютеров или рабочих станций, связанных в сеть, по которой информация передается со скоростью 1 Гбит/с, и специального программного обеспечения, позволяющего всем машинам одновременно работать над одной задачей. Такие системы широко применяются в науке и технике. Кластеры рабочих станций могут включать в себя от нескольких компьютеров до нескольких тысяч. Благодаря низкой цене компонентов отдельные организации могут приобретать такие машины, которые по эффективности являются мини-суперкомпьютерами.

А теперь мы дошли до больших компьютеров размером с комнату, напоминающих компьютеры 60-х годов. В большинстве случаев эти системы — прямые потомки больших компьютеров серии IBM-360. Обычно они работают ненамного быстрее, чем мощные серверы, но у них выше скорость процессов ввода-вывода и обладают они довольно большим пространством на диске — 1 терабайт и более (1 терабайт=10^{12}байт). Такие системы стоят очень дорого и требуют крупных вложений в программное обеспечение, данные и персонал, обслуживающий эти компьютеры. Многие компании считают, что дешевле заплатить несколько миллионов долларов один раз за такую систему, чем даже думать о том, что нужно будет заново программировать все прикладные программы для маленьких компьютеров.

Именно этот класс компьютеров привел к проблеме 2000 года. Проблема возникла из-за того, что в 60-е и 70-е годы программисты, пишущие программы на языке COBOL, представляли год двузначным десятичным числом с целью экономии памяти. Они не смогли предвидеть, что их программное обеспечение будет использоваться через три или четыре десятилетия. Многие компании повторили ту же ошибку, добавив к числу года только два десятичных разряда. Автор этой книги предсказывает, что конец цивилизации произойдет в полночь 31 декабря 9999 года, когда сразу уничтожатся все COBOL-программы, написанные за 8000 лет[1].

Вслед за большими компьютерами идут настоящие суперкомпьютеры. Их процессоры работают с очень высокой скоростью, объем памяти у них составляет множество гигабайтов, диски и сети также работают очень быстро. В последние годы многие суперкомпьютеры стали очень похожи, они почти не отличаются от кластеров рабочих станций, но у них больше составляющих и они работают быстрее.

[1] Необходимо отметить, что в полночь 31 декабря 1999 гада катастрофы не произошло. — *Примеч. перев.*

Суперкомпьютеры используются для решения различных научных и технических задач, которые требуют сложных вычислений, например таких, как моделирование сталкивающихся галактик, разработка новых лекарств, моделирование потока воздуха вокруг крыла аэроплана.

Семейства компьютеров

В этом разделе мы дадим краткое описание трех компьютеров, которые будут использоваться в качестве примеров в этой книге: Pentium II, UltraSPARC II и picojava II.

Pentium II

В 1968 году Роберт Нойс, изобретатель кремниевой интегральной схемы, Гордон Мур, автор известного закона Мура, и Артур Рок, капиталист из Сан-Франциско, основали корпорацию Intel для производства компьютерных микросхем. За первый год своего существования корпорация продала микросхем всего на $3000, но потом объем продаж компании заметно увеличился.

В конце 60-х годов калькуляторы представляли собой большие электромеханические машины размером с современный лазерный принтер и весили около 20 кг. В сентябре 1969 года японская компания Busicom обратилась к корпорации Intel с просьбой выпустить 12 несерийных микросхем для электронной вычислительной машины. Инженер компании Intel Тед Хофф, назначенный на выполнение этого проекта, решил, что можно поместить 4-битный универсальный процессор на одну микросхему, которая будет выполнять те же функции и при этом окажется проще и дешевле. Так в 1970 году появился первый процессор на одной микросхеме, процессор 4004 на 2300 транзисторах.

Заметим, что ни Intel, ни Busicom не имели ни малейшего понятия, какое грандиозное открытие они совершили. Когда компания Intel решила, что стоит попробовать использовать процессор 4004 в других разработках, она предложила купить все права на новую микросхему у компании Busicom за $60000, то есть за сумму, которую Busicom заплатила Intel за разработку этой микросхемы. Busicom сразу приняла предложение Intel, и Intel начала работу над 8-битной версией микросхемы 8008, выпущенной в 1972 году.

Компания Intel не ожидала большого спроса на микросхему 8008, поэтому она выпустила небольшое количество этой продукции. Ко всеобщему удивлению, новая микросхема вызвала большой интерес, поэтому Intel начала разработку еще одного процессора, в котором предел в 16 Кбайт памяти (как у процессора 8008), навязываемый количеством внешних выводов микросхемы, был преодолен. Так появился небольшой универсальный процессор 8080, выпущенный в 1974 году. Как и PDP-8, он произвел революцию на компьютерном рынке и сразу стал массовым продуктом: только компания DEC продала тысячи PDP-8, а Intel — миллионы процессоров 8080.

В 1978 году появился процессор 8086 — 16-битный процессор на одной микросхеме. Процессор 8086 был во многом похож на 8080, но не был полностью совместим с ним. Затем появился процессор 8088 с такой же архитектурой, как и у 8086.

Он выполнял те же программы, что и 8086, но вместо 16-битной шины у него была 8-битная, из-за чего процессор работал медленнее, но стоил дешевле, чем 8086[1]. Когда IBM выбрала процессор 8088 для IBM PC, эта микросхема стала эталоном в производстве персональных компьютеров.

Ни 8088, ни 8086 не могли обращаться к более 1 Мбайт памяти. К началу 80-х годов это стало серьезной проблемой, поэтому компания Intel разработала модель 80286, совместимую с 8086. Основной набор команд остался в сущности таким же, как у процессоров 8086 и 8088, но память была устроена немного по-другому, хотя и могла работать по-прежнему из-за требования совместимости с предыдущими микросхемами. Процессор 80286 использовался в IBM PC/AT и в моделях PS/2. Он, как и 8088, пользовался большим спросом (главным образом потому, что покупатели рассматривали его как более быстрый процессор 8088).

Следующим шагом был 32-битный процессор 80386, выпущенный в 1985 году. Как и 80286, он был более или менее совместим со всеми старыми версиями. Совместимость такого рода оказывалась благом для тех, кто пользовался старым программным обеспечением, и некоторым неудобством для тех, кто предпочитал современную архитектуру, не обремененную ошибками и технологиями прошлого.

Через четыре года появился процессор 80486. Он работал быстрее, чем 80386, мог выполнять операции с плавающей точкой и имел 8 Кбайт кэш-памяти. Кэш-память используется для того, чтобы держать наиболее часто используемые слова внутри центрального процессора и избегать длительного доступа к основной (оперативной) памяти. Иногда кэш-память находится не внутри центрального процессора, а рядом с ним. 80486 содержал встроенные средства поддержки многопроцессорного режима, что давало производителям возможность конструировать системы с несколькими процессорами.

В этот момент Intel, проиграв судебную тяжбу по поводу нарушения правил наименования товаров, выяснила, что номера (например, 80486) не могут быть торговой маркой, поэтому следующее поколение компьютеров получило название Pentium (от греческого слова ΠΕΥΤΕ — пять). В отличие от 80486, у которого был один внутренний конвейер, Pentium имел два, что позволяло работать ему почти в два раза быстрее (конвейеры мы рассмотрим подробно в главе 2).

Когда появилось следующее поколение компьютеров, те, кто рассчитывал на название Sexium (sex по-латыни — шесть), были разочарованы. Название Pentium стало так хорошо известно, что его решили оставить, и новую микросхему назвали Pentium Pro. Несмотря на столь незначительное изменение названия, этот процессор очень сильно отличался от предыдущего. У него была совершенно другая внутренняя организация, и он мог выполнять до пяти команд одновременно.

Еще одно нововведение у Pentium Pro — двухуровневая кэш-память. Процессор содержал 8 Кбайт памяти для часто используемых команд и еще 8 Кбайт для часто используемых данных. В корпусе Pentium Pro рядом с процессором (но не на самой микросхеме) находилась другая кэш-память в 256 Кбайт.

[1] На самом деле разница в стоимости самих микропроцессоров была незначительной. Но компьютеры, собираемые на базе микропроцессора 8088, были дешевле, чем если бы их строили на базе микропроцессора 8086. В то время были распространены 8-битные периферийные устройства, поэтому микропроцессор 8088 позволял упростить сопряжение с внешними устройствами. — *Примеч. научн. ред.*

Вслед за Pentium Pro появился процессор Pentium II, по существу такой же, как и его предшественник, но с особой системой команд для мультимедиа-задач (MMX — multimedia extensions). Эта система команд предназначалась для ускорения вычислений, необходимых при воспроизведении изображения и звука. При наличии MMX специальные сопроцессоры были не нужны. Данные команды имелись в наличии и в более поздних версиях Pentium, но их не было в Pentium Pro. Таким образом, компьютер Pentium II сочетал в себе функции Pentium Pro с мультимедиа-командами.

В начале 1998 года Intel запустил новую линию продукции под названием Celeron. Celeron имел меньшую производительность, чем Pentium II, но зато стоил дешевле. Поскольку у компьютера Celeron такая же архитектура, как у Pentium II, мы не будем обсуждать его в этой книге. В июне 1998 года компания Intel выпустила специальную версию Pentium II — Хеоп. Он имел кэш-память большего объема, его внутренняя шина работала быстрее, были усовершенствованы средства поддержки многопроцессорного режима, но во всем остальном он остался обычным Pentium II, поэтому мы его тоже не будем обсуждать. Компьютеры семейства Intel показаны в табл. 1.4.

Таблица 1.4. Семейство процессоров Intel. Тактовая частота измеряется в МГц (1 МГц = 1 млн циклов/с)

Микросхема	Дата выпуска	Тактовая частота, МГц	Количество транзисторов	Объем памяти	Примечания
4004	4/1971	0,108	**2300**	640 Кбайт	Первый микропроцессор на микросхеме
8008	4/1972	0,08	**3** 500	16 Кбайт	Первый 8-битный микропроцессор
8080	4/1974	2	**6 000**	64 Кбайт	Первый многоцелевой процессор на микросхеме
8086	6/1978	5-10	**29 000**	1 Мбайт	Первый 16-битный процессор на микросхеме
8088	6/1979	5-8	29 000	1 Мбайт	Использовался в IBM PC
80286	2/1982	8-12	134 000	1 Мбайт	Появилась защита памяти
80386	10/1985	16-33	275 000	4 Гбайт	Первый 32-битный процессор
80486	4/1989	25-100	**1** 200 000	4 Гбайт	8 Кбайт кэш-памяти
Pentium	3/1993	60-223	**3** 100 000	4 Гбайт	Два конвейера, у более поздних моделей — MMX
Pentium Pro	3/1995	150-200	**5** 500 000	[1]	Два уровня кэш-памяти
Pentium II	5/1997	233-400	**7** 500 000	64 Гбайт	Pentium Pro + MMX

[1] Шина адреса у микропроцессоров Pentium Pro и Pentium II имеет ширину 36 битов, что позволяет адресовать непосредственно 64 Гбайт. — *Примеч. научи, ред.*

Все микросхемы Intel совместимы со своими предшественниками вплоть до процессора 8086. Другими словами, Pentium II может выполнять программы, написанные для процессора 8086[1]. Совместимость всегда была одним из главных требований при разработке новых компьютеров, чтобы покупатели могли продолжать работать со старым программным обеспечением и не тратить деньги на новое. Конечно, Pentium II во много раз сложнее, чем 8086, поэтому он может выполнять многие функции, которые не способен выполнять процессор 8086. Все эти постепенные доработки в каждой новой версии привели к тому, что архитектура Pentium II не так проста, как могла бы быть, если бы разработчикам процессора Pentium II предоставили 7,5 млн транзисторов и команд, чтобы начать все заново.

Интересно, что хотя закон Мура раньше ассоциировался с числом битов в памяти компьютера, он в равной степени применим и по отношению к процессорам. Если напротив даты выпуска каждой микросхемы поставить число транзисторов на этой микросхеме (количество транзисторов показано в табл. 1.4), мы увидим, что закон Мура действует и здесь. График показан на рис. 1.7.

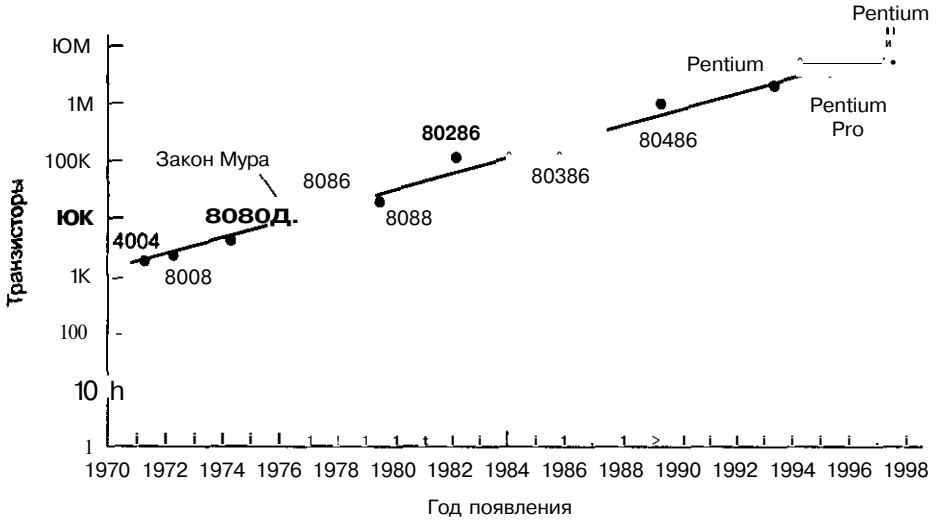


Рис. 1.7. Закон Мура действует и для процессоров

UltraSPARCII

В 70-х годах во многих университетах была очень популярна операционная система UNIX, но персональные компьютеры не подходили для этой операционной системы, поэтому любителям UNIX приходилось работать на мини-компьютерах с разделением времени, таких как PDP-11 и VAX. Энди Бехтольсхайм, аспирант Стэнфордского университета, был очень расстроен тем, что ему нужно посещать

[1] Существуют сотни и тысячи программ, которые не могут быть выполнены на современных быстродействующих микропроцессорах, совместимых с микропроцессором 8086, хотя на более старых (медленных) микропроцессорах они и выполняются. — *Примеч. научн. ред.*

компьютерный центр, чтобы работать с UNIX. В 1981 году он разрешил эту проблему, самостоятельно построив персональную рабочую станцию UNIX из стандартных частей, имеющихся в продаже, и назвал ее SUN-1 (Stanford University Network — сеть Стэнфордского университета).

На Бехтольсхайма скоро обратил внимание Винод Косла, 27-летний индиец, который горел желанием годам к тридцати стать миллионером и уйти от дел. Косла предложил Бехтольсхайму организовать компанию по производству рабочих станций Sun. Он нанял Скота Мак-Нили, другого аспиранта Стэнфордского университета, чтобы тот возглавил производство. Для написания программного обеспечения они наняли Билла Джоя, главного создателя системы UNIX. В 1982 году они вчетвером основали компанию Sun Microsystems. Первый компьютер компании, Sun-1, был оснащен процессором Motorola 68020 и имел большой успех, как и последующие модели Sun-2 и Sun-3, которые также были сконструированы с использованием микропроцессоров Motorola. Эти машины были гораздо мощнее, чем другие персональные компьютеры того времени (отсюда и название «рабочая станция»), и изначально были предназначены для работы в сети. Каждая рабочая станция Sun была оснащена сетевым адаптером Ethernet и программным обеспечением TCP/IP для связи с сетью ARPANET, предшественницей Интернета.

В 1987 году компания Sun, которая к тому времени продавала рабочих станций на полмиллиарда долларов в год, решила разработать свой собственный процессор, основанный на новом революционном проекте калифорнийского университета в Беркли (RISC II). Этот процессор назывался **SPARC (Scalable Processor ARCitecture — наращиваемая архитектура процессора).** Он был использован при производстве рабочей станции Sun-4. Через некоторое время все рабочие станции компании Sun стали производиться на основе этого процессора.

В отличие от многих других компьютерных компаний, Sun решила не заниматься производством процессоров SPARC. Вместо этого она предоставила патент на их изготовление нескольким предприятиям, надеясь, что конкуренция между ними повлечет за собой повышение качества продукции и снижение цен. Эти предприятия выпустили несколько разных микросхем, основанных на разных технологиях, работающих с разной скоростью и отличающихся друг от друга по стоимости. Микросхемы назывались MicroSPARC, HyperSPARK, SuperSPARK и TurboSPARK. Мало чем отличаясь друг от друга, все они были совместимы и могли выполнять одни и те же программы, которые не приходилось изменять.

Компания Sun всегда хотела, чтобы разные предприятия поставляли для SPARK составные части и системы. Нужно было построить целую индустрию, только в этом случае можно было конкурировать с компанией Intel, лидирующей на рынке персональных компьютеров. Чтобы завоевать доверие компаний, которые были заинтересованы в производстве процессоров SPARC, но не хотели вкладывать средства в продукцию, которую будет подавлять Intel, компания Sun создала промышленный консорциум SPARC International для руководства развитием будущих версий архитектуры SPARC. Важно различать архитектуру SPARC, которая представляет собой набор команд, и собственно выполнение этих команд. В этой книге мы будем говорить и об общей архитектуре SPARC, и о процессоре, используемом в рабочей станции SPARC (предварительно обсудив процессоры в третьей и четвертой главах).

Первый SPARC был 32-битным и работал с частотой 36 МГц. Центральный процессор назывался **Ш (Integer Unit — процессор целочисленной арифметики)** и был весьма посредственным. У него было только три основных формата команд и в общей сложности всего 55 команд. С появлением процессора с плавающей точкой добавилось еще 14 команд. Отметим, что компания Intel начала с 8- и 16-битных микросхем (модели 8088, 8086, 80286), а уже потом перешла на 32-битные (модель 80386), а Sun, в отличие от Intel, сразу начала с 32-битных.

Грандиозный перелом в развитии SPARC произошел в 1995 году, когда была разработана 64-битная версия (версия 9) с адресами и регистрами по 64 бит. Первой рабочей станцией с такой архитектурой стал UltraSPARC I, вышедший в свет в 1995 году. Он был полностью совместим с 32-битными версиями SPARC, хотя сам был 64-битным.

В то время как предыдущие машины работали с символьными и числовыми данными, UltraSPARC с самого начала был предназначен для работы с изображениями, аудио, видео и мультимедиа вообще. Среди нововведений, помимо 64-битной архитектуры, появились 23 новые команды, в том числе команды для упаковки и распаковки пикселов из 64-битных слов, масштабирования и вращения изображений, перемещения блоков, а также для компрессии и декомпрессии видео в реальном времени. Эти команды назывались **VIS (Visual Instruction Set)** и предназначались для поддержки мультимедиа. Они были аналогичны командам MMX.

UltraSPARC предназначался для web-серверов с десятками процессоров и физической памятью до 2 Тбайт (терабайт, 1Тбайт = 10^{12} байтов). Тем не менее некоторые версии UltraSPARC могут использоваться и в ноутбуках.

За UltraSPARC I последовали UltraSPARC II и UltraSPARC III. Эти модели отличались друг от друга по скорости, и у каждой из них появлялись какие-то новые особенности. Когда мы будем говорить об архитектуре SPARC, мы будем иметь в виду 64-битную версию компьютера UltraSPARC II (версии 9).

PicoJava II

Язык программирования С придумал один из работников компании Bell Laboratories Деннис Ритчи. Этот язык предназначался для работы в операционной системе UNIX. Из-за большой популярности UNIX С скоро стал доминирующим языком в системном программировании. Через несколько лет Бьярн Строуструп, тоже из компании Bell Laboratories, добавил к С некоторые особенности из объектно-ориентированного программирования, и появился язык С++, который также стал очень популярным.

В середине 90-х годов исследователи в Sun Microsystems думали, как сделать так, чтобы пользователи могли вызывать двоичные программы через Интернет и загружать их как часть web-страниц. Им нравился С++, но он не был надежным в том смысле, что программа, посланная на некоторый компьютер, могла причинить ущерб этому компьютеру. Тогда они решили на основе С++ создать новый язык программирования Java, с которым не было бы подобных проблем. Java — объектно-ориентированный язык, который применяется при решении различных прикладных задач. Поскольку этот язык прост и популярен, мы будем использовать его для примеров.

Поскольку Java — всего лишь язык программирования, можно написать компилятор, который будет преобразовывать его для Pentium, SPARC или любого другого компьютера. Такие компиляторы существуют. Однако этот язык был создан в первую очередь для того, чтобы пересылать программы между компьютерами по Интернету и чтобы пользователям не приходилось изменять их. Но если программа на языке Java компилировалась для SPARC, то когда она пересылалась по Интернету на Pentium, запустить там эту программу было уже нельзя.

Чтобы разрешить эту проблему, компания Sun придумала новую виртуальную машину **JVM (Java Virtual Machine — виртуальная машина Java).** Память у этой машины состояла из 32-битных слов, машина поддерживала 226 команд. Большинство команд были простыми, но выполнение некоторых довольно сложных команд требовало большого количества циклов обращения к памяти.

В компании Sun разработали компилятор, преобразующий программы на языке Java на уровень JVM, и интерпретатор JVM для выполнения этих программ. Этот интерпретатор был написан на языке С и, значит, мог использоваться практически на любом компьютере. Следовательно, чтобы компьютер мог выполнять двоичные программы на языке Java, нужно было всего лишь достать интерпретатор JVM для соответствующего компьютера (например, для Pentium II с системой Windows 98 или для SPARC с системой UNIX) вместе с определенными программами поддержки и библиотеками. Кроме того, большинство браузеров в Интернете содержат интерпретатор JVM, что позволяет легко запускать апплеты (небольшие двоичные программы на Java, связанные со страницами World Wide Web). Большинство этих апплетов поддерживают анимацию и звук.

Интерпретация программ JVM (и любых других программ) происходит медленно. Альтернативный подход — сначала скомпилировать апплет или другую программу JVM для вашей собственной машины, а затем запустить скомпилированную программу. Такая стратегия требует наличия компилятора с JVM на машинный язык внутри браузера и возможности активизировать его, когда необходимо. Эти компиляторы называются **JIT-компиляторами (Just In Time** — «как раз вовремя»), и они широко распространены. Однако эта система создает некоторую задержку между получением JVM-программы и ее выполнением, поскольку JVM-программа компилируется на машинный язык.

Кроме программного обеспечения JVM (JVM-интерпретаторов и JIT-компиляторов) Sun и другие компании разработали микросхемы JVM — процессоры, которые сразу выполняют двоичные программы JVM без какой-либо интерпретации и компиляции. Picojava I и picojava II были разработаны для рынка встроенных систем. На этом рынке требуются мощные и очень дешевые процессоры (цена ниже $50), встраиваемые внутрь пластиковых карточек, телевизоров, телефонов и других устройств, особенно таких, которые обеспечивают связь с внешним миром. Предприятия, имеющие патент на производство микросхем компании Sun, могли производить собственные микросхемы на основе проекта picojava, в той или иной степени изменяя их, включая и убирая процессор с плавающей точкой, преобразуя размер кэш-памяти и т. п.

Ценность микросхемы Java состоит в том, что она способна менять функции в процессе работы. Например, представим себе администратора, у которого есть телефон с процессором Java. Администратору никогда не приходилось читать фак-

сы на крошечном экране телефона, но в один прекрасный день ему это понадобилось. Тогда он звонит провайдеру и просит предоставить ему апплет для просмотра факсов, и таким образом добавляет новую функцию к своему телефону. Но из-за некоторых особенностей прибора и недостатка памяти невозможно использовать интерпретаторы и JIT-компиляторы, поэтому именно в таких случаях необходимы микросхемы JVM.

Picojava II — не физическая микросхема (вы не можете пойти в магазин и купить ее), а проект, который является основой для ряда микросхем, например Sun Microjava 701 и других. Эти микросхемы производятся предприятиями, получившими патент Sun. Мы будем использовать процессор picojava II в качестве иллюстративного примера, поскольку он очень сильно отличается от Pentium II и UltraSPARC II и имеет совершенно другую сферу применения. Picojava II представляет особый интерес для нас, поскольку в главе 4 мы расскажем, как можно создать JVM с помощью микропрограммирования. Тогда мы сможем сравнить спрограммированный JVM с аппаратным обеспечением JVM.

Picojava II содержит два факультативных процессора: кэш-память и процессор с плавающей точкой, которые каждый производитель может включать или не включать в разработку. В целях простоты мы будем рассматривать picojava II как микросхему, хотя на самом деле это не микросхема, а проект микросхемы. Иногда мы будем говорить о микросхеме Sun Microjava 701, которая является воплощением проекта picojava II. Но даже если мы не будем упоминать конкретные микросхемы, читатели должны помнить, что picojava II — это не физическая микросхема, а проект, на основе которого производители разрабатывают разные микросхемы.

Используя Pentium II, UltraSPARC II и picojava II в качестве примеров, мы можем изучить три разных типа процессоров. Первый из них представляет собой CISC с суперскалярным процессором, второй — RISC с суперскалярным процессором. Третий используется во встроенных системах. Эти три процессора сильно отличаются друг от друга, что дает нам возможность лучше увидеть диапазон компьютерных разработок.

Краткое содержание книги

Эта книга о многоуровневых компьютерах и о том, как они организованы (отметим, что почти все современные компьютеры многоуровневые). Подробно мы рассмотрим четыре уровня — цифровой логический уровень, микроархитектурный уровень, уровень архитектуры набора команд и уровень операционной системы. Основные вопросы, которые будут обсуждаться в этой книге, включают общую структуру уровней (и почему уровни построены именно таким образом), типы команд и данных, организацию памяти, адресацию, а также способы построения каждого уровня. Все это называется компьютерной организацией или компьютерной архитектурой.

Мы в первую очередь имеем дело с общими понятиями и не касаемся деталей и строгой математики. По этой причине многие примеры будут сильно упрощены, чтобы сделать упор на основные понятия, а не на детали.

Чтобы разъяснить, как принципы, изложенные в этой книге, могут применяться на практике, мы в качестве примеров будем использовать компьютеры Pentium II, UltraSPARC II и picojava II. Они были выбраны по нескольким причинам. Во-первых, они широко используются, и у читателя наверняка есть доступ хотя бы к одному из них. Во-вторых, каждый из этих компьютеров обладает собственной уникальной архитектурой, что дает основу для сравнения и возможность показать альтернативные варианты. Книги, в которых рассматривается только один компьютер, оставляют у читателя чувство, будто это и есть единственный нормальный компьютер, что является абсурдным в свете огромного числа компромиссов и произвольных решений, которые разработчики вынуждены принимать. Читатель должен рассматривать эти и все другие компьютеры критически и постараться понять, почему дела обстоят именно таким образом и что можно было бы изменить, а не просто принимать их как данность.

Нужно уяснить с самого начала, что эта книга не о том, как программировать Pentium II, UltraSPARC II и picojava II. Эти компьютеры используются только в качестве иллюстративных примеров, и мы не претендуем на их полное описание. Читателям, желающим ознакомиться с этими компьютерами, следует обратиться к публикациям производителей.

Глава 2 знакомит читателей с основными компонентами компьютера: процессорами, памятью, устройствами ввода-вывода. В ней дается краткое описание системной архитектуры и введение к следующим главам.

Главы 3, 4, 5 и 6 касаются каждая одного из уровней, показанных на рис. 1.2. Мы идем снизу вверх, поскольку компьютеры разрабатывались именно таким образом. Структура уровня к в значительной степени определяется особенностями уровня к - 1, поэтому очень трудно понять, как устроен определенный уровень, если не рассмотреть подробно предыдущий, который и определил строение последующего. К тому же с точки зрения обучения логичнее следовать от более простых уровней к более сложным, а не наоборот.

Глава 3 посвящена цифровому логическому уровню, то есть аппаратному обеспечению. В ней рассказывается, что такое вентили и как они объединяются в схемы. В этой главе также вводятся основные понятия булевой алгебры, которая используется для обработки цифровых данных. Кроме того, объясняется, что такое шины, причем особое внимание уделяется популярной шине PCI. В главе приводится много разнообразных примеров, в том числе относящихся к трем компьютерам, упомянутым выше.

Глава 4 знакомит читателя со строением микроархитектурного уровня и принципами его работы. Поскольку функцией этого уровня является интерпретация команд второго уровня, мы сконцентрируемся именно на этом и проиллюстрируем это на примерах. В этой главе также рассказывается о микроархитектурном уровне некоторых конкретных компьютеров.

В главе 5 обсуждается уровень архитектуры команд, который многие называют машинным языком. Здесь мы подробно рассмотрим 3 компьютера, выбранные нами в качестве иллюстративных примеров.

В главе 6 говорится о некоторых командах, об устройстве памяти компьютера, о механизмах управления на уровне операционной системы. В качестве примеров будут использованы операционные системы Windows NT, которая устанавливается на Pentium П, и UNIX, используемая на UltraSPARC П.

Глава 7 — об уровне языка ассемблера. Сюда относится и язык ассемблер, и процесс ассемблирования. Здесь также речь пойдет о компоновке.

В главе 8 обсуждаются параллельные компьютеры, важность которых возрастает с каждым днем. Одни из них действуют на основе нескольких процессоров, которые разделяют общую память, у других общей памяти нет. Одни из них представляют собой суперкомпьютеры, другие — сети рабочих станций Все эти разновидности параллельных компьютеров будут рассмотрены подробно

Глава 9 содержит список рекомендуемой литературы к каждому разделу, а также алфавитный список литературы, цитируемой в этой книге.

Вопросы и задания

1. Объясните следующие термины своими словами:
   1. Транслятор.
   2 Интерпретатор.
   3. Виртуальная машина.
2. Чем отличается интерпретация от трансляции?
3. Может ли компилятор производить данные сразу для микроархитектурного уровня, минуя уровень архитектуры команд? Обоснуйте все доводы за и против.
4. Можете ли вы представить многоуровневый компьютер, у которого уровень внешнего устройства и цифровой логический уровень — не самые нижние уровни? Объясните, почему.
5. Рассмотрим компьютер с идентичными интерпретаторами на первом, втором и третьем уровнях. Чтобы вызвать из памяти, определить и выполнить одну команду, интерпретатору нужно выполнить п команд. Выполнение одной команды первого уровня занимает к не. СКОЛЬКО времени будет занимать выполнение одной команды на втором, третьем и четвертом уровнях?
6. Рассмотрим многоуровневый компьютер, в котором все уровни отличаются друг от друга. Команды каждого уровня в m раз мощнее команд предыдущего уровня, то есть одна команда уровня г может выполнять ту же работу, которую выполняют m команд на уровне г-1. Если для выполнения программы первого уровня требуется к секунд, сколько времени будут выполняться соответствующие программы на втором, третьем и четвертом уровнях, учитывая, что для интерпретации одной команды уровня г+1 требуется п команд уровня г?
7. Некоторые команды уровня операционной системы идентичны командам уровня архитектуры команд. Эти команды сразу выполняются микропрограммой, а не операционной системой. Учитывая ответ на предыдущий вопрос, подумайте, зачем это нужно.
8. В каком смысле аппаратное и программное обеспечение эквивалентны? А в каком не эквивалентны?

9. Одно из следствий идеи фон Неймана хранить программу в памяти компьютера — возможность вносить изменения в программы. Приведите пример, где это может быть полезно (подсказка: подумайте об арифметических операциях над массивами).
10. Работоспособность 75-й модели IBM-360 в 50 раз больше, чем у модели 30, однако время цикла меньше всего лишь в 5 раз. Объясните, почему.
11. На рисунках 1.4 и 1.5 изображены схемы компьютерных систем. Опишите, как происходит процесс ввода-вывода в каждой из этих систем. У какой из них общая производительность больше?
12. В определенный момент времени диаметр транзистора на микропроцессоре составлял один микрон. Каков будет диаметр транзистора на новой модели в следующем году в соответствии с законом Мура?

Глава 2
Организация компьютерных систем

Цифровой компьютер состоит из связанных между собой процессоров, памяти и устройств ввода-вывода. Вторая глава знакомит читателя с этими компонентами и с тем, как они взаимосвязаны. Данная информация послужит основой для подробного рассмотрения каждого уровня в последующих пяти главах. Процессоры, память и устройства ввода-вывода — ключевые понятия, они будут упоминаться при обсуждении каждого уровня, поэтому изучение компьютерной архитектуры мы начнем с них.

Процессоры

На рис. 2.1 показано устройство обычного компьютера. **Центральный процессор** — это мозг компьютера. Его задача — выполнять программы, находящиеся в основной памяти. Он вызывает команды из памяти, определяет их тип, а затем выполняет их одну за другой. Компоненты соединены **шиной,** представляющей собой набор параллельно связанных проводов, по которым передаются адреса, данные и сигналы управления. Шины могут быть внешними (связывающими процессор с памятью и устройствами ввода-вывода) и внутренними.

Процессор состоит из нескольких частей. Блок управления отвечает за вызов команд из памяти и определение их типа. Арифметико-логическое устройство выполняет арифметические операции (например, сложение) и логические операции (например, логическое И).

Внутри центрального процессора находится память для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из нескольких регистров, каждый из которых выполняет определенную функцию. Обычно все регистры одинакового размера. Каждый регистр содержит одно число, которое ограничивается размером регистра. Регистры считываются и записываются очень быстро, поскольку они находятся внутри центрального процессора.

Самый важный регистр — **счетчик команд,** который указывает, какую команду нужно выполнять дальше. Название «счетчик команд» не соответствует действительности, поскольку он ничего не считает, но этот термин употребляется повсеместно[1]. Еще есть **регистр команд,** в котором находится команда, выполняемая в данный

[1] Для этой же цели используется термин «указатель команд». — *Примеч. научи, ред.*

момент. У большинства компьютеров имеются и другие регистры, одни из них многофункциональны, другие выполняют только какие-либо специфические функции.

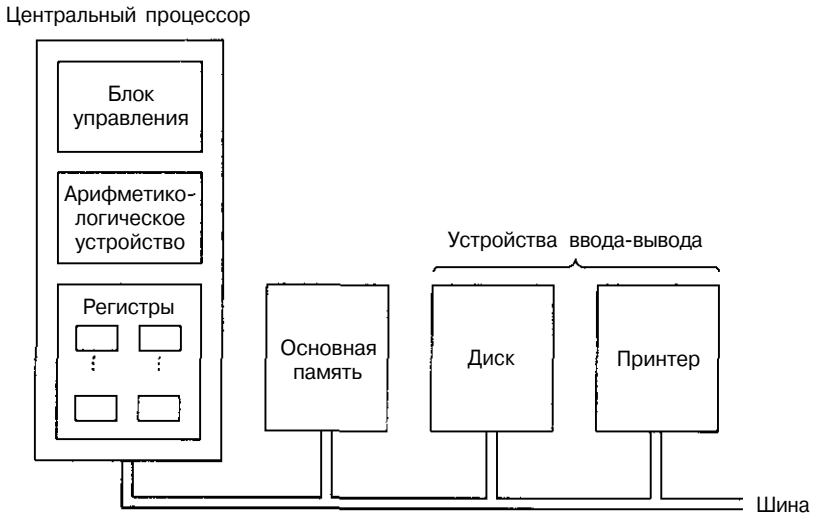


Рис. 2.1. Схема устройства компьютера с одним центральным процессором и двумя устройствами ввода-вывода

Устройство центрального процессора

Внутреннее устройство тракта данных типичного фон-неймановского процессора показано на рис. 2.2. **Тракт данных** состоит из регистров (обычно от 1 до 32), **АЛУ (арифметико-логического устройства)** и нескольких соединяющих шин. Содержимое регистров поступает во входные регистры АЛУ, которые на рис. 2.2 обозначены буквами А и В. В них находятся входные данные АЛУ, пока АЛУ производит вычисления. Тракт данных — важная составная часть всех компьютеров, и мы обсудим его очень подробно.

АЛУ выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Этот выходной регистр может помещаться обратно в один из регистров. Он может быть сохранен в памяти, если это необходимо. На рис. 2.2 показана операция сложения. Отметим, что входные и выходные регистры есть не у всех компьютеров.

Большинство команд можно разделить на две группы: команды типа регистр-память и типа регистр-регистр. Команды первого типа вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ. («Слова» — это такие элементы данных, которые перемещаются между памятью и регистрами[1].) Словом может быть целое число. Устройство памяти мы обсудим ниже в этой главе. Другие команды этого типа помещают регистры обратно в память.

[1] На самом деле размер слова обычно соответствует разрядности регистра данных. Так, например, у 16-битных микропроцессоров 8086 и 8088 слово было 16-битным, а у 32-битных микропроцессоров слово имеет длину 32 бита. — *Примеч. научи, ред*

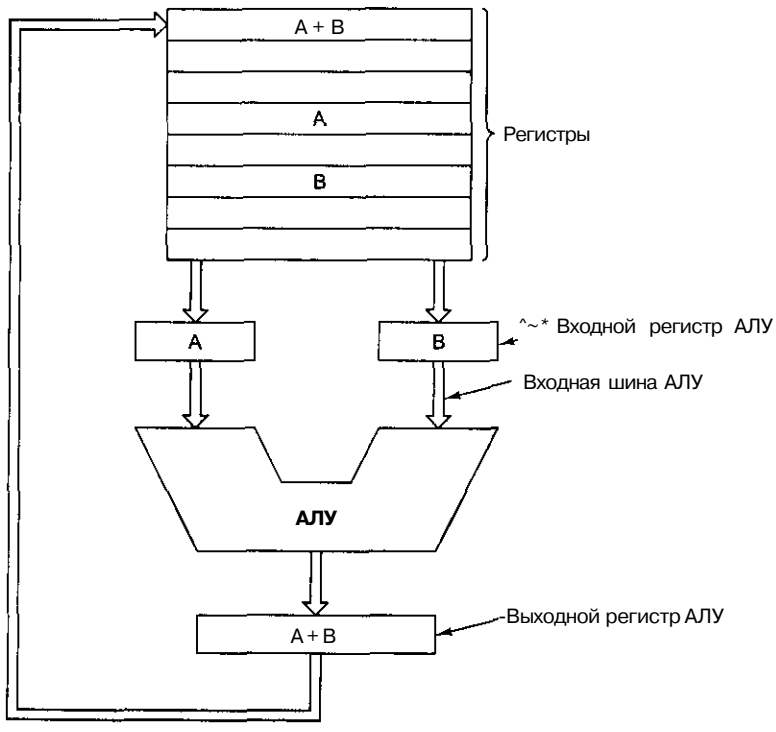


Рис. 2.2. Тракт данных в обычной фон-неймановской машине

Команды второго типа вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из регистров. Этот процесс называется циклом тракта данных. В какой-то степени он определяет, что может делать машина. Чем быстрее происходит цикл тракта данных, тем быстрее компьютер работает.

Выполнение команд

Центральный процессор выполняет каждую команду за несколько шагов:

1) вызывает следующую команду из памяти и переносит ее в регистр команд;
2) меняет положение счетчика команд, который теперь должен указывать на следующую команду[1];
3) определяет тип вызванной команды;
4) если команда использует слово из памяти, определяет, где находится это слово;
5) переносит слово, если это необходимо, в регистр центрального процессора[2];

[1] Это происходит после декодирования текущей команды, а иногда и после ее выполнения. — *Примеч. научи, ред.*

[2] Следует заметить, что бывают команды, которые требуют загрузки из памяти целого множества слов и их обработки в рамках одной-единственной команды. — *Примеч. научн. ред.*

6) выполняет команду;

7) переходит к шагу 1, чтобы начать выполнение следующей команды.

Такая последовательность шагов **(выборка—декодирование—исполнение)** является основой работы всех компьютеров.

Описание работы центрального процессора можно представить в виде программы на английском языке. В листинге 2.1 приведена такая программа-интерпретатор на языке Java. В описываемом компьютере есть два регистра: счетчик команд, который содержит путь к адресу следующей команды, и аккумулятор, в котором хранятся результаты арифметических операций. Кроме того, имеются внутренние регистры, в которых хранится текущая команда (instr), тип текущей команды (instr_type), адрес операнда команды (datajloc) и сам операнд (data). Каждая команда содержит один адрес ячейки памяти. В ячейке памяти находится операнд, например кусок данных, который нужно добавить в аккумулятор.

Листинг 2.1. Интерпретатор для простого компьютера (на языке Java)

```
public class Interp{
    static int PC.            //PC содержит адрес следующей команды
    static int AC;            // аккумулятор, регистр для арифметики
    static int instr.         //регистр для текущей команды
    static int instr_type.    //тип команды (код операции)
    static mt data_loc.      //адрес данных или -1, если его нет
    static int data.          //содержит текущий операнд
    static boolean run_bit = true;  //бит. который можно выключить, чтобы остановить машину

                              public static void interpretCint memory[], int starting_address{
                              //Эта процедура интепретирует программы для простой машины.
которая содержит команды только с одним операндом из памяти Машина содержит регистр AC
(аккумулятор) Он используется для арифметических действий Например, команда ADD суммирует
число из памяти с AC. Интерпретатор работает до тех пор. пока не будет выполнена команда
HALT, вследствие чего бит run_bit поменяет значение на false. Машина состоит из памяти,
счетчика команд, бита run bit и аккумулятора AC Входные параметры состоят из копии
содержимого памяти и начального адреса

        PC=starting_address.
        while (run_bit) {
            instr-memory[PC], //вызывает следующую команду в instr
            PC-PC+1.          //увеличивает значение счетчика команд
            mstr_type=getjnstretype(instr).      //определяет тип команды
            data_loc=find_data(instr, mstrjtype). //находит данные (-1, если данных нет)
            1f(datajoc>=0)   //если data_lock=-l. //значит, операнда нет
               data=memory[data_loc].   //вызов данных
            execute(mstr_type.data),    //выполнение команды
        }
    }

    private static int get_instr_type(mt addr) {.}
    private static int find_dataCint instr. int type) {.}
    private static void executednt type, int data) {..}

}
```

Сама возможность написать программу, имитирующей работу центрального процессора, показывает, что программа не обязательно должна выполняться реальным процессором, относящимся к аппаратному обеспечению. Напротив, вызывать

из памяти, определять тип команд и выполнять эти команды может другая программа. Такая программа называется интерпретатором. Об интерпретаторах мы говорили в главе 1.

Написание программ-интерпретаторов, которые имитируют работу процессора, широко используется при разработке компьютерных систем. После того как разработчики выбрали машинный язык (Я) для нового компьютера, они должны решить, строить ли им процессор, который будет выполнять программы на языке Я, или написать специальную программу для интерпретации программ на языке Я. Если они решают написать интерпретатор, они должны создать аппаратное обеспечение для выполнения этого интерпретатора. Возможны также гибридные конструкции, когда часть команд выполняется аппаратным обеспечением, а часть интерпретируется.

Интерпретатор разбивает команды на маленькие шаги. Таким образом, машина с интерпретатором может быть гораздо проще по строению и дешевле, чем процессор, выполняющий программы без интерпретации. Такая экономия особенно важна, если компьютер содержит большое количество сложных команд с различными опциями. В сущности, экономия проистекает из самой замены аппаратного обеспечения программным обеспечением (интерпретатором).

Первые компьютеры содержали небольшое количество команд, и эти команды были простыми. Но поиски более мощных компьютеров привели, кроме всего прочего, к появлению более сложных команд. Вскоре разработчики поняли, что при наличии сложных команд программы выполняются быстрее, хотя выполнение отдельных команд занимает больше времени. В качестве примеров сложных команд можно назвать выполнение операций с плавающей точкой, обеспечение прямого доступа к элементам массива и т. п. Если обнаруживалось, что две определенные команды часто выполнялись последовательно одна за другой, то вводилась новая команда, заменяющая работу этих двух.

Сложные команды были лучше, потому что некоторые операции иногда перекрывались. Какие-то операции могли выполняться параллельно, для этого использовались разные части аппаратного обеспечения. Для дорогих компьютеров с высокой производительностью стоимость этого дополнительного аппаратного обеспечения была вполне оправданна. Таким образом, у дорогих компьютеров было гораздо больше команд, чем у дешевых. Однако развитие программного обеспечения и требования совместимости команд привели к тому, что сложные команды стали использоваться и в дешевых компьютерах, хотя там во главу угла ставилась стоимость, а не скорость работы.

К концу 50-х годов компания IBM, которая лидировала тогда на компьютерном рынке, решила, что производство семейства компьютеров, каждый из которых выполняет одни и те же команды, имеет много преимуществ и для самой компании, и для покупателей. Чтобы описать этот уровень совместимости, компания IBM ввела термин архитектура. Новое семейство компьютеров должно было иметь одну общую архитектуру и много разных разработок, различающихся по цене и скорости, которые могли выполнять одну и ту же программу. Но как построить дешевый компьютер, который будет выполнять все сложные команды, предназначенные для высокоэффективных дорогостоящих машин?

Решением этой проблемы стала интерпретация. Эта технология, впервые предложенная Уилксом в 1951 году, позволяла разрабатывать простые дешевые компьютеры, которые, тем не менее, могли выполнять большое количество команд. В результате IBM создала архитектуру System/360, семейство совместимых компьютеров, различных по цене и производительности. Аппаратное обеспечение без интерпретации использовалось только в самых дорогих моделях.

Простые компьютеры с интерпретированными командами имели некоторые другие преимущества. Наиболее важными среди них были:

1) возможность фиксировать неправильно выполненные команды или даже восполнять недостатки аппаратного обеспечения;
2) возможность добавлять новые команды при минимальных затратах, даже после покупки компьютера;
3) структурированная организация, которая позволяла разрабатывать, проверять и документировать сложные команды.

В 70-е годы компьютерный рынок быстро разрастался, новые компьютеры могли выполнять все больше и больше функций. Спрос на дешевые компьютеры провоцировал создание компьютеров с использованием интерпретаторов. Возможность разрабатывать аппаратное обеспечение и интерпретатор для определенного набора команд вылилась в создание дешевых процессоров. Полупроводниковые технологии быстро развивались, преимущества низкой стоимости преобладали над возможностями более высокой производительности, и использование интерпретаторов при разработке компьютеров стало широко применимо. Интерпретация использовалась практически во всех компьютерах, выпущенных в 70-е годы, от миникомпьютеров до самых больших машин.

К концу 70-х годов интерпретаторы стали применяться практически во всех моделях, кроме самых дорогостоящих машин с очень высокой производительностью (например, Cray-1 и компьютеров серии Control Data Cyber). Использование интерпретаторов исключало высокую стоимость сложных команд, и разработчики могли вводить все более и более сложные команды, в особенности различные способы определения используемых операндов.

Эта тенденция достигла пика своего развития в разработке компьютера VAX (производитель Digital Equipment Corporation), у которого было несколько сотен команд и более 200 способов определения операндов в каждой команде. К несчастью, архитектура VAX с самого начала разрабатывалась с использованием интерпретатора, а производительности уделялось мало внимания. Это привело к появлению большого количества команд второстепенного значения, которые трудно было выполнять сразу без интерпретации. Данное упущение стало фатальным как для VAX, так и для его производителя (компании DEC). Compaq купил DEC в 1998 году.

Хотя самые первые 8-битные микропроцессоры были очень простыми и содержали небольшой набор команд, к концу 70-х годов даже они стали разрабатываться с использованием интерпретаторов. В этот период основной проблемой для разработчиков стала возрастающая сложность микропроцессоров. Главное преимущество интерпретации заключалось в том, что можно было разработать простой процессор, а вся сложность сводилась к созданию интерпретатора. Таким образом,

разработка сложного аппаратного обеспечения замещалась разработкой сложного программного обеспечения.

Успех Motorola 68000 с большим набором интерпретируемых команд и одновременный провал Zilog Z8000, у которого был столь же обширный набор команд, но не было интерпретатора, продемонстрировали все преимущества использования интерпретаторов при разработке новых машин. Успех Motorola 68000 был несколько неожиданным, учитывая, что Z80 (предшественник Zilog Z8000) пользовался большей популярностью, чем Motorola 6800 (предшественник Motorola 68000). Конечно, важную роль здесь играли и другие факторы, например то, что Motorola много лет занималась производством микросхем, а Exxon (владелец Zilog) долгое время был нефтяной компанией.

Еще один фактор в пользу интерпретации — существование быстрых постоянных запоминающих устройств (так называемых командных ПЗУ) для хранения интерпретаторов. Предположим, что для выполнения обычной интерпретируемой команды Motorola 68000 интерпретатору нужно выполнить 10 команд, которые называются **микрокомандами,** по 100 не каждая, и произвести 2 обращения к оперативной памяти по 500 не каждое. Общее время выполнения команды составит, следовательно, 2000 не, всего лишь в два раза больше, чем в лучшем случае могло бы занять непосредственное выполнение этой команды без интерпретации. А если бы не было специального быстродействующего постоянного запоминающего устройства, выполнение этой команды заняло бы целых 6000 не. Таким образом, важность наличия командных ПЗУ очевидна.

RISC и CISC

В конце 70-х годов проводилось много экспериментов с очень сложными командами, появление которых стало возможным благодаря интерпретации. Разработчики пытались уменьшить пропасть между тем, что компьютеры способны делать, и тем, что требуют языки высокого уровня. Едва ли кто-нибудь тогда думал о разработке более простых машин, так же как сейчас мало кто занимается разработкой менее мощных операционных систем, сетей, редакторов и т. д. (к несчастью).

В компании IBM группа разработчиков во главе с Джоном Коком противостояла этой тенденции; они попытались воплотить идеи Сеймура Крея, создав экспериментальный высокоэффективный мини-компьютер **801.** Хотя IBM не занималась сбытом этой машины, а результаты эксперимента были опубликованы только через несколько лет, весть быстро разнеслась по свету, и другие производители тоже занялись разработкой подобных архитектур.

В 1980 году группа разработчиков в университете Беркли во главе с Дэвидом Паттерсоном и Карло Секвином начала разработку процессоров VLSI без использования интерпретации. Для обозначения этого понятия они придумали термин **RISC** и назвали новый процессор RISC I, вслед за которым вскоре был выпущен RISC II. Немного позже, в 1981 году, Джон Хеннеси в Стенфорде разработал и выпустил другую микросхему, которую он назвал **MIPS.** Эти две микросхемы развились в коммерчески важные продукты SPARC и MIPS соответственно.

Новые процессоры существенно отличались от коммерческих процессоров того времени. Поскольку они не были совместимы с существующей продукцией, раз-

работчики вправе были включать туда новые наборы команд, которые могли бы увеличить общую производительность системы. Так как основное внимание уделялось простым командам, которые могли быстро выполняться, разработчики вскоре осознали, что ключом к высокой производительности компьютера была разработка команд, к выполнению которых можно быстро приступать. Сколько времени занимает выполнение одной команды, было не так важно, как то, сколько команд может быть начато в секунду.

В то время как разрабатывались эти простые процессоры, всеобщее внимание привлекало относительно небольшое количество команд (обычно их было около 50). Для сравнения: число команд в DEC VAX и больших IBM в то время составляло от 200 до 300. RISC — это сокращение от Reduced Instruction Set Computer — компьютер с сокращенным набором команд. RISC противопоставлялся CISC (Complex Instruction Set Computer — компьютер с полным набором команд). В качестве примера CISC можно привести VAX, который доминировал в то время в научных компьютерных центрах. На сегодняшний день мало кто считает, что главное различие RISC и CISC состоит в количестве команд, но название сохраняется до сих пор.

С этого момента началась грандиозная идеологическая война между сторонниками RISC и разработчиками VAX, Intel и больших IBM. По их мнению, наилучший способ разработки компьютеров — включение туда небольшого количества простых команд, каждая из которых выполняется за один цикл тракта данных (см. рис. 2.2), то есть берет два регистра, производит над ними какую-либо арифметическую или логическую операцию (например, сложения или логическое И) и помещает результат обратно в регистр. В качестве аргумента они утверждали, что даже если RISC должна выполнять 4 или 5 команд вместо одной, которую выполняет CISC, притом что команды RISC выполняются в 10 раз быстрее (поскольку они не интерпретируются), он выигрывает в скорости. Следует также отметить, что к этому времени скорость работы основной памяти приблизилась к скорости специальных управляющих постоянных запоминающих устройств, потому недостатки интерпретации были налицо, что повышало популярность компьютеров RISC.

Учитывая преимущества производительности RISC, можно было бы предположить, что такие компьютеры, как Alpha компании DEC, стали доминировать над компьютерами CISC (Pentium и т. д.) на рынке. Однако ничего подобного не произошло. Возникает вопрос: почему?

Во-первых, компьютеры RISC были несовместимы с другими моделями, а многие компании вложили миллиарды долларов в программное обеспечение для продукции Intel. Во-вторых, как ни странно, компания Intel сумела воплотить те же идеи в архитектуре CISC. Процессоры Intel, начиная с 486-го, содержат ядро RISC, которое выполняет самые простые (и обычно самые распространенные) команды за один цикл тракта данных, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие — медленно. Хотя при таком «гибридном» подходе работа происходит не так быстро, как у RISC, данная архитектура имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений.

Принципы разработки современных компьютеров

Прошло уже более двадцати лет с тех пор, как были сконструированы первые компьютеры RISC, однако некоторые принципы разработки можно перенять, учитывая современное состояние технологий аппаратного обеспечения. Если происходит очень резкое изменение в технологиях (например, новый процесс производства делает время цикла памяти в 10 раз меньше, чем время цикла центрального процессора), меняются все условия. Поэтому разработчики всегда должны учитывать возможные технологические изменения, которые могут повлиять на баланс между компонентами компьютера.

Существует ряд принципов разработки, иногда называемых **принципами RISC,** которым по возможности стараются следовать производители универсальных процессоров. Из-за некоторых внешних ограничений, например требования совместимости с другими машинами, приходится время от времени идти на компромисс, но эти принципы — цель, к которой стремится большинство разработчиков. Ниже мы обсудим некоторые из них.

Все команды непосредственно выполняются аппаратным обеспечением. Все обычные команды непосредственно выполняются аппаратным обеспечением. Они не интерпретируются микрокомандами. Устранение уровня интерпретации обеспечивает высокую скорость выполнения большинства команд. В компьютерах типа CISC более сложные команды могут разбиваться на несколько частей, которые затем выполняются как последовательность микрокоманд. Эта дополнительная операция снижает скорость работы машины, но она может быть применима для редко встречающихся команд.

Компьютер должен начинать выполнение большого числа команд. В современных компьютерах используется много различных способов для увеличения производительности, главное из которых — возможность обращаться к как можно большему количеству команд в секунду. Процессор 500-MIPS способен приступать к выполнению 500 млн команд в секунду, и при этом не имеет значения, сколько времени занимает само выполнение этих команд **(MIPS** — это сокращение от **Millions of Instructions Per Second — «миллионы команд в секунду».)** Этот принцип предполагает, что параллелизм может играть главную роль в улучшении производительности, поскольку приступать к большому количеству команд за короткий промежуток времени можно только в том случае, если одновременно может выполняться несколько команд.

Хотя команды некоторой программы всегда расположены в определенном порядке, компьютер может приступать к их выполнению и в другом порядке (так как необходимые ресурсы памяти могут быть заняты) и, кроме того, может заканчивать их выполнение не в том порядке, в котором они расположены в программе. Конечно, если команда 1 устанавливает регистр, а команда 2 использует этот регистр, нужно действовать с особой осторожностью, чтобы команда 2 не считывала регистр до тех пор, пока он не будет содержать нужное значение. Чтобы не допустить подобных ошибок, необходимо вводить большое количество соответствующих записей в память, но производительность все равно становится выше благодаря возможности выполнять несколько команд одновременно.

Команды должны легко **декодироваться.** Предел количества вызываемых команд в секунду зависит от процесса декодирования отдельных команд. Декодирование команд осуществляется для того, чтобы определить, какие ресурсы им необходимы и какие действия нужно выполнить. Полезны любые средства, которые способствуют упрощению этого процесса. Например, используются регулярные команды с фиксированной длиной и с небольшим количеством полей. Чем меньше разных форматов команд, тем лучше.

К памяти должны **обращаться только команды загрузки и сохранения.** Один из самых простых способов разбивания операций на отдельные шаги — потребовать, чтобы операнды для большинства команд брались из регистров и возвращались туда же. Операция перемещения операндов из памяти в регистры может осуществляться в разных командах. Поскольку доступ к памяти занимает много времени, а подобная задержка нежелательна, работу этих команд могут выполнять другие команды, если они не делают ничего, кроме передвижения операндов между регистрами и памятью. Из этого наблюдения следует, что к памяти должны обращаться только команды загрузки и сохранения (LOAD и STORE).

Должно быть большое количество регистров. Поскольку доступ к памяти происходит довольно медленно, в компьютере должно быть много регистров (по крайней мере 32). Если слово однажды вызвано из памяти, при наличии большого числа регистров оно может содержаться в регистре до тех пор, пока будет не нужно. Возвращение слова из регистра в память и новая загрузка этого же слова в регистр нежелательны. Лучший способ избежать излишних перемещений — наличие достаточного количества регистров.

Параллелизм на уровне команд

Разработчики компьютеров стремятся к тому, чтобы улучшить производительность машин, над которыми они работают. Один из способов заставить процессоры работать быстрее — увеличение их скорости, однако при этом существуют некоторые технологические ограничения, связанные с конкретным историческим периодом. Поэтому большинство разработчиков для достижения лучшей производительности при данной скорости работы процессора используют параллелизм (возможность выполнять две или более операций одновременно).

Существует две основные формы параллелизма: параллелизм на уровне команд и параллелизм на уровне процессоров. В первом случае параллелизм осуществляется в пределах отдельных команд и обеспечивает выполнение большого количества команд в секунду. Во втором случае над одной задачей работают одновременно несколько процессоров. Каждый подход имеет свои преимущества. В этом разделе мы рассмотрим параллелизм на уровне команд, а в следующем — параллелизм на уровне процессоров.

Конвейеры

Уже много лет известно, что главным препятствием высокой скорости выполнения команд является их вызов из памяти. Для разрешения этой проблемы разработчики придумали средство для вызова команд из памяти заранее, чтобы они

имелись в наличии в тот момент, когда будут необходимы. Эги команды помещались в набор регистров, который назывался **буфером выборки с упреждением.** Таким образом, когда была нужна определенная команда, она вызывалась прямо из буфера, и не нужно было ждать, пока она считается из памяти. Эта идея использовалась еще при разработке IBM Stretch, который был сконструирован в 1959 году.

В действительности процесс выборки с упреждением подразделяет выполнение команды на два этапа: вызов и собственно выполнение. Идея **конвейера** еще больше продвинула эту стратегию вперед. Теперь команда подразделялась уже не на два, а на несколько этапов, каждый из которых выполнялся определенной частью аппаратного обеспечения, причем все эти части могли работать параллельно.

На рис. 2.3, *а* изображен конвейер из 5 блоков, которые называются стадиями. Стадия С1 вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не будет нужна. Стадия С2 декодирует эту команду, определяя ее тип и тип операндов, над которыми она будет производить определенные действия. Стадия С3 определяет местонахождение операндов и вызывает их или из регистров, или из памяти. Стадия С4 выполняет команду, обычно путем провода операндов через тракт данных (см. рис. 2.2). И наконец, стадия С5 записывает результат обратно в нужный регистр.

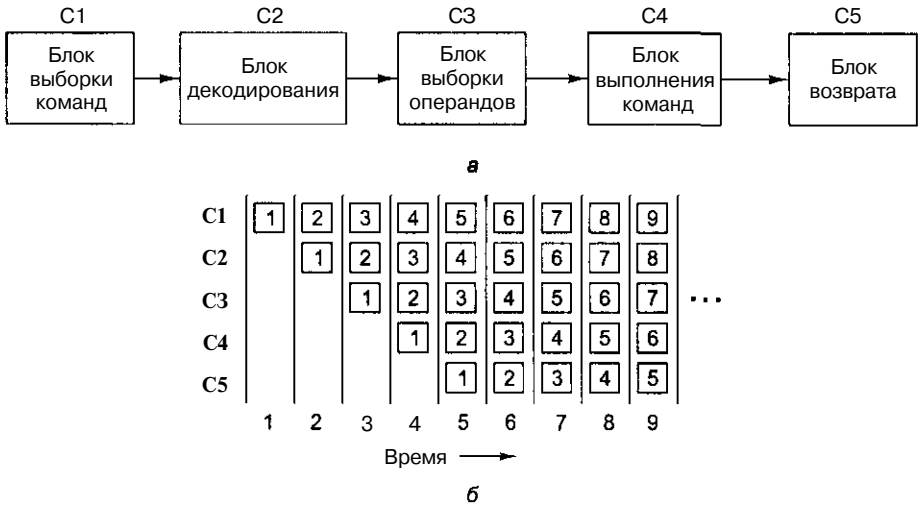


Рис. 2.3. Конвейер из 5 стадий (а); состояние каждой стадии в зависимости от количества пройденных циклов (б). Показано 9 циклов

На рис. 2.3, *б* мы видим, как действует конвейер во времени. Во время цикла 1 стадия С1 работает над командой 1, вызывая ее из памяти. Во время цикла 2 стадия С2 декодирует команду 1, в то время как стадия С1 вызывает из памяти команду 2. Во время цикла 3 стадия С3 вызывает операнды для команды 1, стадия С2 декодирует команду 2, а стадия С1 вызывает третью команду. Во время цикла 4 стадия С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4. Наконец, во время пятого цикла С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие стадии работают над следующими командами.

Чтобы лучше понять принципы работы конвейера, рассмотрим аналогичный пример. Представим себе кондитерскую фабрику» на которой выпечка тортов и их упаковка для отправки производятся раздельно. Предположим, что в отделе отправки находится длинный конвейер, вдоль которого стоят 5 рабочих (или блоков обработки). Каждые 10 секунд (это время цикла) первый рабочий ставит пустую коробку для торта на ленту конвейера. Эта коробка отправляется ко второму рабочему, который кладет в нее торт. После этого коробка с тортом доставляется третьему рабочему, который закрывает и запечатывает ее. Затем она поступает к четвертому рабочему, который ставит на ней ярлык. Наконец, пятый рабочий снимает коробку с конвейерной ленты и помещает ее в большой контейнер для отправки в супермаркет. Примерно таким же образом действует компьютерный конвейер: каждая команда (в случае с кондитерской фабрикой — торт) перед окончательным выполнением проходит несколько шагов обработки.

Возвратимся к нашему конвейеру, изображенному на рис. 2.3. Предположим, что время цикла у этой машины 2 не. Тогда для того, чтобы одна команда прошла через весь конвейер, требуется 10 не. На первый взгляд может показаться, что такой компьютер может выполнять 100 млн команд в секунду, в действительности же скорость его работы гораздо выше. Во время каждого цикла (2 не) завершается выполнение одной новой команды, поэтому машина выполняет не 100 млн, а 500 млн команд в секунду.

Конвейеры позволяют найти компромисс между **временем ожидания** (сколько времени занимает выполнение одной команды) и **пропускной способностью процессора** (сколько миллионов команд в секунду выполняет процессор). Если время цикла составляет Т не, а конвейер содержит п стадий, то время ожидания составит пТ не, а пропускная способность — 1000/Т млн команд в секунду.

Суперскалярные архитектуры

Один конвейер — хорошо, а два — еще лучше. Одна из возможных схем процессора с двойным конвейером показана на рис. 2.4. В основе разработки лежит конвейер, изображенный на рис. 2.3. Здесь общий отдел вызова команд берет из памяти сразу по две команды и помешает каждую из них в один из конвейеров. Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать при использовании ресурсов (например, регистров), и ни одна из них не должна зависеть от результата выполнения другой. Как и в случае с одним конвейером, либо компилятор должен следить, чтобы не возникало неприятных ситуаций (например, когда аппаратное обеспечение выдает некорректные результаты, если команды несовместимы), либо же конфликты выявляются и устраняются прямо во время выполнения команд благодаря использованию дополнительного аппаратного обеспечения.

Сначала конвейеры (как двойные, так и одинарные) использовались только в компьютерах RISC. У 386-го и его предшественников их не было. Конвейеры в процессорах компании Intel появились только начиная с 486-й модели[1].486-й процес-

[1] Необходимо отметить, что параллельное функционирование отдельных блоков процессора использовалось и в предыдущем — 386-м — микропроцессоре. Оно стало прообрщом 5-стадийного конвейера микропроцессора 486. — *Примеч. научи ред.*

сор содержал один конвейер, а Pentium — два конвейера из пяти стадий. Похожая схема изображена на рис. 2.4, но разделение функций между второй и третьей стадиями (они назывались декодирование 1 и декодирование 2) было немного другим. Главный конвейер **(u-конвейер)** мог выполнять произвольные команды. Второй конвейер **(v-конвейер)** мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH).

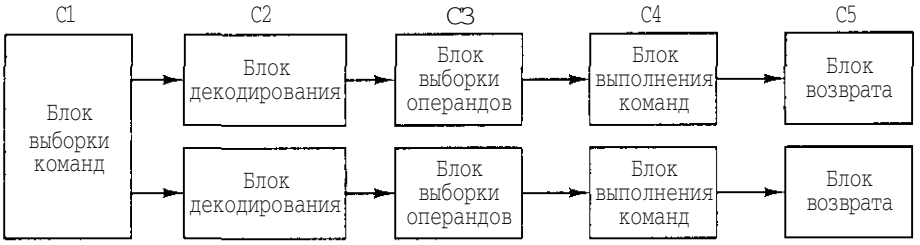


Рис. 2.4. Двойной конвейер из пяти стадий с общим отделом вызова команд

Имеются сложные правила определения, является ли пара команд совместимой для того, чтобы выполняться параллельно. Если команды, входящие в пару, были сложными или несовместимыми, выполнялась только одна из них (в и-конвейере). Оставшаяся вторая команда составляла затем пару со следующей командой. Команды всегда выполнялись по порядку. Таким образом, Pentium содержал особые компиляторы, которые объединяли совместимые команды в пары и могли порождать программы, выполняющиеся быстрее, чем в предыдущих версиях. Измерения показали, что программы, производящие операции с целыми числами, на компьютере Pentium выполняются почти в два раза быстрее, чем на 486-м, хотя у него такая же тактовая частота. Вне всяких сомнений, преимущество в скорости появилось благодаря второму конвейеру.

Переход к четырем конвейерам возможен, но это потребовало бы создания громоздкого аппаратного обеспечения (отметим, что компьютерщики, в отличие от фольклористов, не верят в счастливое число три). Вместо этого используется другой подход. Основная идея — один конвейер с большим количеством функциональных блоков, как показано па рис. 2.5. Pentium II, к примеру, имеет сходную структуру (подробно мы рассмотрим его в главе 4). В 1987 году для обозначения этого подхода был введен термин **суперскалярная архитектура.** Однако подобная идея нашла воплощение еще более 30 лет назад в компьютере CDC 6600. CDC 6600 вызывал команду из памяти каждые 100 не и помещал ее в один из 10 функциональных блоков для параллельного выполнения. Пока команды выполнялись, центральный процессор вызывал следующую команду.

Отметим, что стадия 3 выпускает команды значительно быстрее, чем стадия 4 способна их выполнять. Если бы стадия 3 выпускала команду каждые 10 не, а все функциональные блоки выполняли бы свою работу также за 10 не, то на четвертой стадии всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной. В действительности большинству функциональных блоков четвертой стадии для выполнения команды требуется значительно больше

времени, чем занимает один цикл (это блоки доступа к памяти и блок выполнения операций с плавающей точкой). Как видно из рис. 2 5, на четвертой стадии может быть несколько АЛУ.

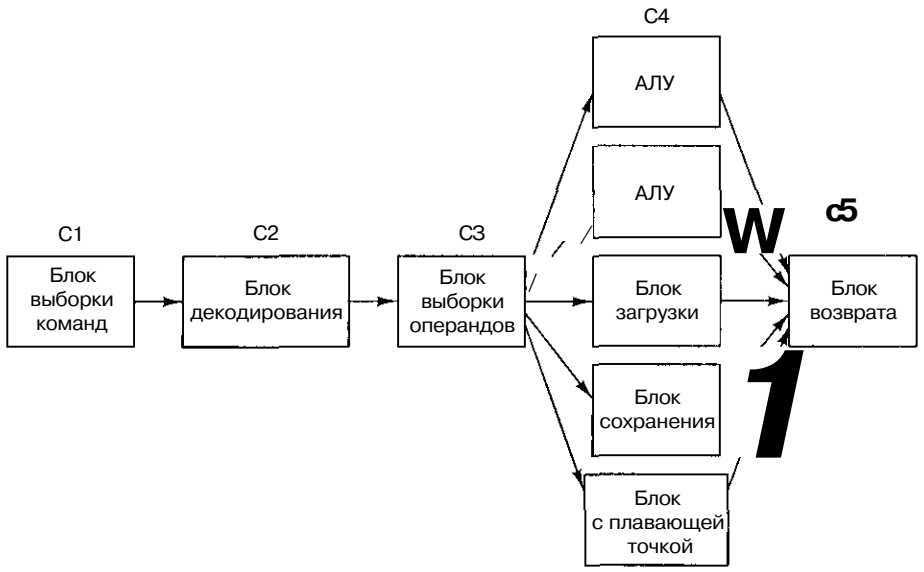


Рис. 2.5. Суперскалярный процессор с пятью функциональными блоками

Параллелизм на уровне процессоров

Спрос на компьютеры, работающие все с более и более высокой скоростью, не прекращается. Астрономы хотят выяснить, что произошло в первую микросекунду после большого взрыва, экономисты хотят смоделировать всю мировую экономику, подростки хотят играть в 3D интерактивные игры со своими виртуальными друзьями через Интернет. Скорость работы процессоров повышается, но у них постоянно возникают проблемы с быстротой передачи информации, поскольку скорость распространения электромагнитных волн в медных проводах и света в оптико-волоконных кабелях по-прежнему остается 20 см/нс, независимо от того, насколько умны инженеры компании Intel. Кроме того, чем быстрее работает процессор, тем сильнее он нагревается[1], и нужно предохранять его от перегрева.

Параллелизм на уровне команд помогает в какой-то степени, но конвейеры и суперскалярная архитектура обычно увеличивают скорость работы всего лишь в 5-10 раз. Чтобы улучшить производительность в 50, 100 и более раз, нужно разрабатывать компьютеры с несколькими процессорами. Ниже мы ознакомимся с устройством таких компьютеров.

[1] Это не совсем точно Есть масса живых примеров, противоречащих этому высказыванию Тепловыделение, конечно, зависит от частоты переключений элементов, но оно зависит и от размеров этих элементов, и от напряжения, от которого они работают. — *Примеч научи ред*

Векторные компьютеры

Многие задачи в физических и технических науках содержат векторы, в противном случае они имели бы очень сложную структуру. Часто одни и те же вычисления выполняются над разными наборами данных в одно и то же время. Структура этих программ позволяет повышать скорость работы благодаря параллельному выполнению команд. Существует два метода, которые используются для быстрого выполнения больших научных программ. Хотя обе схемы во многих отношениях схожи, одна из них считается расширением одного процессора, а другая — параллельным компьютером.

Массивно-параллельный процессор (array processor) состоит из большого числа сходных процессоров, которые выполняют одну и ту же последовательность команд применительно к разным наборам данных. Первым в мире таким процессором был ILLIAC IV (Университет Иллинойса). Он изображен на рис. 2.6. Первоначально предполагалось сконструировать машину, состоящую из четырех секторов, каждый из которых содержит решетку 8x8 элементов процессор/память. Для каждого сектора имелся один блок контроля. Он рассылал команды, которые выполнялись всеми процессорами одновременно, при этом каждый процессор использовал свои собственные данные из своей собственной памяти (загрузка данных происходила во время инициализации). Из-за очень высокой стоимости был построен только один такой сектор, но он мог выполнять 50 млн операций с плавающей точкой в секунду. Если бы при создании машины использовалось четыре сектора и она могла бы выполнять 1 млрд операций с плавающей точкой в секунду, то мощность такой машины в два раза превышала бы мощность компьютеров всего мира.

Рис. 2.6. Массивно-параллельный процессор ILLIAC IV

Для программистов **векторный процессор (vector processor)** очень похож на массивно-параллельный процессор (array processor). Как и массивно-параллельный процессор, он очень эффективен при выполнении последовательности операций над парами элементов данных. Но, в отличие от первого (array processor), все операции сложения выполняются в одном блоке суммирования, который имеет

конвейерную структуру. Компания Cray Research, основателем которой был Сеймур Крей, выпустила много векторных процессоров, начиная с модели Cray-1 (1974) и по сей день. Cray Research в настоящее время входит в состав SGI.

Оба типа процессоров работают с массивами данных. Оба они выполняют одни и те же команды, которые, например, попарно складывают элементы для двух векторов. Но если у массивно-параллельного процессора (array processor) есть столько же суммирующих устройств, сколько элементов в массиве, векторный процессор (vector processor) содержит **векторный регистр,** который состоит из набора стандартных регистров. Эти регистры последовательно загружаются из памяти при помощи одной команды. Команда сложения попарно складывает элементы двух таких векторов, загружая их из двух векторных регистров в суммирующее устройство с конвейерной структурой. В результате из суммирующего устройства выходит другой вектор, который или помещается в векторный регистр, или сразу используется в качестве операнда при выполнении другой операции с векторами.

Массивно-параллельные процессоры (array processor) выпускаются до сих пор, но занимают незначительную сферу компьютерного рынка, поскольку они эффективны при решении только таких задач, которые требуют одновременного выполнения одних и тех же вычислений над разными наборами данных. Массивно-параллельные процессоры (array processor) могут выполнять некоторые операции гораздо быстрее, чем векторные компьютеры (vector computer), но они требуют большего количества аппаратного обеспечения, и для них сложно писать программы. Векторный процессор (vector processor), с другой стороны, можно добавлять к обычному процессору. В результате те части программы, которые могут быть преобразованы в векторную форму, выполняются векторным блоком, а остальная часть программы — обычным процессором.

Мультипроцессоры

Элементы массивно-параллельного процессора связаны между собой, поскольку их работу контролирует один блок управления. Система нескольких параллельных процессоров, разделяющих общую память, называется **мультипроцессором.** Поскольку каждый процессор может записывать или считывать информацию из любой части памяти, их работа должна согласовываться программным обеспечением, чтобы не допустить каких-либо пересечений.

Возможны разные способы воплощения этой идеи. Самый простой из них — наличие одной шипы, соединяющей несколько процессоров и одну общую память. Схема такого мультипроцессора показана на рис. 2.7, *а.* Такие системы производят многие компании.

Нетрудно понять, что при наличии большого числа быстро работающих процессоров, которые постоянно пытаются получить доступ к памяти через одну и ту же шину, будут возникать конфликты. Чтобы разрешить эту проблему и повысить производительность компьютера, были разработаны различные модели. Одна из них изображена на рис. 2.7, *б.* В таком компьютере каждый процессор имеет свою собственную локальную память, которая недоступна для других процессоров. Эта память используется для программ и данных, которые не нужно разделять между несколькими процессорами. При доступе к локальной памяти главная шина не используется, и, таким образом, поток информации в этой шине снижается. Возможны и другие варианты решения проблемы (например, кэш-память).

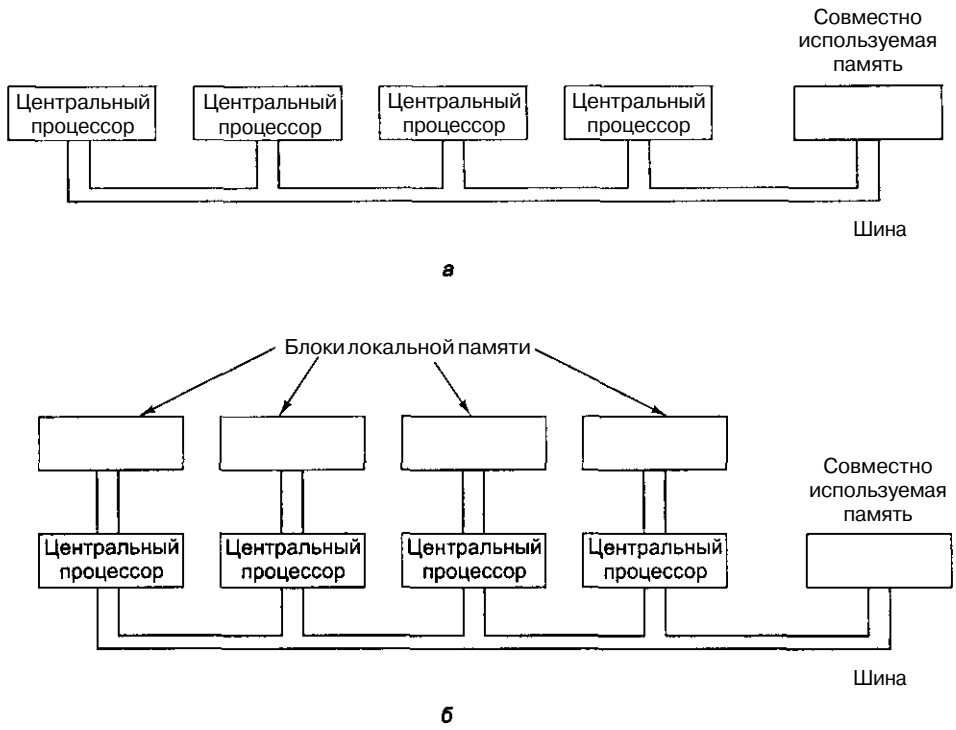


Рис. 2.7. Мультипроцессор с одной шиной и одной общей памятью (а); мультипроцессор, в котором для каждого процессора имеется собственная локальная память (б)

Мультипроцессоры имеют преимущество перед другими видами параллельных компьютеров, поскольку с единой разделенной памятью очень легко работать. Например, представим, что программа ищет раковые клетки на сделанном через микроскоп снимке ткани. Фотография в цифровом виде может храниться в общей памяти, при этом каждый процессор обследует какую-нибудь определенную область фотографии. Поскольку каждый процессор имеет доступ к общей памяти, обследование клетки, которая начинается в одной области и продолжается в другой, не представляет трудностей.

Мультикомпьютеры

Мультипроцессоры с небольшим числом процессоров (≤ 64) сконструировать довольно легко, а вот создание больших мультипроцессоров представляет некоторые трудности. Сложность заключается в том, чтобы связать все процессоры с памятью. Чтобы избежать таких проблем, многие разработчики просто отказались от идеи разделенной памяти и стали создавать системы, состоящие из большого числа взаимосвязанных компьютеров, у каждого из которых имеется своя собственная память, а общей памяти нет. Такие системы называются мультикомпьютерами.

Процессоры мультикомпьютера отправляют друг другу послания (это несколько похоже на электронную почту, но гораздо быстрее). Каждый компьютер не обязательно связывать со всеми другими, поэтому обычно в качестве топологий используются

2D, 3D, деревья и кольца. Чтобы послания могли дойти до места назначения, они должны проходить через один или несколько промежуточных компьютеров. Тем не менее время передачи занимает всего несколько микросекунд. Сейчас создаются и запускаются в работу мультикомпьютеры, содержащие около 10 000 процессоров.

Поскольку мультипроцессоры легче программировать, а мультикомпьютеры — конструировать, появилась идея создания гибридных систем, которые сочетают в себе преимущества обоих видов машин. Такие компьютеры представляют иллюзию разделенной памяти, при этом в действительности она не конструируется и не требует особых денежных затрат. Мы рассмотрим мультипроцессоры и мультикомпьютеры подробнее в главе 8.

Основная память

Память — часть компьютера, где хранятся программы и данные. Можно также употреблять термин «запоминающее устройство». Без памяти, откуда процессоры считывают и куда записывают информацию, не было бы цифровых компьютеров со встроенными программами.

Бит

Основной единицей памяти является двоичный разряд, который называется **битом.** Бит может содержать 0 или 1. Эта самая маленькая единица памяти. (Устройство, в котором хранятся только нули, вряд ли могло быть основой памяти. Необходимы по крайней мере две величины.)

Многие полагают, что в компьютерах используется бинарная арифметика, потому что это «эффективно». Они имеют в виду (хотя сами это редко осознают), что цифровая информация может храниться благодаря различию между разными величинами какой-либо физической характеристики, например напряжения или тока. Чем больше величин, которые нужно различать, тем меньше различий между смежными величинами и тем менее надежна память. Двоичная система требует различения всего двух величин, следовательно, это самый надежный метод кодирования цифровой информации. Если вы не знакомы с двоичной системой счисления, смотрите Приложение А.

Считается, что некоторые компьютеры, например большие IBM, используют **и** десятичную, и двоичную арифметику. На самом деле здесь применяется так называемый **двоично-десятичный код.** Для хранения одного десятичного разряда используется 4 бита. Эти 4 бита дают 16 комбинаций для размещения 10 различных значений (от 0 до 9). При этом 6 оставшихся комбинаций не используются. Ниже показано число 1944 в двоично-десятичной и чисто двоичной системах счисления; в обоих случаях используется 16 битов:

десятичное: 0001 10010100 0100 двоичное: 0000011110011000

16 битов в двоично-десятичном формате могут хранить числа от 0 до 9999, то есть всего 10000 различных комбинаций, а 16 битов в двоичном формате — 65536 комбинаций. Именно по этой причине говорят, что двоичная система эффективнее.

Однако представим, что могло бы произойти, если бы какой-нибудь гениальный молодой инженер придумал очень надежное электронное устройство, которое могло бы хранить разряды от 0 до 9, разделив участок напряжения от 0 до 10 В на 10 интервалов. Четыре таких устройства могли бы хранить десятичное число от 0 до 9999, то есть 10 000 комбинаций. А если бы те же устройства использовались для хранения двоичных чисел, они могли бы содержать всего 16 комбинаций. Естественно, в этом случае десятичная система была бы более эффективной.

Адреса памяти

Память состоит из ячеек, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется адресом, По адресу программы могут ссылаться на определенную ячейку. Если память содержит n ячеек, они будут иметь адреса от 0 до n-1. Все ячейки памяти содержат одинаковое число битов. Если ячейка состоит из k битов, она может содержать любую из 2^k комбинаций. На рис. 2.8 показаны 3 различных способа организации 96-битной памяти. Отметим, что соседние ячейки по определению имеют последовательные адреса.

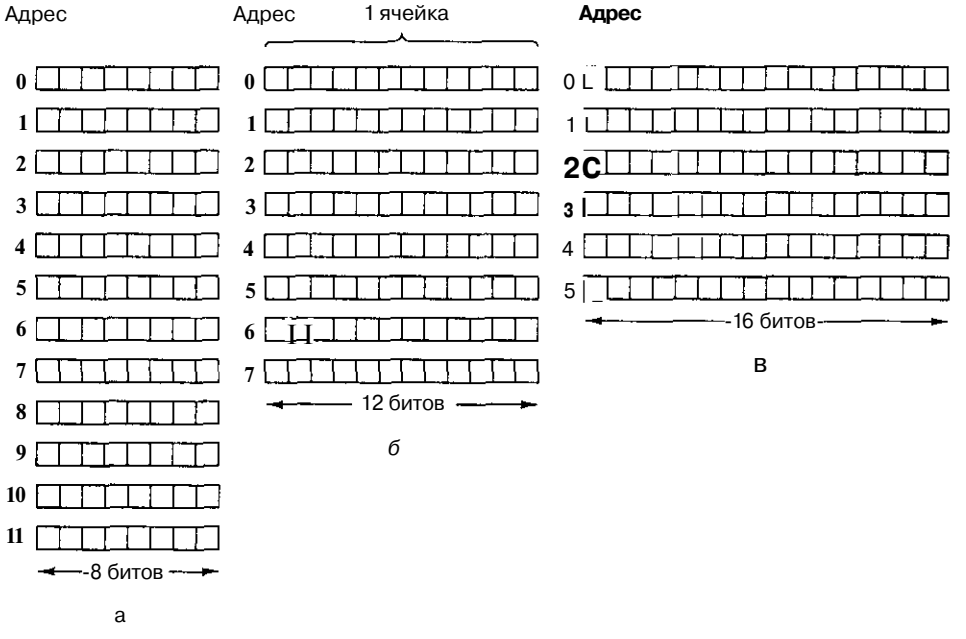


Рис. 2.6. Три способа организации 96-битной памяти

В компьютерах, где используется двоичная система счисления (включая восьмеричное и шестнадцатеричное представление двоичных чисел), адреса памяти также выражаются в двоичных числах. Если адрес состоит из m битов, максимальное число адресованных ячеек будет составлять 2^m. Например, адрес для обращения к памяти, изображенной на рис. 2.8, *а,* должен состоять по крайней мере из

4 битов, чтобы выражать все числа от 0 до 11. При устройстве памяти, показанном на рис. 2.8, *б* и 2.8, *в,* достаточно 3-битного адреса. Число битов в адресе определяет максимальное количество адресованных ячеек памяти и не зависит от числа битов в ячейке. 12-битные адреса нужны и памяти с 2^{12} ячеек по 8 битов каждая, и памяти с 2^{12} ячеек по 64 бита каждая.

В табл. 2.1 показано число битов в ячейке для некоторых коммерческих компьютеров.

Таблица 2.1. Число битов в ячейке для некоторых моделей коммерческих компьютеров

Компьютер	Число битов в ячейке
Burroughs B1700	1
IBM PC	8
DECPDP-8	12
IBM 1130	16
DECPDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Syber	60

Ячейка— минимальная единица, к которой можно обращаться, В последние годы практически все производители выпускают компьютеры с 8-битными ячейками, которые называются байтами, Байты группируются в **слова.** Компьютер с 32-битными словами имеет 4 байта на каждое слово, а компьютер с 64-битными словами — 8 байтов на каждое слово. Такая единица, как слово, необходима, поскольку большинство команд производят операции над целыми словами (например, складывают два слова). Таким образом, 32-битная машина будет содержать 32-битные регистры и команды для манипуляций с 32-битными словами, тогда как 64-битная машина будет иметь 64-битные регистры и команды для перемещения, сложения, вычитания и других операций над 64-битными словами.

Упорядочение байтов

Байты в слове могут нумероваться слева направо или справа налево. На первый взгляд может показаться, что между этими двумя вариантами нет разницы, но мы скоро увидим, что выбор имеет большое значение. На рис. 2.9, *а* изображена часть памяти 32-битного компьютера, в котором байты пронумерованы слева направо (как у компьютеров SPARC или больших IBM). Рисунок 2.9,*б* показывает аналогичную репрезентацию 32-битного компьютера с нумерацией байтов справа налево (как у компьютеров Intel).

Важно понимать, что в обеих системах 32-битное целое число (например, 6) представлено битами 110 в трех крайних правых битах слова, а остальные 29 битов

представлены нулями. Если байты нумеруются слева направо, биты 110 находятся в байте 3 (или 7, или 11 и т. д.). Если байты нумеруются справа налево, биты 110 находятся в байте 0 (или 4, или 8 и т. д.). В обоих случаях слово, содержащее это целое число, имеет адрес 0.

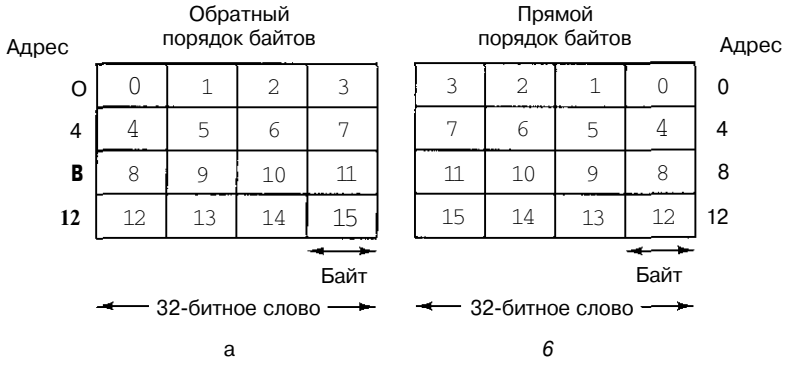


Рис. 2.9. Память с нумерацией байтов слева направо (а), память с нумерацией байтов справа налево (б)

Если компьютеры содержат только целые числа, никаких сложностей не возникает. Однако многие прикладные задачи требуют использования не только целых чисел, но и цепочек символов и других типов данных. Рассмотрим, например, простую запись данных персонала, состоящую из цепочки символов (имя сотрудника) и двух целых чисел (возраст и номер отдела). Цепочка символов завершается одним или несколькими байтами 0, чтобы заполнить слово. На рис. 2.10, а представлена схема с нумерацией байтов слева направо, а на рис. 2.10, *б* — с нумерацией байтов справа налево для записи «Jim Smith, 21 год, отдел 260» (1x256+4=260).

Рис. 2.10. Запись данных сотрудника для машины с нумерацией байтов слева направо (а); та же запись для машины с нумерацией байтов справа налево {б), результат преобразования записи из нумерации слева направо в нумерацию справа налево (в); результат перестановки байтов в предыдущем случае (г)

Оба эти представления хороши и внутренне последовательны. Проблемы начинаются тогда, когда один из компьютеров пытается переслать эту запись на Другой компьютер по сети. Предположим, что машина с нумерацией байтов слева на-

право пересылает запись на компьютер с нумерацией байтов справа налево по одному байту, начиная с байта 0 и заканчивая байтом 19. Для простоты будем считать, что биты не инвертируются при передаче. Таким образом, байт 0 переносится из первой машины на вторую в байт 0 и т. д., как показано на рис. 2.10, *в*.

Компьютер, получивший запись, имя печатает правильно, но возраст получается $21x2^{24}$, и номер отдела тоже искажается. Такая ситуация возникает, поскольку при передаче записи порядок букв в слове меняется так, как нужно, но при этом порядок байтов целых чисел тоже изменяется, что приводит к неверному результату.

Очевидное решение этой проблемы — наличие программного обеспечения, которое инвертировало бы байты в слове после того, как сделана копия. Результат такой операции изображен на рис. 2.10, *г*. Мы видим, **что** числа стали правильными, но цепочка символов превратилась в «MIJTIMS», при этом «Н» вообще поместилась отдельно. Цепочка переворачивается потому, что компьютер сначала считывает байт 0 (пробел), затем байт 1 (М) и т. д.

Простого решения не существует. Есть один способ, но он неэффективен. (Нужно перед каждой единицей данных помещать заголовок, информирующий, какой тип данных последует за ним — цепочка, целое число и т. д. Это позволит компьютеру-получателю производить только необходимые преобразования.) Ясно, что отсутствие стандарта упорядочивания байтов является главным неудобством при обмене информацией между разными машинами.

Код с исправлением ошибок

Память компьютера время от времени может делать ошибки из-за всплесков напряжения на линии электропередачи и по другим причинам. Чтобы бороться с такими ошибками, используются коды с обнаружением и исправлением ошибок. При этом к каждому слову в памяти особым образом добавляются дополнительные биты. Когда слово считывается из памяти, эти биты проверяются на наличие ошибок.

Чтобы понять, как обращаться с ошибками, необходимо внимательно изучить, что представляют собой эти ошибки. Предположим, что слово состоит из m битов данных, к которым мы прибавляем г дополнительных битов (контрольных разрядов). Пусть общая длина слова будет п (то есть п=т+г). n-битную единицу, содержащую m битов данных и **г** контрольных разрядов, часто называют **кодированным словом.**

Для любых двух кодированных слов, например 10001001 и 10110001, можно определить, сколько соответствующих битов в них различается. В данном примере таких бита три. Чтобы определить количество различающихся битов, нужно над двумя кодированными словами произвести логическую операцию ИСКЛЮЧАЮЩЕЕ ИЛИ и сосчитать число битов со значением 1 в полученном результате. Число битовых позиций, по которым различаются два слова, называется **интервалом Хэмминга.** Если интервал Хэмминга для двух слов равен d, это значит, что достаточно d битовых ошибок, чтобы превратить одно слово в другое. Например, интервал Хэмминга кодированных слов 11110001 и 00110000 равен 3, поскольку для превращения первого слова во второе достаточно 3 ошибок в битах.

Память состоит из m-битных слов, и следовательно, существует 2^m вариантов сочетания битов. Кодированные слова состоят из п битов, но из-за способа под-

счета контрольных разрядов допустимы только $2^{Ш}$ из 2" кодированных слов. Если в памяти обнаруживается недопустимое кодированное слово, компьютер знает, что произошла ошибка. При наличии алгоритма для подсчета контрольных разрядов можно составить полный список допустимых кодированных слов и из этого списка найти два слова, для которых интервал Хэмминга будет минимальным. Это интервал Хэмминга полного кода.

Свойства проверки и исправления ошибок определенного кода зависят от его интервала Хэмминга. Чтобы обнаружить d ошибок в битах, необходим код с интервалом d+1, поскольку d ошибок не могут изменить одно допустимое кодированное слово на другое допустимое кодированное слово Соответственно, чтобы исправить d ошибок в битах, необходим код с интервалом 2d+1, поскольку в этом случае допустимые кодированные слова так сильно отличаются друг от друга, что даже если произойдет d изменений, изначальное кодированное слово будет ближе к ошибочному, чем любое другое кодированное слово, поэтому его без труда можно будет определить.

В качестве простого примера кода с обнаружением ошибок рассмотрим код, в котором к данным присоединяется один бит четности. Бит четности выбирается таким образом, что число битов со значением 1 в кодированном слове четное (или нечетное). Интервал этого кода равен 2, поскольку любая ошибка в битах приводит к кодированному слову с неправильной четностью. Другими словами, достаточно двух ошибок в битах для перехода от одного допустимого кодированного слова к другому допустимому слову. Такой код может использоваться для обнаружения одиночных ошибок. Если из памяти считывается слово, содержащее неверную четность, поступает сигнал об ошибке. Программа не сможет продолжаться, но зато не будет неверных результатов.

В качестве простого примера кода с исправлением ошибок рассмотрим код с четырьмя допустимыми кодированными словами:

0000000000,0000011111, ШИОООООи 1111111111

Интервал этого кода равен 5 Это значит, что он может исправлять двойные ошибки. Если появляется кодированное слово 0000000111, компьютер знает, что изначальное слово должно быть 0000011111 (если произошло не более двух ошибок). При наличии трех ошибок, если, например, слово 0000000000 изменилось на 0000000111, этот метод недопустим.

Представим, что мы хотим разработать код с m битами данных и г контрольных разрядов, который позволил бы исправлять все ошибки в битах. Каждое из $2^{т}$ допустимых слов имеет п недопустимых кодированных слов, которые отличаются от допустимого одним битом. Они образуются инвертированием каждого из п битов в n-битном кодированном слове. Следовательно, каждое из $2^{т}$ допустимых слов требует п+1 возможных сочетаний битов, приписываемых этому слову (п возможных ошибочных вариантов и один правильный). Поскольку общее число различных сочетаний битов равно $2^{П}$, то $(п+1)2^{та} \leq 2^{п}$. Так как n-ш+г, следовательно, $(т+г+1) \leq 2^{г}$. Эта формула дает нижний предел числа контрольных разрядов, необходимых для исправления одиночных ошибок. В табл 2.2 показано необходимое количество контрольных разрядов для слов разного размера.

Таблица 2.2. Число контрольных разрядов для кода, способного исправлять одиночные ошибки

Размер слова	Количество контрольных разрядов	Общий размер	На сколько процентов увеличилась длина слова
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Этого теоретического нижнего предела можно достичь, используя метод Ричарда Хэмминга. Но прежде чем обратиться к этому алгоритму, давайте рассмотрим простую графическую схему, которая четко иллюстрирует идею кода с исправлением ошибок для 4-битных слов. Диаграмма Венна на рис. 2.11 содержит 3 круга, A, B и C, которые вместе образуют семь секторов. Давайте закодируем в качестве примера слово из 4 битов 1100 в сектора AB, ABC, AC и BC, по одному биту в каждом секторе (в алфавитном порядке). Кодирование показано на рис. 2.11, *а*.

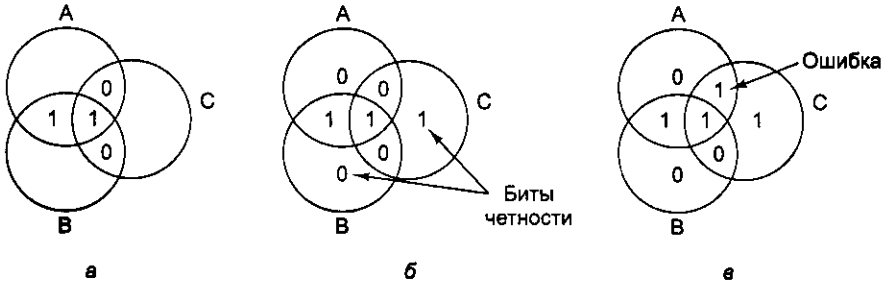


Рис. 2.11. Кодировка числа 1100 (а); добавляются биты четности *(б);* ошибка в секторе AC (в)

Далее мы добавим бит четности к каждому из трех пустых секторов, чтобы получилась положительная четность, как показано на рис. 2.11, *б*. По определению сумма битов в каждом из трех кругов, A, B, и C, должна быть четной. В круге A находится 4 числа: 0, 0, 1 и 1, которые в сумме дают четное число 2. В круге B находятся числа 1, 1, 0 и 0, которые также при сложении дают четное число 2. То же имеет силу и для круга C. В данном примере получилось так, что все суммы одинаковы, но вообще возможны случаи с суммами 0 и 4. Рисунок соответствует кодированному слову, состоящему из 4 битов данных и 3 битов четности.

Предположим, что бит в секторе AC изменился с 0 на 1, как показано на рис. 2.11, *в*. Компьютер видит, что круги A и C имеют отрицательную четность. Единственный способ исправить ошибку, изменив только один бит, — возвращение биту AC значения 0. Таким способом компьютер может исправлять одиночные ошибки автоматически.

А теперь посмотрим, как может использоваться алгоритм Хэмминга при создании кодов с исправлением ошибок для слов любого размера. В коде Хэмминга к

слову, состоящему из m битов, добавляется r битов четности, при этом образуется слово длиной m+r битов. Биты нумеруются с единицы (а не с нуля), причем первым считается крайний левый. Все биты, номера которых — степени двойки, являются битами четности; остальные используются для данных. Например, к 16-битному слову нужно добавить 5 битов четности. Биты с номерами 1, 2, 4, 8 и 16 — биты четности, а все остальные — биты данных. Всего слово содержит 21 бит (16 битов данных и 5 битов четности). В рассматриваемом примере мы будем использовать положительную четность (выбор произвольный).

Каждый бит четности проверяет определенные битовые позиции. Общее число битов со значением 1 в проверяемых позициях должно быть четным. Ниже указаны позиции проверки для каждого бита четности:

Бит 1 проверяет биты 1, 3, 5,7, 9,11, 13,15,17,19, 21.
Бит 2 проверяет биты 2, 3, 6, 7,10,11,14,15,18,19.
Бит 4 проверяет биты 4, 5,6, 7,12,13,14,15, 20, 21.
Бит 8 проверяет биты 8,9,10, И, 12,13,14, 15.
Бит 16 проверяет биты 16,17,18,19, 20, 21.

В общем случае бит b проверяется битами $b_1, b_2, \ldots, b_{Jt}$ такими что $b_i+b_2+\ldots+b_r=b$. Например, бит 5 проверяется битами 1 и 4, поскольку 1+4=5. Бит 6 проверяется битами 2 и 4, поскольку 2+4=6 и т. д.

На рис. 2.12 показано построение кода Хэмминга для 16-битного слова 1111000010101110 Соответствующим 21-битным кодированным словом является 001011100000101101110. Чтобы увидеть, как происходит исправление ошибок, рассмотрим, что произойдет, если бит 5 изменит значение из-за резкого скачка напряжения на линии электропередачи. В результате вместо кодированного слова 001011100000101101110 получится 001001100000101101 ПО. Будут проверены 5 битов четности. Вот результаты проверки:

Бит четности 1 неправильный (биты 1, 3, 5, 7,9, 11, 13, 15, 17, 19, 21 содержат пять единиц).

Бит четности 2 правильный (биты 2, 3, 6,7,10,11,14,15,18,19 содержат шесть единиц).

Бит четности 4 неправильный (биты 4,5,6,7,12,13,14,15,20,21 содержат пять единиц).

Бит четности 8 правильный (биты 8,9,10,11,12,13,14,15 содержат две единицы).

Битчетности 16 правильный (биты 16,17,18,19,20,21 содержат четыре единицы).

Общее число единиц в битах 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21 должно быть четным, поскольку в данном случае используется положительная четность. Неправильным должен быть один из битов, проверяемых битом четности 1 (а именно 1, 3,5,7,9,11,13,15,17,19 и 21). Бит четности 4 тоже неправильный. Это значит, что изменил значение один из следующих битов: 4,5,6,7,12,13,14,15,20,21. Ошибка должна быть в бите, который содержится в обоих списках. В данном случае общими являются биты 5,7,13,15 и 21. Поскольку бит четности 2 правильный, биты 7 и 15 исключаются. Правильность бита четности 8 исключает наличие ошибки в бите 13. Наконец, бит 21 также исключается, поскольку бит четности 16 правильный. В итоге остается бит 5, в котором и содержится ошибка. Поскольку этот бит имеет значение 1, он должен принять значение 0. Именно таким образом исправляются ошибки.

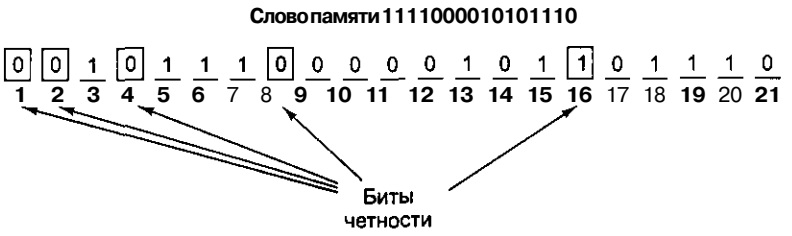


Рис. 2.12. Построение кода Хэмминга для слова 1111000010101110с помощью добавления 5 контрольных разрядов к ,16 битам данных

Чтобы найти неправильный бит, сначала нужно подсчитать все биты четности. Если они правильные, ошибки нет (или есть, но больше одной). Если обнаружились неправильные биты четности, то нужно сложить их номера. Сумма, полученная в результате, даст номер позиции неправильного бита. Например, если биты четности 1 и 4 неправильные, а 2,8 и 16 правильные, то ошибка произошла в бите 5 (1+4).

Кэш-память

Процессоры всегда работали быстрее, чем память. Процессоры и память совершенствовались параллельно, поэтому это несоответствие сохранялось. Поскольку на микросхему можно помещать все больше и больше транзисторов, разработчики процессоров использовали эти преимущества для создания конвейеров и суперскалярной архитектуры, что еще больше повышало скорость работы процессоров. Разработчики памяти обычно использовали новые технологии для увеличения емкости, а не скорости, что еще больше усугубляло проблему. На практике такое несоответствие в скорости работы приводит к следующему: после того как процессор дает запрос памяти, должно пройти много циклов, прежде чем он получит слово, которое ему нужно. Чем медленнее работает память, тем дольше процессору приходится ждать, тем больше циклов должно пройти.

Как мы уже говорили выше, есть два пути решения этой проблемы. Самый простой из них — начать считывать информацию из памяти, когда это необходимо, и при этом продолжать выполнение команд, но если какая-либо команда попытается использовать слово до того, как оно считалось из памяти, процессор должен приостанавливать работу. Чем медленнее работает память, тем чаще будет возникать такая проблема и тем больше будет проигрыш в работе. Например, если отсрочка составляет 10 циклов, весьма вероятно, что одна из 10 следующих команд попытается использовать слово, которое еще не считалось из памяти.

Другое решение проблемы — сконструировать машину, которая не приостанавливает работу, но следит, чтобы программы-компиляторы не использовали слова до того, как они считаются из памяти. Однако это не так просто осуществить на практике. Часто при выполнении команды загрузки машина не может выполнять другие действия, поэтому компилятор вынужден вставлять пустые команды, которые не производят никаких операций, но при этом занимают место в памяти. В действительности при таком подходе простаивает не аппаратное, а программное обеспечение, но снижение производительности при этом такое же.

На самом деле эта проблема не технологическая, а экономическая. Инженеры знают, как построить память, которая будет работать так же быстро, как и процессор, но при этом ее приходится помещать прямо на микросхему процессора (поскольку информация через шину поступает очень медленно). Установка большой памяти на микросхему процессора делает его больше и, следовательно, дороже, и даже если бы стоимость не имела значения, все равно существуют ограничения в размерах процессора, который можно сконструировать. Таким образом, приходится выбирать между быстрой памятью небольшого размера и медленной памятью большого размера. Мы бы предпочли память большого размера с высокой скоростью работы по низкой цене.

Интересно отметить, что существуют технологии сочетания маленькой и быстрой памяти с большой и медленной, что позволяет получить и высокую скорость работы, и большую емкость по разумной цене. Маленькая память с высокой скоростью работы называется **кэш-памятью** (от французского слова cacher «прятать»[1]; читается «кэш»). Ниже мы кратко опишем, как используется кэш-память и как она работает. Более подробное описание см. в главе 4.

Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово, сначала он обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэш-памяти, среднее время доступа значительно сокращается.

Таким образом, успех или неудача зависит от того, какая часть слов находится в кэш-памяти. Давно известно, что программы не обращаются к памяти наугад. Если программе нужен доступ к адресу A, то скорее всего после этого ей понадобится доступ к адресу, расположенному поблизости от A. Практически все команды обычной программы (за исключением команд перехода и вызова процедур) вызываются из последовательных участков памяти. Кроме того, большую часть времени программа тратит на циклы, когда ограниченный набор команд выполняется снова и снова. Точно так же при манипулировании матрицами программа скорее всего будет обращаться много раз к одной и той же матрице, прежде чем перейдет к чему-либо другому.

То, что при последовательных отсылках к памяти в течение некоторого промежутка времени используется только небольшой ее участок, называется **принципом локальности.** Этот принцип составляет основу всех систем кэш-памяти. Идея состоит в следующем: когда определенное слово вызывается из памяти, оно вместе с соседними словами переносится в кэш-память, что позволяет при очередном запросе быстро обращаться к следующим словам. Общее устройство процессора, кэш-памяти и основной памяти показано на рис. 2.13. Если слово считывается или записывается к раз, компьютеру понадобится сделать 1 обращение к медленной основной памяти и к-1 обращений к быстрой кэш-памяти. Чем больше к, тем выше общая производительность.

---

[1] В английском «cash* получило значение «наличные (карманные) деньги», то есть то, что под рукой А уже из него и образовался термин «кэш», который относят к сверхоперативной памяти. — *Примеч научн, ред.*

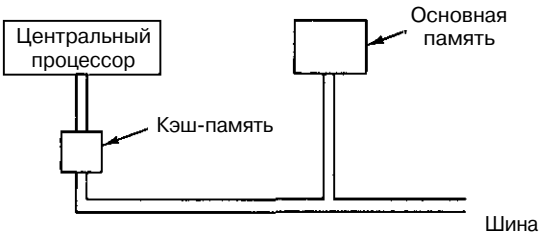


Рис. 2.13. Кэш-память по логике вещей должна находиться между процессором и основной памятью. В действительности существует три возможных варианта расположения кэш-памяти

Мы можем сделать более строгие вычисления. Пусть с — время доступа к кэш-памяти, m — время доступа к основной памяти и **h** — **коэффициент совпадения,** который показывает соотношение числа ссылок к кэш-памяти и общего числа всех ссылок. В нашем примере h=(k~l)/k. Таким образом, мы можем вычислить среднее время доступа:

среднее время доступа =c+(1-h)m.

Если h—И и все обращения делаются только к кэш-памяти, то время доступа стремится к с. С другой стороны, если h—»0 и каждый раз нужно обращаться к основной памяти, то время доступа стремится к с+ш: сначала требуется время с для проверки кэш-памяти (в данном случае безуспешной), а затем время m для обращения к основной памяти. В некоторых системах обращение к основной памяти может начинаться параллельно с исследованием кэш-памяти, чтобы в случае неудачного поиска цикл обращения к основной памяти уже начался. Однако эта стратегия требует способности останавливать процесс обращения к основной памяти в случае результативного обращения к кэш-памяти, что делает разработку такого компьютера более сложной.

Основная память и кэш-память делятся на блоки фиксированного размера с учетом принципа локальности. Блоки внутри кэш-памяти обычно называют **строками кэш-памяти (cache lines).** Если обращение к кэш-памяти нерезультативно, из основной памяти в кэш-память загружается вся строка, а не только необходимое слово. Например, если строка состоит из 64 байтов, обращение к адресу 260 повлечет за собой загрузку в кэш-память всей строки, то есть с 256-го по 319-й байт. Возможно, через некоторое время понадобятся другие слова из этой строки. Такой путь обращения к памяти более эффективен, чем вызов каждого слова по отдельности, потому что вызвать **к** слов 1 раз можно гораздо быстрее, чем 1 слово к раз. Если входные сообщения кэш-памяти содержат более одного слова, это значит, что будет меньше таких входных сообщений и, следовательно, меньше непроизводительных затрат.

Разработка кэш-памяти очень важна для процессоров с высокой производительностью. Первый вопрос — размер кэш-памяти. Чем больше размер, тем лучше работает память, но тем дороже она стоит. Второй вопрос — размер строки кэш-памяти. Кэш-память объемом 16 Кбайт можно разделить на 1К строк по 16 байтов, 2К строк по 8 байтов и т. д. Третий вопрос — как устроена кэш-память, то есть как она определяет, какие именно слова содержатся в ней в данный момент. Устройство кэш-памяти мы рассмотрим подробно в главе 4.

Четвертый вопрос — должны ли команды и данные находиться вместе в общей кэш-памяти. Проще разработать **смежную кэш-память,** в которой хранятся и данные, и команды. При этом вызов команд и данных автоматически уравновешивается. Тем не менее в настоящее время существует тенденция к использованию **разделенной кэш-памяти,** когда команды хранятся в одной кэш-памяти, а данные — в другой. Такая структура также называется **Гарвардской (Harvard Architecture),** поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Маге III, который был создай Говардом Айкеном в Гарварде. Современные разработчики пошли по этому пути, поскольку сейчас широко используются процессоры с конвейерами, а при такой организации должна быть возможность одновременного доступа и к командам, и к данным (операндам). Разделенная кэш-память позволяет осуществлять параллельный доступ, а общая — нет. К тому же, поскольку команды обычно не меняются во время выполнения, содержание командной кэш-памяти никогда не приходится записывать обратно в основную память.

Наконец, пятый вопрос — количество блоков кэш-памяти. В настоящее время очень часто кэш-память первого уровня располагается прямо на микросхеме процессора, кэш-память второго уровня — не на самой микросхеме, но в корпусе процессора, а кэш-память третьего уровня — еще дальше от процессора.

Сборка модулей памяти и их типы

Со времен появления полупроводниковой памяти и до начала 90-х годов все микросхемы памяти производились, продавались и устанавливались на плату компьютера по отдельности. Эти микросхемы вмещали от 1 Кбит до 1 Мбит информации и выше. В первых персональных компьютерах часто оставлялись пустые разъемы, чтобы покупатель в случае необходимости мог вставить дополнительные микросхемы.

В настоящее время распространен другой подход. Группа микросхем (обычно 8 или 16) монтируется на одну крошечную печатную плату и продается как один **блок. Он называется SIMM (Single Inline Memory Module — модуль памяти, имеющий выводы с одной стороны)** или **DIMM (Dual Inline Memory Module — модуль памяти, у которого выводы расположены с двух сторон).** У первого из них контакты расположены только на одной стороне печатной платы (выводы на второй стороне дублируют первую), а у второго — на обеих сторонах. Схема SIMM изображена на рис. 2.14.

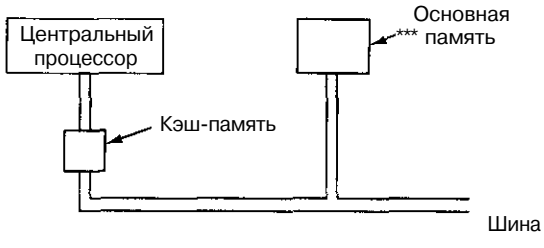


Рис. 2.14. Модуль SIMM в 32 Мбайт. Модулем управляют две микросхемы

Обычный модуль SIMM содержит 8 микросхем по 32 Мбит (4 Мбайт) каждая. Таким образом, весь модуль вмещает 32 Мбайт информации. Во многие компьютеры встраивается 4 модуля, следовательно, при использовании модулей SIMM по 32 Мбайт общий объем памяти составляет 128 Мбайт. При необходимости данные модули SIMM можно заменить модулями с большей вместимостью (64 Мбайт и выше).

У первых модулей SIMM было 30 контактов, и они могли передавать 8 битов информации за один раз. Остальные контакты использовались для адресации и контроля. Более поздние модули содержали уже 72 контакта и передавали 32 бита информации за один раз. Для компьютера Pentium, который требовал одновременной передачи 64 битов, эти модули соединялись по два, и каждый из них доставлял половину требуемых битов. В настоящее время стандартным способом сборки является модуль DIMM. У него на каждой стороне платы находится по 84 позолоченных контакта, то есть всего 168. DIMM способен передавать 64 бита данных за раз. Вместимость DIMM обычно составляет 64 Мбайт и выше. В электронных записных книжках обычно используется модуль DIMM меньшего размера, который называется **SO-DIMM (Small Outline DIMM).** Модули SIMM и DIMM могут содержать бит четности или код исправления ошибок, однако, поскольку вероятность возникновения ошибок в модуле 1 ошибка в 10 лет, в большинстве обычных компьютеров методы обнаружения и исправления ошибок не применяются.

Вспомогательная память

Не важно, каков объем основной памяти: он все равно всегда будет слишком мал. Мы всегда хотим хранить в памяти компьютера больше информации, чем она может вместить С развитием технологий людям приходят в голову такие вещи, которые раньше считались совершенно фантастическими. Например, можно вообразить, что Библиотека Конгресса решила представить в цифровой форме и продать полное содержание всех хранящихся в ней изданий в одной статье («Все человеческие знания всего за $49»). В среднем каждая книга содержит 1 Мбайт текста и 1 Мбайт сжатых рисунков. Таким образом, для размещения 50 млн книг понадобится 10^{14} байт или 100 Тбайт памяти. Для хранения всех существующих художественных фильмов (50 000) необходимо примерно столько же места. Такое количество информации в настоящее время невозможно разместить в основной памяти, и вряд ли можно будет это сделать в будущем (по крайней мере, в ближайшие несколько десятилетий).

Иерархическая структура памяти

Иерархическая структура памяти является традиционным решением проблемы хранения большого количества данных Она изображена на рис. 2.15. На самом верху находятся регистры процессора. Доступ к регистрам осуществляется быстрее всего. Дальше идет кэш-память, объем которой сейчас составляет от 32 Кбайт до нескольких мегабайт. Затем следует основная память, которая в настоящее

время может вмещать от 16 Мбайт до десятков гигабайтов. Далее идут магнитные диски и, наконец, накопители на магнитной ленте и оптические диски, которые используются для хранения архивной информации.

Рис. 2.15. Пятиуровневая организация памяти

По мере продвижения по структуре сверху вниз возрастают три параметра. Во-первых, увеличивается время доступа. Доступ к регистрам занимает несколько наносекунд, доступ к кэш-памяти — немного больше, доступ к основной памяти — несколько десятков наносекунд. Дальше идет большой разрыв: доступ к дискам занимает по крайней мере 10 мкс, а время доступа к магнитным лентам и оптическим дискам вообще может измеряться в секундах (поскольку эти накопители информации еще нужно взять и поместить в соответствующее устройство).

Во-вторых, увеличивается объем памяти. Регистры могут содержать в лучшем случае 128 байтов, кэш-память — несколько мегабайтов, основная память — десятки тысяч мегабайтов, магнитные диски — от нескольких гигабайтов до нескольких десятков гигабайтов. Магнитные ленты и оптические диски хранятся автономно от компьютера, поэтому их объем ограничивается только финансовыми возможностями владельца.

В-третьих, увеличивается количество битов, которое вы получаете за 1 доллар. Стоимость объема основной памяти измеряется в долларах за мегабайт[1], объем магнитных дисков — в пенни за мегабайт, а объем магнитной ленты — в долларах за гигабайт или еще дешевле.

Регистры, кэш-память и основную память мы уже рассмотрели. В следующих разделах мы расскажем о магнитных дисках, а затем приступим к изучению оптических дисков. Накопители на магнитных лентах мы рассматривать не будем, поскольку они очень редко используются; к тому же о них практически нечего сказать.

[1] Заметим, что стоимость памяти постоянно уменьшается, в то время как ее объем — увеличивается. Закон Мура применим и здесь. Сегодня 1 Мбайт оперативной памяти стоит около 10 центов. —*Примеч. научи, ред.*

Магнитные диски

Магнитный диск состоит из одного или нескольких алюминиевых дисков[1] с магнитным слоем. Изначально они были 50 см в диаметре, но сейчас их диаметр составляет от 3 до 12 см, а у портативных компьютеров — меньше 3 см, причем этот параметр продолжает уменьшаться. Головка диска, содержащая индукционную катушку, двигается над поверхностью диска, опираясь на воздушную подушку. Отметим, что у дискет головка касается поверхности. Когда через головку проходит положительный или отрицательный ток, он намагничивает поверхность под головкой. При этом магнитные частицы намагничиваются направо или налево в зависимости от полярности тока. Когда головка проходит над намагниченной областью, в ней (в головке) возникает положительный или отрицательный ток, что дает возможность считывать записанные ранее биты. Поскольку диск вращается под головкой, поток битов может записываться, а потом считываться. Конфигурация дорожки диска показана на рис. 2.16.

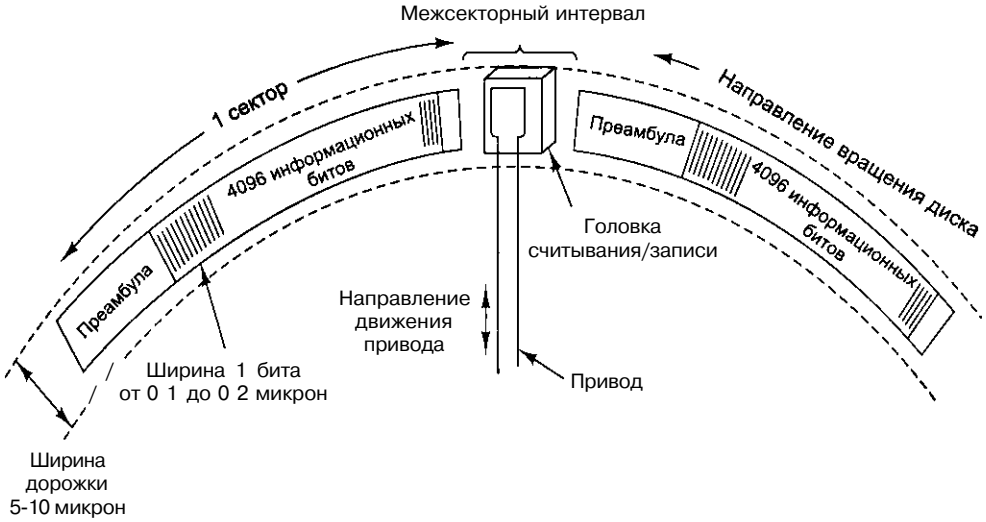


Рис. 2.16. Кусок дорожки диска (два сектора)

Дорожкой называется круговая последовательность битов, записанных на диск за его полный оборот. Каждая дорожка делится на **секторы** фиксированной длины. Каждый сектор обычно содержит 512 байтов данных. Перед данными располагается **преамбула (preamble),** которая позволяет головке синхронизироваться перед чтением или записью. После данных идет код с исправлением ошибок (код Хэмминга или чаще **код Рида—Соломона,** который может исправлять много ошибок, а не только одиночные). Между соседними секторами находится **межсекторный интервал.** Многие производители указывают размер неформатированного диска (как будто каждая дорожка содержит только данные), но более честно было бы указывать вместимость форматированного диска, когда не учитываются пре-

[1] В настоящее время фирма IBM делает их из стекла. — *Примеч научи, ред.*

амбулы, коды с исправлением ошибок и межсекторные интервалы. Емкость форматированного диска обычно на 15% меньше емкости неформатированного диска.

У всех дисков есть кронштейны, они могут перемещаться туда и обратно по радиусу на разные расстояния от шпинделя, вокруг которого вращается диск. На разных расстояниях от оси записываются разные дорожки. Таким образом, дорожки представляют собой ряд концентрических кругов, расположенных вокруг шпинделя. Ширина дорожки зависит от величины головки и от точности ее перемещения. На сегодняшний момент диски имеют от 800 до 2000 дорожек на см[1], то есть ширина каждой дорожки составляет от 5 до 10 микрон (1 микрон=1/1000 мм). Следует отметить, что дорожка — это не углубление на поверхности диска, а просто кольцо намагниченного материала, которое отделяется от других дорожек небольшими пограничными областями.

Плотность записи битов на концентрических дорожках различная, в зависимости от расстояния от центра диска. Плотность записи зависит главным образом от качества поверхности диска и чистоты воздуха. Плотность записи современных дисков разнится от 50 000 до 100 000 бит/см. Чтобы достичь высокого качества поверхности и достаточной чистоты воздуха, диски герметично запечатываются, что защищает их от попадания грязи. Такие диски называются винчестерами. Впервые они были выпущены фирмой IBM. У них было 30 Мбайт фиксированной памяти и 30 Мбайт сменной памяти. Возможно, эти диски 30-30 ассоциировались с ружьями «Винчестер» 30-30[2] Большинство магнитных дисков состоит из нескольких пластин, расположенных друг под другом, как показано на рис. 2.17. Каждая поверхность снабжена рычагом и головкой. Рычаги скреплены таким образом, что одновременно могут перемещаться на разные расстояния от оси. Совокупность дорожек, расположенных на одном расстоянии от центра, называется **цилиндром.**

Производительность диска зависит от многих факторов. Чтобы считать или записать сектор, головка должна переместиться на нужное расстояние от оси. Этот процесс называется **поиском.** Среднее время поиска между дорожками, взятыми наугад, составляет от 5 до 15 мс, а поиск между последовательными дорожками занимает около 1 мс. Когда головка помещается на нужное расстояние от центра, выжидается некоторое количество времени (оно называется **временем ожидания сектора),** пока нужный сектор не оказывается иод головкой. Большинство дисков вращаются со скоростью 3600, 5400 или 7200 оборотов в минуту. Таким образом, среднее время ожидания сектора (половина оборота) составляет от 4 до 8 мс. Существуют также диски со скоростью вращения 10800 оборотов в минуту (180 оборотов в секунду). Время передачи информации зависит от плотности записи и скорости вращения. При скорости передачи от 5 до 10 Мбайт в секунду[3] время передачи одного сектора (512 байтов) составляет от 25 до 100 мкс. Следовательно, время поиска и время ожидания сектора определяет время передачи информации. Ясно, что считывание секторов из разных частей диска неэффективно.

---

[1] Плотность хранения информации на магнитных дисках также постоянно увеличивается, и сейчас на 1 см поверхности уже размещается более 10000 дорожек — *Примеч. научи, ред.*

[2] Двуствольное ружье 30-го калибра. — *Примеч. перев*

[3] В современных винчестерах скорость линейного чтения уже превысила 40 Мбайт в секунду —*Примеч. научн ред*

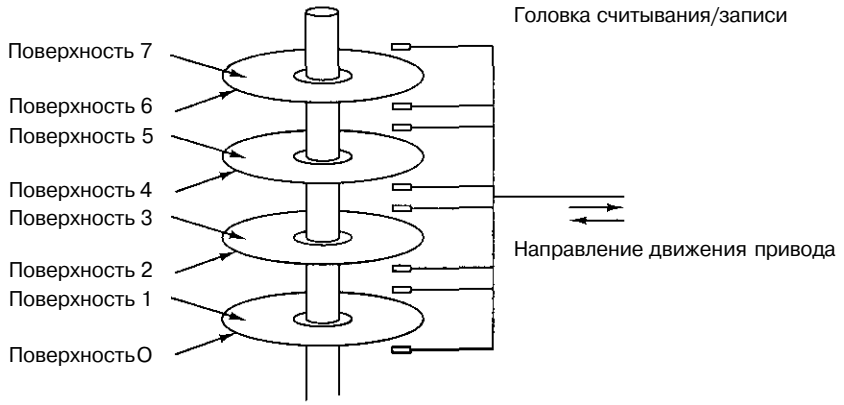


Рис. 2.17. Винчестер с четырьмя дисками

Следует упомянуть, что из-за наличия преамбул, кодов с исправлением ошибок, промежутков между секторами, а также из-за того, что определенное время затрачивается на поиск дорожки и на ожидание сектора, существует огромная разница между максимальной скоростью передачи данных, когда необходимые данные разбросаны в разных частях диска, и той, когда они находятся в одном месте и считываются последовательно. Максимальная скорость передачи данных в первом случае достигается в тот момент, когда головка расположена над первым битом данных. Однако такая скорость работы может сохраняться только на одном секторе. Для некоторых приложений, например мультимедиа, имеет значение именно средняя скорость передачи за некоторый период с учетом необходимого времени поиска и времени ожидания сектора.

Поскольку диски вращаются со скоростью от 60 до 120 оборотов в секунду, они нагреваются и расширяются, то есть физически изменяются в размерах. Некоторые диски должны периодически совершать рекалибровку механизмов перемещения, чтобы компенсировать эти расширения. Поэтому мощность привода, перемещающего головки над поверхностью диска, периодически меняется. При таких рекалибровках могут возникать трудности с использованием прикладных программ мультимедиа, которые ожидают более или менее непрерывного потока битов, поступающего с максимальной скоростью передачи последовательной информации (с одной части диска). Для работы с прикладными программами мультимедиа некоторые производители выпускают специальные **аудио-видеодиски,** которые не совершают термических рекалибровок.

Немного сообразительности, и старая школьная математическая формула для вычисления длины окружности c=2яг откроет, что линейная длина внешних дорожек больше, чем длина внутренних. Поскольку все магнитные диски вращаются с постоянной угловой скоростью независимо от того, где находятся головки, возникает очевидная проблема. Раньше при производстве дисков изготовители создавали максимально возможную плотность записи на внутренней дорожке, и при продвижении от центра диска плотность записи постепенно снижалась. Если дорожка содержит, например, 18 секторов, то каждый из них занимает дугу в 20°, и не важно, на каком цилиндре находится эта дорожка.

В настоящее время используется другая стратегия. Цилиндры делятся на зоны (на диске их обычно от 10 до 30). При продвижении от центра диска число секторов на дорожке в каждой зоне возрастает. Это изменение усложняет процедуру хранения информации на дорожке, но зато повышает емкость диска, что считается более важным. Все секторы имеют одинаковый размер. К счастью, хоть какие-то вещи в жизни никогда не изменяются.

С диском связан так называемый контроллер — микросхема, которая управляет диском. Некоторые контроллеры содержат целый процессор. В задачи контроллера входит получение от программного обеспечения таких команд, как READ, WRITE и FORMAT (то есть запись всех преамбул), управление перемещением рычага, обнаружение и исправление ошибок, преобразование 8-битных байтов, считываемых из памяти, в непрерывный поток битов и наоборот. Некоторые контроллеры производят буферизацию совокупности секторов и кэширование секторов для дальнейшего потенциального использования, а также устраняют поврежденные секторы. Необходимость последней функции вызвана наличием секторов с поврежденным, то есть постоянно намагниченным, участком. Когда контроллер обнаруживает поврежденный сектор, он заменяет его одним из свободных секторов, которые выделяются специально для этой цели в каждом цилиндре или зоне.

Дискеты

С изобретением персонального компьютера появилась необходимость каким-то образом распространять программное обеспечение. Решением этой проблемы послужила дискета (floppy disk — «гибкий диск»; назван так, потому что первые дискеты были гибкими физически) — небольшой сменный носитель информации. Дискеты были придуманы фирмой IBM. Изначально на них записывалась информация по обслуживанию больших машин (для сотрудников фирмы). Но производители компьютеров вскоре переняли эту идею и стали использовать дискеты в качестве удобного средства записи программного обеспечения и его продажи.

Дискеты обладают теми же общими характеристиками, что и диски, которые мы только что рассматривали, с тем лишь различием, что головки жестких дисков перемещаются над поверхностью диска на воздушной подушке, а у дискет головки касаются поверхности. В результате и сами дискеты, и головки очень быстро изнашиваются. Поэтому когда не происходит считывание и запись информации, головки убираются с поверхности, а компьютер останавливает вращение диска. Это позволяет продлить срок службы дискет. Но при этом, если поступает команда считывания или записи, происходит небольшая задержка (примерно полсекунды) перед тем, как мотор начнет работать.

Существует два вида дискет: 5,25 дюймов и 3,5 дюйма[1]. Каждая из них может быть или с низкой плотностью записи (Low-Density, сокращение LD), или с высокой плотностью записи (High-Density, сокращение HD). Дискеты на 3,5 дюйма выпускаются в жесткой защитной упаковке, поэтому в действительности они не

[1] Дискеты размером 5,25 дюйма уже несколько лет как вышли из обращения В 2001 году производители персональных компьютеров выпустили стандарт, согласно которому и дискеты размером 3,5 дюйма должны будут окончить свое существование, так как в новые компьютеры не будут устанавливаться приводы для работы с этими дискетами. — *Примеч. научи, ред.*

гибкие. Поскольку 3-дюймовые дискеты вмещают больше данных и лучше защищены от внешних воздействий, они, по существу, заменили старые 5-дюймовые. Наиболее важные параметры всех 4 типов дискет показаны в табл. 2.3.

Таблица 2.3. Параметры четырех видов дискет

Параметры	LD5.25	HD5.25	LD3,5	HD3.5
Размер, дюймы	5,25	5,25	3,5	3,5
Емкость	360 Кбайт	1,2 Мбайт	720 Кбайт	1,44 Мбайт
Количество дорожек	40	60	80	80
Количество секторов в дорожке	9	15	9	18
Количество головок	2	2	2	2
Число оборотов в мин.	300	360	300	300
Скорость передачи данных, Кбит/с	250	500	250	500
Тип	Гибкий	Гибкий	[1]	Жесткий

Диски IDE

Диски современных персональных компьютеров развились из диска машины IBM PC XT. Это был диск Seagate на 10 Мбайт, управляемый контроллером Xebec на встроенной карте. У этого диска было 4 головки, 306 цилиндров и по 17 секторов на дорожке. Контроллер мог управлять двумя дисками. Операционная система считывала с диска и записывала на диск информацию. Для этого она передавала параметры в регистры процессора и вызывала систему **BIOS (Basic Input Output System — базовую систему ввода-вывода),** расположенную во встроенном ПЗУ. Система BIOS запрашивала машинные команды для загрузки регистров контроллера, которые начинали передачу данных.

Сначала контроллер помещался на отдельной плате, а позже, начиная с **IDE-дисков (Integrated Drive Electronics — устройство со встроенным контроллером),** которые появились в середине 80-х годов, стал встраиваться в материнскую плату[2]. Однако соглашения о вызовах системы BIOS не изменялись, поскольку необходимо было обеспечить совместимость с более старыми версиями. Обращение к секторам производилось по номерам головки, цилиндра и сектора, причем головки и цилиндры нумеровались с 0, а секторы — с 1. Вероятно, такая ситуация сложилась из-за ошибки одного из программистов BIOS, который написал свой шедевр на ассемблере 8088. Имея 4 бита для номера головки, 6 битов для сектора и 10 битов для цилиндра, диск мог содержать максимум 16 головок, 63 сектора и 1024 цилиндра, то есть всего 1 032 192 сектора. Емкость такого диска составляла 528 Мбайт, и в те времена эта цифра считалась огромной (а вы бы стали сегодня осыпать упреками новую машину, которая не способна работать с дисками объемом более 1 Тбайт?).

[1] Сам магнитный диск — гибкий, жестким является только футляр, в котором он расположен. — *Примеч. научи, ред.*

[2] Встраиваться он стал в сам винчестер, то есть на печатную плату, расположенную в корпусе винчестера. На материнской плате размещается иторая часть контроллера этого интерфейса. — *Примеч. научн.ред.*

Вскоре появились диски объемом более 528 Мбайт, но у них была другая геометрия (4 головки, 32 сектора, 2000 цилиндров). Операционная система не могла обращаться к ним из-за того, что соглашения о вызовах системы BIOS не менялись (требование совместимости). В результате контроллеры начали выдавать ложную информацию, делая вид, что геометрия диска соответствовала системе BIOS. Но на самом деле виртуальная геометрия просто накладывалась на реальную. Хотя этот метод действовал, он затруднял работу операционных систем, которые размещали данные определенным образом, чтобы сократить время поиска.

В конце концов на смену IDE дискам пришли EIDE-диски (Extended **IDE** — усовершенствованные IDE), поддерживающие дополнительную схему адресации **LBA (Logical Block Adressing),** которая просто нумерует секторы от 0 до $2^{24}-1$. Контроллер должен переделывать адреса LBA в адреса головки, сектора и цилиндра, но зато объем диска превышает 528 Мбайт. EIDE диски и контроллеры также имеют другие усовершенствования. Например, они способны контролировать 4 диска вместо двух, у них более высокая скорость передачи данных, и они могут управлять приводом для чтения CD-ROM.

Изначально IDE- и EIDE-диски производились только для систем Intel, поскольку данный интерфейс является точной копией шины IBM PC. Тем не менее в настоящее время некоторые другие компьютеры также используют эти диски из-за их низкой стоимости.

SCSI-диски

SCSI-диски не отличаются от IDE-дисков с точки зрения расположения цилиндров, дорожек и секторов, но они имеют другой интерфейс и более высокую скорость передачи данных, SCSI-диски восходят к изобретателю дискеты Говарду Шугарту (Hovard Shugart). В 19791 оду его компания выпустила диск SASI (Shugart Associates System Interface). В 1986 году Институт американских государственных стандартов после длительных обсуждений внес некоторые преобразования в этот диск и изменил его название на **SCSI (Small** Computer System **Interface — интерфейс малых вычислительных систем).** Аббревиатура SCSI произносится как «скази». Версии, работающие с более высокой скоростью, получили названия Fast SCSI (10 МГц), Ultra SCSI (20 МГц) и Ultra2 SCSI (40 МГц). Каждая из этих разновидностей также имела 16-битную версию. Основные параметры всех этих версий приведены в табл. 2.4.

Таблица 2.4. Некоторые возможные параметры SCSI

Название	Количество разрядов	Шина (МГц)	Скорость передачи данных по шине, Мбайт/с
SCSI-1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80

Поскольку у дисков SCSI высокая скорость передачи данных, они используются в большинстве рабочих станций UNIX, которые производятся Sun, HP, SGI и другими компаниями. Эти диски также встраиваются в компьютеры Macintosh и сетевые серверы Intel.

SCSI — это не просто интерфейс жесткого диска. Это шина, к которой могут подсоединяться контроллер SCSI и до семи дополнительных устройств. Ими могут быть один или несколько жестких дисков SCSI, компакт-диски, устройства для записи компакт-дисков, сканеры, накопители на магнитной ленте и другие периферийные устройства. Каждое устройство имеет свой идентификационный код от О до 7 (до 15 для 16-битных версий). У каждого устройства есть два разъема: один — входной, другой — выходной. Кабели соединяют выходной разъем одного устройства с входным разъемом следующего устройства и т. д. Это похоже на соединение лампочек в елочной гирлянде. Последнее устройство в цепи должно завершать цепь, чтобы отражения от концов шины не искажали другие данные в шине. Обычно контроллер помещается на встроенной карте и является первым звеном цепи, хотя это не обязательно.

Самый обычный кабель для 8-битного SCSI имеет 50 проводов, 25 из которых (заземления) спарены с 25 другими, чтобы обеспечить хорошую помехоустойчивость, которая необходима для высокой скорости работы. Из 25 проводов 8 используются для данных, 1 — для контроля четности, 9 — для управления, а оставшиеся сохраняются для будущего применения. 16-битным и 32-битным устройствам требуется еще 1 кабель для дополнительных сигналов. Кабели могут быть несколько метров в длину, чтобы обеспечивать связь с внешними устройствами (сканерами и т. п.).

Контроллеры и периферийные устройства SCSI могут быть или задатчиками, или приемниками. Обычно контроллер, действующий как задатчик, посылает команды дискам и другим периферийным устройствам, которые, в свою очередь, являются приемниками. Команды представляют собой блоки до 16 байтов, которые сообщают приемнику, что нужно делать. Команды и ответы на них оформляются в виде фраз, при этом используются различные сигналы контроля для разграничения фраз и разрешения конфликтных ситуаций, которые возникают, если несколько устройств одновременно пытаются использовать шину. Это очень важно, так как SCSI позволяет всем устройствам работать одновременно, что сильно повышает производительность среды, поскольку активизируется сразу несколько процессов (в качестве примеров можно привести UNIX или Windows NT). В системах IDE и EIDE если работает одно из устройств, другие не могут действовать одновременно с ним.

RAID-массивы

Производительность процессоров за последнее десятилетие сильно возросла, увеличиваясь почти вдвое каждые 1,5 года. Однако с производительностью дисков дело обстоит иначе. В 70-х годах среднее время поиска в мини-компьютерах составляло от 50 до 100 миллисекунд. Сейчас время поиска составляет около 10 миллисекунд. Во многих отраслях технической промышленности (например, в автомобильной или авиационной) увеличение производительности в 5 или 10 раз за два десятилетия считалось бы грандиозным, но в компьютерной промышленности

эти цифры вызывают недоумение Таким образом, разрыв между производительностью процессоров и дисков становился все больше и больше

Как мы уже видели, для тою чтобы увеличить скорость работы процессора, используется параллельная обработка данных Уже на протяжении многих лет разным людям приходит в голову мысль, что было бы неплохо сделать так, чтобы устройства ввода-вывода также могли работать параллельно В 1988 году Паттерсон, Гибсон и Кате в своей статье предложили 6 разных типов организации дисков, которые могли использоваться для увеличения производительности, надежности или того и другого Эти идеи были сразу заимствованы производителями компьютеров, что привело к появлению нового класса устройств ввода-вывода под названием **RAID** Паттерсон, Гибсон и Кате определили RAID как **Redundunt Array of Inexpensive Disks** — «избыточный массив недорогих дисков», но позже буква I в аббревиатуре стала заменять слово Independent (независимый) вместо изначального слова Inexpensive (недорогой) Может быть, в этом случае у производителей появилась возможность выпускать дорогостоящие диски? RAID-массиву противопоставлялся **SLED (Single Large Expensive Disk** — «один большой дорогостоящий диск»)

Основная идея RAID состоит в следующем Рядом с компьютером (обычно большим сервером) устанавливается бокс с дисками, контроллер диска замещается RAID-контроллером, данные копируются на RAID-массив, а затем производятся обычные действия Иными словами, операционная система воспринимает RAID как SLED, при этом у RAID-массива выше производительность и надежность Поскольку SCSI-диски обладают высокой производительностью при довольно низкой цене, при этом один контроллер может управлять несколькими дисками (до 7 дисков на 8-битных моделях SCSI и до 15 на 16-битных), то естественно, что большинство устройств RAID состоит из RAID SCSI-контроллера и бокса SCSI-дисков, которые операционная система воспринимает как один большой диск Таким образом, чтобы использовать RAID-массив, не требуется никаких изменений в программном обеспечении, что очень выгодно для многих системных администраторов

Системы RAID имеют несколько преимуществ Во-первых, как мы уже сказали, программное обеспечение воспринимает RAID как один большой диск Во-вторых, данные на всех RAID распределены по дискам таким образом, чтобы можно было осуществлять параллельные операции Несколько различных способов распределения данных были предложены Паттерсоном, Гибсоном и Катсом Сейчас они известны как RAID-массив нулевого уровня, RAID-массив первого уровня ит д до RAID-массива пятого уровня Кроме того, существует еще несколько уровней, которые мы не будем обсуждать Термин «уровень» несколько неудачный, поскольку здесь нет никакой иерархической структуры Просто существует б разных типов организации дисков

RAID-массив нулевого уровня показан на рис 2 18, а Он представляет собой виртуальный диск, разделенный на полосы, зоны (strips) no k секторов каждая, при этом секторы с 0 по к-1 — полоса 0, секторы с **к** по 2к-1 — полоса 1 и т д Для к=1 каждая полоса — это сектор, для к=2 каждая полоса — это два сектора и т д RAID-массив нулевого уровня последовательно записывает полосы по кругу, как показано на рис 2 18, *а* На этом рисунке изображен RAID-массив с 4 дисками Такое распределение данных по нескольким дискам называется **разметкой (striping)**

Например, если программное обеспечение вызывает команду для считывания блока данных, состоящего из четырех последовательных полосок и начинающегося на границе между полосками, то RAID-контроллер разбивает эту команду на 4 отдельные команды, каждую для одного из четырех дисков, и выполняет их параллельно. Таким образом, мы получаем устройство параллельного ввода-вывода без изменения программного обеспечения.

RAID-массив нулевого уровня лучше всего работает с большими запросами, чем больше запрос, тем лучше. Если полосок в запросе больше, чем дисков в RAID-массиве, то некоторые диски получают по несколько запросов, и как только такой диск завершает выполнение первого запроса, он приступает к следующему. Задача контроллера состоит в том, чтобы разделить запрос должным образом, послать нужные команды соответствующим дискам в правильной последовательности, а затем правильно записать результаты в память. Производительность при таком подходе очень высокая, и осуществить его несложно.

RAID-массив нулевого уровня хуже всего работает с операционными системами, которые время от времени запрашивают данные по одному сектору за раз. В этом случае результаты будут, конечно, правильными, но не будет никакого параллелизма и, следовательно, никакого выигрыша в производительности. Другой недостаток такой структуры состоит в том, что надежность у нее потенциально ниже, чем у SLED. Рассмотрим RAID-массив, состоящий из четырех дисков, на каждом из которых могут происходить сбои в среднем каждые 20 000 часов. Сбои в таком RAID-массиве будут случаться примерно через каждые 5000 часов, при этом все данные могут быть утеряны. У SLED сбои происходят также в среднем каждые 20 000 часов, но так как это один диск, его надежность в 4 раза выше. Поскольку в описанной разработке нет никакой избыточности, это не настоящий[1] RAID-массив.

Следующая разновидность — RAID-массив первого уровня. Он показан на рис. 2.18, *б* и, в отличие от RAID-массива нулевого уровня, является настоящим RAID-массивом[2]. Он дублирует все диски, таким образом получается 4 изначальных диска и 4 резервные копии. При записи информации каждая полоса записывается дважды. При считывании может использоваться любая из двух копий, при этом одновременно может происходить загрузка информации с большего количества дисков, чем у RAID-массива нулевого уровня. Следовательно, производительность при записи будет такая же, как у обычного диска, а при считывании — гораздо выше (максимум в два раза). Отказоустойчивость отличная: если происходит сбой на диске, вместо него используется копия. Восстановление состоит просто в установке нового диска и копировании всей информации с резервной копии на пего.

В отличие от нулевого и первого уровней, которые работают с полосами секторов, RAID-массив второго уровня имеет дело со словами, а иногда даже с байтами. Представим, что каждый байт виртуального диска разбивается на два кусочка по 4 бита, затем к каждому из них добавляется код Хэмминга, и таким образом получается слово из 7 битов, у которого 1,2 и 4 — биты четности. Затем представим, что 7 дисков, изображенные на рис. 2.18, *в,* были синхронизированы по позиции рыча-

[1] На самом деле настоящий, но нулевого уровня. — *Примеч. научи, ред.*

[2] На рис. 2.18, б изображен RAID уровня 0+1, а не 1-го уровня. —*Примеч. научн.ред.*

га и позиции вращения. Тогда было бы возможно записать слово из 7 битов с кодом Хэмминга на 7 дисков, по 1 биту на диск

Подобная схема использовалась в так называемых думающих машинах СМ-2. К 32-битному слову с данными добавлялось 6 битов четности (код Хэмминга). В результате получалось 38-битное кодированное слово, к которому добавлялся дополнительный бит четности, и это слово записывалось на 39 дисков. Общая производительность была огромной, так как одновременно могло записываться 32 сектора данных При утрате одного из дисков проблем также не возникало, поскольку потеря одного диска означала потерю одного бита в каждом 39-битном слове, а с этим код Хэмминга справлялся моментально.

С другой стороны, эта схема требует, чтобы все диски были синхронизированы по вращению Кроме того, она имеет смысл, только если имеется достаточно большое количество дисков (даже при наличии 32 дисков для данных и 6 дисков для битов четности накладные расходы составляют 19 процентов). К тому же требуется большая работа контроллера, поскольку он должен вычислять контрольную сумму кода Хэмминга каждый раз при передаче бита.

RAID-массив третьего уровня представляет собой упрощенную версию RAID-массива второго уровня. Он изображен на рис. 2.18, г. Здесь для каждого слова данных вычисляется 1 бит четности и записывается на диск четности. Как и в RAID-массиве второго уровня, диски должны быть точно синхронизированы, поскольку каждое слово данных распределено на несколько дисков.

На первый взгляд может показаться, что один бит четности только обнаруживает, но не исправляет ошибки. Если речь идет о случайных необнаруженных ошибках, это наблюдение верно Однако если речь идет о сбое на диске, бит четности обеспечивает исправление 1-битной ошибки, поскольку позиция неправильного бита известна. Если происходит сбой, контроллер выдает информацию, что все биты равны 0 Если в слове возникает ошибка четности, бит с диска, на котором произошел сбой, должен быть 1, и следовательно, он исправляется. Хотя RAID-массивы второго и третьего уровней обеспечивают очень высокую скорость передачи данных, число запросов устройств ввода-вывода в секунду не больше, чем при наличии одного диска.

RAID-массиавы четвертого и пятого уровней опять работают с полосами, а не со словами с битами четности, и не требуют синхронизации дисков RAID-массив четвертого уровня (см. рис. 2.18, *д)* устроен так же, как RAID-массив нулевого уровня, с тем различием, что у RAID-массива четвертого уровня имеется дополнительный диск, на который записываются полосы четности Например, пусть каждая полоса состоит из к байтов Все полосы должны находиться в отношении «исключающего ИЛИ», и полоса четности для проверки этого отношения также должна состоять из к байтов Если происходит сбой на диске, утраченные байты могут быть вычислены заново при использовании информации с диска четности

Такая разработка предохраняет от потерь на диске, но обладает очень низкой производительностью в случае небольших исправлений. Если изменяется 1 сектор, необходимо считывать информацию со всех дисков, для того чтобы опять вычислить четность, которая должна быть записана заново Вместо этого можно считать с диска прежние данные и прежнюю четность и из них вычислить новую четность. Но даже с такой оптимизацией процесса при наличии небольших исправлений требуется произвести два считывания и две записи.

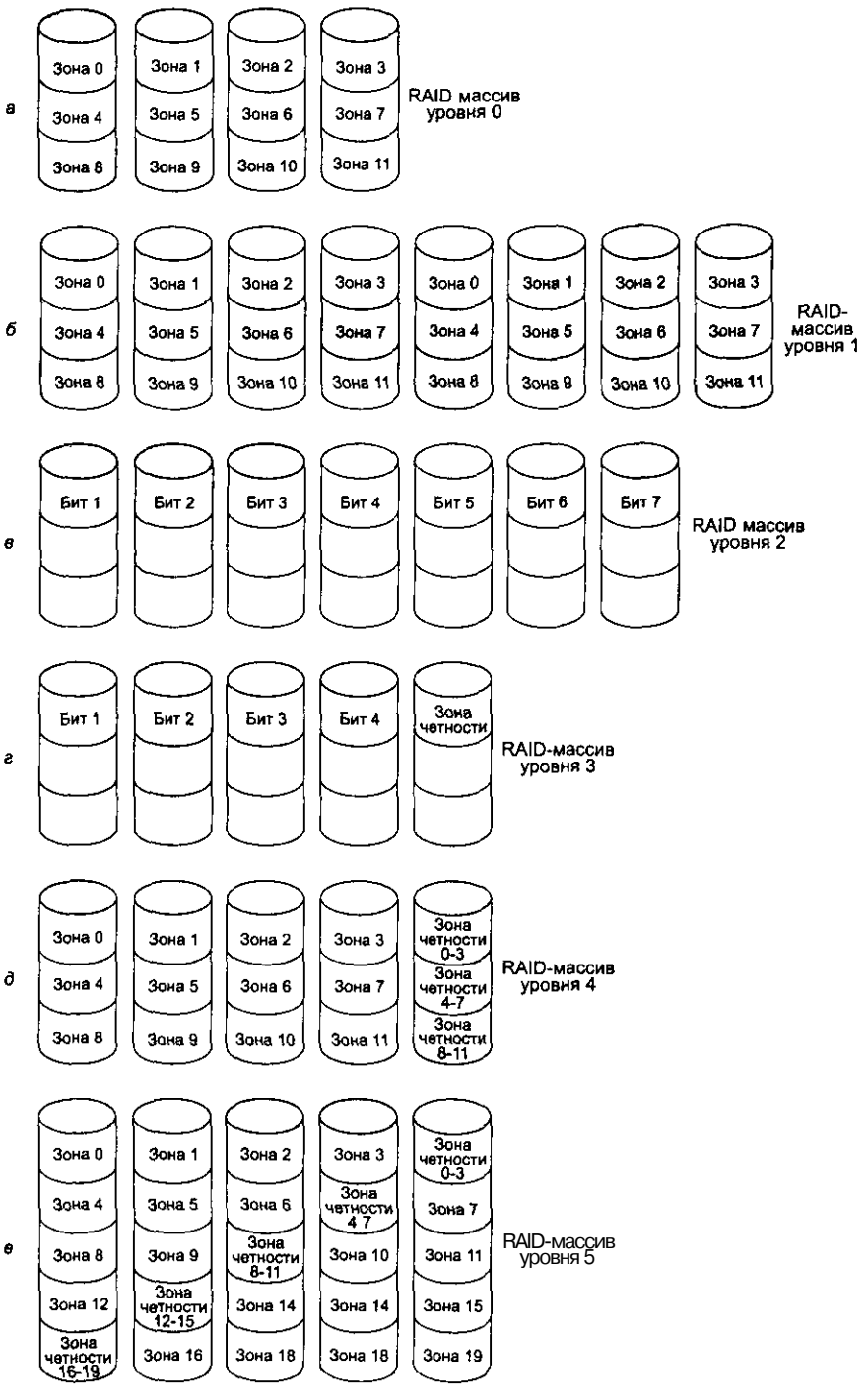


Рис. 2.18. RAID-массивы с нулевого по пятый уровень Резервные копии и диски четности закрашены серым цветом

Такие трудности при загрузке на диск четности могут быть препятствием высокой производительности. Эта проблема устраняется в RAID-массиве пятого уровня, в котором биты четности распределяются равномерно по всем дискам и записываются по кругу, как показано на рис. 2.18, *е*. Однако в случае сбоя на диске восстановить содержание утраченного диска достаточно сложно, хотя и возможно.

Компакт-диски

В последние годы помимо магнитных дисков стали доступны оптические диски. Они обладают более высокой плотностью записи, чем обычные магнитные диски[1]. Оптические диски изначально использовались для записи телевизионных программ, но позже они стали использоваться как средства хранения информации в компьютерной технике. Из-за потенциально огромной емкости оптические диски стали предметом многих исследований, и их усовершенствование происходило довольно быстро.

Первые оптические диски были изобретены голландской корпорацией Philips для хранения кинофильмов. Они имели 30 см в диаметре, выпускались под маркой LaserVision, но нигде, кроме Японии, не пользовались популярностью.

В 1980 году корпорация Philips вместе с Sony разработала CD (Compact Disc — компакт-диск), который быстро вытеснил виниловые диски, использовавшиеся для музыкальных записей.

Описание технических деталей компакт-диска было опубликовано в официальном Международном Стандарте (IS 10149), который часто называют **Красной книгой** (по цвету обложки). Международные Стандарты издаются Международной Организацией Стандартизации, которая представляет собой аналог таких национальных групп стандартизации, как ANSI, DIN и т. и. У каждой такой группы есть свой IS-номер (International Standard — Международный Стандарт). Международный Стандарт технических характеристик диска был опубликован для того, чтобы компакт-диски от разных музыкальных издателей и проигрыватели от разных производителей стали совместимыми. Все компакт-диски должны быть 120 мм в диаметре и 1,2 мм в толщину, а диаметр отверстия в середине должен составлять 15 мм. Аудио-компакт-диски были первым средством хранения цифровой информации, которое вышло на массовый рынок потребления. Предполагается, что они будут использоваться на протяжении ста лет. Пожалуйста, сравните в 2080 году работу самой последней разработки и первой партии компакт-дисков.

Компакт-диск изготавливается с использованием очень мощного инфракрасного лазера, который выжигает отверстия диаметром 0,8 микрон в специальном стеклянном контрольном диске. По этому контрольному диску делается шаблон с выступами в тех местах, где лазер прожег отверстия. В шаблон вводится жидкая смола (поликарбонат), и таким образом получается компакт-диск с тем же набором отверстий, что и в стеклянном диске. На смолу наносится очень тонкий слой алюминия, который в свою очередь покрывается защитным лаком. После этого наклеивается этикетка. Углубления в нижнем слое смолы в английском языке называются термином **«впадина»** (pit), а ровные пространства между впадинами называются термином «площадка»- (land).

---

[1] Это утверждение ошибочно. — *Примеч. научи, ред.*

Во время воспроизведения лазерный диод небольшой мощности светит инфракрасным светом с длиной волны 0,78 микрон на сменяющиеся впадины и площадки Лазер находится на той стороне диска, где слой смолы, поэюму впадины для лазера оказываются выступами на ровной поверхности Так как впадины имеют высоту в четверть длины волны света лазера, длина волны света, отраженного от впадины, составляет половину длины волны света, отраженного от окружающей выступ ровной поверхности В результате, если свет отражается от выступа, фотодетектор проигрывателя получает меньше света, чем при отражении от площадки Именно таким образом проигрыватель отличает впадину от площадки Хотя, казалось бы, проще всего использовать впадину для записи 0, а площадку для записи 1, более надежно использовать переход впадина/площадка или площадка/впадина для 1 и его отсутствие для 0

Впадины и площадки записываются по спирали Запись начинается на некотором расстоянии от отверстия в центре диска и продвигается к краю, занимая 32 мм диска Спираль проходит 22 188 оборотов вокруг диска (примерно 600 на 1 мм) Если ее распрямить, ее длина составит 5,6 км Спираль изображена на рис 2 19

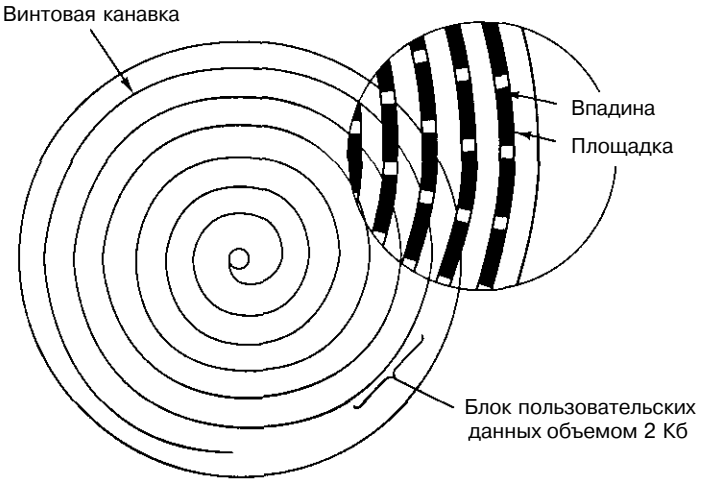


Рис. 2.19. Структура записи компакт-диска

Чтобы музыка звучала нормально, впадины и площадки должны сменяться с постоянной линейной скоростью Следовательно, скорость вращения компакт-диска должна постепенно снижаться по мере продвижения считывающей головки от центра диска к внешнему краю Когда головка находится на внутренней стороне диска, скорость вращения составляет 530 оборотов в минуту, чтобы достичь желаемой скорости 120 см/с Когда головка находится на внешней стороне диска, скорость вращения падает до 200 оборотов в минуту, чтобы обеспечить такую же линейную скорость Диск с постоянной линейной скоростью отличается от магнитного диска, который работает с постоянной угловой скоростью, независимо от того, где находится головка в настоящий момент Кроме того, скорость вращения компакт-диска (530 оборотов в минуту) очень сильно отличается от скорости вращения магнитных дисков, которая составляет от 3600 до 7200 оборотов в минуту

В 1984 году Philips и Sony начали использовать компакт-диски для хранения компьютерных данных. Они опубликовали **Желтую книгу,** в которой определили точный стандарт того, что они назвали **CD-ROM (Compact Disc - Read Only Memory — компакт-диск — постоянное запоминающее устройство).** Чтобы влиться в широко развернувшийся к тому времени рынок аудио-компакт-дисков, компьютерные компакт-диски должны были быть такого же размера, как аудио-диски, механически и оптически совместимыми с ними и производиться по той же технологии. Вследствие такого решения потребовались моторы, работающие с низкой скоростью и способные менять скорость. Стоимость производства одного компакт-диска составляла в среднем меньше 1 доллара.

В Желтой книге определено форматирование компьютерных данных. В ней также описаны усовершенствованные приемы исправления ошибок, что является существенным шагом, поскольку компьютерщики, в отличие от любителей музыки, придают очень большое значение ошибкам в битах. Разметка компакт-диска состоит в кодировании каждого байта 14-битным символом. Как мы видели выше, 14 битов достаточно для того, чтобы закодировать кодом Хэмминга 8-битный байт, при этом останется два лишних бита. На самом деле используется более мощная система кодировки. *Перевод из 16-битной в 8-битную систему для считывания информации производится аппаратным обеспечением с помощью поисковых таблиц.*

На следующем уровне 42 последовательных символа формируют фрейм из 588 битов. Каждый фрейм содержит 192 бита данных (24 байта). Оставшиеся 396 битов используются для исправления ошибок и контроля. У аудио- и компьютерных компакт-дисков эта система одинакова.

У компьютерных компакт-дисков каждые 98 фреймов группируются в **сектор,** как показано на рис. 2.20. Каждый сектор начинается с преамбулы из 16 байтов, первые 12 из которых - O0FFFFFFFFFFFFFFFFFFO0 (в шестнадцатеричной системе), что дает возможность проигрывателю определять начало сектора. Следующие 3 байта содержат номер сектора, который необходим, поскольку поиск на компакт-диске, на котором данные записаны по спирали, гораздо сложнее, чем на магнитном диске, где данные записаны на концентрических дорожках. Чтобы найти определенный сектор, программное обеспечение подсчитывает, куда приблизительно нужно направляться; туда помещается считывающая головка, а затем начинается поиск преамбулы, чтобы установить, насколько верен был подсчет. Последний байт преамбулы определяет тип диска.

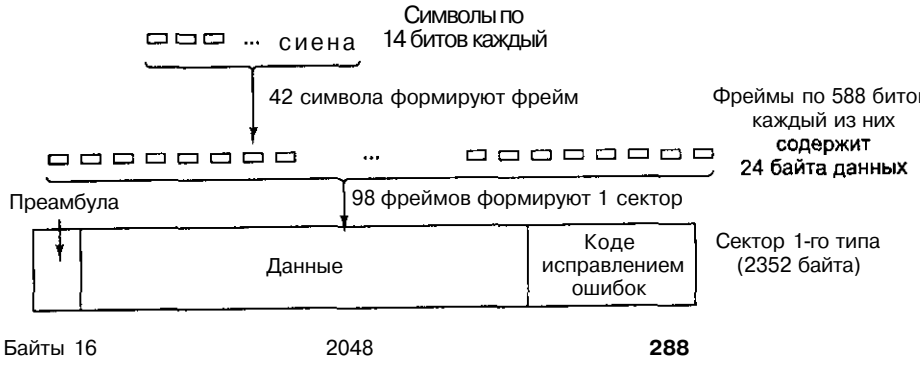


Рис. 2.20. Схема расположения данных на компакт-диске

Желтая книга определяет 2 типа дисков. Тип 1 использует расположение данных, показанное на рис 2 20, где преамбула составляет 16 байтов, данные — 2048 байтов, а код с исправлением ошибок — 228 байтов (код Рида—Соломона) Тип 2 объединяет данные и коды с исправлением ошибок в поле данных на 2336 байтов. Такая схема применяется для приложений, которые не нуждаются в исправлении ошибок (или, точнее, которые не могут выделить время для этого), например аудио и видео Отметим, что для обеспечения высокой степени надежности используются три схемы исправления ошибок в пределах символа, в пределах фрейма и в пределах сектора Одиночные ошибки в битах исправляются на самом нижнем уровне, пакеты ошибок — на уровне фреймов, а все остаточные ошибки — на уровне секторов Для обеспечения такой надежности необходимо 98 фреймов по 588 битов (7203 байта), чтобы поддерживать 2048 байтов полезной нагрузки Таким образом, эффективность составляет всего 28%

Односкоростные устройства для чтения компакт-дисков считывают 75 секторов/с, что обеспечивает скорость передачи данных 153 600 байт/с при диске первого типа **и** 175 200 баит/с при диске второго типа Двухскоростные устройства работают в два раза быстрее и т д , до самой высокой скорости Стандартный аудио-компакт-диск располагает емкостью для 74 минут музыки, что соответствует 681 984 000 байтов Это число равно 650 Мбайт, так как 1 Мбайт=2^{20} байтов (1 048 576 байт), а не 1 000 000 байтов

Отметим, что даже устройство для чтения компакт-дисков со скоростью, обозначаемой как 32х (4 915 200 байт/с), не сравнимо с быстрым магнитным диском SCSI-2 (10 Мбайт/с), несмотря на то, что многие устройства для чтения компакт-дисков используют интерфейс SCSI (кроме того, применяется интерфейс EIDE). Когда вы понимаете, что время поиска составляет несколько сотен миллисекунд, становится ясно, что устройства для чтения компакт-дисков по производительности сильно уступают магнитным дискам, хотя емкость компакт-дисков гораздо выше[1].

В 1986 году корпорация Philips опубликовала **Зеленую книгу,** добавив графику и возможность помещать аудио-, видео- и обычные данные в одном секторе, что было необходимо для мультимедийных компакт-дисков

Последняя проблема, которую нужно было разрешить при разработке компакт-дисков, — совместимость файловой системы Чтобы можно было использовать один и тот же компакт-диск на разных компьютерах, необходимо было соглашение по поводу файловой системы компакт-дисков Чтобы выпустить такое соглашение, представители разных компьютерных компаний встретились на озере Тахо в Хай-Сьерраз (the High Sierras) на границе Калифорнии и Невады и разработали файловую систему, которую они назвали **High Sierra** Позднее эта система превратилась в Международный Стандарт (IS 9660) Существует три уровня этого стандарта На первом уровне допустимы имена файлов до 8 символов, за именем файла может следовать расширение до трех символов (соглашение для наименования файлов в MS-DOS) Имена файлов могут содержать только прописные буквы, цифры и символ подчеркивания Директории могут вкладываться одна в другую, причем

[1] Емкость компакт-дисков на два порядка ниже емкости современных магнитных дисков — *Примеч научн ред*

допускается не более 8 иерархических ступеней. Имена директорий могут не содержать расширения. На первом уровне требуется, чтобы все файлы были смежными, что не представляет особых трудностей в случае с носителем, на который информация записывается только один раз. Любой компакт-диск, который соответствует Международному Стандарту IS 9660 первого уровня, может быть прочитан с использованием системы MS-DOS, компьютеров Apple, Unix и практически любого другого компьютера. Производители компакт-дисков считают это свойство большим плюсом.

Второй уровень Международного Стандарта IS 9660 допускает имена файлов до 32 символов, а на третьем уровне допускается несмежное расположение файлов. Расширения Rock Ridge (названные так причудливо в честь города в фильме Джина Уайлдера «Горящие седла») допускают очень длинные имена файлов (для Unix), UID, GID и символические связи, но компакт-диски, не соответствующие первому уровню, не будут читаться на всех компьютерах

Компакт-диски стали очень популярны для распространения компьютерных игр, художественных фильмов, энциклопедий, атласов и различного рода справочников. В настоящее время на компакт-дисках выпускается бóльшая часть коммерческого программного обеспечения. Сочетание большой вместимости и низкой цены делает компакт-диски подходящими для бесчисленного множества приложений.

CD-R

Вначале оборудование, необходимое для изготовления контрольных компакт-дисков (как аудио-, так и компьютерных), было очень дорогим Но, как это обычно происходит в компьютерной промышленности, ничего не остается дорогим долгое время. К середине 90-х годов записывающие устройства для компакт-дисков размером не больше проигрывателя стали обычными и общедоступными, их можно было приобрести в любом магазине компьютерной техники. Эти устройства все еще отличались от магнитных дисков, поскольку информацию, записанную однажды на компакт-диск, уже нельзя было стереть. Тем не менее они быстро нашли сферу применения в качестве дополнительных носителей информации, а основными носителями продолжали служить жесткие диски. Кроме того, отдельные лица и начинающие компании могли выпускать свои собственные компакт-диски небольшими партиями или производить контрольные диски и отправлять их на крупные коммерческие предприятия, занимающиеся изготовлением копий. Такие диски называются **CD-R (CD-Recordable).**

CD-R производится на основе поликарбонатных заготовок. Такие же заготовки используются при производстве компакт-дисков Однако диски CD-R отличаются от компакт-дисков тем, что CD-R содержат канавку шириной 0,6 мм, чтобы направлять лазер при записи Канавка имеет синусоидальное отклонение 0,3 мм на частоте ровно 22,05 кГц для обеспечения постоянной обратной связи, чтобы можно было точно определить скорость вращения и в случае необходимости отрегулировать ее. CD-R выглядит как обычный диск, только он не серебристого, а золотистого цвета, так как для изготовления отражающего слоя вместо алюминия

используется настоящее золото. В отличие от обычных компакт-дисков с физическими углублениями, CD-R моделируются с помощью изменения отражательной способности впадин и площадок Для этого между слоем поликарбоната и отражающим слоем золота помещается слой красителя, как показано на рис 2.21. Используется два вида красителей: цианин зеленого цвета и пталоцианин желтовато-оранжевого цвета. Химики могут спорить до бесконечности, какой из них лучше. Эти красители сходны с теми красителями, которые используются в фотографии, и именно поэтому Kodak и Fuji являются главными производителями дисков CD-R.

На начальной стадии слой красителя прозрачен, что дает возможность свету лазера проходить сквозь него и отражаться от слоя золота. При записи информации мощность лазера увеличивается до 8-16 мВт. Когда луч достигает красителя, краситель нагревается, и в результате разрушается химическая связь. Такое изменение молекулярной структуры создает темное пятно. При чтении (когда мощность лазера составляет 0,5 мВт) фотодетектор улавливает разницу между темными пятнами, где краситель был поврежден, и прозрачными областями, где краситель не тронут. Это различие воспринимается как различие между впадинами и площадками даже при чтении на обычном устройстве для считывания компакт-дисков или на аудио-проигрывателе.

Ни один новый вид компакт-дисков не обходился без публикации параметров в книге определенного цвета. В случае с CD-R это была **Оранжевая книга,** вышедшая в 1989 году. Этот документ определяет диск CD-R, а также новый формат, **CD-ROM XA,** который позволяет записывать информацию на CD-R постепенно: несколько секторов сегодня, несколько секторов завтра, несколько секторов через месяц. Группа последовательных секторов, записываемых за 1 раз, называется **дорожкой компакт-диска.**

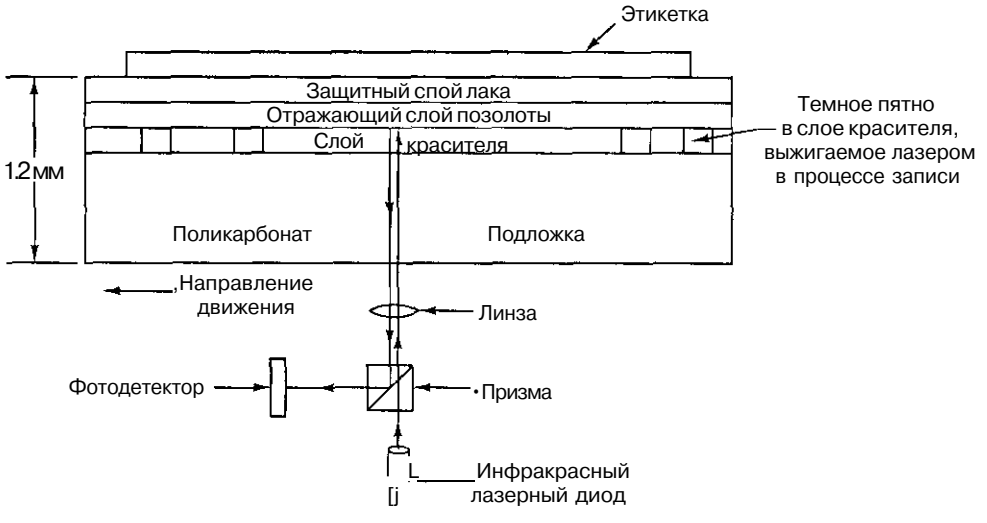


Рис. 2.21. Поперечное сечение диска CD-R и лазера (масштаб не соблюдается). Обычный компакт-диск имеет сходную структуру, но у него отсутствует слой красителя и вместо слоя золота используется слой алюминия с выемками

Одним из первых применений CD-R был фото-компакт-диск фирмы Kodak. При такой системе клиент приносит экспонированную пленку и старый фото-компакт-диск в проявочную машину и получает свой старый компакт-диск, на который после старых снимков записаны новые. Новый пакет данных, полученный в результате сканирования негативов, записывается на компакт-диск в виде отдельной дорожки. Такой способ записи необходим, поскольку заготовки дисков CD-R слишком дорого стоят, поэтому записывать каждую новую пленку на новый диск невыгодно.

Однако с появлением такого типа записи возникла новая проблема. До появления Оранжевой книги у всех компакт-дисков был единый **VTOC (Volume Table of Contents — оглавление диска).** При такой системе дозаписывать диск было невозможно. Решением данной проблемы стало предложение давать для каждой дорожки компакт-диска отдельный VTOC. В список файлов VTOC могут включаться все файлы из предыдущих дорожек или некоторые из них. После того как диск CD-R вставлен в считывающее устройство, операционная система начинает искать среди дорожек самый последний VTOC, который выдает текущее состояние диска. Если в текущий VTOC включить только некоторые, а не все файлы из предыдущих дорожек, может создаться впечатление, что файлы были удалены. Дорожки можно группировать в **сессии. В** этом случае мы говорим о **многосессионных** компакт-дисках. Стандартные аудио-проигрыватели не могут работать с многосекционными компакт-дисками, поскольку они ожидают единый VTOC в начале диска.

Каждая дорожка должна записываться непрерывно без остановок. Поэтому жесткий диск, от которого поступают данные, должен работать достаточно быстро, чтобы вовремя их доставлять. Если файлы, которые нужно скопировать, расположены в разных частях жесткого диска, длительное время поиска может послужить причиной остановки потока данных на CD-R и, следовательно, причиной недобора данных буфера. В результате недобора данных буфера у вас появится замечательная блестящая (правда, немного дорогая) подставка для стаканов и бутылок. Программное обеспечение CD-R обычно предлагает параметр сбора всех необходимых файлов в виде блока последовательных данных. То есть до передачи файлов на CD-R создается копия компакт-диска в 650 Мбайт. Однако этот процесс обычно удваивает время записи, требует наличия 650 Мбайт свободного дискового пространства и не защищает от того, что жесткие диски начинают совершать рекалибровку в случае перегрева.

С появлением CD-R у отдельных лиц и компаний появилась возможность без труда копировать компьютерные и музыкальные компакт-диски, что часто происходит с нарушением авторских прав. Были придуманы разные средства, препятствующие производству пиратской продукции и затрудняющие чтение компакт-дисков с помощью программного обеспечения, разработанного не производителем данного диска. Один из таких способов — запись на компакт-диск информации о том, что длина всех файлов составляет несколько гигабайт. Это препятствует копированию файлов на жесткий диск с использованием обычного программного обеспечения. Настоящие размеры файлов включаются в программное обеспечение производителя данного компакт-диска или прячутся где-нибудь на компакт-диске (часто в зашифрованном виде). При другом подходе в избранные секторы вставляются заведомо неправильные коды с исправлением ошибок. Программ-

ное обеспечение, прилагаемое к данному компакт-диску, зафиксирует эти ошибки, а обычное программное обеспечение проверит эти коды с исправлением ошибок и не будет работать, если они заведомо правильные. Кроме того, возможно использование нестандартных промежутков между дорожками и других физических «дефектов».

CD-RW

Хотя люди и привыкли к таким носителям информации, которые нельзя перезаписывать (такими носителями являются, например, бумага или фотопленка), все равно существует спрос на перезаписываемые компакт-диски. В настоящее время появилась технология **CD-RW (CD-Rewritable — перезаписываемый компакт-диск).** При этом используется носитель такого же размера, как и CD-R. Однако вместо красителя (цианина или пталоцианина) при производстве CD-RW используется сплав серебра, индия, сурьмы и теллура для записывающего слоя. Этот сплав имеет два состояния: кристаллическое и аморфное, которые обладают разной отражательной способностью.

Устройства для записи компакт-дисков снабжены лазером с тремя вариантами мощности. При самой высокой мощности лазер расплавляет сплав, переводя его из кристаллического состояния с высокой отражательной способностью в аморфное состояние с низкой отражательной способностью, так получается впадина. При средней мощности сплав расплавляется и возвращается обратно в естественное кристаллическое состояние, при этом впадина превращается снова в площадку. При низкой мощности лазер определяет состояние материала (для считывания информации), никакого перехода состояний при этом не происходит.

CD-RW не заменили CD-R, поскольку заготовки дисков CD-RW гораздо дороже заготовок CD-R. Кроме того, для приложений, поддерживающих жесткие диски, большим плюсом является тот факт, что с CD-R нельзя случайно стереть информацию.

DVD

Основной формат компакт-дисков использовался с 1980 года. С тех пор технологии продвинулись вперед, поэтому оптические диски с высокой емкостью сейчас вполне доступны по цене и пользуются большим спросом. Голливуд с радостью заменил бы аналоговые видеозаписи на цифровые диски, поскольку они лучше по качеству, их дешевле производить, они дольше служат, занимают меньше места на полке в магазине и их не нужно перематывать. Компании, выпускающие бытовую технику, занимаются поисками нового массового продукта, а многие компьютерные компании хотят добавить к своему программному обеспечению мультимедиа.

Такое развитие технологий и спроса на продукцию трех чрезвычайно богатых и мощных индустрии привело к появлению **DVD** (изначально сокращение от **Digital Video Disk — цифровой видеодиск,** а сейчас официально **Digital Versatile Disk — цифровой универсальный диск).** Диски DVD в целом похожи на компакт-диски. Как и обычные компакт-диски, они имеют 120 мм в диаметре, создаются на основе поликарбоната и содержат впадины и площадки, которые освеща-

ются лазерным диодом и считываются фотодетектором. Однако существует несколько различий:

1. Впадины меньшего размера (0,4 микрона вместо 0,8 микрона, как у обычного компакт-диска).
2. Более плотная спираль (0,74 микрона между дорожками вместо 1,6 микрона).
3. Красный лазер (с длиной волны 0,65 микрона вместо 0,78 микрона).

В совокупности эти усовершенствования дали семикратное увеличение емкости (до 4,7 Гбайт). Считывающее устройство для DVD 1x работает со скоростью 1,4 Мбайт/с (скорость работы считывающего устройства для компакт-дисков составляет 150 Кбайт/с). К несчастью, из-за перехода к красному лазеру потребовались DVD-проигрыватели с двумя лазерами или со сложной оптической системой, чтобы можно было читать существующие музыкальные и компьютерные компакт-диски. Таким образом, не все DVD-проигрыватели могут работать со старыми компакт-дисками. Кроме того, не всегда возможно считывание дисков CD-R и CD-RW.

Достаточно ли 4,7 Гбайт? Может быть. Если использовать сжатие MPEG-2 (стандарт IS 13346), DVD-диск объемом 4,7 Гбайт может вместить полноэкранную видеозапись на 133 минуты с высокой разрешающей способностью (720x480) вместе с озвучиванием на 8 языках и субтитрами на 32 других языках. Около 92% фильмов, снятых в Голливуде, по длительности меньше 133 минут. Тем не менее для некоторых приложений (например, игр мультимедиа или справочных изданий) может понадобиться больше места, а Голливуд мог бы записывать по несколько фильмов на один диск. Поэтому было разработано 4 формата:

1. Односторонние однослойные (4,7 Гбайт).
2. Односторонние двуслойные (8,5 Гбайт).
3. Двусторонние однослойные (9,4 Гбайт).
4. Двусторонние двуслойные (17 Гбайт).

Зачем так много форматов? Если говорить коротко, основная причина — убеждения компаний. Philips и Sony считали, что нужно выпускать односторонние диски с двойным слоем, а Toshiba и Time Warner хотели производить двусторонние диски с одним слоем. Philips и Sony думали, что покупатели не захотят переворачивать диски, а компания Time Warner полагала, что если поместить два слоя на одну сторону диска, он не будет работать. Компромиссное решение — выпускать все варианты, а рынок уже сам определит, какой из вариантов выживет.

При двуслойной технологии на нижний отражающий слой помещается полуотражающий слой. В зависимости от того, где фокусируется лазер, он отражается либо от одного слоя, либо от другого. Чтобы обеспечить надежное считывание информации, впадины и площадки нижнего слоя должны быть немного больше по размеру, поэтому его емкость немного меньше, чем у верхнего слоя.

Двусторонние диски создаются путем склеивания двух односторонних дисков по 0,6 мм. Чтобы толщина всех версий была одинаковой, односторонний диск толщиной 0,6 мм приклеивается к пустой подложке (возможно, в будущем эта под-

ложка будет содержать 133 минуты рекламы, в надежде, что покупатели заинтересуются, что там на нем). Структура двустороннего диска с двойным слоем показана на рис. 2.22.

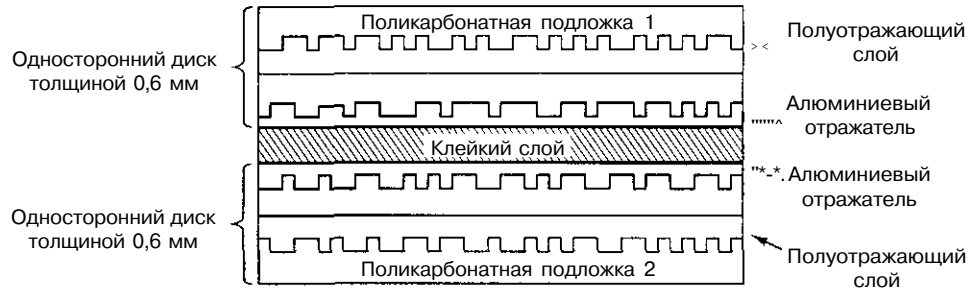


Рис. 2.22. Двусторонний диск DVD с двойным слоем

DVD был разработан корпорацией, состоящей из 10 компаний по производству бытовой техники, семь из которых были японскими, в тесном сотрудничестве с главными студиями Голливуда (японские компании являлись владельцами некоторых из этих студий). Ни компьютерная, ни телекоммуникационная промышленность не были вовлечены в разработку, и в результате упор был сделан на использование DVD для видеопрокатов и распродаж Перечислим некоторые стандартные особенности DVD: возможность исключать непристойные сцены из фильма (чтобы родители могли превращать фильм типа NC17[1] в фильм, который можно смотреть детям), шестиканальный звук, поддержка для перемасштабирования. Последняя особенность позволяет DVD-проигрывателю решать, как обрезать правый и левый край фильмов (у которых соотношение ширины и высоты 3:2) так, чтобы они подходили к современным телевизорам (с форматом 4:3).

Еще одна особенность, которая, вероятно, никогда не пришла бы в голову разработчикам компьютерных технологий, — намеренная несовместимость дисков для Соединенных Штатов и для европейских стран и другие стандарты для других континентов. Голливуд ввел такую систему, потому что новые фильмы всегда сначала выпускаются на экраны в Соединенных Штатах и только после появления видеокассет отправляются в Европу. Это делается для того, чтобы европейские магазины видеопродукции не могли покупать видеозаписи в Америке слишком рано (вследствие этого мог сократиться объем продаж новых фильмов в Европе). Если бы Голливуд стоял во главе компьютерной промышленности, то в Америке были бы дискеты 3,5 дюйма, а в Европе — 9 см.

Поскольку DVD-диски пользуются большой популярностью, возможно, что в скором времени диски DVD-R (на которых возможна запись информации) и DVD-RW (на которых возможна перезапись информации) станут продуктами массового потребления. Однако успех DVD не гарантирован, поскольку кабельные компании планируют доставлять фильмы несколько другим способом — по кабелю, и борьба уже началась.

[1] NC17 — фильмы, содержащие сцены секса и насилия и не предназначенные для просмотра детьми. — *Примеч. перев.*

Процесс ввода-вывода

Как мы сказали в начале этой главы, компьютерная система состоит из трех основных компонентов: центрального процессора, памяти (основной и вспомогательной) и устройств ввода-вывода (принтеров, сканеров и модемов). До сих пор мы рассматривали центральные процессоры и память. Теперь мы займемся изучением устройств ввода-вывода и тем, как они связываются с остальными компонентами системы.

Шины

Большинство персональных компьютеров и рабочих станций имеют физическую структуру, сходную с той, которая изображена на рис. 2.23. Обычное устройство представляет собой металлический корпус с большой интегральной схемой на дне, которая называется **материнской** платой. Материнская плата содержит микросхему процессора, несколько разъемов для модулей DIMM и различные микросхемы поддержки Она также содержит шину, протянутую вдоль нее, и несколько разъемов для подсоединения плат устройств ввода-вывода. Иногда может быть две шины одна с высокой скоростью передачи данных (для современных плат устройств ввода-вывода), а другая с низкой скоростью передачи данных (для старых плат устройств в вода-вывода).

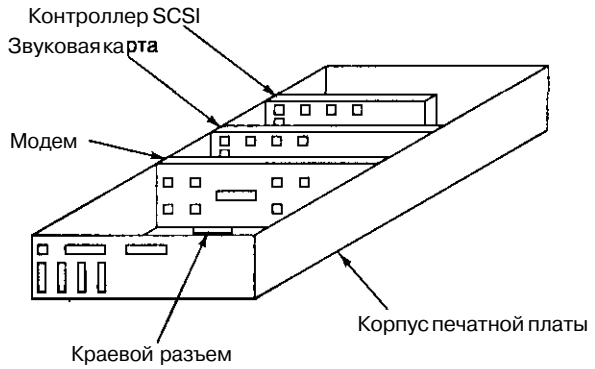


Рис. 2.23. Физическая структура персонального компьютера

Логическая структура обычного персонального компьютера показана на рис. 2.24. У данного компьютера имеется одна шина для соединения центрального процессора, памяти и устройств ввода-вывода, однако большинство систем содержат две и более шин Каждое устройство ввода-вывода состоит из двух частей: одна из **них** содержит большую часть электроники и называется **контроллером,** а другая представляет собой само устройство ввода-вывода, например дисковод. Контроллер обычно содержится на плате, которая втыкается в свободный разъем. Исключение представляют контроллеры, являющиеся обязательными (например, клавиатура), которые иногда располагаются на материнской плате. Хотя дисплей (монитор) и не является факультативным устройством, соответствующий контроллер иногда рас-

полагается на встроенной плате, чтобы пользователь мог по желанию выбирать платы с графическими ускорителями или без них, устанавливать дополнительную память и т. д. Контроллер связывается с самим устройством кабелем, который подсоединяется к разъему на задней стороне корпуса.

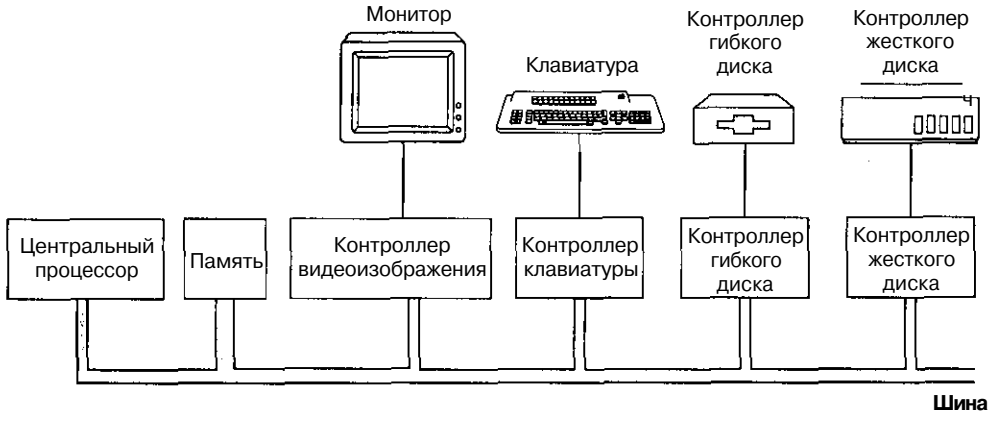


Рис. 2.24. Логическая структура обычного персонального компьютера

Контроллер управляет своим устройством ввода-вывода и регулирует доступ к шине для этого. Например, если программа запрашивает данные с диска, она посылает команду контроллеру диска, который затем отправляет команды поиска и другие команды на диск. После нахождения соответствующей дорожки и сектора диск начинает передавать контроллеру данные в виде потока битов. Задача контроллера состоит в том, чтобы разбить поток битов на куски и записывать каждый такой кусок в память по мере их накопления. Отдельный кусок обычно представляет собой одно или несколько слов. Если контроллер считывает данные из памяти или записывает их в память без участия центрального процессора, то говорят, что осуществляется **прямой доступ к памяти (Direct Memory Access,** сокращенно **DMA).** Когда передача данных заканчивается, контроллер вызывает **прерывание,** вынуждая центральный процессор приостановить работу текущей программы и начать выполнение особой процедуры. Эта процедура называется **программой обработки прерывания** и нужна, чтобы проверить ошибки, произвести необходимые действия в случае их обнаружения и сообщить операционной системе, что процесс ввода-вывода завершен. Когда программа обработки прерывания завершена, процессор возобновляет работу программы, которая была приостановлена в момент прерывания.

Шина используется не только контроллерами ввода-вывода, но и процессором для передачи команд и данных. А что происходит, если процессор и контроллер ввода-вывода хотят получить доступ к шине одновременно? В этом случае особая микросхема, которая называется **арбитром шины,** решает, чья очередь первая. Обычно предпочтение отдается устройствам ввода-вывода, поскольку работу дисков и других движущихся устройств нельзя прерывать, так как это может привести к потере данных. Когда ни одно устройство ввода-вывода не функционирует, центральный процессор может полностью распоряжаться шиной для связи с па-

мятью. Однако если какое-нибудь устройство ввода-вывода находится в действии, оно будет запрашивать доступ к шине и получать его каждый раз, когда ему это необходимо. Такой процесс называется **занятием цикла памяти** и замедляет работу компьютера.

Такая система успешно использовалась в первых персональных компьютерах, поскольку все их компоненты работали примерно с одинаковой скоростью. Однако как только центральные процессоры, память и устройства ввода-вывода стали работать быстрее, возникла проблема: шина больше не могла справляться с такой нагрузкой. В случае с закрытыми системами, например рабочими станциями, решением данной проблемы стала разработка новой шины с более высокой скоростью передачи данных для следующей модели машины. Поскольку никто никогда не переносил устройства ввода-вывода со старой модели на новую, такой подход работал успешно.

И все же в мире персональных компьютеров многие заменяли процессор более усовершенствованным, но при этом хотели подсоединить свой старый принтер, сканер **и** модем **к** новой системе. Кроме того, существовала целая обширная отрасль промышленности, которая выпускала широкий спектр устройств ввода-вывода для шины IBM PC, и производители этих устройств были совершенно не заинтересованы в том, чтобы начинать все разработки заново. Компания IBM прошла этот тяжелый путь, выпустив после серии IBM PC серию PS/2. У PS/2 была новая шина с более высокой скоростью передачи данных, но большинство производителей клонов продолжали использовать старую шину PC, которая сейчас называется шиной **ISA (Industry Standard Architecture — стандартная промышленная архитектура).** Большинство производителей дисков и устройств ввода-вывода также продолжали выпускать контроллеры для старой модели, поэтому **IBM** оказалась в весьма неприятной ситуации, поскольку она в тот момент была единственным производителем персональных компьютеров, несовместимых с серией IBM. В конце концов компания была вынуждена вернуться к производству компьютеров на основе шины ISA. Отметим, что ISA также может быть сокращением от Instruction Set Architecture (архитектура набора команд), если речь идет об уровнях компьютера. А если речь идет о шинах, аббревиатура ISA означает Industry Standard Architecture (стандартная промышленная архитектура).

Тем не менее, несмотря на то, что из-за влияния рынка никаких изменений не произошло, старая шина работала слишком медленно, поэтому что-то нужно было предпринять. Данная ситуация привела к тому, что другие компании начали производить компьютеры с несколькими шинами, одна из которых была старой шиной ISA или **EISA (Extended ISA — расширенная архитектура промышленного стандарта).** EISA — последователь ISA, совместимый со старыми версиями. В настоящее время самой популярной из них является шина **PCI (Peripheral Component Interconnect — взаимодействие периферийных компонентов).** Она была разработана компанией Intel, при этом было решено сделать все патенты всеобщим достоянием, чтобы вся компьютерная промышленность (в том числе и конкуренты компании) могла перенять эту идею.

Существует много различных конфигураций шины PCI. Наиболее типичная из них показана на рис. 2.25. В такой конфигурации центральный процессор об-

щается с контроллером памяти по специальному средству связи с высокой скоростью передачи данных. Контроллер соединяется с памятью и шиной PCI непосредственно, и таким образом, передача данных между центральным процессором и памятью происходит не через шину PCI. Однако периферийные устройства с высокой скоростью передачи данных, например SCSI-диски, могут подсоединяться прямо к шине PCI. Кроме того, шина PCI имеет параллельное соединение с шиной ISA, чтобы можно было использовать контроллеры ISA и соответствующие устройства. Машина такого типа обычно содержит 3 или 4 пустых разъема PCI и еще 3 или 4 пустых разъема ISA, чтобы покупатели имели возможность вставлять и старые карты ввода-вывода ISA (для медленно работающих устройств), и новые карты PCI (для устройств с высокой скоростью работы[1]).

В настоящее время существует много разных видов устройств ввода-вывода. Некоторые наиболее распространенные из них описываются ниже.

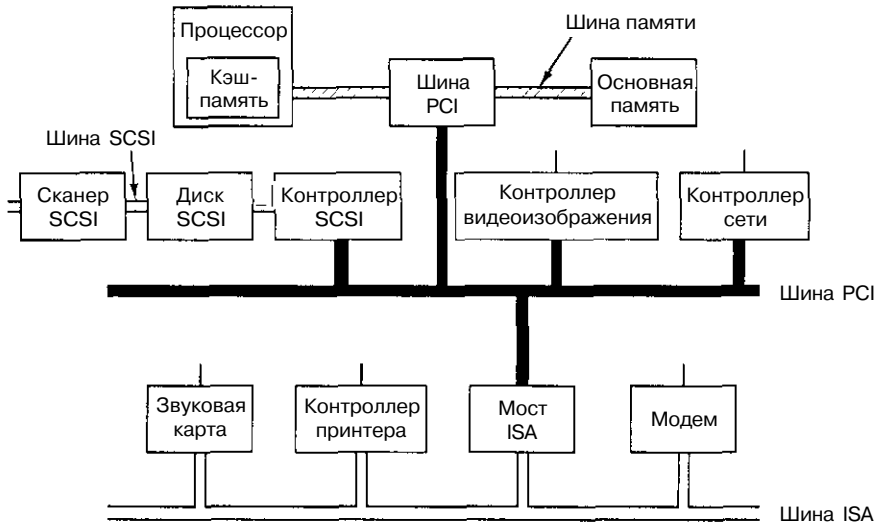


Рис. 2.25. Обычный современный персональный компьютер с шиной PCI и шиной ISA. Модем и звуковая карта — устройства ISA; SCSI-контроллер — устройство PCI

Терминалы

Терминалы компьютера состоят из двух частей: клавиатуры и монитора. В больших компьютерах эти части соединены в одно устройство и связаны с самим компьютером обычным или телефонным проводом. В авиакомпаниях, банках и различных отраслях промышленности, где работают с такими компьютерами, эти устройства до сих пор широко применимы. В мире персональных компьютеров клавиатура и монитор — независимые устройства. Но и в том и в другом случае технология этих двух частей одна и та же.

[1] Необходимо отметить, что в настоящее время существующие стандарты на персональный компьютер уже не содержат шину ISA. — *Примеч. научн. ред.*

Клавиатуры

Существует несколько видов клавиатур. У первых компьютеров IBM PC под каждой клавишей находился переключатель, который давал ощутимую отдачу и щелкал при нажатии клавиши. Сегодня у самых дешевых клавиатур при нажатии клавиш происходит лишь механический контакт с печатной платой. У клавиатур получше между клавишами и печатной платой кладется слой из эластичного материала (особого типа резины). Под каждой клавишей находится небольшой купол, который прогибается в случае нажатия клавиши. Проводящий материал, находящийся внутри купола, замыкает схему. У некоторых клавиатур под каждой клавишей находится магнит, который при нажатии клавиши проходит через катушку и таким образом вызывает электрический ток. Также используются другие методы, как механические, так и электромагнитные.

В персональных компьютерах при нажатии клавиши происходит процедура прерывания и запускается программа обработки прерывания (эта программа является частью операционной системы). Программа обработки прерывания считывает регистр аппаратного обеспечения в контроллер клавиатуры, чтобы получить номер клавиши, которая была нажата (от 1 до 102). Когда клавишу отпускают, происходит второе прерывание. Так, если пользователь нажимает клавишу **SHIFT,** затем нажимает и отпускает клавишу «M>, а затем отпускает клавишу SHIFT, операционная система понимает, что ему нужна заглавная, а не строчная буква «M». Обработка совокупности клавиш SHIFT, CTRL и ALT совершается только программным обеспечением (сюда же относится известное сочетание клавиш CTRL-ALT-DEL, которое используется для перезагрузки всех компьютеров IBM PC и их клонов).

Мониторы с электронно-лучевой трубкой

Монитор представляет собой коробку, содержащую **электронно-лучевую трубку** и ее источники питания. Электронно-лучевая трубка включает в себя электронную пушку, которая выстреливает пучок электронов на фосфоресцентный экран в передней части трубки, как показано на рис. 2.26, *а.* (Цветные мониторы содержат три электронные пушки: одну для красного, вторую для зеленого и третью для синего цвета.) При горизонтальной развертке пучок электронов (луч) развертывается по экрану примерно за 50 мкс, образуя почти горизонтальную полосу на экране. Затем луч совершает горизонтальный обратный ход к левому краю, чтобы начать следующую развертку. Устройство, которое так, линия за линией, создает изображение, называется устройством **растровой развертки.**

Горизонтальная развертка контролируется линейно возрастающим напряжением, которое воздействует на пластины горизонтального отклонения, расположенные слева и справа от электронной пушки. Вертикальная развертка контролируется более медленно возрастающим напряжением, которое воздействует на пластины вертикального отклонения, расположенные под и над электронной пушкой. После определенного количества разверток (от 400 до 1000) напряжение на пластинах вертикального и горизонтального отклонения спадает, и луч возвращается в верхний левый угол экрана. Полное изображение возобновляется от 30 до 60 раз в секунду[1]. Движения луча показаны на рис. 2.26, *6.* Хотя мы описали работу элект-

[1] Современные электрон но-лучевые мониторы могут иметь рефреш (частоту обновления изображения, вычерчиваемого лучом на экране) до 150 и более раз в секунду. Эта частота, естественно, обратно пропорционально зависит от количества строк, из которых строится изображение. — *Примеч. научн. ред.*

ронно-лучевых трубок, в которых для развертки луча по экрану используются электрические поля, во многих моделях вместо электрических используются магнитные поля (особенно в дорогостоящих мониторах).

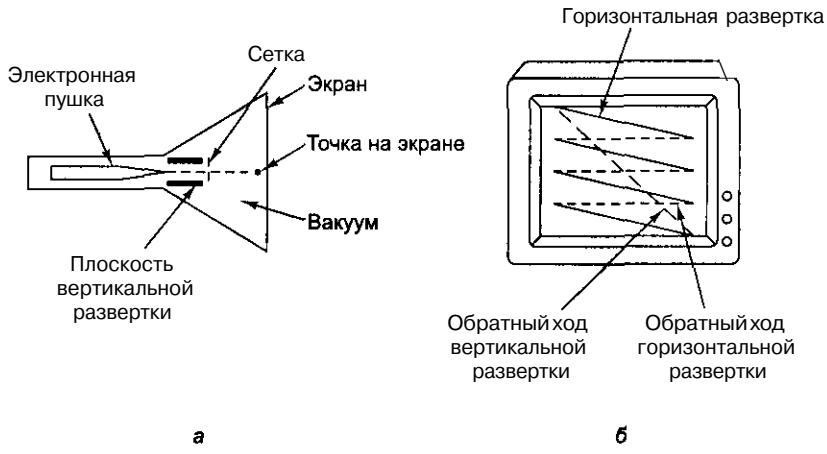


Рис. 2.26. Поперечное сечение электронно-лучевой трубки (а); схема развертки электронно-лучевой трубки (б)

Для получения на экране изображения из точек внутри электронно-лучевой трубки находится сетка. Когда на сетку воздействует положительное напряжение, электроны возбуждаются, луч направляется на экран, который через некоторое время начинает светиться. Когда используется отрицательное напряжение, электроны отталкиваются и не проходят через сетку, и экран не зажигается. Таким образом напряжение, воздействующее на сетку, вызывает появление соответствующего набора битов на экране. Такой механизм позволяет переводить двоичный электрический сигнал на дисплей, состоящий из ярких и темных точек.

Жидкокристаллические мониторы

Электронно-лучевые трубки слишком громоздкие и тяжелые для использования в портативных компьютерах, поэтому для таких экранов необходима совершенно другая технология. В таких случаях чаще всего используются **жидкокристаллические дисплеи.** Эта технология чрезвычайно сложна, имеет несколько вариантов воплощения и быстро меняется, поэтому мы из необходимости сделаем ее описание по возможности кратким и простым.

Жидкие кристаллы представляют собой вязкие органические молекулы, которые двигаются, как молекулы жидкостей, но при этом имеют структуру, как у кристалла. Они были открыты австрийским ботаником Рейницером (Rheinitzer) в 1888 году и впервые стали применяться при изготовлении различных дисплеев (для калькуляторов, часов и т. п.) в 1960 году. Когда молекулы расположены в одну линию, оптические качества кристалла зависят от направления и поляризации воздействующего света. При использовании электрического поля линия молекул, а следовательно, и оптические свойства могут изменяться. Если воздействовать лучом света на жидкий кристалл, интенсивность света, исходящего из самого жидкого

кристалла, может контролироваться с помощью электричества. Это свойство используется при создании индикаторных дисплеев.

Экран жидкокристаллического дисплея состоит из двух стеклянных параллельно расположенных пластин, между которыми находится герметичное пространство с жидким кристаллом. К обеим пластинам подсоединяются прозрачные электроды. Искусственный или естественный свет за задней пластиной освещает экран изнутри. Электроды, подведенные к пластинам, используются для того, чтобы создать электрические поля в жидком кристалле. На различные части экрана воздействует разное напряжение, и таким образом можно контролировать изображение. К передней и задней пластинам экрана приклеиваются поляроиды, поскольку технология дисплея требует использования поляризованного света. Общая структура показана на рис. 2.27, *а*.

В настоящее время используются различные типы жидкокристаллических дисплеев, но мы рассмотрим только один из них — **дисплей со скрученным нематиком.** В этом дисплее на задней пластине находятся крошечные горизонтальные желобки, а на передней — крошечные вертикальные желобки, как показано на рис. 2.27, *б*. При отсутствии электрического поля молекулы направляются к этим желобкам. Так как они (желобки) расположены перпендикулярно друг к другу, молекулы жидкого кристалла оказываются скрученными на 90°.

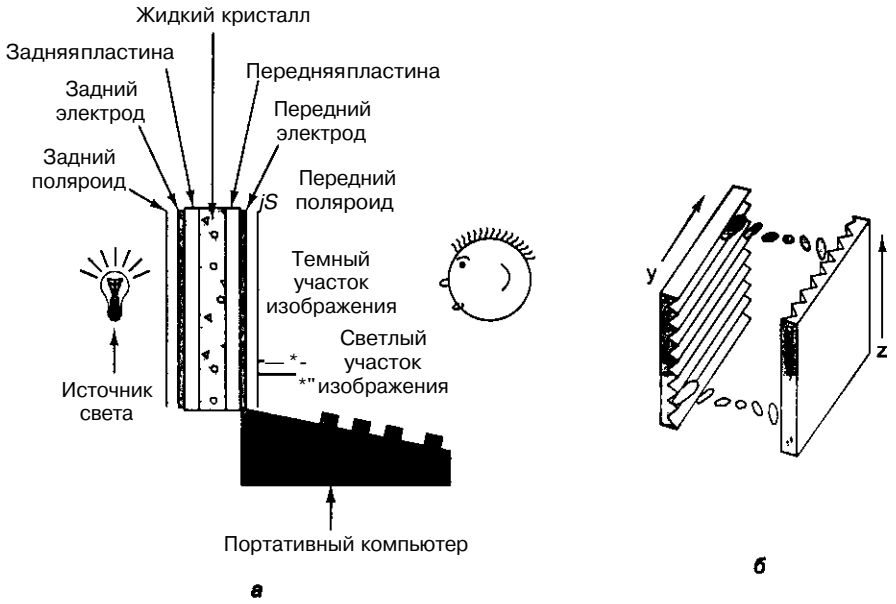


Рис. 2.27. Структура экрана на жидких кристаллах (а); желобки на передней и задней пластинах, расположенные перпендикулярно друг к другу (б)

На задней пластине дисплея находится горизонтальный поляроид. Он пропускает только горизонтально поляризованный свет. На передней пластине дисплея находится вертикальный поляроид. Он пропускает только вертикально поляризованный свет. Если бы между пластинами не было жидкого кристалла, горизон-

тально поляризованный свет, пропущенный поляроидом на задней пластине, блокировался бы поляроидом на передней пластине, что делало бы экран полностью черным.

Однако скрученная кристаллическая структура молекул, сквозь которую проходит свет, разворачивает плоскость поляризации света. При отсутствии электрического поля жидкокристаллический экран будет полностью освещен. Если подавать напряжение к определенным частям пластины, скрученная структура разрушается, блокируя прохождение света в этих частях.

Для подачи напряжения обычно используются два подхода. В дешевом **пассивном матричном индикаторе** оба электрода содержат параллельные провода. Например, на дисплее размером 640x480 электрод задней пластины содержит 640 вертикальных проводов, а электрод передней пластины — 480 горизонтальных проводов. Если подавать напряжение на один из вертикальных проводов, а затем посылать импульсы на один из горизонтальных, можно изменить напряжение в определенной позиции пиксела и, таким образом, сделать нужную точку темной. Если то же самое повторить со следующим пикселом и т. д., можно получить темную полосу развертки, аналогичную полосам в электронно-лучевых трубках. Обычно изображение на экране перерисовывается 60 раз в секунду, чтобы создавалось впечатление постоянной картинки (так же, как в электронно-лучевых трубках).

Второй подход — применение **активного матричного индикатора.** Он стоит гораздо дороже, чем пассивный матричный индикатор, но зато дает изображение лучшего качества, что является большим преимуществом. Вместо двух наборов перпендикулярно расположенных проводов у активного матричного индикатора имеется крошечный элемент переключения в каждой позиции пиксела на одном из электродов. Меняя состояние переключателей, можно создавать на экране произвольную комбинацию напряжений в зависимости от комбинации битов.

До сих пор мы описывали, как работают монохромные мониторы. Достаточно сказать, что цветные мониторы работают на основе тех же общих принципов, что и монохромные, но детали гораздо сложнее. Чтобы разделить белый цвет на красный, зеленый и синий, в каждой позиции пиксела используются оптические фильтры, поэтому эти цвета могут отображаться независимо друг от друга. Из сочетания этих трех основных цветов можно получить любой цвет.

Символьные терминалы

Обычно используются три типа терминалов: символьные терминалы, графические терминалы и терминалы RS-232-C. Все эти терминалы в качестве входных данных получают набор с клавиатуры, но при этом они отличаются друг от друга тем, каким образом компьютер обменивается с ними информацией, и тем, каким образом передаются выходные данные. Ниже мы кратко опишем каждый из этих типов.

В персональном компьютере существует два способа вывода информации на экран: символьный и графический. На рис. 2.28 показано, как происходит символьное отображение информации на экране (клавиатура считается отдельным устройством). На серийной плате связи находится область памяти, которая называется **видеопамятью,** а также несколько электронных устройств для получения доступа к шине и генерирования видеосигналов.

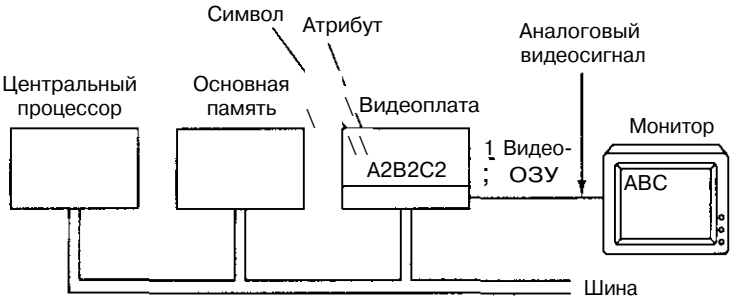


Рис. 2.28. Схема получения выходного сигнала на экране персонального компьютера

Чтобы отобразить на экране символы, центральный процессор копирует их в видеопамять в виде байтов. С каждым символом связывается атрибутивный байт, который описывает, какой именно символ должен быть изображен на экране. Атрибуты могут содержать указания на цвет символа, его интенсивность, а также на то, мигает он или нет. Таким образом, изображение 25x80 символов требует наличия 4000 байтов видеопамяти (2000 для символов и 2000 для атрибутов). Большинство плат содержат больше памяти для хранения нескольких изображений.

Видеоплата должна время от времени посылать символы из видео-ОЗУ и порождать необходимый сигнал, чтобы приводить в действие монитор. За один раз посылается целая строка символов, поэтому можно вычислять отдельные строки развертки. Этот сигнал является аналоговым сигналом с высокой частотой, и он контролирует развертку электронного луча, который рисует символы на экране. Так как выходными данными платы является видеосигнал, монитор должен находиться не дальше, чем в нескольких метрах от компьютера, чтобы предотвратить искажение.

Графические терминалы

При втором способе вывода информации на экран видеопамять рассматривается не как массив символов 25x80, а как массив элементов изображения, которые называются **пикселами.** Каждый пиксел может быть включен или выключен. Он представляет один элемент информации. В персональных компьютерах монитор может содержать 640x480 пикселов, но чаще используются мониторы 800x600 и более. Мониторы рабочих станций обычно содержат 1280x960 пикселов и более. Терминалы, отображающие биты, а не символы, называются **графическими терминалами.** Все современные видеоплаты могут работать или как символьные, или как графические терминалы под контролем программного обеспечения.

Основная идея работы терминала показана на рис. 2.28. Однако в случае с графическим изображением видео-ОЗУ рассматривается как большой массив битов. Программное обеспечение может задавать любую комбинацию битов, и она сразу же будет отображаться на экране. Чтобы нарисовать символы, программное обеспечение может, например, назначить для каждого символа прямоугольник 9x14 и заполнять его необходимыми битами. Такой подход позволяет программному обеспечению создавать разнообразные шрифты и сочетать их по желанию. Аппаратное обеспечение только отображает на экране массив битов. Для цветных мониторов каждый пиксел содержит 8, 16 или 24 бита.

Графические терминалы обычно используются для поддержки мониторов, содержащих несколько окон. **Окном** называется область экрана, используемая одной программой. Если одновременно работает несколько программ, на экране появляется несколько окон, при этом каждая программа отображает результаты независимо от других программ.

Хотя графические терминалы универсальны, у них есть два больших недостатка. Во-первых, они требуют большого объема видео-ОЗУ. В настоящее время обычно используются мониторы 640x480 (VGA), 800x600 (SVGA), 1024x768 (XVGA) и 1280x960. Отметим, что у всех этих мониторов отношение ширины и высоты 4:3, что соответствует соотношению сторон телевизионных экранов. Чтобы получить цвет, необходимо 8 битов для каждого из трех основных цветов, или 3 байта на пиксел. Следовательно, для монитора 1024x768 требуется 2,3 Мбайт видео-ОЗУ.

Из-за требования такого большого объема памяти приходится идти на компромисс. При этом для указания цвета используется 8-битный номер. Этот номер является индексом таблицы аппаратного обеспечения, которая называется цветовой **палитрой** и включает в себя 256 разделов, каждый из которых содержит 24 бита. Биты указывают на сочетание красного, зеленого и синего цветов. Такой подход, называемый **индексацией цветов,** сокращает необходимый объем видео-ОЗУ на 2/3, но допускает только 256 цветов. Обычно каждое окно на экране отображается отдельно, но при этом используется только одна цветовая палитра. К тому же, когда на экране присутствуют несколько окон, правильно передаются цвета только одного из них.

Второй недостаток графических терминалов — низкая производительность. Поскольку программисты осознали, что они могут управлять каждым пикселом во времени и пространстве, они, естественно, хотят осуществить эту возможность. Хотя данные могут копироваться из видео-ОЗУ на монитор без прохождения через главную шину, при доставке данных в видео-ОЗУ без использования шины не обойтись. Чтобы отобразить цветное изображение на полный экран размером 1024x768, необходимо копировать 2,3 Мбайт данных в видео-ОЗУ для каждого кадра. Для двигающегося видеоизображения должно сменяться по крайней мере 25 кадров в секунду, а скорость передачи данных должна составлять 57,6 Мбайт/с. Шина (E)ISA не может выдержать такую нагрузку, поэтому необходимо использовать видеокарты PCI, но даже в этом случае приходится идти на компромисс.

Еще одна проблема, связанная с производительностью, — как прокручивать экран. Можно скопировать все биты в программное обеспечение, но это очень сильно перегрузит центральный процессор. Не удивительно, что многие видеокарты оснащены специальным аппаратным обеспечением, которое двигает части экрана не с помощью копирования битов, а путем изменения базовых регистров.

Терминалы RS-232-C

Одни компании производят компьютеры, а другие выпускают терминалы (особенно для больших компьютеров). Чтобы (почти) любой терминал мог работать с (почти) любым компьютером, Ассоциация стандартов в электронной промышленности разработала стандартный интерфейс для терминалов под названием RS-232-C.

Терминалы RS-232-C содержат стандартизированный разъем с 25 выводами. Стандарт RS-232-C определяет размер и форму разъема, уровни напряжения и значение сигнала на каждом выводе.

Если компьютер и терминал разделены, чаще всего их можно соединить только по телефонной сети. К несчастью, телефонная сеть не может передавать сигналы, требуемые стандартом RS-232-C, поэтому для преобразования сигнала между компьютером и телефоном, а также между терминалом и телефоном помещается устройство, называемое модемом (модулятор-демодулятор). Ниже мы кратко рассмотрим устройство модемов.

На рис. 2.29 показано расположение компьютера, модемов и терминала при использовании телефонной линии. Если терминал находится достаточно близко от компьютера, так, что их можно связать обычным проводом, модемы не подсоединяются, но в этом случае используются те же кабели и разъемы RS-232-C, хотя выводы, связанные с модемом, не нужны.

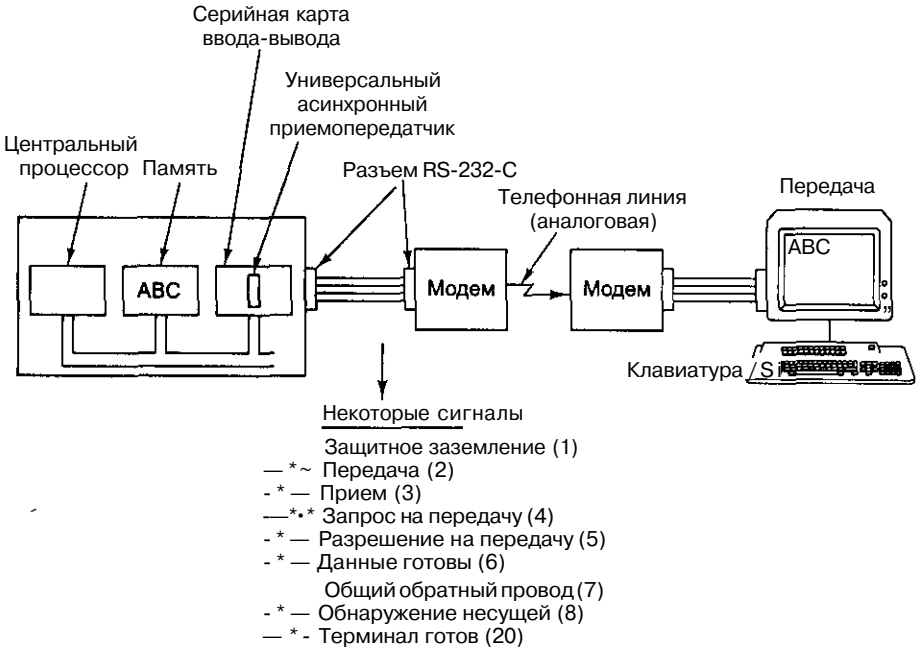


Рис. 2.29. Соединение терминала RS-232-C с компьютером. В списке сигналов в скобках указаны номера выводов

Чтобы обмениваться информацией, и компьютер, и терминал должны содержать микросхему UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик), а также логическую схему для доступа к шине. Чтобы отобразить на экране символ, компьютер вызывает этот символ из основной памяти и передает его UART, который затем отправляет его по кабелю RS-232-C бит за битом. В UART поступает сразу целый символ (1 байт), который преобразуется в последовательность битов, и они передаются один за другим с определенной скоростью. UART добавляет к каждому символу начальный и конечный биты, чтобы отделить один символ от другого. При скорости передачи 110 бит/с используется 2 конечных бита.

В терминале другой **UART** получает **биты и** восстанавливает целый символ, который затем отображается на экране. Входная информация, которая поступает с клавиатуры терминала, преобразуется в терминале из целых символов в последовательность битов, а затем UART в компьютере восстанавливает целые символы.

Стандарт RS-232-C определяет около 25 сигналов, но на практике используются только некоторые из них (большинство из которых может опускаться, если терминал непосредственно соединен с компьютером проводом, без модемов). Штыри 2 и 3 предназначены для отправки и получения данных соответственно. По каждому выводу проходит односторонний поток битов (один в одном направлении, а другой в противоположном). Когда терминал или компьютер включен, он выдает сигнал готовности терминала (то есть устанавливает 1), чтобы сообщить модему, что он включен. Сходным образом модем выдает сигнал готовности набора данных, чтобы сообщить об их наличии. Когда терминалу или компьютеру нужно послать данные, он выдает сигнал запроса о разрешении пересылки. Если модем разрешает пересылку, он должен выдать сигнал о том, что путь для пересылки свободен. Другие выводы выполняют различные функции определения состояний, проверки и синхронизации.

Мыши

Время идет, а люди работают за компьютером, все меньше и меньше вдаваясь в принципы его работы. Компьютеры серии ENIAC использовались только теми, кто их конструировал. В 50-е годы с компьютерами работали только высококвалифицированные программисты. Сейчас за компьютерами работают многие люди, при этом они не знают (или даже не хотят знать), как работают компьютеры и как они программируются.

Много лет назад у большинства компьютеров был интерфейс с командной строкой, в которой набирались различные команды. Поскольку многие неспециалисты считали такие интерфейсы недружелюбными или даже враждебными, компьютерные фирмы разработали специальные интерфейсы с возможностью указания на экран. Для создания такой возможности чаще всего используется мышь.

Мышь — это маленькая пластиковая коробка, которая лежит на столе рядом с клавиатурой. Если ее двигать по столу, курсор на экране тоже будет двигаться, позволяя пользователям указывать на элементы экрана. У мыши есть одна, две **или три** кнопки, нажатие на которые дает возможность пользователям выбирать строки меню. Было очень много споров по поводу того, сколько кнопок должно быть у мыши. Наивные пользователи предпочитали одну (так как в этом случае невозможно нажать не ту кнопку), но более продвинутые предпочитали несколько кнопок, чтобы можно было на экране выполнять сложные действия.

Существует три типа мышей: механические, оптические и оптомеханические. У мышей первого типа снизу торчат резиновые колесики, оси которых расположены перпендикулярно друг к другу. Если мышь передвигается в вертикальном направлении, то вращается одно колесо, а если в горизонтальном, то другое. Каждое колесико приводит в действие резистор (потенциометр). Если измерить изменения сопротивления, можно узнать, на сколько провернулось колесико, и таким образом вычислить, на какое расстояние передвинулась мышь в каждом направле-

нии. В последние годы такие мыши были практически полностью вытеснены новой моделью, в которой вместо колес используется шарик, который слегка высовывается снизу. Такая мышь изображена на рис. 2.30.

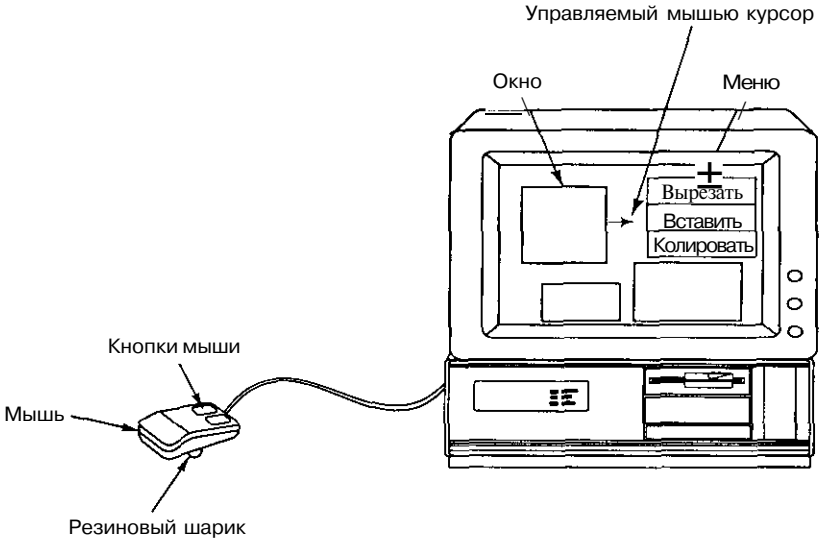


Рис. 2.30. Использование мыши для указания на строки меню

Следующий тип — оптическая мышь. У нее нет ни колес, ни шарика. Вместо этого используются светодиод и фотодетектор, расположенный в нижней части мыши. Оптическая мышь перемещается по поверхности особого пластикового коврика, который содержит прямоугольную решетку с линиями, близко расположенными друг к другу. Когда мышь двигается по решетке, фотодетектор воспринимает пересечения линий, наблюдая изменения в количестве света, отражаемого от светодиода. Электронное устройство внутри мыши подсчитывает количество пересеченных линий в каждом направлении.

Третий тип — оптомеханическая мышь. У нее, как и у более современной механической мыши, есть шарик, который вращает два вывода, расположенных перпендикулярно друг к другу. Выводы связаны с кодировщиками. В каждом кодировщике имеются прорези, через которые проходит свет. Когда мышь двигается, выводы вращаются и световые импульсы воздействуют на детекторы каждый раз, когда между светодиодом и детектором появляется прорезь. Число воспринятых детектором импульсов пропорционально количеству перемещения.

Хотя мыши можно устанавливать по-разному, обычно используется следующая система: компьютеру передается последовательность из 3 байтов каждый раз, когда мышь проходит определенное минимальное расстояние (например, 0,01 дюйма). Обычно эти характеристики передаются в последовательном потоке битов. Первый байт содержит целое число, которое указывает, на какое расстояние переместилась мышь в направлении x с прошлого раза. Второй байт содержит ту же информацию для направления y. Третий байт указывает на текущее состояние кнопок мыши. Иногда для каждой координаты используются два байта.

Программное обеспечение принимает эту информацию по мере поступления и преобразует относительные движения, передаваемые мышью, в абсолютную позицию. Затем оно отображает стрелочку на экране в позиции, соответствующей расположению мыши. Если указать стрелочкой на определенный элемент экрана и щелкнуть кнопкой мыши, компьютер может вычислить, какой именно элемент компьютерной информации, соответствующий данному элементу на экране, был выбран.

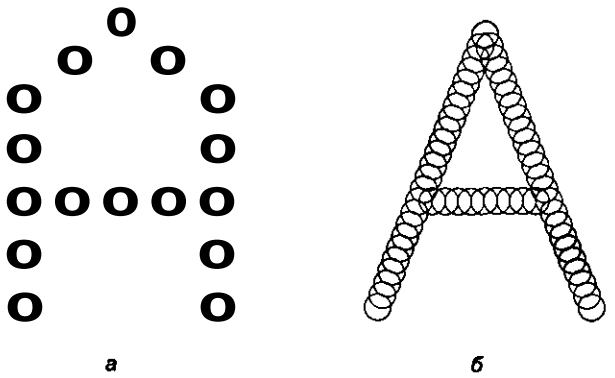
Принтеры

Иногда пользователю нужно напечатать созданный документ или страницу, полученную из World Wide Web, поэтому компьютеры могут быть оснащены принтером. В этом разделе мы опишем некоторые наиболее распространенные типы монохромных (то есть черно-белых) и цветных принтеров.

Монохромные принтеры

Самыми дешевыми являются **матричные принтеры,** у которых печатающая головка последовательно проходит каждую строку печати. Головка содержит от 7 до 24 игл, возбуждаемых электромагнитным полем. Дешевые матричные принтеры имеют 7 игл для печати, скажем, 80 символов в строке в матрице 5x7. В результате строка печати состоит из 7 горизонтальных линий, а каждая из этих линий состоит из 5x80=400 точек. Каждая точка может печататься или не печататься в зависимости от того, какая нужна буква. На рис. 2.31, *а* показана буква «A», напечатанная на матрице 5x7.

Качество печати можно повышать двумя способами: использовать большее количество игл и создавать наложение точек. На рис. 2.31, *б* показана буква «A», напечатанная с использованием 24 игл, в результате чего получилось пересечение точек. Для получения таких пересечений обычно требуется несколько проходов по одной строке печати, поэтому чем выше качество печати, тем медленнее работает принтер. Большинство принтеров можно настраивать, создавая различные варианты соотношения качества и скорости.



а *б*

Рис. 2.31. Буква «A» на матрице 5x7 (а); буква «A», напечатанная с использованием 24 игл. Получается наложение точек (б)

Матричные принтеры дешевы (особенно в отношении расходных материалов) и очень надежны, но работают медленно, шумно, и качество печати очень низкое. Однако они широко применимы, по крайней мере, в трех областях. Во-первых, они очень популярны для печати на больших листах (более 30 см). Во-вторых, ими очень удобно пользоваться при печати на маленьких отрезках бумаги (например, кассовых чеках, уведомлениях о снятии денег с кредитных карт, посадочных талонах в авиакомпаниях). В-третьих, они используются для распечатывания одновременно нескольких листов с вложенной между ними копировальной бумагой, и эта технология самая дешевая.

Дома удобно использовать недорогие **струйные принтеры.** Подвижная печатающая головка содержит картридж с чернилами. Она двигается горизонтально над бумагой, а чернила в это время выпрыскиваются из крошечных выпускных отверстий. Внутри каждого отверстия капля чернил нагревается до критической точки и в конце концов вырывается наружу Единственное место, куда она может попасть из отверстия, — лист бумаги. Затем выпускное отверстие охлаждается, в результате создается вакуум, который втягивает следующую каплю. Скорость работы принтера зависит от того, насколько быстро повторяется цикл нагревания/охлаждения. Струйные принтеры обычно имеют разрешающую способность от 300 dpi (dots per inch — точек на дюйм) до 720 dpi, хотя существуют струйные принтеры с разрешающей способностью 1440 dpi. Они достаточно дешево стоят, работают бесшумно и дают хорошее качество печати, однако отличаются низкой скоростью, используют очень дорогие картриджи и производят распечатки, смоченные чернилами.

Вероятно, самым удивительным изобретением в области печатных технологий со времен Иоганна Гутенберга, который изобрел подвижную литеру в XV веке, является **лазерный принтер.** Это устройство сочетает хорошее качество печати, универсальность, высокую скорость работы и умеренную стоимость. В лазерных принтерах используется почти такая же технология, как в фотокопировальных устройствах. Многие компании производят устройства, совмещающие свойства копировальной машины, принтера и иногда также факса.

Основное устройство принтера показано на рис. 2.32. Главной частью этого принтера является вращающийся барабан (в некоторых более дорогостоящих системах вместо барабана используется лента). Перед печатью каждого листа барабан получает напряжение около 1000 вольт и окружается фоточувствительным материалом. Свет лазера проходит вдоль барабана (по длине) почти как пучок электронов в электронно-лучевой трубке, только вместо напряжения для сканирования барабана используется вращающееся восьмиугольное зеркало. Луч света модулируется, и в результате получается определенный набор темных и светлых участков. Участки, на которые воздействует луч, теряют свой электрический заряд.

После того как нарисована строка точек, барабан немного поворачивается для создания следующей строки. В итоге первая строка точек достигает резервуара с тонером (электростатически чувствительным черным порошком). Тонер притягивается к тем точкам, которые заряжены, и так формируется визуальное изображение строки. Через некоторое время барабан с тонером прижимается к бумаге, оставляя на ней отпечаток изображения. Затем лист проходит через горячие валики, и изображение закрепляется. После этого барабан разряжается, и остатки тонера счищаются с него. Теперь он готов к печатанию следующей страницы.

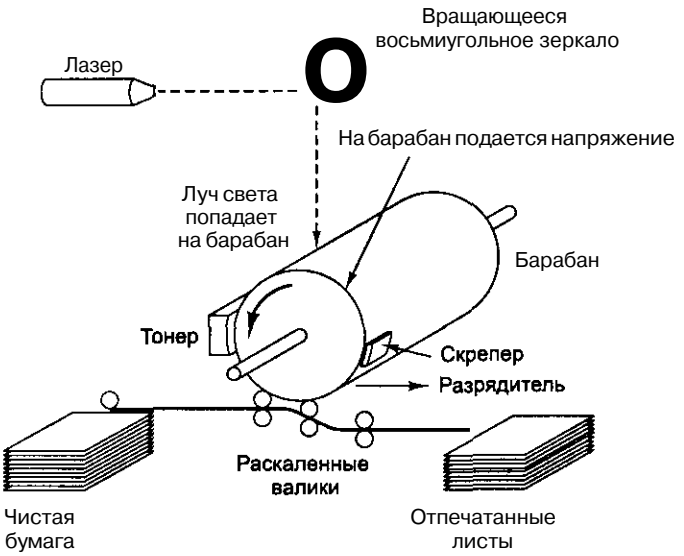


Рис. 2.32. Работа лазерного принтера

Едва ли нужно говорить, что этот процесс представляет собой чрезвычайно сложную комбинацию физики, химии, механики и оптики. Тем не менее некоторые производители выпускают агрегаты, которые называются **печатающими** устройствами. Изготовители лазерных принтеров сочетают печатающие устройства с их собственным аппаратным и программным обеспечением. Аппаратное обеспечение состоит из быстроработающего процессора, а также нескольких мегабайтов памяти для хранения полного изображения в битовой форме и различных шрифтов, одни из которых встроены, а другие загружаются из памяти. Большинство принтеров принимают команды, описывающие страницу, которую нужно напечатать (в противоположность принтерам, принимающим изображения в битовой форме от центрального процессора). Эти команды обычно даются на языке PCL или PostScript.

Лазерные принтеры с разрешающей способностью 300 dpi и выше могут печатать черно-белые фотографии, но технология при этом гораздо сложнее, чем может показаться на первый взгляд. Рассмотрим фотографию, отсканированную с разрешающей способностью 600 dpi, которую нужно напечатать на принтере с такой же разрешающей способностью (600 dpi). Сканированное изображение содержит 600x600 пикселов/дюйм, каждый пиксел характеризуется определенной степенью серого цвета от 0 (белый цвет) до 255 (черный цвет). Принтер может печатать с разрешающей способностью 600 dpi, но каждый напечатанный пиксел может быть либо черного цвета (когда есть тонер), либо белого цвета (когда нет тонера). Степени серого печататься не могут.

Для печати таких изображений используется так называемая **обработка полутонов** (как при печати серийных плакатов). Изображение разбивается на ячейки, каждая 6x6 пикселов. Каждая ячейка может содержать от 0 до 36 черных пикселов. Человеческому глазу ячейка с большим количеством черных пикселов кажется

темнее, чем ячейка с небольшим количеством черных пикселов. Серые тона в диапазоне от 0 до 255 передаются следующим образом. Этот диапазон делится на 37 зон. Серые тона от 0 до 6 расположены в зоне 0, от 7 до 13 — в зоне 1 и т. д. (зона 36 немного меньше, чем другие, потому что 256 на 37 без остатка не делится). Когда встречаются тона зоны 0, ячейка оставляется белой, как показано на рис. 2.33, *а*. Тона зоны 1 передаются одним черным пикселом в ячейке. Тона зоны 2 — двумя пикселами в ячейке, как показано на рис. 2.33, *б*. Изображение серых тонов других зон показано на рис. 2 33, *в — е* Если фотография отсканирована с разрешающей способностью 600 dpi, после подобной обработки полутонов разрешающая способность напечатанного изображения снижается до 100 ячеек/дюйм. Данная разрешающая способность называется градацией полутонов и измеряется в **lpi** (lines per inch — количество строк на дюйм)

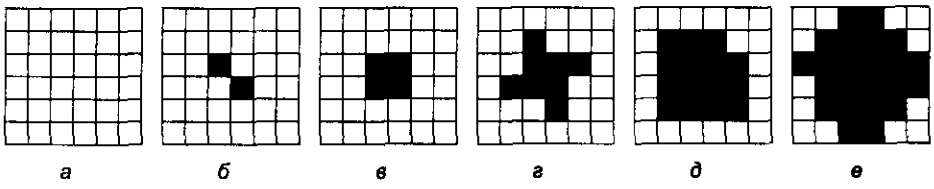


Рис. 2.33. Изображение серых полутонов различных зон 0-6 (а); 14-20 (б), 28-34 *(в)*, 56-62(г), 105-111 (д); 161-167 (е)

Цветныепринтеры

Цветные изображения могут передаваться двумя способами: с помощью поглощенного света и с помощью отраженного света. Поглощенный свет используется, например, при создании изображений в электронно-лучевых мониторах. В данном случае изображение строится путем аддитивного наложения трех основных цветов' красного, зеленого и синего. Отраженный свет используется при создании цветных фотографий и картинок в глянцевых журналах. В этом случае поглощается свет с определенной длиной волны, а остальной свет отражается Такие изображения создаются путем субтрактивного наложения трех основных цветов: голубого (красный полностью поглощен), желтого (синий полностью поглощен) и сиреневого (зеленый полностью поглощен). Теоретически путем смешивания голубых, желтых и сиреневых чернил можно получить любой цвет. Но на практике очень сложно получить такие чернила, которые полностью поглощали бы весь свет и в результате давали черный цвет. По этой причине практически во всех цветных печатающих устройствах используются чернила четырех цветов: голубого, желтого, сиреневого и черного. Такая система цветов называется CYMK (C — Cyan (голубой), Y — Yellow (желтый) M — Magenta (сиреневый) и K — ЫacK (черный)). Из слова «black» берется последняя буква, чтобы не путать с Blue (синий). Мониторы, напротив, используют поглощенный свет и наложение красного, зеленого и синего цветов для создания цветного изображения.

Полный набор цветов, который может производить монитор или принтер, называется цветовой шкалой Не существует такого устройства, которое полностью передавало бы цвета окружающего нас мира. В лучшем случае устройство дает всего

256 степеней интенсивности каждого цвета, и в итоге получается только 16 777 216 различных цветов. Несовершенство технологий еще больше сокращает это число, а оставшиеся цвета не передают полного цветового спектра. Кроме того, цветовосприятие связано не только с физическими свойствами света, но и с работой палочек и колбочек в сетчатке глаза.

Из всего этого следует, что превратить красивое цветное изображение, которое замечательно смотрится на экране, в идентичное печатное изображение очень сложно. Среди основных проблем можно назвать следующие:

1. Цветные мониторы используют поглощенный свет; цветные принтеры используют отраженный свет.
2. Электронно-лучевая трубка производит 256 оттенков каждого цвета; цветные принтеры должны совершать обработку полутонов.
3. Мониторы имеют темный фон; бумага имеет светлый фон.
4. Системы цветов RGB (Red, Green, Blue — красный, зеленый, синий) и CYMK отличаются друг от друга.

Чтобы цветные печатные изображения соответствовали реальной действительности (или хотя бы изображениям на экране), необходима калибровка оборудования, сложное программное обеспечение и компетентность пользователя.

Для цветной печати используются пять технологий, и все они основаны на системе CYMK. Самыми дешевыми являются **цветные струйные принтеры.** Они работают так же, как и монохромные струйные принтеры, но вместо одного картриджа в них находится четыре (для голубых, желтых, сиреневых и черных чернил). Они хорошо печатают цветную графику, сносно печатают фотографии и при этом не очень дорого стоят (отметим, что сами принтеры дешевые, а картриджи довольно дорогие).

Для получения лучших результатов должны использоваться особые чернила **и** особая бумага. Существует два вида чернил. **Чернила на** основе красителя состоят из красителей, растворенных в жидкой среде. Они дают яркие цвета и легко вытекают из картриджа. Главным недостатком таких чернил является то, что они быстро выгорают под воздействием ультрафиолетовых лучей, которые содержатся в солнечном свете. **Чернила на основе пигмента** содержат твердые частицы пигмента, погруженные в жидкость. Жидкость испаряется с бумаги, а пигмент остается. Чернила не выгорают, но зато дают не такие яркие краски, как чернила на основе красителя. Кроме того, частицы пигмента часто засоряют выпускные отверстия картриджей, поэтому их нужно периодически чистить. Для печати фотографий необходима мелованная или глянцевая бумага. Эти особые виды бумаги были созданы специально для того, чтобы удерживать капельки чернил и не давать им растекаться.

Следующий тип принтеров — **принтеры с твердыми чернилами.** В этих принтерах содержится 4 твердых блока специальных восковых чернил, которые затем расплавляются. Перед началом печати должно пройти 10 минут (время, необходимое для того, чтобы расплавить чернила). Горячие чернила выпрыскиваются на бумагу, где они затвердевают и закрепляются после прохождения листа между двумя валиками.

Третий тип цветных принтеров — **цветные лазерные принтеры.** Они работают так же, как их монохромные братья, только они составляют четыре отдельных изображения (голубого, желтого, сиреневого и черного цвета) и используют четыре разных тонера. Поскольку полное изображение в битовой форме обычно составляется заранее, для изображения с разрешающей способностью 1200x1200 dpi на листе в 80 квадратных дюймов нужно 115 млн пикселов. Так как каждый пиксел состоит из 4 битов, принтеру нужно 55 Мбайт памяти только для хранения изображения в битовой форме, не считая памяти, необходимой для внутренних процессоров, шрифтов и т. п. Это требование делает цветные лазерные принтеры очень дорогими, но зато они очень быстро работают и дают высокое качество печати. К тому же полученные изображения сохраняются на протяжении длительного времени.

Четвертый тип принтеров — **принтеры с восковыми чернилами.** Они содержат широкую ленту из четырехцветного воска, которая разделяется на отрезки размером с лист бумаги. Тысячи нагревательных элементов растапливают воск, когда бумага проходит под лентой. Воск закрепляется на бумаге в форме пикселов с использованием системы CYMK. Такие принтеры когда-то были очень популярны, но сейчас их вытеснили другие типы принтеров с более дешевыми расходными материалами.

Пятый тип принтеров работает на основе технологии **сублимации.** Это слово содержит некоторые фрейдистские нотки[1], однако в науке под сублимацией понимается переход твердых веществ в газообразные без прохождения через стадию жидкости. Таким материалом является, например, сухой лед (замороженный углекислый газ). В принтере, работающем на основе процесса сублимации, контейнер с красителями CYMK двигается над термической печатающей головкой, которая содержит тысячи программируемых нагревательных элементов. Красители мгновенно испаряются и впитываются специальной бумагой. Каждый нагревательный элемент может производить 256 различных температур. Чем выше температура, тем больше красителя осаждается и тем интенсивнее получается цвет. В отличие от всех других цветных принтеров, данный принтер способен воспроизводить цвета практически сплошного спектра, поэтому процедура обработки полутонов не нужна. Процесс сублимации часто используется при изготовлении моментальных снимков. Такие снимки делаются на специальной дорогостоящей бумаге.

Модемы

С появлением большого количества компьютеров в последние годы возникла необходимость установить связь между компьютерами. Например, можно связать свой домашний компьютер с компьютером на работе, с поставщиком услуг Интернета или банковской системой, Для обеспечения такой связи часто используется телефонная линия.

Однако грубая телефонная линия не подходит для передачи компьютерных сигналов, которые обычно передают 0 как 0 В, а 1 — от 3 до 5 В, как показано на рис. 2.34, *а.* Двухуровневые сигналы страдают от сильного искажения во время

[1] Сублимация в психологии означает психический процесс преобразования и переключения энергии влечений на цели социальной деятельности и культурного творчества; термин введен 3. Фрейдом. - *Примеч. перев.*

передачи по телефонной линии, которая предназначена для передачи голоса, а искажения ведут к ошибкам в передаче. Тем не менее синусоидальный сигнал с частотой от 1000 до 2000 Гц, который называется **несущим сигналом,** может передаваться с относительно небольшими искажениями, и это свойство используется при передаче данных в большинстве телекоммуникационных систем.

Поскольку синусоидальная волна полностью предсказуема, она не передает никакой информации. Однако изменяя амплитуду, частоту или фазу, можно передавать последовательность нулей и единиц, как показано на рис. 2.34. Этот процесс называется **модуляцией.** При **амплитудной модуляции** (рис 2.34, *б)* используется два уровня напряжения, для 0 и 1 соответственно. Если цифровые данные передаются с очень низкой скоростью, то при передаче 1 слышен громкий шум, а при передаче 0 шум отсутствует.

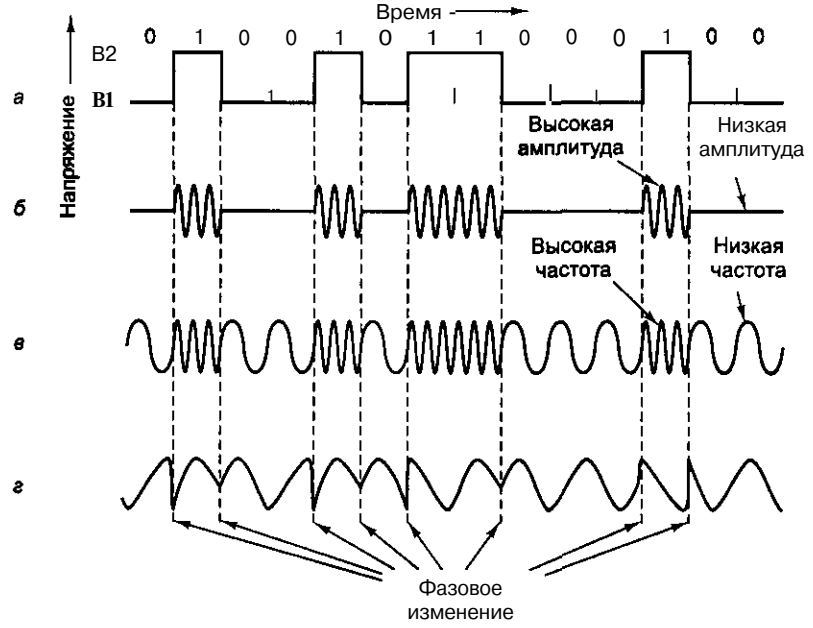


Рис. 2.34. Последовательная передача двоичного числа 01001011000100 по телефонной линии двухуровневый сигнал (а), амплитудная модуляция (б); частотная модуляция (в); фазовая модуляция (г)

При **частотной модуляции** (рис. 2.34, б) уровень напряжения не изменяется, но частота несущего сигнала различается для 1 и для 0. В этом случае при передаче цифровых данных можно услышать два тона: один из них соответствует 0, а другой — 1. Частотная модуляция иногда называется **частотной манипуляцией.**

При простой фазовой модуляции (рис. 2.34, г) амплитуда и частота сохраняются на одном уровне, а фаза несущего сигнала изменяется на 180 градусов, когда данные меняются с 0 на 1 или с 1 на 0. В более сложных системах фазовой модуляции в начале каждого неделимого временного отрезка фаза несущего сигнала резко сдвигается на 45,135, 225 или 315 градусов, чтобы передавать 2 бита за один временной отрезок. Это называется **дибитной фазовой кодировкой.** Например,

сдвиг по фазе на 45° представляет 00, сдвиг по фазе на 135° — 01 и т. д Существуют системы для передачи трех и более битов за один временной отрезок Число таких временных интервалов (то есть число потенциальных изменений сигнала в секунду) называется скоростью в бодах При передаче двух или более битов за 1 временной отрезок скорость передачи битов будет превышать скорость в бодах. Отметим, что термины «бод» и «бит» часто путают.

Если данные состоят из последовательности 8-битных символов, было бы желательно иметь средство связи для передачи 8 битов одновременно, то есть 8 пар проводов. Так как телефонные линии, предназначенные для передачи голоса, обеспечивают только один канал связи, биты должны пересылаться последовательно один за другим (или в группах по два, если используется дибитная кодировка). Устройство, которое получает символы из компьютера в форме двухуровневых сигналов (по одному биту в отрезок времени) и передает биты по одному или по два в форме амплитудной, фазовой или частотной модуляции, называется модемом. Для указания на начало и конец каждого символа в начале и конце 8-битной цепочки ставятся начальный и конечный биты, таким образом, всего получается 10 битов.

Модем посылает отдельные биты каждого символа через равные временные отрезки Например, скорость 9600 бод означает, что сигнал меняется каждые 104 мкс Второй модем, получающий информацию, преобразует модулированный несущий сигнал в двоичное число. Биты поступают в модем через равные промежутки времени Если модем определил начало символа, его часы сообщают, когда нужно начать считывать поступающие биты

Современные модемы передают данные со скоростью от 28 800 бит/с до 57 600 бит/с, что обычно соответствует более низкой скорости в бодах. Они сочетают несколько технологий для передачи нескольких битов за 1 бод, модулируя амплитуду, частоту и фазу. Почти все современные модемы являются **дуплексными,** то есть могут передавать информацию в обоих направлениях одновременно, используя различные частоты. Модемы и линии связи, которые не могут передавать информацию в обоих направлениях одновременно (как железная дорога, по которой поезда могут ходить и в северном, и в южном направлениях, но не в одно и то же время), называются **полудуплексными.** Линии связи, которые могут передавать информацию только в одном направлении, называются **симплексными.**

ISDN

В начале 80-х годов европейские компании почтовой, телефонной и телеграфной связи разработали стандарт цифровой телефонии под названием **ISDN (Integrated Services Digital Network — цифровая сеть с предоставлением комплексных** услуг). Она давала возможность горожанам иметь дома сигнализацию, связанную со специальными учреждениями, а также предназначалась для выполнения других своеобразных функций Компании настойчиво рекламировали эту идею, но без особого успеха Вдруг появился World Wide Web, и людям понадобился цифровой доступ к Интернету. Тут-то и обнаружилось совершенно потрясающее применение ISDN (хотя вовсе не благодаря разработчикам этой сети). С тех пор она стала очень популярной в США и других странах.

Когда клиент телефонной компании подписывается на ISDN, телефонная компания заменяет старую аналоговую линию новой цифровой. (В действительности сама линия не меняется, меняется только оборудование на обоих концах.) Новая

линия содержит два независимых цифровых канала, каждый со скоростью передачи данных 64 000 бит/с, плюс канал для сигналов со скоростью передачи 16 000 бит/с. Оборудование необходимо для того, чтобы объединить все три канала в один цифровой канал со скоростью передачи данных 144 000 бит/с. Предприятия могут приобретать 30-канальную линию ISDN.

ISDN не только быстрее передает данные, чем аналоговый канал, но и быстрее устанавливает соединение (не дольше 1 секунды), не требует наличия аналогового модема, а также более надежен, то есть дает меньше ошибок, чем аналоговый канал. Кроме того, ISDN имеет ряд дополнительных особенностей, которых нет у аналоговых каналов.

Структура связи ISDN показана на рис. 2.35. Поставщик предоставляет цифровой канал, который передает биты. Что означают эти биты — личное дело отправителя и получателя. Между оборудованием клиента и поставщика помещается устройство для взаимной связи NT1 с T-интерфейсом на одной стороне и U-интерфейсом на другой. В США клиенты должны покупать собственное устройство NT1, а во многих европейских странах — брать напрокат у поставщика.

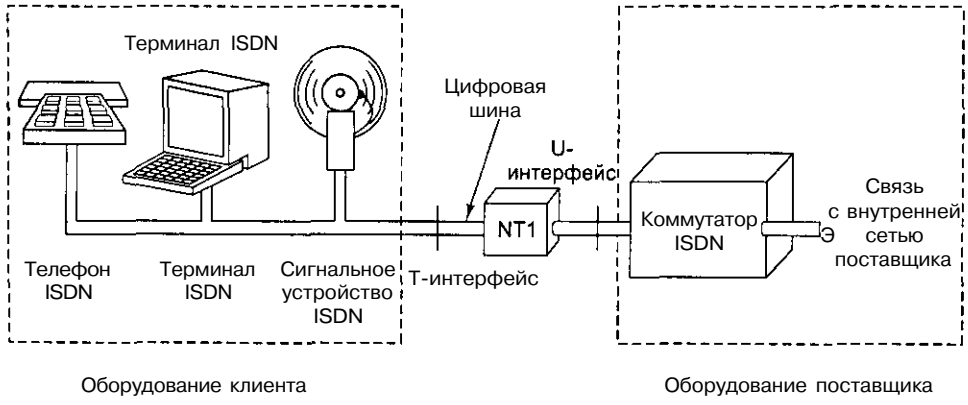


Рис. 2.35. ISDN для домашнего использования

Коды символов

У каждого компьютера есть набор символов, который он использует. Как минимум этот набор включает 26 заглавных и 26 строчных букв[1], цифры от 0 до 9, а также некоторые специальные символы: пробел, точка, запятая, минус, символ возврата каретки и т. д.

Для того чтобы передавать эти символы в компьютер, каждому из них приписывается номер: например, a=1, Ь=2,..., z=26, +=27, -=28. Отображение символов в целые числа называется кодом символов. Важно отметить, что связанные между собой компьютеры должны иметь один и тот же код, иначе они не смогут обмениваться информацией. По этой причине были разработаны стандарты. Ниже мы рассмотрим два самых важных из них.

[1] Для английского языка. — *Примеч. перев.*

ASCII

Один широко распространенный код называется **ASCII (American Standard Code for Information Interchange — американский стандартный код** для **обмена информацией)** Каждый символ ASCII-кода содержит 7 битов, таким образом, всего может быть 128 символов (табл 2 5) Коды от 0 до 1F (в шестиадцатеричной системе счисления) соответствуют управляющим символам, которые не печатаются

Многие непечатные символы ASCII предназначены для передачи данных Например, послание может состоять из символа начала заголовка SOH (Start of Header), самого заголовка, символа начала текста STX (Start of Text), самого текста, символа конца текста ETX (End of Text) и, наконец, символа конца передачи EOT (End of Transmission) Однако на практике послания, отправляемые по телефонным линиям и сетям, форматируются по-другому, так что непечатные символы передачи ASCII практически не используются

Печатные символы ASCII наглядны Они включают буквы верхнего и нижнего регистров, цифры, знаки пунктуации и некоторые математические символы.

Таблица 2.5. Таблица кодов ASCII

Число	Команда	Значение	Число	Команда	Значение
0	NUL	Null (Пустой указатель)	10	DLE	Data Link Escape (Выход из системы передачи)
1	SOH	Start of Heading {Начало заголовка)	11	DC1	Device Control 1 (Управление устройством)
2	STX	Start of Text (Начало текста)	12	DC2	Device Control 2 (Управление устройством)
3	ETX	End of Text (Конец текста)	13	DC3	Device Control 3 (Управление устройством)
4	EOT	End of Transmission (Конец передачи)	14	DC4	Device Control 4 (Управление устройством)
5	ENQ	ENQunty (Запрос)	15	NAK	Negative AcKnolidgement (Неподтверждение приема)
6	ACK	ACKnoligement (Подтверждение приема)	16	SYN	SYNcronous idle (Простой)
7	BEL	Bell (Символ звонка)	17	ETB	End of Transmission Block (Конец блока передачи)
8	BS	Backspace (Отступ назад)	18	CAN	CANcel (Отмена)
9	HT	Horizontal Tab (Горизонтальная табуляция)	19	EM	End of Medium (Конец носителя)
A	LF	Line Feed (Перевод строки)	1A	SUB	SUBstitute (Подстрочный индекс)
B	VT	Vertical Tab (Вертикальная табуляция)	1B	ESC	ESCape (Выход)

Число	Команда	Значение	Число	Команда	Значение
C	FF	From Feed (Перевод страницы)	1C	FS	File Separator (Разделитель файлов)
D	**CR**	Carnage **Return** (Возврат каретки)	1D	GS	Group Separator (Разделитель группы)
E	**SO**	Shift Out {Переключение на дополнительный регистр)	1E	RS	Record Separator (Разделитель записи)
=	**SI**	Shift In (Переключение настандартный регистр)	1F	US	Unit Separator (Разделитель модуля)

Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ
20	(пробел)	**30**	**0**	**40**	**@**	**50**	**P**	**60**	·	**70**	**P**
21	!	**31**	**1**	**41**	**A**	**51**	**Q**	**61**	a	**71**	**q**
22	·	**32**	**2**	**42**	**B**	**52**	**R**	**62**	**Ь**	**72**	**r**
23	#	**33**	**3**	**43**	**C**	**53**	**S**	**63**	**c**	**73**	s
24	Ф	**34**	**4**	**44**	**D**	**54**	**T**	**64**	**d**	**74**	**t**
25	%	**35**	**5**	**45**	**E**	**55**	**И**	**65**	e	**75**	и
26	&	**36**	**6**	**46**	**F**	**56**	V	**66**	**f**	**76**	V
27	·	**37**	**7**	**47**	**G**	**57**	W	**67**	**g**	**77**	W
28	(	**38**	**8**	**48**	**H**	**58**	X	**68**	**h**	**78**	x
29	)	**39**	**9**	**49**	**I**	**59**	Y	**69**	i	**79**	**y**
2A	·	**ЗА**	**;**	**4A**	**J**	**5A**	Z	**6A**	**J**	**7A**	z
2B	+	**ЗВ**	**;**	**4B**	**K**	**5B**	[	**6B**	**k**	**7B**	{
2C	,	**ЗС**	<	**4C**	**L**	**5C**	\	**6C**	**l**	**7C**	!
2D	-	**3D**	=	**4D**	**M**	**5D**	]	**6D**	**m**	**7D**	}
2E		**ЗЕ**	>	**4E**	**N**	**5E**	-	**6E**	**n**	**7E**	~
2F	/	**3F**	9	**4F**	**O**	**5F**	_	**6F**	**o**	**7F**	DEL

UNICODE

Компьютерная промышленность развивалась преимущественно в США, что привело к появлению кода ASCII. Этот код подходит для английского языка, но не очень удобен для других языков. Во французском языке есть надстрочные знаки (например, système), в немецком — умляуты (например, far) и т. д. В некоторых европейских языках есть несколько букв, которых нет в ASCII, например, немецкое 3 или датское 0. Некоторые языки имеют совершенно другой алфавит (например, русский или арабский), а у некоторых вообще нет алфавита (например, китайский). Компьютеры распространились по всему свету, и поставщики программного обеспечения хотят реализовывать свою продукцию не только в англоязычных, но и в тех странах, где большинство пользователей не говорят по-английски и где нужен другой набор символов.

Первой попыткой расширения ASCII был IS 646, который добавлял к ASCII еще 128 символов, в результате чего получился 8-битный код под названием **Latin-1.** Добавлены были в основном латинские буквы со штрихами и диакритическими знаками. Следующей попыткой был IS 8859, который ввел понятие **кодовая страница.** Кодовая страница — набор из 256 символов для определенного языка или группы языков. IS 8859-1 - это Latin-1. IS 8859-2 включает славянские языки с латинским алфавитом (например, чешский, польский и венгерский). IS 8859-3 содержит символы турецкого, мальтийского, эсперанто и галисийского языков и т. д. Главным недостатком такого подхода является то, что программное обеспечение должно следить, с какой именно кодовой страницей оно имеет дело в данный момент, и при этом невозможно смешивать языки. К тому же эта система не охватывает японский и китайский языки.

Группа компьютерных компаний разрешила эту проблему, создав новую систему под названием UNICODE, и объявила эту систему международным стандартом (IS 10646). UNICODE поддерживается некоторыми языками программирования (например, Java), некоторыми операционными системами (например, Windows NT) и многими приложениями. Вероятно, эта система будет распространяться по всему миру.

Основная идея UNICODE — приписывать каждому символу единственное постоянное 16-битное значение, которое называется **указателем кода.** Многобайтные символы и escape-последовательности не используются. Поскольку каждый символ состоит из 16 битов, писать программное обеспечение гораздо проще.

Так как символы UNICODE состоят из 16 битов, всего получается 65 536 кодовых указателей. Поскольку во всех языках мира в общей сложности около 200 000 символов, кодовые указатели являются очень скудным ресурсом, который нужно распределять с большой осторожностью. Около половины кодов уже распределено, и консорциум, разработавший UNICODE, постоянно рассматривает предложения на распределение оставшейся части. Чтобы ускорить принятие UNICODE, консорциум использовал Latin-1 в качестве кодов от 0 до 255, легко преобразуя ASCII в UNICODE.

Во избежание излишней растраты кодов каждый диакритический знак имеет свой собственный код. А сочетание диакритических знаков с буквами — задача программного обеспечения.

Вся совокупность кодов разделена на блоки, каждый блок содержит 16 кодов. Каждый алфавит в UNICODE имеет ряд последовательных зон. Приведем некоторые примеры (в скобках указано число задействованных кодов): латынь (336), греческий (144), русский (256), армянский (96), иврит (112), деванагари (128), гурмуки(128), ория(128),телугу (128)иканнада(128). Отметим, что каждому из этих языков приписано больше кодов, чем в нем есть букв. Это было сделано отчасти потому, что во многих языках у каждой буквы есть несколько вариантов. Например, каждая буква в английском языке представлена в двух вариантах: там есть строчные и заглавные буквы. В некоторых языках буквы имеют три или более форм, выбор которых зависит от того, где находится буква: в начале, конце или середине слова.

Кроме того, некоторые коды были приписаны диакритическим знакам (112), знакам пунктуации (112), подстрочным и надстрочным знакам (48), знакам валют (48), математическим символам (256), геометрическим фигурам (96) и рисункам (192).

Затем идут символы для китайского, японского и корейского языков. Сначала идут *1024* фонетических символа (например, катакана и бопомофо), затем иероглифы, используемые в китайском и японском языках (20 992), а затем слоги корейского языка (11 156).

Чтобы пользователи могли создавать новые символы для особых целей, существует еще 6400 кодов.

Хотя UNICODE разрешил многие проблемы, связанные с интернационализацией, он все же не мог разрешить абсолютно все проблемы. Например, латинский алфавит упорядочен, а иероглифы — нет, поэтому программа для английского языка может расположить слова «cat? и «dog» по алфавиту, сравнив значение кодов первых букв, а программе для японского языка нужны дополнительные таблицы, чтобы можно было вычислять, в каком порядке расположены символы в словаре.

Еще одна проблема состоит в том, что постоянно появляются новые слова. 50 лет назад никто не говорил об апплетах, киберпространстве, гигабайтах, лазерах, модемах, «смайликах» или видеопленках. С появлением новых слов в английском языке новые коды не нужны. А вот в японском нужны. Кроме новых терминов, необходимо также добавить по крайней мере 20 000 новых имен собственных и географических названий (в основном китайских). Шрифт Брайля, которым пользуются слепые, вероятно, тоже должен быть задействован. Представители различных профессиональных кругов также заинтересованы в наличии каких-либо особых символов. Консорциум по созданию UNICODE рассматривает все новые предложения и выносит по ним решения.

UNICODE использует один и тот же код для символов, которые выглядят почти одинаково, но имеют несколько значений или пишутся немного по-разному в китайском и японском языках (как если бы английские текстовые процессоры всегда писали слово «blue» как «blew», потому что они произносятся одинаково). Одни считают такой подход оптимальным для экономии скудного запаса кодов, другие рассматривают его как англо-саксонский культурный империализм (а вы думали, что приписывание символам 16-битных значений не носит политического характера?). Дело усложняется тем, что полный японский словарь содержит 50 000 иероглифических знаков (не считая собственных имен), поэтому при наличии 20 992 кодов приходится делать выбор и чем-то жертвовать. Далеко не все японцы считают, что консорциум компьютерных компаний, даже если некоторые из них японские, является идеальным форумом, чтобы принимать решения, чем именно нужно жертвовать.

Краткое содержание главы

Компьютерные системы состоят из трех типов компонентов: процессоров, памяти и устройств ввода-вывода. Задача процессора заключается в том, чтобы последовательно вызывать команды из памяти, декодировать и выполнять их. Цикл вызов—декодирование—выполнение всегда можно представить в виде алгоритма. Вызов, декодирование и выполнение команд определенной программы иногда выполняются программой-интерпретатором, работающей на более низком уровне. Для повышения скорости работы во многих компьютерах имеется один или не-

сколько конвейеров или суперскалярная архитектура с несколькими функциональными блоками, которые действуют параллельно.

Широко распространены системы с несколькими процессорами. Компьютеры с параллельной обработкой включают векторные процессоры, в которых одна и та же операция выполняется одновременно над разными наборами данных, мультипроцессоры, в которых несколько процессоров разделяют общую память, и мультикомпьютеры, в которых у каждого компьютера есть своя собственная память, но при этом компьютеры связаны между собой и пересылают друг другу сообщения.

Память можно разделить на основную и вспомогательную. Основная память используется для хранения программ, которые выполняются в данный момент. Время доступа невелико (максимум несколько десятков наносекунд) и не зависит от адреса, к которому происходит обращение Кэш-память еще больше сокращает время доступа. Память может быть оснащена кодом с исправлением ошибок для повышения надежности.

Время доступа к вспомогательной памяти, напротив, гораздо больше (от нескольких миллисекунд и более) и зависит от расположения считываемых и записываемых данных. Наиболее распространенные виды вспомогательной памяти — магнитные ленты, магнитные диски и оптические диски. Магнитные диски существуют в нескольких вариантах: дискеты, винчестеры, IDE-диски, SCSI-диски и RAID-массивы. Среди оптических дисков можно назвать компакт-диски, диски CD-R и DVD.

Устройства ввода-вывода используются для передачи информации в компьютер и из компьютера Они связаны с процессором и памятью одной или несколькими шинами В качестве примеров можно назвать терминалы, мыши, принтеры и модемы. Большинство устройств ввода-вывода используют код ASCII, хотя UNICODE уже стремительно распространяется по всему миру.

Вопросы и задания

1. Рассмотрим машину с трактом данных, который изображен на рис. 2.2. Предположим, что загрузка регистров АЛУ занимает 5 не, работа АЛУ — 10 не, а помещение результата обратно в регистр — 5 не Какое максимальное число миллионов команд в секунду способна выполнять эта машина при отсутствии конвейера?
2. Зачем нужен шаг 2 в списке шагов, приведенном в разделе «Выполнение команд»? Что произойдет, если этот шаг пропустить?
3. На компьютере 1 выполнение каждой команды занимает 10 не, а на компьютере 2 - 5 не. Можете ли вы с уверенностью сказать, что компьютер 2 работает быстрее? Аргументируйте ответ.
4. Предположим, что вы разрабатываете компьютер на одной микросхеме для использования во встроенных системах. Вся память находится на микросхеме и работает с той же скоростью, что и центральный процессор. Рассмотрите принципы, изложенные в разделе «Принципы разработки современных компьютеров», и скажите, важны ли они в данном случае (высокая производительность желательна).

5 Можно ли добавить кэш-память к процессорам, изображенным на рис. 2.7, *б?* Если можно, то какую проблему нужно будет решить в первую очередь?

6. В некотором вычислении каждый последующий шаг зависит от предыдущего Что в данном случае более уместно: векторный процессор или конвейер? Объясните, почему.

7. Чтобы конкурировать с недавно изобретенным печатным станком, один средневековый монастырь решил наладить массовое производство рукописных книг. Для этого в большом зале собралось огромное количество писцов. Настоятель монастыря называл первое слово книги, и все писцы записывали его. Затем настоятель называл второе слово, и все писцы записывали его. Этот процесс повторялся до тех пор, пока не была прочитана вслух и переписана вся книга. На какую из систем параллельной обработки информации (см. раздел «Параллелизм на уровне процессоров») эта система больше всего похожа?

8. При продвижении сверху вниз по пятиуровневой иерархической структуре памяти время доступа возрастает. Каково отношение к времени доступа оптического диска и к регистровой памяти? (Предполагается, что диск уже вставлен.)

9. Сосчитайте скорость передачи данных в человеческом глазу, используя следующую информацию. Поле зрения состоит приблизительно из 10^6 элементов (пикселов). Каждый пиксел может сводиться к наложению трех основных цветов, каждый из которых имеет 64 степени интенсивности. Временное разрешение 100 миллисекунд.

10. Генетическая информация у всех живых существ кодируется в молекулах ДНК. Молекула ДНК представляет собой линейную последовательность четырех основных нуклеотидов: A, C, G и T. Геном человека содержит приблизительно $3x10^9$нуклеотидов в форме 100 000 генов. Какова общая информационная емкость человеческого генома (в битах)? Какова средняя информационная емкость гена (в битах)?

11. Какие из перечисленных ниже видов памяти возможны? Какие из них приемлемы? Объясните, почему.

    1) 10-битный адрес, 1024 ячейки, размер ячейки 8 битов;
    2) 10-битный адрес, 1024 ячейки, размер ячейки 12 битов;
    3) 9-битный адрес, 1024 ячейки, размер ячейки 10 битов;
    4) 11-битный адрес, 1024 ячейки, размер ячейки 10 битов;
    5) 10-битный адрес, 10 ячеек, размер ячейки 1024 бита;
    6) 1024-битный адрес, 10 ячеек, размер ячейки 10 битов.

12. Социологи могут получить 3 возможных ответа на вопрос «Верите ли вы в фей?»: да, нет, не знаю. Учитывая это, одна компьютерная компания решила создать машину для обработки данных социологических опросов. Этот компьютер имеет тринарную память, то есть каждый байт (или трайт?) состоит из 8 тритов, а каждый трит может принимать значение 0, 1 или 2. Сколько

нужно тритов для хранения 6-битного числа? Напишите выражение для числа тритов, необходимых для хранения n битов.

13. Компьютер может содержать 268 435 456 байтов памяти. Почему разработчики выбрали такое странное число вместо какого-нибудь хорошо запоминающегося, например 250 000 000?
14. Придумайте код Хэмминга для разрядов от 0 до 9.
15. Придумайте код для разрядов от 0 до 9 с интервалом Хэмминга 2.
16. В коде Хэмминга некоторые биты «пустые» в том смысле, что они используются для проверки и не несут никакой информации. Какой процент пустых битов содержится в посланиях, полная длина которых (данные + биты проверки) 2"-1? Сосчитайте значение этого выражения при n от 3 до 10.
17. Ошибки при передаче данных по телефонной линии часто происходят «вспышками» (искажается сразу много последовательных битов). Поскольку код Хэмминга может исправлять только одиночные ошибки в символе, в данном случае он не подходит, так как шум может исказить n последовательных битов. Придумайте метод передачи текста в коде ASCII по телефонной линии, где шум может исказить 100 последовательных битов. Предполагается, что минимальный интервал между двумя искажениями составляет тысячи символов. *Подсказка:* подумайте о порядке передачи битов.
18. Сколько времени занимает считывание диска с 800 цилиндрами, каждый из которых содержит 5 дорожек по 32 сектора? Сначала считываются все сектора дорожки 0, начиная с сектора 0, затем все сектора дорожки 1, начиная с сектора $0_т$ и т. д. Оборот совершается за 20 мс, поиск между соседними цилиндрами занимает 10 мс, а в случае расположения считываемых данных в разных частях диска — до 50 мс. Переход от одной дорожки цилиндра к другой происходит мгновенно.
19. Диск, изображенный на рис. 2.16, имеет 64 сектора на дорожке и скорость вращения 7200 оборотов в минуту. Какова скорость передачи данных на одной дорожке?
20. Компьютер содержит шину с временем цикла 25 не. За 1 цикл он может считывать из памяти или записывать в память 32-битное слово. Компьютер имеет диск Ultra-SCSI, который использует шину и передает информацию со скоростью 40 Мбайт/с. Центральный процессор обычно вызывает из памяти и выполняет одну 32-битную команду каждые 25 не. Насколько диск замедляет работу процессора?
21. Представьте, что вы записываете часть операционной системы, отвечающую за управление диском. Логически вы представляете себе диск как последовательность блоков от 0 на внутренней стороне до какого-либо максимума снаружи. Когда создаются файлы, вам приходится размещать свободные сектора. Вы можете двигаться от наружного края внутрь или наоборот. Имеет ли значение, какую стратегию выбрать? Поясните свой ответ.
22. Система адресации LBA использует 24 бита для обращения к сектору. Каков максимальный объем диска, с которым она может работать?

23. RAID третьего уровня может исправлять единичные битовые ошибки, используя только i диск четности. А что происходит в RAID-массиве второго уровня? Он ведь тоже может исправлять единичные ошибки, но использует при этом несколько дисков.
24. Какова точная емкость (в байтах) компакт-диска второго типа, содержащего данные на 74 минуты?
25. Чтобы прожигать отверстия в диске CD-R, лазер должен включаться и выключаться очень быстро. Какова длительность одного состояния (включения или выключения) в наносекундах, если компакт-диск первого типа прокручивается со скоростью 4x?
26. Чтобы вместить фильм длительностью 133 минуты на односторонний DVD с одним слоем, требуется небольшая компрессия. Вычислите, насколько нужно сжать фильм. Предполагается, что для записи дорожки изображения нужно 3,5 Гбайт, разрешающая способность изображения 720x480 пикселов с 24-битным цветом и в секунду меняется 30 кадров.
27. Скорость передачи данных между центральным процессором и связанной с ним памятью на несколько порядков выше, чем скорость передачи данных с механических устройств ввода-вывода. Каким образом это несоответствие может вызвать снижение производительности? Как можно смягчить такое снижение производительности?
28. Графический терминал имеет монитор 1024x768. Изображение на мониторе меняется 75 раз в секунду. Как часто меняется отдельный пиксел?
29. Производитель говорит, что его цветной графический терминал может воспроизводить 2^{24} различных цветов. Однако аппаратное обеспечение имеет только 1 байт для каждого пиксела. Каким же образом получается столько цветов?
30. Монохромный лазерный принтер может печатать на одном листе 50 строк по 80 символов в определенном шрифте. Символ в среднем занимает пространство 2x2 мм, причем тонер занимает 25% этого пространства, а оставшаяся часть остается белой. Толщина слоя тонера составляет 25 микрон. Картридж с тонером имеет размер 25x8x2 см. На сколько страниц хватит картриджа?
31. Когда текст в ASCII-коде с проверкой на четность передается асинхронно со скоростью 2880 символов/с через модем, передающий информацию со скоростью 28 800 бит/с, сколько процентов битов от всех полученных содержат данные?
32. Компания, выпускающая модемы, разработала новый модем с частотной модуляцией, который использует 16 частот вместо 2. Каждая секунда делится на n равных временных отрезков, каждый из которых содержит один из 16 возможных тонов. Сколько битов в секунду может передавать этот модем при использовании синхронной передачи?
33. Оцените, сколько символов (включая пробелы) содержит обычная книга по информатике. Сколько битов нужно для того, чтобы закодировать книгу

в ASCII с проверкой на четность? Сколько компакт-дисков нужно для хранения 10 000 книг по информатике? Сколько двухсторонних, двухслойных DVD-дисков нужно для хранения такого же количества книг?

34. Декодируйте следующий двоичный текст ASCII: 1001001 0100000 1001100 1001111 1010110 1000101 0100000 1011001 1001Ш 1010101 0101110.

35. Напишите процедуру *hamming (ascii, encoded),* которая переделывает 7 последовательных битов *ascii* в 11 -битное целое кодированное число *encoded.*

36. Напишите функцию *distance (code, n, k),* которая на входе получает массив *code* из *n* символов по *k* битов каждый и возвращает дистанцию символа.

Глава 3
Цифровой логический уровень

В самом низу иерархической схемы на рис. 1.2 находится цифровой логический уровень, или аппаратное обеспечение компьютера. В этой главе мы рассмотрим различные аспекты цифровой логики, что должно послужить основой для изучения более высоких уровней в последующих главах. Предмет изучения находится на границе информатики и электротехники, но материал является самодостаточным, поэтому предварительного ознакомления с аппаратным обеспечением и электротехникой не потребуется.

Основные элементы, из которых конструируются цифровые компьютеры, чрезвычайно просты. Сначала мы рассмотрим эти основные элементы, а также специальную двузначную алгебру (булеву алгебру), которая используется при конструировании этих элементов. Затем мы рассмотрим основные схемы, которые можно построить из вентилей в различных комбинациях, в том числе схемы для выполнения арифметических действий. Следующая тема — как можно комбинировать вентили для хранения информации, то есть как устроена память. После этого мы перейдем к процессорам и к тому, как процессоры на одной микросхеме обмениваются информацией с памятью и периферическими устройствами. Затем мы рассмотрим различные примеры промышленного производства.

Вентили и булева алгебра

Цифровые схемы могут конструироваться из небольшого числа простых элементов путем сочетания этих элементов в различных комбинациях. В следующих разделах мы опишем эти основные элементы, покажем, как их можно сочетать, а также введем математический метод, который можно использовать при анализе их работы.

Вентили

Цифровая схема — это схема, в которой есть только два логических значения. Обычно сигнал от 0 до 1 В представляет одно значение (например, 0), а сигнал от 2 до 5 В — другое значение (например, 1). Напряжение за пределами указанных величин недопустимо. Крошечные электронные устройства, которые называются вен-

тилями, могут вычислять различные функции от этих двузначных сигналов. Эти вентили формируют основу аппаратного обеспечения, на которой строятся все цифровые компьютеры.

Описание принципов работы вентилей не входит в задачи этой книги, поскольку это относится к **уровню физических устройств,** который находится ниже уровня 0. Тем не менее мы очень кратко рассмотрим основной принцип, который не так уж и сложен. Вся современная цифровая логика основывается на том, что транзистор может работать как очень быстрый бинарный переключатель. На рис. 3.1, а изображен биполярный транзистор, встроенный в простую схему. Транзистор имеет три соединения с внешним миром; **коллектор, базу** и **эмиттер.** Если входное напряжение У„„ниже определенного критического значения, транзистор выключается и действует как очень большое сопротивление. Это приводит к выходному сигналу V_{out}, близкому к V_{cc} (напряжению, подаваемому извне), обычно +5 В для данного типа транзистора. Если V_{in} превышает критическое значение, транзистор включается и действует как провод, вызывая заземление сигнала V_{out} (по соглашению О В).

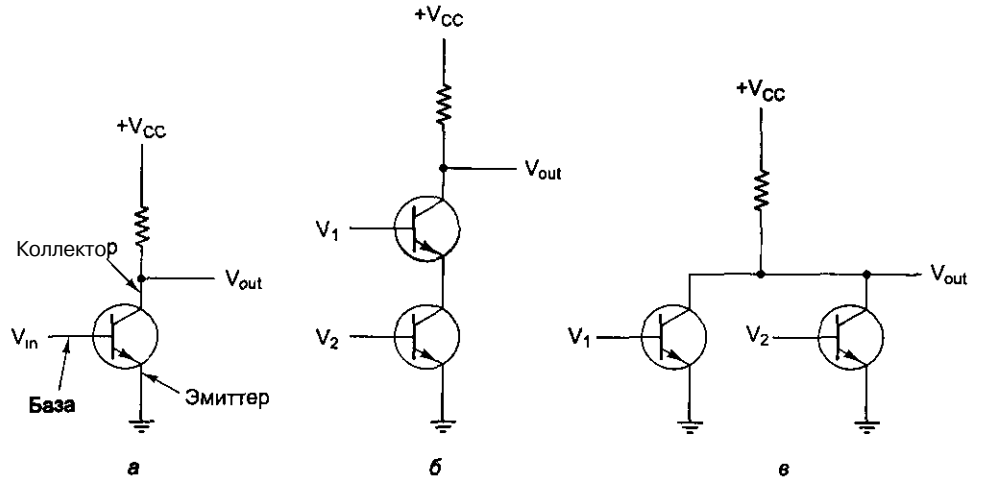


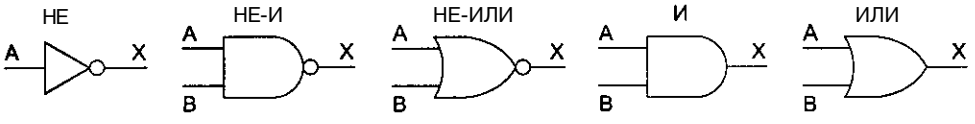
Рис. 3.1. Транзисторный инвертор (а); вентиль НЕ-И (б); вентиль НЕ-ИЛИ (в)

Важно отметить, что если напряжение V_{in} низкое, то V_{out} высокое, и наоборот. Эта схема, таким образом, является инвертором, превращающим логический 0 в логическую 1 и логическую 1 в логический 0. Резистор (ломаная линия) нужен для ограничения количество тока, проходящего через транзистор, чтобы транзистор не сгорел. На переключение с одного состояния на другое обычно требуется несколько наносекунд.

На рис. 3.1, *б* два транзистора соединены последовательно. Если и напряжение V,, и напряжение V_2 высокое, то оба транзистора будут служить проводниками и снижать V_{out}. Если одно из входных напряжений низкое, то соответствующий транзистор будет выключаться и напряжение на выходе будет высоким. Другими словами, V_{out} будет низким тогда и только тогда, когда и напряжение V], и напряжение $У_г$ высокое.

На рис. 3.1, *в* два транзистора соединены параллельно. Если один из входных сигналов высокий, будет включаться соответствующий транзистор и снижать выходной сигнал Если оба напряжения на входе низкие, то выходное напряжение будет высоким

Эти три схемы образуют три простейших вентиля Они называются вентилями НЕ, НЕ-И и НЕ-ИЛИ. Вентили НЕ часто называют **инверторами.** Мы будем использовать оба термина. Если мы примем соглашение, что высокое напряжение (Vcc) — это логическая 1, а низкое напряжение («земля») — логический 0, то мы сможем выражать значение на выходе как функцию от входных значений. Значки, которые используются для изображения этих трех типов вентилей, показаны на рис. 3.2, ***а*** — *в*. Там же приводится поведение функции для каждой схемы. На этих рисунках A и B — это входные сигналы, а X — выходной сигнал. Каждая строка таблицы определяет выходной сигнал для различных комбинаций входных сигналов.



A	X
0	1
1	0

а

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

б

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

в

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

г

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

д

Рис. 3.2. Значки для изображения 5 основных вентилей. Поведение функции для каждого вентиля

Если выходной сигнал (см. рис. 3.1, *б)* подать в инвертор, мы получим другую схему, противоположную вентилю НЕ-И, то есть такую схему, у которой выходной сигнал равен 1 тогда и только тогда, когда оба входных сигнала равны 1. Такая схема называется вентилем И; ее схематическое изображение и описание соответствующей функции даны на рис. 3.2, *г.* Точно так же вентиль НЕ-ИЛИ может быть связан с инвертором. Тогда получится схема, **у** которой выходной сигнал равен 1 в том случае, если хотя бы один из входных сигналов — 1, и равен 0, если оба входных сигнала равны 0. Изображение этой схемы, ко горая называется вентилем ИЛИ, а также описание соответствующей функции даны на рис. 3.2, *д.* Маленькие кружочки в схемах инвертора, вентиля НЕ-И и вентиля НЕ-ИЛИ называются **инвертирующими выходами** Они также могут использоваться в другом контексте для указания на инвертированный сигнал.

Пять вентилей, изображенных на рис. 3.2, составляют основу цифрового логического уровня. Из предшествующего обсуждения должно быть ясно, что вентили НЕ-И и НЕ-ИЛИ требуют два транзистора каждый, а вентили И и ИЛИ — три транзистора каждый. По этой причине во многих компьютерах используются вен-

тили НЕ-И и НЕ-ИЛИ, а не И и ИЛИ. (На практике все вентили выполняются несколько по-другому, но НЕ-И и НЕ-ИЛИ все равно проще, чем И и ИЛИ.) Следует упомянуть, что вентили могут иметь более двух входов. В принципе вентиль НЕ-И, например, может иметь произвольное количество входов, но на практике больше восьми обычно не бывает.

Хотя устройство вентилей относится к уровню физических устройств, мы все же упомянем основные серии производственных технологий, так как они часто упоминаются в литературе. Две основные технологии — **биполярная** и **МОП** (металл-оксид-полупроводник). Среди биполярных технологий можно назвать **ТТЛ** (транзисторно-транзисторную логику), которая служила основой цифровой электроники на протяжении многих лет, и **ЭСЛ** (эмиттерно-связанную логику), которая используется в тех случаях, когда требуется высокая скорость выполнения операций.

Вентили МОП работают медленнее, чем ТТЛ и ЭСЛ, но потребляют гораздо меньше энергии и занимают гораздо меньше места, поэтому можно компактно расположить большое количество таких вентилей. Вентили МОП имеют несколько разновидностей: p-канальный МОП-прибор, n-канальный МОП-прибор и комплиментарный МОП. Хотя МОП-транзисторы конструируются не так, как биполярные транзисторы, они обладают такой же способностью функционировать, как электронные переключатели. Современные процессоры и память чаще всего производятся с использованием технологии комплиментарных МОП, которая работает при напряжении +3,3 В. Это все, что мы можем сказать об уровне физических устройств. Читатели, желающие узнать больше об этом уровне, могут обратиться к литературе, приведенной в главе 9.

Булева алгебра

Чтобы описать схемы, которые строятся путем сочетания различных вентилей, нужен особый тип алгебры, в которой все переменные и функции могут принимать только два значения: 0 и 1. Такая алгебра называется **булевой алгеброй.** Она названа в честь английского математика Джорджа Буля (1815-1864). На самом деле в данном случае мы говорим об особом типе булевой алгебры, а именно об **алгебре релейных схем,** но термин «булева алгебра» очень часто используется в значении «алгебра релейных схем», поэтому мы не будем их различать.

Как и в обычной алгебре (то есть в той, которую изучают в школе), в булевой алгебре есть свои функции. Булева функция имеет одну или несколько переменных и выдает результат, который зависит только от значений этих переменных. Можно определить простую функцию f, сказав, что f(A)=1, если A=0, и f(A)=-O, если A=1. Такая функция будет функцией НЕ (см. рис. 3.2, *а).*

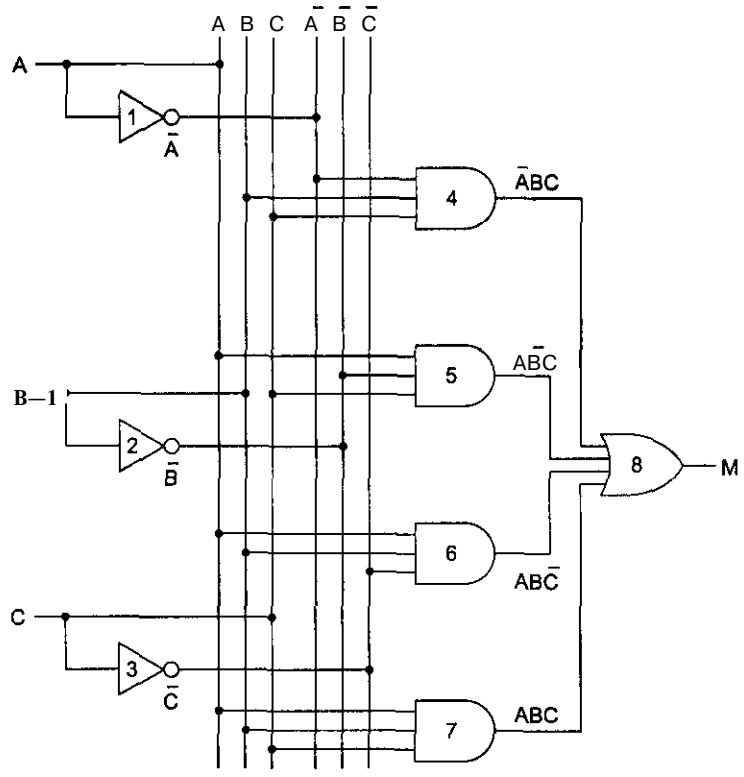
Так как булева функция от n переменных имеет только 2" возможных комбинаций значений переменных, то такую функцию можно полностью описать в таблице с 2" строками. В каждой строке будет даваться значение функции для разных комбинаций значений переменных. Такая таблица называется **таблицей истинности.** Все таблицы на рис. 3.2 представляют собой таблицы истинности. Если мы

договоримся всегда располагать строки таблицы истинности по порядку номеров, то есть для двух переменных в порядке 00, 01, 10, 11, то функцию можно полностью описать 2"-битным двоичным числом, которое получается, если считывать по вертикали колонку результатов в таблице истинности Таким образом, НЕ-И — это 1110, НЕ-ИЛИ - 1000, И - 0001 и ИЛИ - 0111. Очевидно, что существует только 16 булевых функций от двух переменных, которым соответствуют 16 возможных 4-битных цепочек. В обычной алгебре, напротив, есть бесконечное число функций от двух переменных, и ни одну из них нельзя описать, дав таблицу значений этой функции для всех возможных значений переменных, поскольку каждая переменная может принимать бесконечное число значений.

На рис. 3.3, *а* показана таблица истинности для булевой функции от трех переменных: M-f(A, B, C) Это функция большинства, которая принимает значение 0, если большинство переменных равно 0, и 1, если большинство переменных равно 1. Хотя любая булева функция может быть определена с помощью таблицы истинности, с возрастанием количества переменных такой тип записи становится громоздким. Поэтому вместо таблиц истинности часто используется другой тип записи.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

а



б

Рис. 3.3. Таблица истинности для функции большинства от трех переменных (а), схема для этой функции (б)

Чтобы увидеть, каким образом осуществляется этот другой тип записи, отметим, что любую булеву функцию можно определить, указав, какие комбинации значений переменных дают значение функции 1. Для функции, приведенной на рис. 3.3, а, существует 4 комбинации переменных, которые дают значение функции 1. Мы будем рисовать черту над переменной, чтобы показать, что ее значение инвертируется. Отсутствие черты означает, что значение переменной не инвертируется. Кроме того, мы будем использовать знак умножения (точку) для обозначения булевой функции И (знак умножения может опускаться) и + для обозначения булевой функции ИЛИ. Например, $A\bar{B}C$ принимает значение 1, только если A=1, B=0 и C=1. $A\bar{H}+BC$ принимает значение 1, только если (A=1 и B=0) или (B=1 и C=0). В таблице на рис. 3.3, *а* функция принимает значение 1 в четырех строках: $AB\bar{C}$, $A\bar{B}C$, $\bar{A}BC$ И $\bar{A}\bar{B}C$. Функция M принимает значение истины (то есть 1), если одно из этих четырех условий истинно. Следовательно, мы можем написать

$M=AB\bar{C}+A\bar{B}C+\bar{A}BC+\bar{A}\bar{B}C$.

Это компактная запись таблицы истинности. Таким образом, функцию от n переменных можно описать суммой максимум 2^n произведений, при этом в каждом произведении будет по n множителей. Как мы скоро увидим, такая формулировка особенно важна, поскольку она ведет прямо к реализации данной функции с использованием стандартных вентилей.

Важно понимать различие между абстрактной булевой функцией и ее реализацией с помощью электронной схемы. Булева функция состоит из переменных, например A, B и C, и операторов И, ИЛИ и НЕ. Булева функция описывается с помощью таблицы истинности или специальной записи, например:

$F=A\bar{B}C+ABC$.

Булева функция может реализовываться с помощью электронной схемы (часто различными способами) с использованием сигналов, которые представляют входные и выходные переменные, и вентилей, например, И, ИЛИ и НЕ.

Реализация булевых функций

Как было сказано выше, представление булевой функции в виде суммы максимум 2^n произведений делает возможной реализацию этой функции. На рисунке 3.3 можно увидеть, как это осуществляется. На рисунке 3.3, *б* входные сигналы A, B и C показаны с левой стороны, а функция M, полученная на выходе, показана с правой стороны. Поскольку необходимы дополнительные величины (инверсии) входных переменных, они образуются путем провода сигнала через инверторы 1,2 и 3. Чтобы сделать рисунок понятней, мы нарисовали 6 вертикальных линий, 3 из которых связаны с входными переменными, а 3 другие — с их инверсиями. Эти линии обеспечивают передачу входного сигнала к вентилям. Например, вентили 5, 6 и 7 в качестве входа используют A. В реальной схеме эти вентили, вероятно, будут непосредственно соединены проводом с A без каких-либо промежуточных вертикальных проводов.

Схема содержит четыре вентиля И, по одному для каждого члена в уравнении для M (то есть по одному для каждой строки в таблице истинности с результатом 1). Каждый вентиль И вычисляет одну из указанных строк таблицы истинное-

ти. В конце концов все данные произведения суммируются (имеется в виду операция ИЛИ) для получения конечного результата.

Посмотрите на рис. 3.3, *б,* В этой книге мы будем использовать следующее соглашение: если две линии на рисунке пересекаются, связь подразумевается только в том случае, если на пересечении указана жирная точка. Например, выход вентиля 3 пересекает все 6 вертикальных линий, но связан он только с С. Отметим, что другие авторы могут использовать другие соглашения.

Из рисунка 3.3 должно быть ясно, как реализовать схему для любой булевой функции:

1. Составить таблицу истинности для данной функции.
2. Обеспечить инверторы, чтобы порождать инверсии для каждого входного сигнала.
3. Нарисовать вентиль И для каждой строки таблицы истинности с результатом 1.
4. Соединить вентили И с соответствующими входными сигналами.
5. Вывести выходы всех вентилей И в вентиль ИЛИ.

Мы показали, как реализовать любую булеву функцию с использованием вентилей НЕ, И и ИЛИ. Однако гораздо удобнее строить схемы с использованием одного типа вентилей. К счастью, можно легко преобразовать схемы, построенные по предыдущему алгоритму, в форму НЕ-И или НЕ-ИЛИ. Чтобы осуществить такое преобразование, все, что нам нужно, — это способ воплощения НЕ, И и ИЛИ с помощью одного типа вентилей. На рисунке 3.4 показано, как это можно сделать, используя только вентили НЕ-И или только вентили НЕ-ИЛИ. Отметим, что существуют также другие способы подобного преобразования.

Для того чтобы реализовать булеву функцию с использованием только вентилей НЕ-И или только вентилей НЕ-ИЛИ, можно сначала следовать алгоритму, описанному выше, и сконструировать схему с вентилями НЕ и И и ИЛИ. Затем нужно заменить многовходовые вентили эквивалентными схемами с использованием двухвходовых вентилей. Например, A+B+C+D можно поменять на (A+B)+(C+D), используя три двухвходовых вентиля. Затем вентили НЕ и И и ИЛИ заменяются схемами, изображенными на рис. 3.4.

Хотя такая процедура и не приводит к оптимальным схемам с точки зрения минимального числа вентилей, она демонстрирует, что подобное преобразование осуществимо. Вентили НЕ-И и НЕ-ИЛИ считаются полными, потому что можно вычислить любую булеву функцию, используя только вентили НЕ-И или только вентили НЕ-ИЛИ. Ни один другой вентиль не обладает таким свойством, вот почему именно эти два типа вентилей предпочтительны при построении схем.

Эквивалентность схем

Разработчики схем часто стараются сократить число вентилей, чтобы снизить цену, уменьшить занимаемое схемой место, сократить потребление энергии и т. д. Чтобы упростить схему, разработчик должен найти другую схему, которая может вычислять ту же функцию, но при этом требует меньшего количества вентилей (или может работать с более простыми вентилями, например двухвходовыми вместо четырехвходовых). Булева алгебра является ценным инструментом в поиске эквивалентных схем.

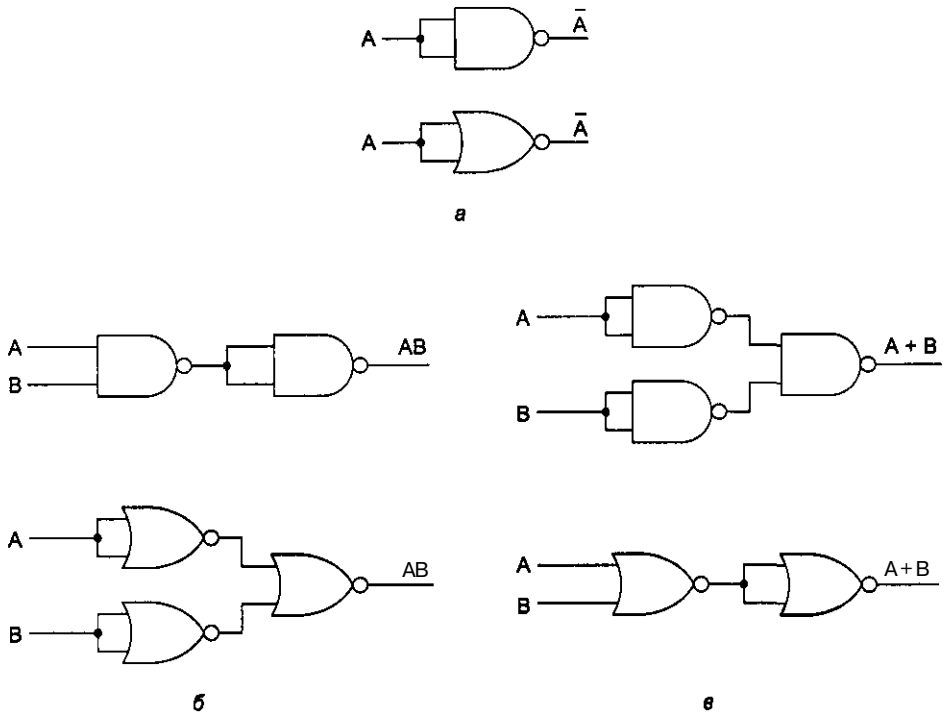
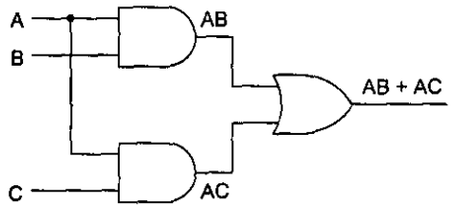


Рис. 3.4. Конструирование вентилей НЕ (а), И (б) и ИЛИ *(в)* с использованием только вентилей НЕ-И или только вентилей НЕ-ИЛИ

В качестве примера использования булевой алгебры рассмотрим схему и таблицу истинности для AB+AC (рис. 3.5, *а)*. Хотя мы это еще не обсуждали, многие правила обычной алгебры имеют силу для булевой алгебры. Например, выражение AB+AC может быть преобразовано в A(B+C) с помощью дистрибутивного закона. На рис. 3.5, *б* показана схема и таблица истинности для A(B+C). Две функции являются эквивалентными тогда и только тогда, когда обе функции принимают одно и то же значение для всех возможных переменных. Из таблиц истинности на рис. 3.5 ясно видно, что A(B+C) эквивалентно AB+AC. Несмотря на эту эквивалентность, схема на рис. 3.5, *б* лучше, чем схема на рис. 3.5, а, поскольку она содержит меньше вентилей.

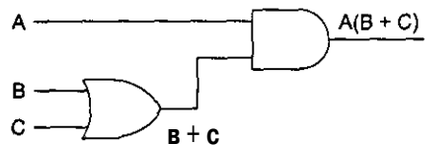
Обычно разработчик исходит из определенной булевой функции, а затем применяет к ней законы булевой алгебры, чтобы найти более простую функцию, эквивалентную исходной. На основе полученной функции можно конструировать схему.

Чтобы использовать данный подход, нам нужны некоторые равенства из булевой алгебры. В табл. 3.1 показаны некоторые основные законы. Интересно отметить, что каждый закон имеет две формы. Одну форму из другой можно получить, меняя И на ИЛИ и 0 на 1. Все законы можно легко доказать, составив их таблицы истинности. Почти во всех случаях результаты очевидны, за исключением законов Де Моргана, законов поглощения и дистрибутивного закона A+BC=(A+B)(A+C). Законы Де Моргана распространяются на выражения с более чем двумя переменными, например $\bar{A}\bar{B}\bar{C}=\bar{A}+\bar{B}+C$.



A	B	C	AB	AC	AB+AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

а



A	B	C	A	B+C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

б

Рис. 3.5. Две эквивалентные функции: AB+AC (а); A(B+C) *(б)*.

Таблица 3 . 1 . Некоторые законы булевой алгебры

Названиязаконов	**И**	**ИЛИ**
Законы тождества	1A=A	0+A=A
Законы нуля	OA=0	1+A=1
Законы идемпотентности	AA=A	A+A=A
Законы инверсии	**AĀ"=0**	A+$\bar{A}$=1
Коммуникативные законы	AB=BA	A+B=B+A
Ассоциативные законы	(AB)C=A(BC)	(A+B)+C=A+£B+C)
Дистрибутивные законы	A+BC=(A+B)(A+C)	A(B+C)=AB+AC
Законы поглощения	A(A+B)=A	**A+AB=A**
ЗаконыДеМоргана	$\overline{AB}$=Д+$\bar{B}$	**$\overline{A+B}$=ДВ**

Законы Де Моргана предполагают альтернативную запись. На рис. 3.6, *а* форма И дается с отрицанием, которое показывается с помощью инвертирующих входов и выходов. Таким образом, вентиль ИЛИ с инвертированными входными сигналами эквивалентен вентилю НЕ-И. Из рис. 3.6, *б,* на котором изображена вторая форма закона Де Моргана, ясно, что вместо вентиля НЕ-ИЛИ можно нарисовать вентиль И с инвертированными входами. С помощью отрицания обеих форм закона Де Моргана мы приходим к эквивалентным репрезентациям вентилей И и ИЛИ (см. рис. 3.6, *в* и 3.6, *г)*. Аналогичные символические изображения существуют для различных форм закона Де Моргана (например, n-входовый вентиль НЕ-И становится вентилем ИЛИ с инвертированными входами).

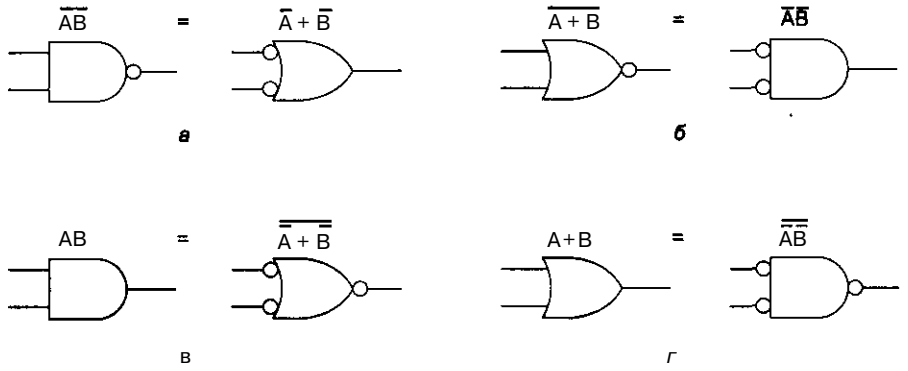
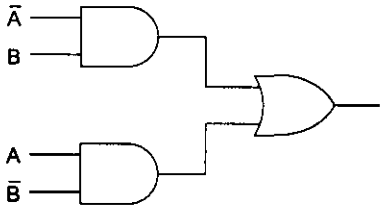


Рис. 3.6. Альтернативные обозначения некоторых вентилей: НЕ-И (а); НЕ-ИЛИ (б); И (е); ИЛИ (г)

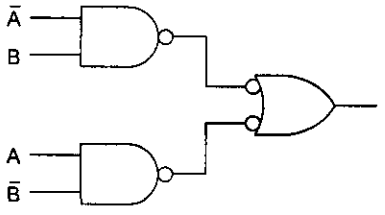
Используя уравнения, указанные на рис. 3.6, и аналогичные уравнения для многовходовых вентилей, можно легко преобразовать сумму произведений в чистую форму НЕ-И или чистую форму НЕ-ИЛИ. В качестве примера рассмотрим функцию ИСКЛЮЧАЮЩЕЕ ИЛИ (рис. 3.7, *а)*. Стандартная схема, выражающая сумму произведений, показана на рис. 3.7, *б.* Чтобы перейти к форме НЕ-И, нужно линии, соединяющие выходы вентилей И с входом вентиля ИЛИ, нарисовать с инвертирующими входами и выходами, как показано на рис. 3.7, *в.* Затем, применяя рис. 3.6, *а,* мы приходим к рис. 3.7, *г.* Переменные Ā и B̄ можно получить из А и В, используя вентили НЕ-И или НЕ-ИЛИ с объединенными входами. Отметим, что инвертирующие входы (выходы) могут перемещаться вдоль линии по желанию, например, от выходов входных вентилей к входам выходного вентиля.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

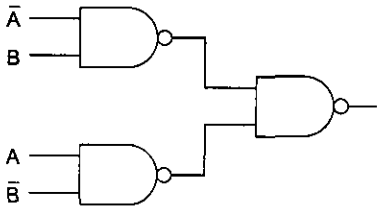
а



б



в



г

Рис. 3.7. Таблица истинности для функции ИСКЛЮЧАЮЩЕЕ ИЛИ (а); Три схемы для вычисления этой функции (б), *(в), (г)*

Очень важно отметить, что один и тот же вентиль может вычислять **разные** функции в зависимости от используемых соглашений. На рис. 3.8, *а* мы показали выход определенного вентиля, F, для различных комбинаций входных сигналов. И входные, и выходные сигналы показаны в вольтах. Если мы примем соглашение, что О В — это логический ноль, а 3,3 В или 5 В — логическая единица, мы получим таблицу истинности, показанную на рис. 3.8, *б*, то есть функцию И. Такое соглашение называется **позитивной логикой.** Однако если мы примем **негативную логику,** то есть условимся, что О В — это логическая единица, а 3,3 В или 5 В — логический ноль, то мы получим таблицу истинности, показанную на рис. 3.8, *в,* то есть функцию ИЛИ.

A	B	F
0 V	0 V	0 V
0 V	5 V	0 V
5 V	0 V	0 V
5 V	5 V	5 V

а

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

б

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

в

Рис. 3.8. Электрические характеристики устройства (а); позитивная логика (б); негативная логика (s)

Таким образом, **все** зависит от того, какое соглашение выбрано для отображения вольт в логических величинах. В этой книге мы будем использовать позитивную логику. Случаи использования негативной логики будут оговариваться отдельно.

Основные цифровые логические схемы

В предыдущих разделах мы увидели, как реализовать простейшие схемы с использованием отдельных вентилей. На практике в настоящее время схемы очень редко конструируются вентиль за вентилем, хотя когда-то это было распространено. Сейчас стандартные блоки представляют собой модули, которые содержат ряд вентилей. В следующих разделах мы рассмотрим эти стандартные блоки более подробно и увидим, как они используются и как их можно построить из отдельных вентилей.

Интегральные схемы

Вентили производятся и продаются не по отдельности, а в модулях, которые называются **интегральными схемами (ИС)** или **микросхемами.** Интегральная схема представляет собой квадратный кусочек кремния размером примерно 5x5 мм, на котором находится несколько вентилей[1]. Маленькие интегральные схемы обычно

[1] Следует заметить, что эти сведения относятся к семидесятым годам прошлого века. В настоящее время степень интеграции стала выше на несколько порядков, и такие простейшие интегральные схемы в вычислительной технике уже давно не используются. — *Примеч. научи, ред.*

помещаются в прямоугольные пластиковые или керамические корпуса размером от 5 до 15 мм в ширину и от 20 до 50 мм в длину. Вдоль длинных сторон располагается два параллельных ряда выводов около 5 мм в длину, которые можно втыкать в разъемы или впаивать в печатную плату. Каждый вывод соединяется с входом или выходом какого-нибудь вентиля, или с источником питания, или с «землей». Корпус с двумя рядами выводов снаружи и интегральными схемами внутри официально называется двурядным корпусом (Dual Inline Package, сокращенно DIP), но все называют его микросхемой, стирая различие между куском кремния и корпусом, в который он помещается. Большинство корпусов имеют 14, 16, 18, 20, 22, 24, 28,40, 64 или 68 выводов. Для больших микросхем часто используются корпуса, у которых выводы расположены со всех четырех сторон или снизу.

Микросхемы можно разделить на несколько классов с точки зрения количества вентилей, которые они содержат. Эта классификация, конечно, очень грубая, но иногда она может быть полезна:

- МИС (малая интегральная схема): от 1 до 10 вентилей.
- СИС (средняя интегральная схема): от 1 до 100 вентилей.
- БИС (большая интегральная схема): от 100 до 100 000 вентилей.
- СБИС (сверхбольшая интегральная схема): более 100 000 вентилей.

Эти схемы имеют различные свойства и используются для различных целей.

МИС обычно содержит от двух до шести независимых вентилей, каждый из которых может использоваться отдельно, как описано в предыдущих разделах. На рис. 3.9 изображена обычная микросхема МИС, содержащая четыре вентиля НЕ-И. Каждый из этих вентилей имеет два входа и один выход, что требует наличия 12 выводов. Кроме того, микросхеме требуется питание (V_{cc}) и «земля» (GND). Они разделяются всеми вентилями. На корпусе рядом с выводом 1 обычно имеется паз, чтобы можно было определить, что это вывод 1. Чтобы избежать путаницы на диаграмме, по соглашению не показываются неиспользованные вентили, источник питания и «земля».

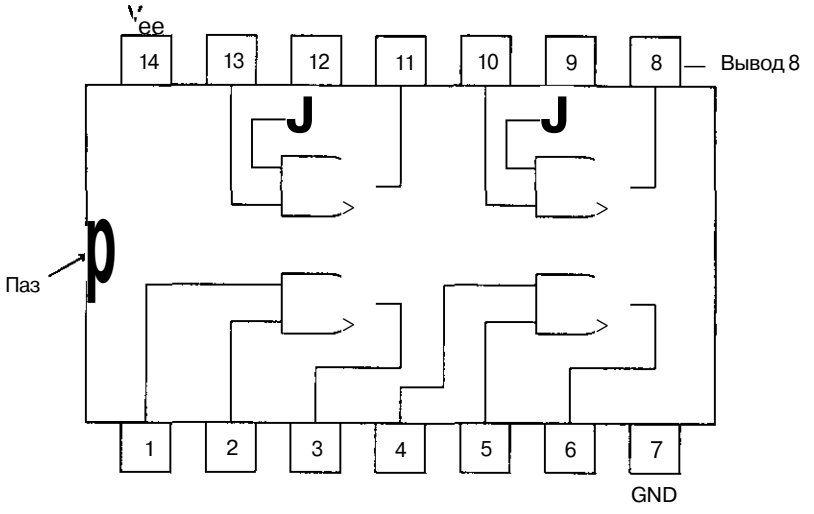


Рис. 3.9. Микросхема МИС, содержащая 4 вентиля

Подобные микросхемы стоят несколько центов. Каждая микросхема МИС содержит несколько вентилей и примерно до 20 выводов. В 70-е годы компьютеры конструировались из большого числа таких микросхем, но в настоящее время на одну микросхему помещается целый центральный процессор и существенная часть памяти (кэш-памяти).

Для удобства мы считаем, что у вентиля появляются изменения на выходе, как только появляются изменения на входе. На самом деле существует определенная **задержка вентиля,** которая включает в себя время прохождения сигнала через микросхему и время переключения. Время задержки обычно составляет от 1 до 10 не.

В настоящее время стало возможным помещать до 10 млн транзисторов на одну микросхему[1]. Так как любая схема может быть сконструирована из вентилей НЕ-И, может создаться впечатление, что производитель способен изготовить микросхему, содержащую 5 млн вентилей НЕ-И. К несчастью, для создания такой микросхемы потребуется 15 000 002 выводов. Поскольку стандартный вывод занимает 0,1 дюйм, микросхема будет более 18 км в длину, что отрицательно скажется на покупательной способности. Поэтому чтобы использовать преимущество данной технологии, нужно разработать такие схемы, у которых количество вентилей сильно превышает количество выводов. В следующих разделах мы рассмотрим простые микросхемы МИС, в которых несколько вентилей соединены определенным образом между собой для вычисления некоторой функции, но при этом требуется небольшое число внешних выводов

Комбинационные схемы

Многие применения цифровой логики требуют наличия схем с несколькими входами и несколькими выходами, в которых выходные сигналы определяются текущими входными сигналами. Такая схема называется **комбинационной схемой.** Не все схемы обладают таким свойством. Например, схема, содержащая элементы памяти, может генерировать выходные сигналы, которые зависят от значений, хранящихся в памяти. Микросхема, которая реализует таблицу истинности (например, приведенную на рис. 3.3, *а),* является типичным примером комбинационной схемы. В этом разделе мы рассмотрим наиболее часто используемые комбинационные схемы.

Мультиплексоры

На цифровом логическом уровне **мультиплексор** представляет собой схему с *2"* входами, одним выходом и п линиями управления, которые выбирают один из входов. Выбранный вход соединяется с выходом. На рис. 3.10 изображена схема восьмивходового мультиплексора. Три линии управления А, В и С кодируют 3-битное число, которое указывает, какая из восьми линий входа должна соединяться с вентилем ИЛИ и, следовательно, с выходом. Вне зависимости от того, какое значение будет на линиях управления, семь вентилей И будут всегда выдавать на выходе 0, а оставшийся может выдавать или 0, или 1 в зависимости от значения

[1] Не стоит забывать закон Мура. Ядро процессора Pentium IV содержит уже 42 млн транзисторои, и очевидно, это не предел — *Примеч научн ред.*

выбранной линии входа. Каждый вентиль И запускается определенной комбинацией линий управления. Схема мультиплексора показана на рис. 3.10. Если к этому добавить источник питания и «землю», то мультиплексор можно запаковать в корпус с 14 выводами.

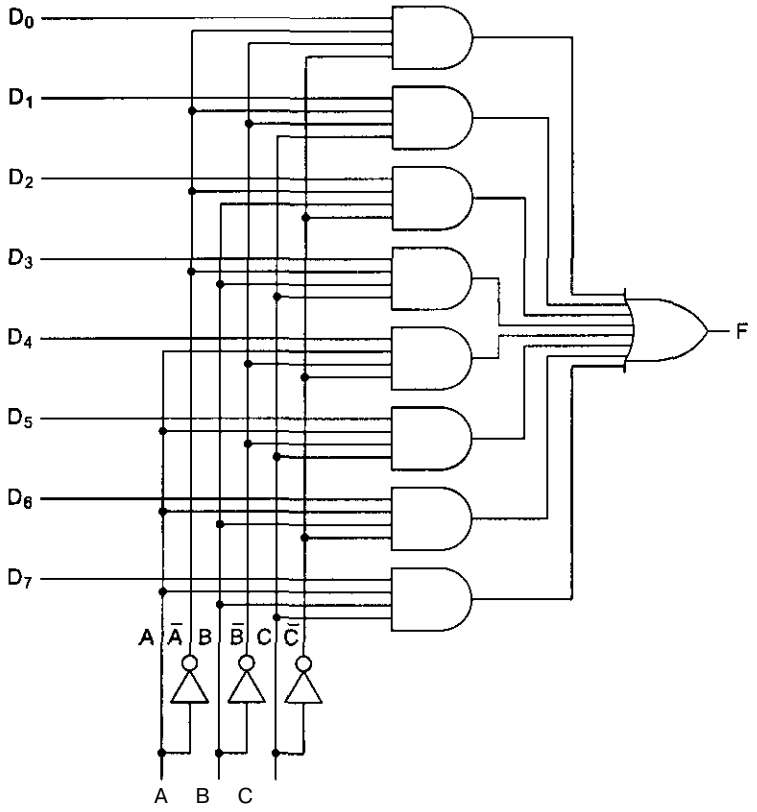


Рис. 3.10. Схема восьмивходового мультиплексора

Используя мультиплексор, мы можем реализовать функцию большинства (см. рис. 3.3, а), как показано на рис. 3.11, б. Для каждой комбинации A, B и C выбирается одна из входных линий. Каждый вход соединяется или с V_{cc} (логическая 1), или с «землей» (логический 0). Алгоритм соединения входов очень прост: входной сигнал D_i такой же, как значение в строке i в таблице истинности. На рис. 3.3, *а* в строках 0, 1, 2 и 4 значение функции равно 0, поэтому соответствующие входы заземляются; в оставшихся строках значение функции равно 1, поэтому соответствующие входы соединяются с логической 1. Таким способом можно реализовать любую таблицу истинности с тремя переменными, используя микросхему на рис. 3.11, *а*.

Мы уже видели, как мультиплексор может использоваться для выбора одного из нескольких входов и как он может реализовать таблицу истинности. Его также можно использовать в качестве преобразователя параллельного кода в последова-

тельный. Если подать 8 битов данных на линии входа, а затем переключать линии управления последовательно от 000 до i11 (это двоичные числа), 8 битов поступят на линию выхода последовательно. Обычно такое преобразование осуществляется при вводе информации с клавиатуры, поскольку каждое нажатие клавиши определяет 7- или 8-битное число, которое должно передаваться последовательно по телефонной линии.

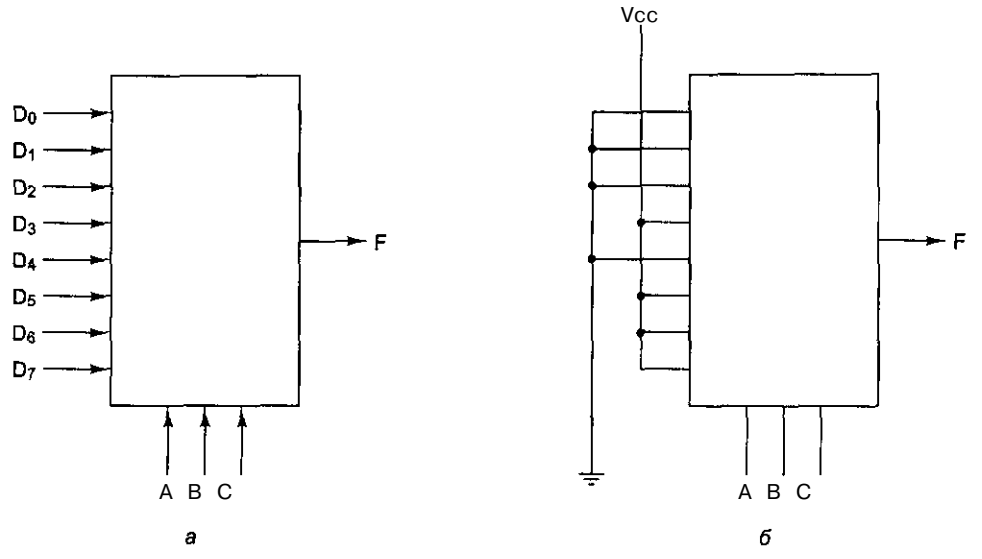


Рис. 3.11. Мультиплексор, построенный на СИС (а), тот же мультиплексор, смонтированный для вычисления функции большинства (б)

Противоположностью мультиплексора является демультиплексор, который соединяет единственный входной сигнал с одним из 2" выходов в зависимости от значений п линий управления. Если бинарное значение линий управления равно к, то выбирается выход к.

Декодеры

В качестве второго примера рассмотрим схему, которая получает на входе п-битное число и использует его для того, чтобы выбрать (то есть установить на значение 1) одну из 2" выходных линий. Такая схема называется декодером. Пример декодера для п=3 показан на рис. 3.12.

Чтобы понять, зачем нужен декодер, представим себе память, состоящую из 8 микросхем, каждая из которых содержит 1 Мбайт. Микросхема 0 имеет адреса от 0 до 1 Мбайт, микросхема 1 — адреса от 1 Мбайт до 2 Мбайт и т. д. Три старших двоичных разряда адреса используются для выбора одной из восьми микросхем. На рис. 3.12 эти три бита — три входа A, B и C В зависимости от входных сигналов ровно одна из восьми выходных линий (D_0,..., D_7) принимает значение 1; остальные линии принимают значение 0. Каждая выходная линия запускает одну из восьми микросхем памяти. Поскольку только одна линия принимает значение 1, запускается только одна микросхема.

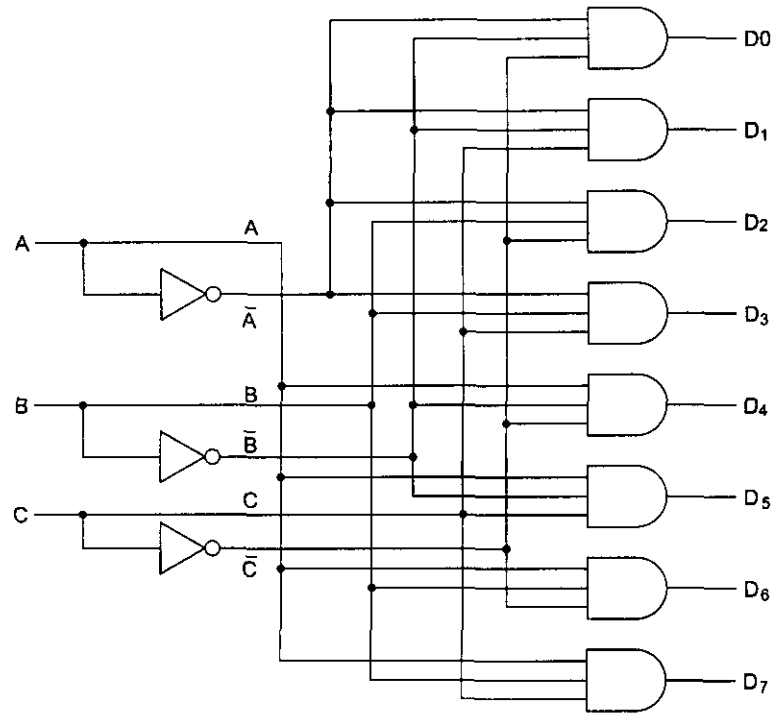


Рис. 3.12. Схема декодера, содержащего 3 входа и 8 выходов

Принцип работы схемы, изображенной на рис. 3.12, не сложен. Каждый вентиль И имеет три входа, из которых первый или A, или $\bar{A}$, второй или B, или $\bar{B}$, а третий или C, или $\bar{C}$. Каждый вентиль запускается различной комбинацией входов: Do — сочетанием A B C, Di — A $\bar{B}$ C и т. д.

Компараторы

Еще одна полезная схема — компаратор. Компаратор сравнивает два слова, которые поступают на вход. Компаратор, изображенный на рис. 3.13, принимает два входных сигнала, A и B, каждый длиной 4 бита, и выдает 1, если они равны, и O, если они не равны. Схема основывается на вентиле ИСКЛЮЧАЮЩЕЕ ИЛИ, который выдает 0, если сигналы на входе равны, и 1, если сигналы на входе не равны. Если все четыре входных слова равны, все четыре вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ должны выдавать 0. Эти четыре сигнала затем поступают в вентиль ИЛИ. Если в результате получается 0, значит, слова, поступившие на вход, равны; в противном случае они не равны. В нашем примере мы использовали вентиль ИЛИ в качестве конечной стадии, чтобы поменять значение полученного результата: 1 означает равенство, а 0 — неравенство.

Программируемые логические матрицы

Ранее мы рассказывали, что любую функцию (таблицу истинности) можно представить в виде суммы произведений и, следовательно, воплотить в схеме, исполь-

зуя вентили И и ИЛИ. Для вычисления сумм произведений служит так называемая **программируемая логическая матрица** (рис. 3.14). Эта микросхема содержит входы для 12 переменных. Дополнительные сигналы (инверсии) генерируются внутри самой микросхемы. В итоге всего получается 24 входных сигнала. Какой именно входной сигнал поступает в определенный вентиль И, определяется по матрице 24x50 бит. Каждая из входных линий к 50 вентилям И содержит плавкую перемычку. При выпуске с завода все 1200 перемычек остаются нетронутыми. Чтобы запрограммировать матрицу, покупатель выжигает выбранные перемычки, прикладывая к схеме высокое напряжение.

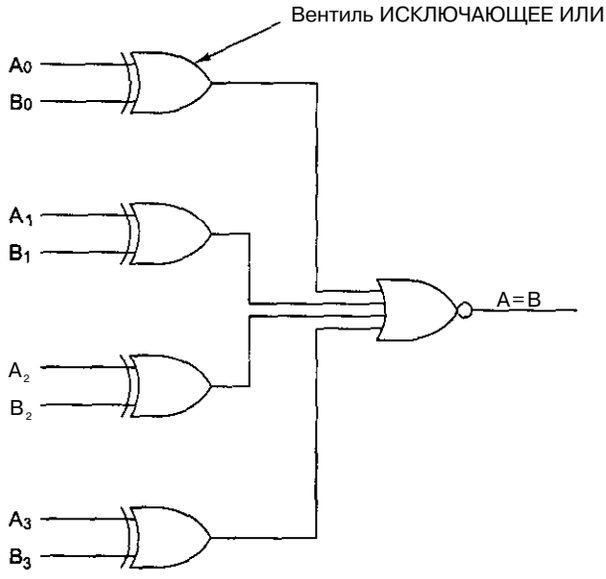


Рис. 3.13. Простой четырехразрядный компаратор

Выходная часть схемы состоит из шести вентилей ИЛИ, каждый из которых содержит до 50 входов, что соответствует наличию 50 выходов у вентилей И. Какие из потенциально возможных связей действительно существуют, зависит от того, как была запрограммирована матрица 50x6. Микросхема имеет 12 входных выводов, 6 выходных выводов, питание и «землю» (то есть всего 20 выводов).

Приведем пример использования программируемой логической матрицы. Рассмотрим схему, изображенную на рис. 3.3, *б*. Она содержит три входа, четыре вентиля И, один вентиль ИЛИ и три инвертора. Если запрограммировать нашу матрицу определенным образом, она сможет вычислять ту же функцию, используя три из 12 входов, четыре из 50 вентилей И и один из 6 вентилей ИЛИ. (Четыре вентиля И должны вычислять ABC, ABC, ABC И ABC; вентиль ИЛИ принимает эти 4 произведения в качестве входных данных.) Можно сделать так, чтобы та же программируемая логическая матрица вычисляла одновременно сумму четырех функций одинаковой сложности. Для простых функций ограничивающим фактором является число входных переменных, для более сложных — вентили И и ИЛИ.

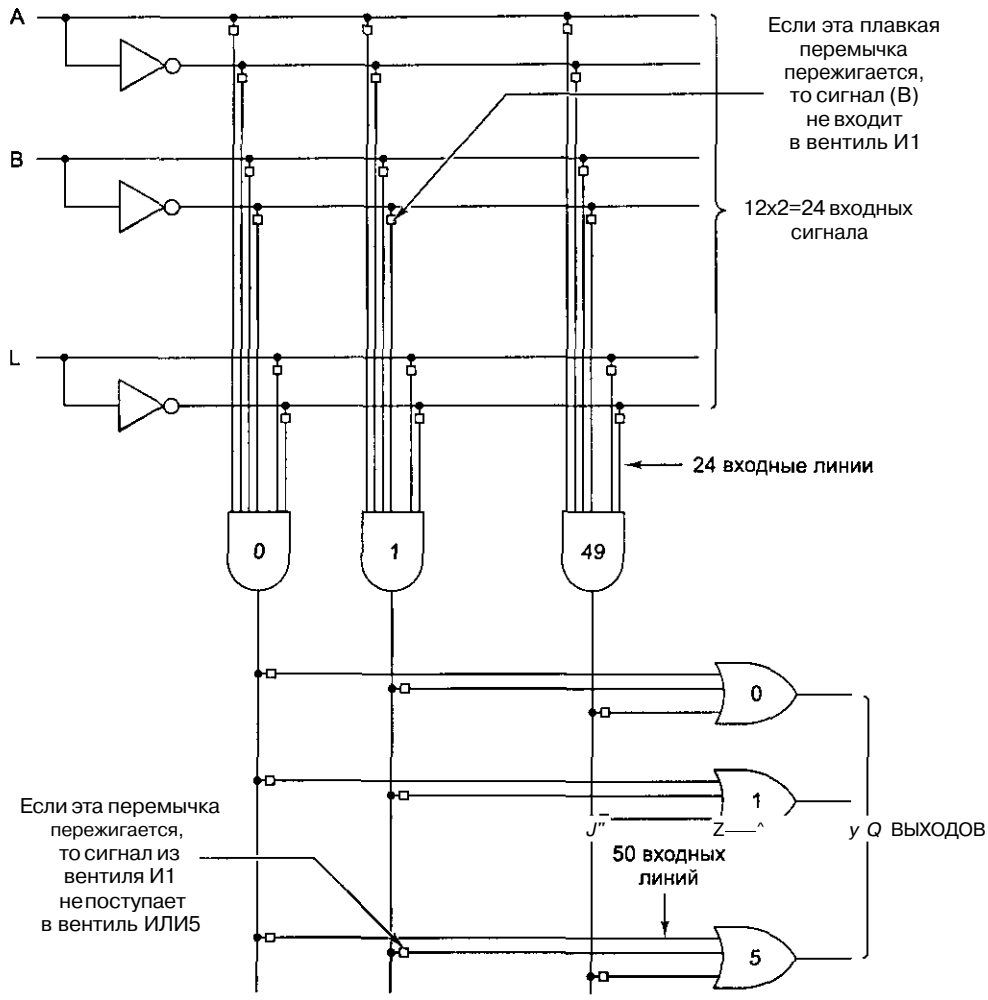


Рис. 3.14. Программируемая логическая матрица с 12 входами и 6 выходами. Маленькие квадратики — плавкие перемычки, выжигаемые для задания функции, которую нужно вычислить, Плавкие перемычки упорядочиваются в двух матрицах. Верхняя матрица — для вентилей И, а нижняя матрица — для вентилей ИЛИ

Матрицы, программируемые в условиях эксплуатации, все еще используются. Однако предпочтение отдается матрицам, которые изготавливаются на заказ. Они разрабатываются заказчиком и выпускаются производителем в соответствии с запросами заказчика. Такие программируемые логические матрицы гораздо дешевле.

А теперь мы можем обсудить три разных способа воплощения таблицы истинности, приведенной на рис. 3.3, *а*. Если в качестве компонентов использовать МИС, нам нужны 4 микросхемы. С другой стороны, мы можем обойтись одним мультиплексором, построенным на СИС, как показано на рис. 3.11, *б*. Наконец, мы можем использовать лишь четвертую часть программируемой логической матрицы. Очевидно, если необходимо вычислять много функций, использование программируе-

мой логической матрицы более эффективно, чем применение двух других методов. Для простых схем предпочтительнее более дешевые МИС и СИС.

Арифметические схемы

Перейдем от СИС общего назначения к комбинационным схемам СИС, которые используются для выполнения арифметических операций. Мы начнем с простой 8-разрядной схемы сдвига, затем рассмотрим структуру сумматоров и, наконец, изучим арифметико-логические устройства, которые играют существенную роль в любом компьютере.

Схемы сдвига

Первой арифметической схемой СИС, которую мы рассмотрим, будет схема сдвига, содержащая 8 входов и 8 выходов (рис. 3.15). Восемь входных битов подаются на линии $D_0,\ldots, D_7$. Выходные данные, которые представляют собой входные данные, сдвинутые на 1 бит, поступают на линии So,. •., S7. Линия управления С определяет направление сдвига: 0 — налево, 1 — направо.

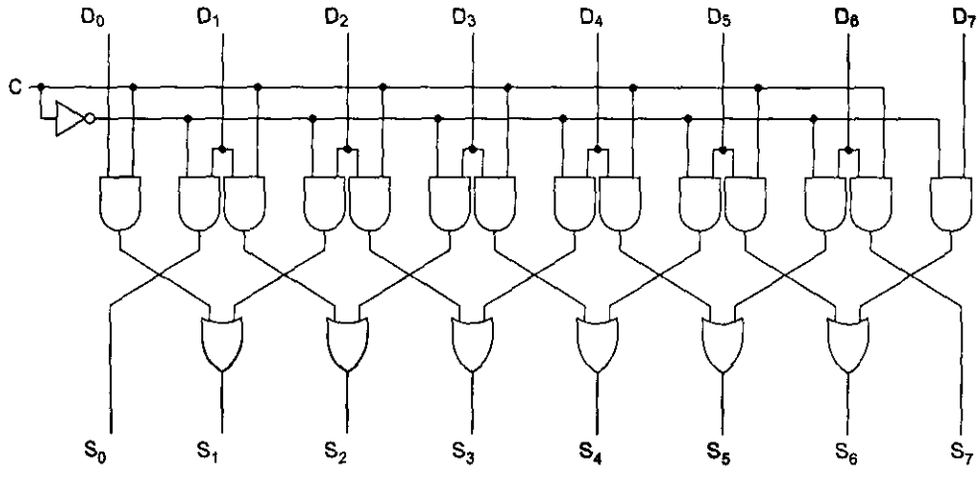


Рис. 3.15. Схема сдвига

Чтобы понять, как работает такая схема, рассмотрим пары вентилей И (кроме крайних вентилей) Если С=1, правый член каждой пары включается, пропуская через себя соответствующий бит. Так как правый вентиль И соединен с входом вентиля ИЛИ, который расположен справа от этого вентиля И, происходит сдвиг вправо. Если С=0, включается левый вентиль И из пары, и тогда происходит сдвиг влево.

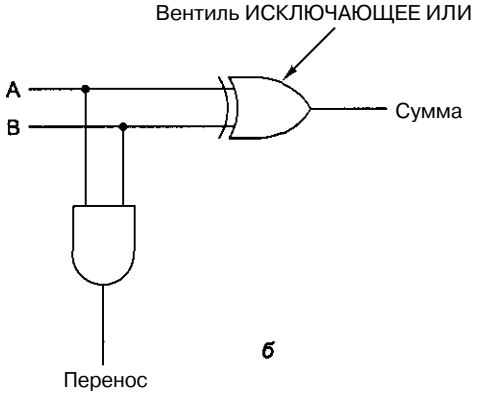
Сумматоры

Компьютер, который не умеет складывать целые числа, практически немыслим. Следовательно, схема для выполнения операций сложения является существенной частью любого процессора. Таблица истинности для сложения одноразряд-

ных целых чисел показана на рис. 3.16, а. Здесь имеется два результата: сумма входных переменных A и B и перенос на следующую (левую) позицию. Схема для вычисления бита суммы и бита переноса показана на рис. 3.16,*б*. Такая схема обычно называется полусумматором.

A	B	Сумма	Перенос
0	0	**0**	0
0	1	**1**	0
1	0	**1**	0
1	1	**0**	1

а



б

Рис. 3.16. Таблица истинности для сложения одноразрядных чисел (а); схема полусумматора (б)

Полусумматор подходит для сложения битов нижних разрядов двух многобитовых слов. Но он не годится для сложения битов в середине слова, потому что не может осуществлять перенос в эту позицию. Поэтому необходим **полный сумматор** (рис. 3.17). Из схемы должно быть ясно, что полный сумматор состоит из двух полусумматоров. Сумма равна 1, если нечетное число переменных A, B и *Вход переноса* принимает значение 1 (то есть если единице равна или одна из переменных, или все три). *Выход переноса* принимает значение 1, если или A и B одновременно равны 1 (левый вход в вентиль ИЛИ), или если один из них равен 1, а *Вход переноса* также равен 1. Два полусумматора порождают и биты суммы, и биты переноса.

Чтобы построить сумматор, например, для двух 16-битных слов, нужно продублировать схему, изображенную на рис. 3.17, *б,* 16 раз. Перенос производится в левый соседний бит. Перенос в самый правый бит соединен с 0. Такой сумматор называется **сумматором со сквозным переносом.** Прибавление 1 к числу 111... 111 не осуществится до тех пор, пока перенос не пройдет весь путь от самого правого бита к самому левому. Существуют более быстрые сумматоры, работающие без подобной задержки. Естественно, предпочтение обычно отдается им.

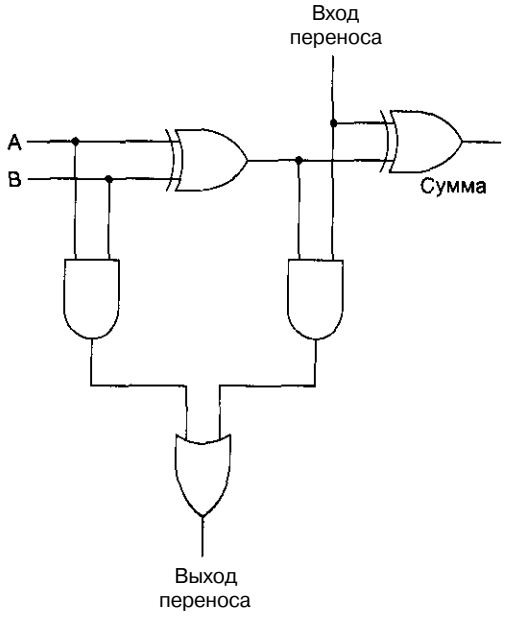
Рассмотрим пример более быстрого сумматора. Разобьем 32-разрядный сумматор на 2 половины: нижнюю 16-разрядную и верхнюю 16-разрядную. Когда начинается сложение, верхний сумматор еще не может приступить к работе, поскольку он не узнает значение переноса, пока не совершится 16 суммирований в нижнем сумматоре.

Однако можно сделать одно преобразование. Вместо одного верхнего сумматора можно получить два верхних сумматора, продублировав соответствующую часть аппаратного обеспечения. Тогда схема будет состоять из трех 16-разрядных сум-

маторов: одного нижнего и двух верхних U0 и U1, которые работают параллельно. В сумматор U0 в качестве переноса поступает 0, а в сумматор U1 в качестве переноса поступает 1. Оба верхних сумматора начинают работу одновременно с нижним сумматором, но только один из результатов суммирования в двух верхних сумматорах будет правильным. После сложения 16 нижних разрядов становится известно значение переноса в верхний сумматор, и тогда можно определить правильный ответ. При таком подходе время сложения сокращается в два раза. Такой сумматор называется **сумматором с выбором переноса.** Можно разбить каждый 16-разрядный сумматор на два 8-разрядных и т. д.

A	B	Вход переноса	Сумма	Выход переноса
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

а



б

Рис. 3.17. Таблица истинности для полного сумматора (а); схема для полного сумматора {б)

Арифметико-логические устройства

Большинство компьютеров содержат одну схему для выполнения операций И, ИЛИ и сложения над двумя машинными словами. Обычно такая схема для п-битных слов состоит из п идентичных схем для индивидуальных битовых позиций. На рис. 3.18 изображена такая схема, которая называется арифметико-логическим устройством, или АЛУ. Это устройство может вычислять одну из 4 следующих функций: А И В, А ИЛИ В, $\bar{B}$ и A+B. Выбор функции зависит от того, какие сигналы поступают на линии F_0 и F,: 00,01,10 или 11 (в двоичной системе счисления) Отметим, что здесь A+B означает арифметическую сумму А и В, а не логическую операцию И.

В левом нижнем углу схемы находится двухразрядный декодер, который порождает сигналы включения для четырех операций. Выбор операции определяет-

ся сигналами управления F_0 и Fj. В зависимости от значений F_0 и Fi выбирается одна из четырех линий разрешения, и тогда выходной сигнал выбранной функции проходит через последний вентиль ИЛИ.

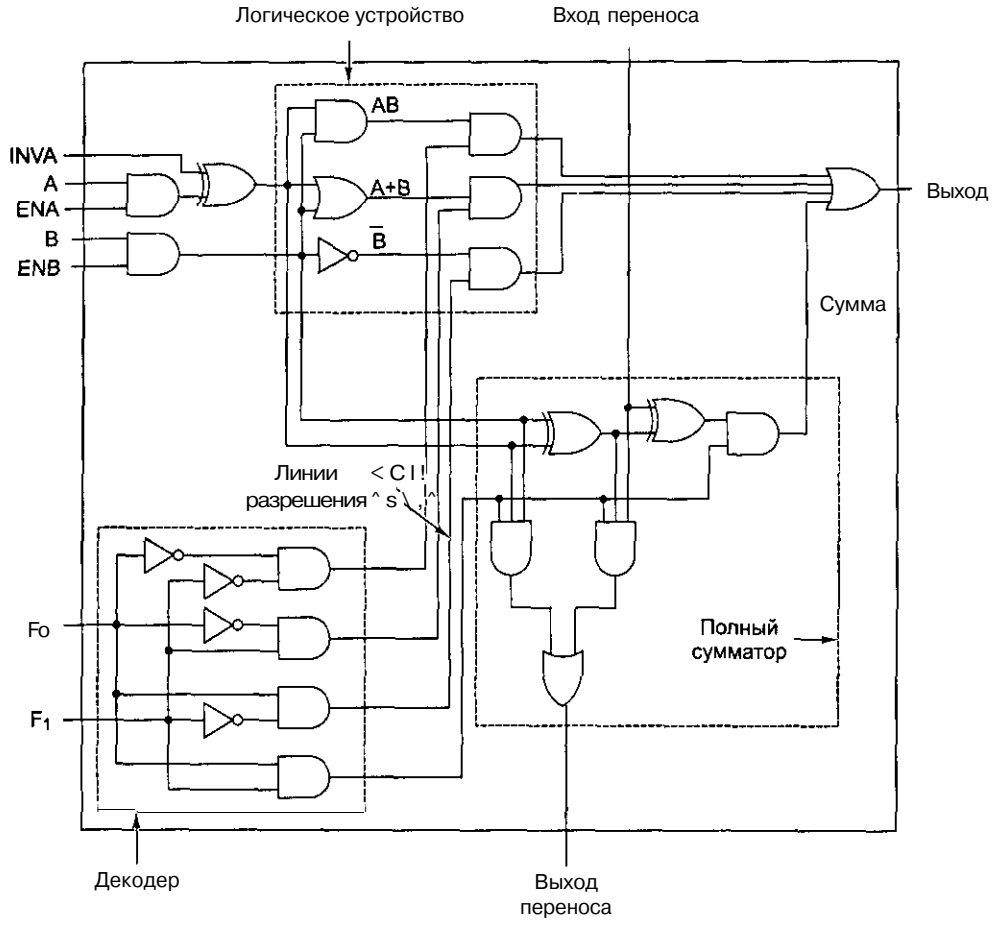


Рис. 3.18. Одноразрядное АЛУ

В верхнем левом углу схемы находится логическое устройство для вычисления А И В, А ИЛИ В и $\overline{B}$, но по крайней мере один из этих результатов проходит через последний вентиль ИЛИ в зависимости от того, какую из разрешающих линий выбрал декодер. Так как ровно один из выходных сигналов декодера будет равен 1, то и запускаться будет ровно один из четырех вентилей И. Остальные три вентиля будут выдавать 0 независимо от значений А и В.

АЛУ может выполнять не только логические и арифметические операции над А и В, но и делать их равными нулю, отрицая ENA (сигнал разрешения А) или ENB (сигнал разрешения В). Можно также получить *X,* установив INVA (инверсию А). Зачем нужны ENA, ENB и INVA, мы рассмотрим в главе 4. При нормаль-

ных условиях и ENA, и ENB равны 1, чтобы разрешить поступление обоих входных сигналов, а сигнал INVA равен 0. В этом случае A и B просто поступают в логическое устройство без изменений.

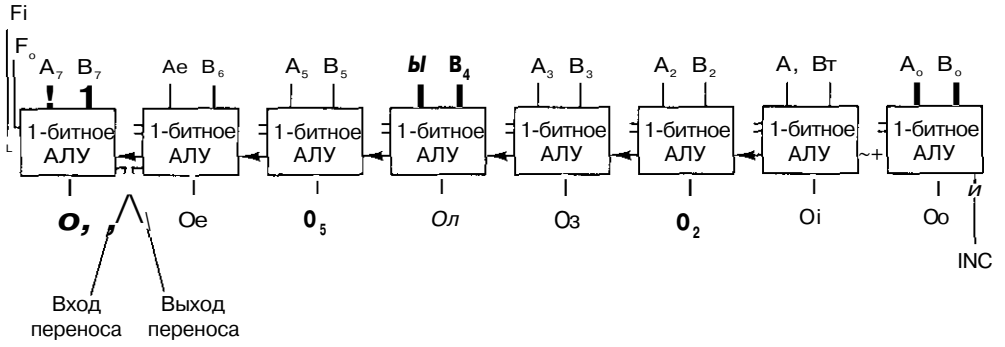


Рис. 3-19. Восемь одноразрядных секций, соединенных в 8-разрядное АЛУ Сигналы разрешения и инверсии не показаны для упрощения схемы

В нижнем правом углу находится полный сумматор для подсчета суммы A и B и для осуществления переносов. Переносы необходимы, поскольку несколько таких схем могут быть соединены для выполнения операций над целыми словами. Одноразрядные схемы, подобные той, которая изображена на рис. 3.18, называются разрядными микропроцессорными секциями. Они позволяют разработчику сконструировать АЛУ любой желаемой ширины. На рис. 3.19 показана схема 8-разрядного АЛУ, составленного из восьми **одноразрядных секций.** Сигнал INC (увеличение на единицу) нужен только для операций сложения. Он дает возможность вычислять такие суммы, как A+1 и A+B+1.

Тактовые генераторы

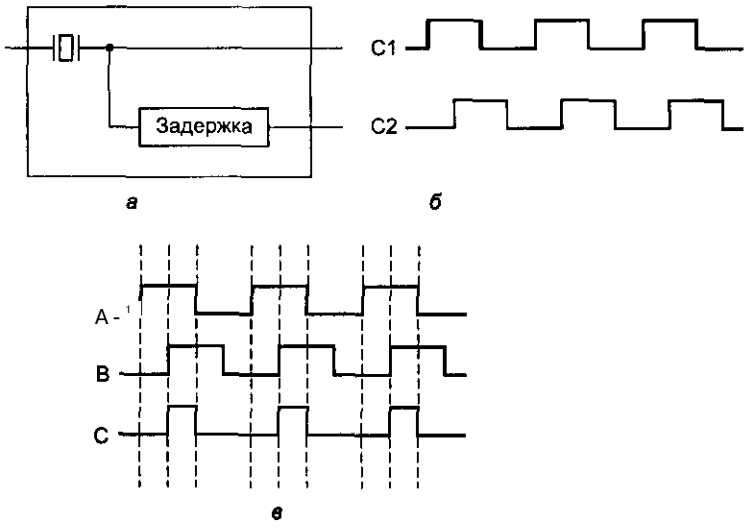
Во многих цифровых схемах все зависит от порядка, в котором выполняются действия. Иногда одно действие должно предшествовать другому, иногда два действия должны происходить одновременно. Для контроля временных отношений в цифровые схемы встраиваются тактовые генераторы, чтобы обеспечить синхронизацию. **Тактовый генератор** — это схема, которая вызывает серию импульсов. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется временем такта. Частота импульсов обычно от 1 до 500 МГц, что соответствует времени такта от 1000 не до 2 не. Частота тактового генератора обычно контролируется кварцевым генератором, чтобы достичь высокой точности.

В компьютере за время одного такта может произойти много событий. Если они должны осуществляться в определенном порядке, то такт следует разделить на подтакты. Чтобы достичь лучшего разрешения, чем у основного тактового генератора, нужно сделать ответвление от задающей линии тактового генератора и вставить схему с определенным временем задержки. Таким образом порождается

вторичный сигнал тактового генератора, который сдвинут ио фазе относительно первичного (рис 3 20, а) Временная диаграмма (рис 3 20, б) обеспечивает четыре начала отсчета времени для дискретных событии

1 Нарастающий фронт С1
2 Задний фронт С1
3 Нарастающий фронт С2
4 Задний фронт С2

Связав различные события с различными фронтами, можно достичь требуемой последовательности выполнения действий Если в пределах одного такта требуется более четырех начал отсчета, можно сделать еще несколько ответвлений от задающей линии с различным временем задержки



Рис, 3.20. Тактовый генератор {а), временная диаграмма для тактового генератора (б), порождение асинхронных тактовых импульсов {в)

В некоторых схемах важны временные интервалы, а не дискретные моменты времени Например, некоторое событие может происходить в любое время, когда уровень импульса С1 высокий, а не на нарастающем фронте Другое событие может происходить только в том случае, когда уровень импульса С2 высокий Если необходимо более двух интервалов, нужно обеспечить больше линий передачи синхронизирующих импульсов или сделать так, чтобы состояния с высоким уровнем импульса у двух тактовых генераторов частично пересекались во времени. В последнем случае можно выделить 4 отдельных интервала СТ И С^, С1 И С2, С1 И С2 и СТ И С2

Тактовые генераторы могут быть синхронными В этом случае время состояния с высоким уровнем импульса равно времени состояния с низким уровнем импульса (рис 3 20, *б)* Чтобы получить асинхронную серию импульсов, нужно сдвинуть сигнал задающего генератора, используя цепь задержки Затем нужно

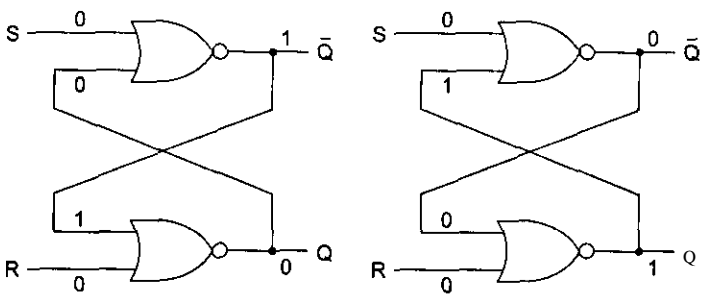
соединить полученный сигнал с изначальным сигналом с помощью логической функции И (см. рис. 3.20, *в,* сигнал С),

Память

Память является необходимым компонентом любого компьютера. Без памяти не было бы компьютеров, по крайней мере таких, какие есть сейчас. Память используется как для хранения команд, которые нужно выполнить, так и данных. В следующих разделах мы рассмотрим основные компоненты памяти, начиная с уровня вентилей. Мы увидим, как они работают и как из них можно получить память большой емкости.

Защелки

Чтобы создать один бит памяти, нам нужна схема, которая каким-то образом «запоминает» предыдущие входные значения. Такую схему можно сконструировать из двух вентилей НЕ-ИЛИ, как показано на рис. 3.21, *а.* Аналогичные схемы можно построить из вентилей НЕ-И. Мы не будем упоминать эти схемы в дальнейшем, поскольку они, по существу, идентичны схемам с вентилями НЕ-ИЛИ.



A	B	НЕ-ИЛИ
0	0	1
0	1	0
1	0	0
1	1	0

Рис. 3.21. Защелка НЕ-ИЛИ в состоянии 0 (а); защелка НЕ-ИЛИ в состоянии 1 *(б);* таблица истинности для функции НЕ-ИЛИ (й)

Схема, изображенная на рис. 3.21, *а,* называется SR-защелкой. У нее есть два входа: S (setting — установка) и R (resetting — сброс). У нее также есть два комплементарных[1] (дополнительных) выхода: Q и $\overline{Q}$. В отличие от комбинационной схемы, выходные сигналы защелки не определяются текущими входными сигналами.

Чтобы увидеть, как это осуществляется, предположим, что S=0 и R=0 (вообще они равны 0 большую часть времени). Чтобы провести доказательство, предположим также, что Q=0. Так как Q возвращается в верхний вентиль НЕ-ИЛИ и оба входа этого вентиля равны 0, то его выход, Q, равен 1. Единица возвращается в нижний вентиль, у которого в итоге один вход равен 0, а другой — 1, а на выходе получается Q=0. Такое положение вещей, по крайней мере, состоятельно (рис. 3.21, *а).*

[1] От англ. *complementary* — дополняющий. — *Примеч. пер.*

А теперь давайте представим, что Q=1, а R и S все еще равны 0. Верхний вентиль имеет входы 0 и 1 и выход $\overline{Q}$ (то есть 0), который возвращается в нижний вентиль. Такое положение вещей, изображенное на рис. 3.21, *б,* также состоятельно. Положение, когда оба выхода равны 0, несостоятельно, поскольку в этом случае оба вентиля имели бы на входе два нуля, что привело бы к единице на выходе, а не к нулю. Точно так же невозможно иметь оба выхода равных 1, поскольку это привело бы к входным сигналам 0 и 1, что вызывает на выходе 0, а не 1. Наш вывод прост: при R=S=0 защелка имеет два стабильных состояния, которые мы будем называть 0 и 1 в зависимости от Q.

А сейчас давайте рассмотрим действие входных сигналов на состояние защелки. Предположим, что S принимает значение 1, в то время как Q=0. Тогда входные сигналы верхнего вентиля будут 1 и 0, что приведет к выходному сигналу $\overline{Q}$=0. Это изменение делает оба входа в нижний вентиль равными 0 и, следовательно, выходной сигнал равным 1. Таким образом, установка S на значение 1 переключает состояние с 0 на 1. Установка R на значение 1, когда защелка находится в состоянии 0, не вызывается изменений, поскольку выход нижнего вентиля НЕ-ИЛИ равен 0 и для входов 10, и для входов 11.

Используя подобную аргументацию, легко увидеть, что установка S на значение 1 при состоянии защелки 1 (то есть при Q=1) не вызывает изменений, но установка R на значение 1 приводит к изменению состояния защелки. Таким образом, если S принимает значение 1, то Q будет равно 1 независимо от предыдущего состояния защелки. Сходным образом переход R на значение 1 вызывает Q=0. Схема «запоминает», какой сигнал был в последний раз: S или R. Используя это свойство, мы можем конструировать компьютерную память.

Синхронные SR-защелки

Часто бывает удобно сделать так, чтобы защелка меняла состояние только в определенные моменты. Чтобы достичь этой цели, мы немного изменили основную схему и получили **синхронную SR-защелку** (рис. 3.22).

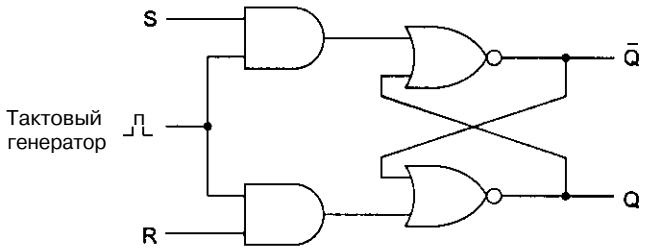


Рис. 3.22. Синхронная SR-защелка

Эта схема имеет дополнительный синхронизирующий вход, который обычно равен 0. Если этот вход равен 0, то оба выхода вентилей И равны 0 независимо от S и R, и защелка не меняет состояние. Когда значение синхронизирующего входа равно 1, действие вентилей И исчезает и состояние защелки становится зависимым от S и R. Для обозначения того факта, что синхронизирующий вход равен 1

(то есть состояние схемы зависит от значений S и R), часто используется термин **стробировать.**

До сих пор мы скрывали, что происходит, если S=R=1. И по понятным причинам: когда и R, и S в конце концов возвращаются к 0, схема становится недетерминированной. Единственное состоятельное положение при S=R=1 — это Q=Q=0, но как только оба входа возвращаются к 0, защелка должна перейти в одно из двух стабильных состояний. Если один из входов принимает значение 0 раньше, чем другой, оставшийся в состоянии 1 «побеждает», потому что когда один из входов равен 1, он управляет состоянием защелки. Если оба входа переходят к 0 одновременно (что маловероятно), защелка переходит в одно из своих состояний наугад.

Синхронные D-защелки

Чтобы разрешить неопределенность SR-защелки (неопределенность возникает в случае, если S=R=1), нужно предотвратить появление подобной неопределенности. На рис. 3.23 изображена схема защелки только с одним входом D. Так как входной сигнал в нижний вентиль И всегда является обратным кодом входного сигнала в верхний вентиль И, ситуация, когда оба входа равны 1, никогда не возникает. Когда D=1 и синхронизирующий вход равен 1, защелка переходит в состояние Q=1. Когда D=0 и синхронизирующий вход равен 1, защелка переходит в состояние Q=0. Другими словами, когда синхронизирующий вход равен 1, текущее значение D отбирается и сохраняется в защелке. Такая схема, которая называется **синхронной** D-защелкой, представляет собой память объемом 1 бит. Значение, которое было сохранено, всегда доступно на выходе Q. Чтобы загрузить в память текущее значение D, нужно пустить положительный импульс по линии синхронизирующего сигнала.

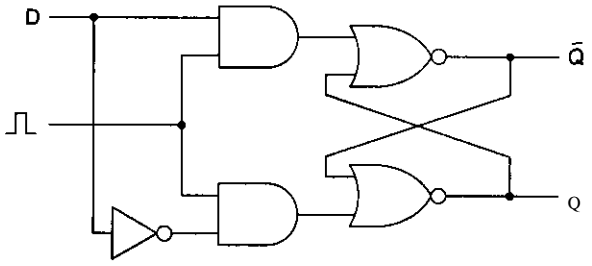


Рис. 3.23. Синхронная D-защелка

Такая схема требует наличия 11 транзисторов. Более сложные схемы могут хранить 1 бит, имея всего 6 транзисторов. На практике обычно используются последние.

Триггеры (flip-flops)

Многие схемы выбирают значение на определенной линии в определенный момент времени и запоминают его. В такой схеме, которая называется **триггером,**

переход состояния происходит не тогда, когда синхронизирующий сигнал равен 1, а во время перехода синхронизирующего сигнала с 0 на 1 (нарастающий фронт) или с 1 на 0 (задний фронт). Следовательно, длина синхронизирующего импульса не имеет значения, поскольку переходы происходят быстро.

Подчеркнем еще раз различие между триггером и защелкой. Триггер запускается фронтом сигнала, а защелка запускается уровнем сигнала. Обратите внимание, что в литературе эти термины часто путаются. Многие авторы используют термин «триггер», когда речь идет о защелке, и наоборот[1].

Существует несколько подходов к разработке триггеров. Например, если бы существовал способ генерирования очень короткого импульса на нарастающем фронте синхронизирующего сигнала, этот импульс можно было бы подавать в D-защелку. В действительности такой способ существует. Соответствующая схема показана на рис. 3.24, *а*.

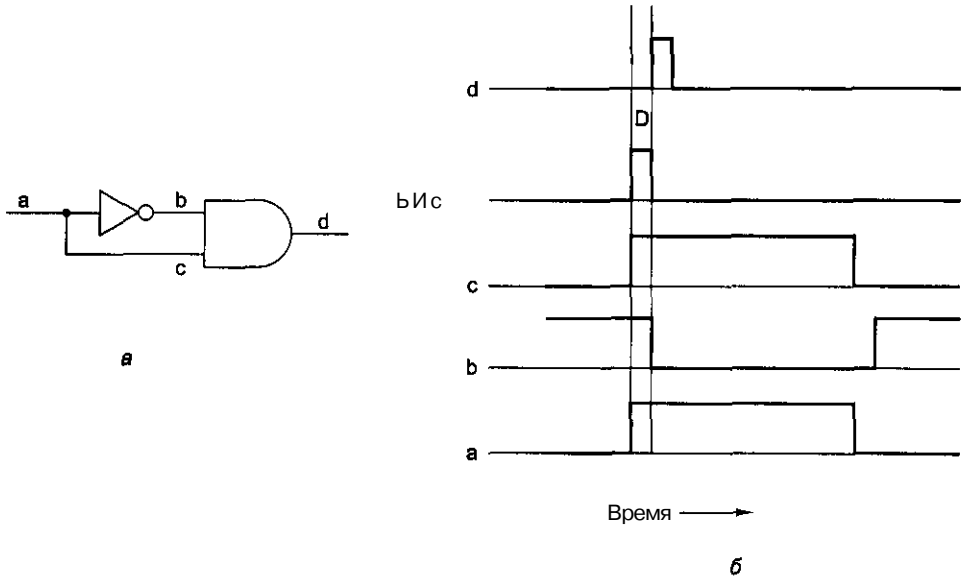


Рис. 3.24. Генератор импульса (а); временная диаграмма для четырехточекнасхеме(б)

На первый взгляд может показаться, что выход вентиля И всегда будет нулевым, поскольку функция И от любого сигнала с его инверсией дает 0, но на самом деле ситуация несколько более тонкая. При прохождении сигнала через инвертор происходит небольшая, но все-таки не нулевая задержка. Данная схема работает именно благодаря этой задержке. Предположим, что мы измеряем напряжение в четырех точках a, Ь, с и d. Входовый сигнал в точке a представляет собой длинный синхронизирующий импульс (см. нижний график на рис. 3.24, *б)*. Сигнал в точке b показан над ним. Отметим, что этот сигнал инвертирован и подается с некоторой

[1] В отечественной литературе термин «защелка* (latch) не используется, і оворят о триггерах, Однако при этом вводится понятие Т-триггера, который здесь называется настоящим триггером. — *Примеч. научн.ред*

задержкой. Время задержки зависит от типа инвертора и обычно составляет несколько наносекунд.

Сигнал в точке с тоже подается с задержкой, но эта задержка обусловлена только временем прохождения сигнала (со скоростью света). Если физическое расстояние между a и c, например, 20 микрон, тогда задержка на распространение сигнала составляет 0,0001 не, что, конечно, незначительно по сравнению со временем, которое требуется на прохождение сигнала через инвертор. Таким образом, сигнал в точке с практически идентичен сигналу в точке a.

Когда входные сигналы b и c подвергаются операции И, в результате получается короткий импульс, длина которого (Д) равна вентильной задержке инвертора (обычно 5 не и меньше). Выходной сигнал вентиля И — данный импульс, сдвинутый из-за задержки вентиля И (см. верхний график на рис. 3 24, *б).* Этот временной сдвиг означает только то, что D-защелка активизируется с определенной задержкой после нарастающего фронта синхронизирующего импульса Он никак не влияет на длину импульса. В памяти со временем цикла в 50 не импульс в 5 не (который сообщает, когда нужно выбирать линию D) достаточно короткий, и в этом случае полная схема может быть такой, какая изображена на рис. 3.25. Следует упомянуть, что такая схема триггера проста для понимания, но на практике обычно используются более сложные триггеры.

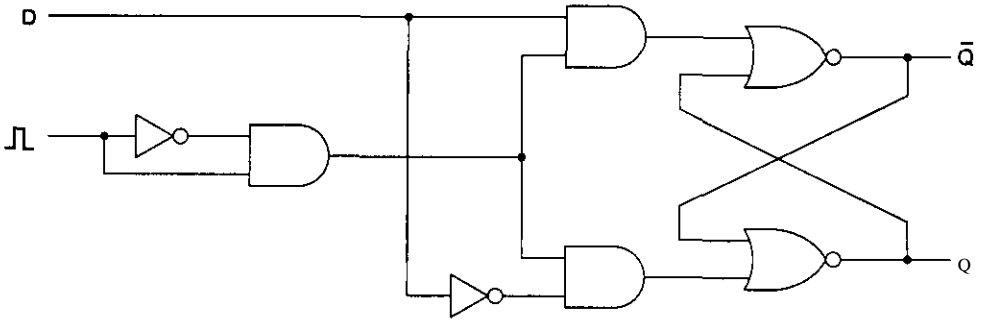
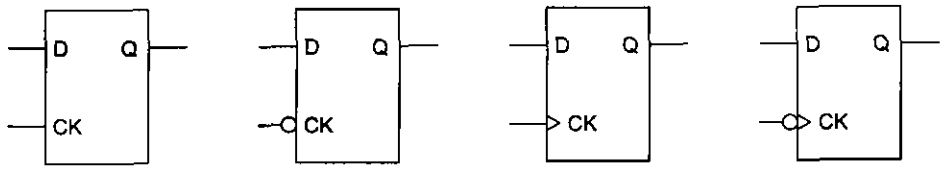


Рис. 3.25. D-триггер

Стандартные изображения защелок и триггеров показаны на рис 3 26 На рис.3 26, *а* изображена защелка, состояние которой загружается тогда, когда синхронизирующий сигнал CK (от слова clock) равен 1, в противоположность защелке, изображенной на рис. 3 26,*б,* у которой синхронизирующий сигнал обычно равен 1, но переходит на 0, чтобы загрузить состояние из D. На рис. 3.26, *виг* изображены триггеры. То, что это триггеры, а не защелки, показано с помощью уголка при синхронизирующем входе. Триггер на рис. 3.26, *в* изменяет состояние на возрастающем фронте синхронизирующего импульса (переход от 0 к 1), тогда как триггер на рис 3 26, *г* изменяет состояние на заднем фронте (переход от 0 к 1). Многие (хотя не все) защелки и триггеры также имеют выход £У, а у некоторых есть два дополнительных входа. *Set* (установка) или *Preset* (предварительная установка) и *Reset* (сброс) или *Clear* (очистка). Первый вход *(Set* или *Preset)* устанавливает Q=1, а второй *(Reset или Clear)* — Q=0.



а б в *г*

Рис. 3.26. D-защелки и D-триггеры

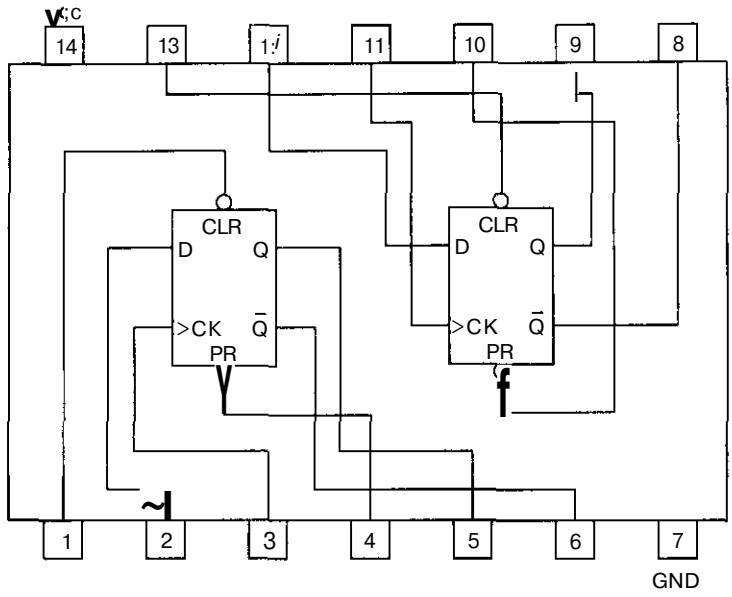
Регистры

Существуют различные конфигурации триггеров. На рисунке 3.27, *а* изображена схема, содержащая два независимых D-триггера с сигналами предварительной установки и очистки. Хотя эти два триггера находятся на одной микросхеме с 14 выводами, они не связаны между собой. Совершенно по-другому устроен восьмиразрядный триггер, изображенный на рис. 3.27, *б*. Здесь, в отличие от предыдущей схемы, у восьми триггеров нет выхода Q и линий предварительной установки и все синхронизирующие линии связаны вместе и управляются выводом 11. Сами триггеры того же типа, что на рис. 3.26, *г,* но инвертирующие входы аннулируются инвертором, связанным с выводом 11, поэтому триггеры запускаются при переходе от 0 к 1. Все восемь сигналов очистки также объединены, поэтому когда вывод 1 переходит в состояние 0, все триггеры также переходят в состояние 0. Если вам не понятно, почему вывод 11 инвертируется на входе, а затем инвертируется снова при каждом сигнале CK, то ответ прост: входной сигнал не имеет достаточной мощности, чтобы запустить все восемь триггеров; входной инвертор на самом деле используется в качестве усилителя.

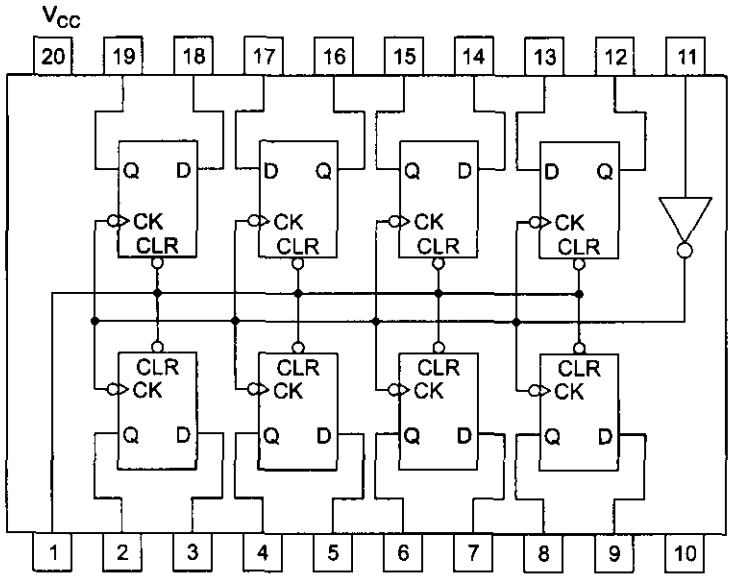
Одна из причин объединения линий синхронизации и линий очистки в микросхеме на рис. 3.27, *б* ~ экономия выводов. С другой стороны, микросхема данной конфигурации несколько отличается от восьми несвязанных триггеров. Эта микросхема используется в качестве одного 8-разрядного регистра. Две такие микросхемы могут работать параллельно, образуя 16-разрядный регистр. Для этого нужно связать соответствующие выводы 1 и 11. Регистры и их применение мы рассмотрим более подробно в главе 4.

Организация памяти

Хотя мы и совершили переход от простой памяти в 1 бит (см. рис. 3.23) к 8-разрядной памяти (см. рис. 3.27, *б),* чтобы построить память большого объема, требуется другой способ организации, при котором можно обращаться к отдельным словам. Пример организации памяти, которая удовлетворяет этому критерию, показан на рис. 3.28. Эта память содержит четыре 3-битных слова. Каждая операция считывает или записывает целое 3-битное слово. Хотя общий объем памяти (12 битов) ненамного больше, чем у нашего 8-разрядного триггера, такая память требует меньшего количества выводов, и, что особенно важно, подобная организация применима при построении памяти большого объема.



а



б

Рис. 3.27. Два D-триггера (а); восьмиразрядный триггер (б)

Хотя структура памяти, изображенная на рис. 3.28, может на первый взгляд показаться сложной, на самом деле она очень проста благодаря своей регулярной структуре. Она содержит 8 входных линий (3 входа для данных — 1_0, Ь и 1_2; 2 входа

для адресов — Ao и A,; 3 входа для управления — CS (Chip Select — выбор элемента памяти), RD (для различия между считыванием и записью) и OE (Output Enable — разрешение выдачи выходных сигналов)) и 3 выходные линии для данных — Oo, O_t и Or- Такую память в принципе можно поместить в корпус с 14 выводами (включая питание и «землю»), а 8-разрядный триггер требует наличия 20 выводов.

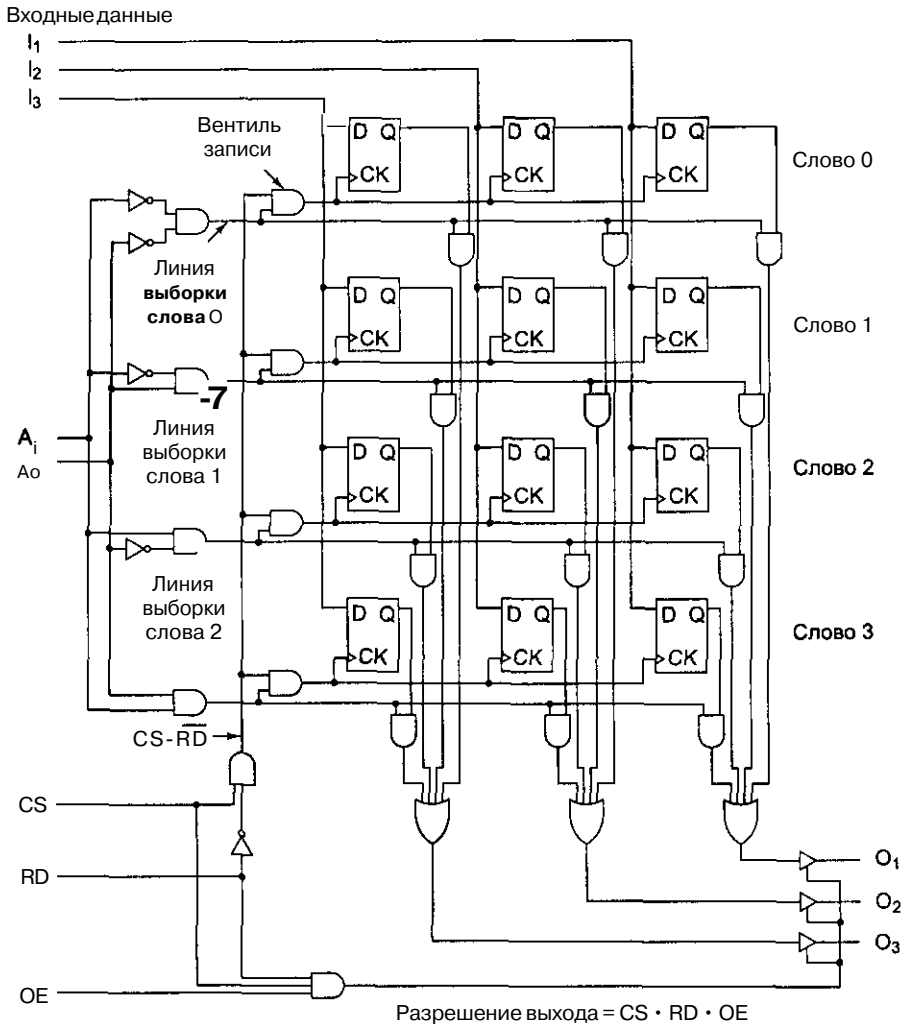


Рис. 3.28. Логическая блок-схема для памяти 4X3. Каждый ряд представляет одно из 3-битных слов. При операции считывания и записи всегда считывается или записывается целое слово

Чтобы выбрать микросхему памяти, внешняя логика должна установить CS на 1, а также установить RD на 1 для чтения и на 0 для записи. Две адресные линии должны указывать, какое из четырех 3-битных слов нужно считывать или записывать. При операции считывания входные линии для данных не используются.

Выбирается слово и помещается на выходные линии для данных. При операции записи биты, находящиеся на входных линиях для данных, загружаются в выбранное слово памяти; выходные линии при этом не используются.

А теперь давайте посмотрим, как работает память, изображенная на рис. 3.28. Четыре вентиля И для выбора слов в левой части схемы формируют декодер. Входные инверторы расположены так, что каждый вентиль запускается определенным адресом. Каждый вентиль приводит в действие линию выбора слов (для слов 0, 1, 2 и 3). Когда микросхема должна производить запись, вертикальная линия CS • 1Ш получает значение 1, запуская один из 4 вентилей записи. Выбор вентиля зависит от того, какая именно линия выбора слов равна 1. Выходной сигнал вентиля записи приводит в действие все сигналы СК для выбранного слова, загружая входные данные в триггеры для этого слова. Запись производится только в том случае, если CS равно 1, а RD равно 0, при этом записывается только слово, выбранное адресами Ао и Аь остальные слова не меняются

Процесс считывания сходен с процессом записи. Декодирование адреса происходит точно так же, как и при записи. Но в данном случае линия CS • $\overline{RD}$ принимает значение 0, поэтому все вентили записи блокируются и ни один из триггеров не меняется. Вместо этого линия выбора слов запускает вентили И, связанные с битами Q выбранного слова Таким образом, выбранное слово передает свои данные в четырехвходовые вентили ИЛИ, расположенные в нижней части схемы, а остальные три слова выдают 0. Следовательно, выход вентилей ИЛИ идентичен значению, сохраненному в данном слове. Остальные три слова никак не влияют на выходные данные.

Мы могли бы разработать схему, в которой три вентиля ИЛИ соединялись бы с тремя линиями вывода данных, но это вызвало бы некоторые проблемы. Мы рассматривали линии ввода данных и линии вывода данных как разные линии На практике же используются одни и те же линии. Если бы мы связали вентили ИЛИ с линиями вывода данных, микросхема пыталась бы выводить данные (то есть задавать каждой линии определенную величину) даже в процессе записи, мешая нормальному вводу данных. По этой причине желательно каким-то образом соединять вентили ИЛИ с линиями вывода данных при считывании и полностью разъединять их при записи. Все, что нам нужно, — электронный переключатель, который может устанавливать и разрушать связь за несколько наносекунд.

К счастью, такие переключатели существуют На рис. 3 29, *а* показано символическое изображение так называемого **буферного элемента без инверсии.** Он содержит вход для данных, выход для данных и вход управления. Когда вход управления равен 1, буферный элемент работает как провод (см. рис. 3.29, *б).* Когда вход управления равен 0, буферный элемент работает как разомкнутая цепь (см рис. 3.29, б), как будто кто-то отрезал выход для данных от остальной части схемы кусачками. Соединение может быть восстановлено за несколько наносекунд, если сделать сигнал управления равным 1.

На рис. 3.29, г показан **буферный** элемент с **инверсией,** который действует как обычный инвертор, когда сигнал управления равен 1, и отделяет выход от остальной части схемы, когда сигнал управления равен 0. Оба буферных элемента представляют собой **устройства с тремя состояниями,** поскольку они могут выдавать 0,1 или вообще не выдавать сигнала (в случае с разомкнутой цепью). Буфер-

ные элементы, кроме того, усиливают сигналы, поэтому они могут справляться с большим количеством сигналов одновременно. Иногда они используются в схемах именно по этой причине, даже если их свойства переключателя не нужны.

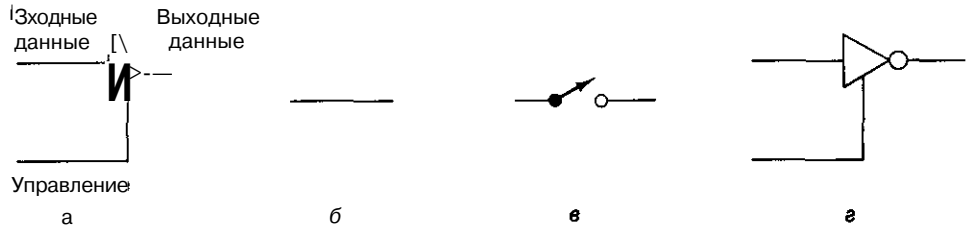


Рис. 3.29. Буферный элемент без инверсии (а); действие буферного элемента без инверсии, когда сигнал управления равен 1 (б); действие буферного элемента без инверсии, когда сигнал управления равен 0 {s); буферный элемент с инверсией (г)

Сейчас уже должно быть понятно, для чего нужны три буферных элемента без инверсии на линиях вывода данных. Когда CS, RD и OE все равны 1, то сигнал разрешения выдачи выходных данных также равен 1, в результате чего запускаются буферные элементы и слово помещается на выходные линии. Когда один из сигналов CS, RD и OE равен 0, выходы отсоединяются от остальной части схемы.

Микросхемы памяти

Преимущество памяти, изображенной на рис. 3.28, состоит в том, что подобная структура применима при разработке памяти большого объема. Мы нарисовали схему 4x3 (для 4 слов по 3 бита каждое). Чтобы расширить ее до размеров 4x8, нужно добавить еще 5 колонок триггеров по 4 триггера в каждой, а также 5 входных и 5 выходных линий. Чтобы перейти от размера 4x3 к размеру 8x3, мы должны добавить еще четыре ряда триггеров по три триггера в каждом, а также адресную линию A_2. При такой структуре число слов в памяти должно быть степенью двойки для максимальной эффективности, а число битов в слове может быть любым.

Поскольку технология изготовления интегральных схем хорошо подходит для производства микросхем с внутренней структурой повторяемой плоской поверхности, микросхемы памяти являются идеальным применением для этого. С развитием технологии число битов, которое можно вместить в одной микросхеме, постоянно увеличивается, обычно в два раза каждые 18 месяцев (закон Мура). С появлением больших микросхем маленькие микросхемы не всегда устаревают из-за компромиссов между преимуществами емкости, скорости, мощности, цены и сопряжения. Обычно самые большие современные микросхемы пользуются огромным спросом и, следовательно, стоят гораздо дороже за 1 бит, чем микросхемы небольшого размера.

При любом объеме памяти существует несколько различных способов организации микросхемы. На рис. 3.30 показаны две возможные структуры микросхемы в 4 Мбит: 512 Kx8 и 4096 Kx1. (Размеры микросхем памяти обычно даются в битах, а не в байтах, поэтому здесь мы будем придерживаться этого соглашения.) На рис. 3.30, *а* можно видеть 19 адресных линий для обращения к одному из 2^{19} байтов и 8 линий данных для загрузки или хранения выбранного байта.

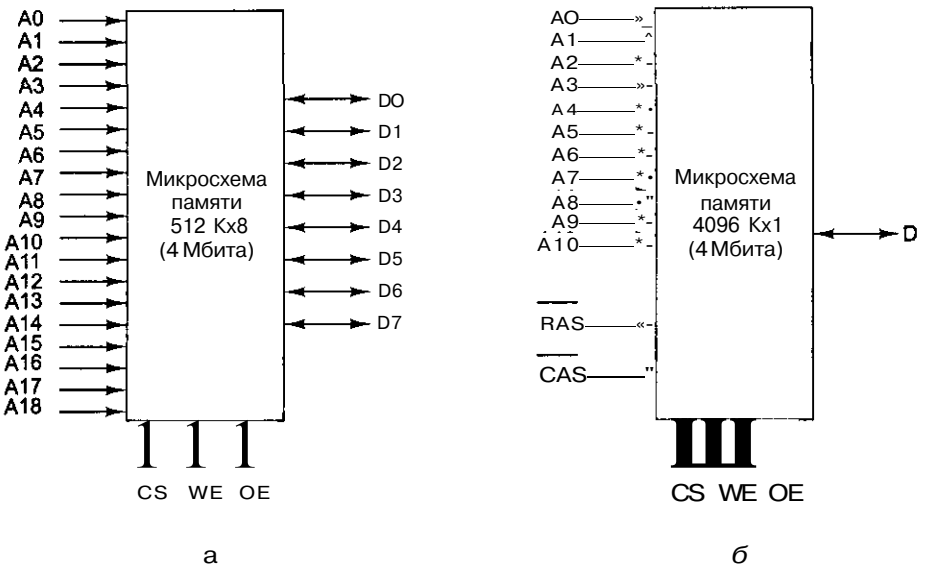


Рис. 3.30. Два способа организации памяти объемом 4 Мбит

Сделаем небольшое замечание по поводу терминологии. На одних выводах высокое напряжение вызывает какое-либо действие, на других — низкое напряжение. Чтобы избежать путаницы, мы будем употреблять термин **«установить сигнал»-,** когда вызывается какое-то действие, вместо того чтобы говорить, что напряжение повышается или понижается. Таким образом, для одних выводов установка сигнала значит установку на 1, а для других — установку на 0. Названия выводов, которые устанавливаются на 0, содержат сверху черту. Сигнал CS устанавливается на 1, а сигнал $\overline{CS}$ — на 0. Противоположный термин — «сбросить».

А теперь вернемся к нашей микросхеме. Поскольку обычно компьютер содержит много микросхем памяти, нужен сигнал для выбора необходимой микросхемы, такой, чтобы нужная нам микросхема реагировала на вызов, а остальные нет. Сигнал $\overline{CS}$ (Chip Select — выбор элемента памяти) используется именно для этой цели. Он устанавливается, чтобы запустить микросхему. Кроме того, нужен способ отличия считывания от записи. Сигнал $\overline{WE}$ (Write Enable — разрешение записи) используется для указания того, что данные должны записываться, а не считываться. Наконец, сигнал $\overline{OE}$ (Output Enable — разрешение выдачи выходных сигналов) устанавливается для выдачи выходных сигналов. Когда этого сигнала нет, выход отсоединен от остальной части схемы.

На рис. 3.30, *б* используется другая схема адресации. Микросхема представляет собой матрицу 2048x2048 однобитных ячеек, что составляет 4 Мбит. Чтобы обратиться к микросхеме, сначала нужно выбрать строку. Для этого И-битный номер этой строки подается на адресные выводы. Затем устанавливается сигнал $\overline{RAS}$ (Row Address Strobe — строб адреса строки). После этого на адресные выводы подается номер столбца и устанавливается сигнал CAS (Column Address Strobe — строб адреса столбца). Микросхема реагирует на сигнал, принимая или выдавая 1 бит данных.

Большие микросхемы памяти часто производятся в виде матриц mxn, обращение к которым происходит по строке и столбцу. Такая организация памяти сокращает число необходимых выводов, но, с другой стороны, замедляет обращение к микросхеме, поскольку требуется два цикла адресации: один для строки, а другой для столбца. Чтобы ускорить этот процесс, в некоторых микросхемах можно вызывать адрес ряда, а затем несколько адресов столбцов для доступа к последовательным битам ряда.

Много лет назад самые большие микросхемы памяти обычно были устроены так, как показано на рис. 3.30, *б*. Поскольку слова выросли от 8 до 32 битов и выше, использовать подобные микросхемы стало неудобно. Чтобы из микросхем 4096 Kxl построить память с 32-битными словами, требуется 32 микросхемы, работающие параллельно. Эти 32 микросхемы имеют общий объем, по крайней мере, 16 Мбайт. Если использовать микросхемы 512 Kx8, то потребуется всего 4 микросхемы, но при этом объем памяти будет составлять 2 Мбайт. Чтобы избежать наличия 32 микросхем, большинство производителей выпускают семейства микросхем с длиной слов 1,4, 8 и 16 битов.

ОЗУ и ПЗУ

Все виды памяти, которые мы рассматривали до сих пор, имеют одно общее свойство: в них можно и записывать информацию, и считывать ее. Такая память называется **ОЗУ (оперативное запоминающее устройство).** Существует два типа ОЗУ: статическое и динамическое. **Статическое ОЗУ** конструируется с использованием D-триггеров. Информация в ОЗУ сохраняется на протяжении всего времени, пока к нему подается питание: секунды, минуты, часы и даже дни. Статическое ОЗУ работает очень быстро. Обычно время доступа составляет несколько наносекунд. По этой причине статическое ОЗУ часто используется в качестве кэш-памяти второго уровня.

В динамическом ОЗУ, напротив, триггеры не используются. Динамическое ОЗУ представляет собой массив ячеек, каждая из которых содержит транзистор и крошечный конденсатор. Конденсаторы могут быть заряженными и разряженными, что позволяет хранить нули и единицы. Поскольку электрический заряд имеет тенденцию исчезать, каждый бит в динамическом ОЗУ должен **обновляться** (перезаряжаться) каждые несколько миллисекунд, чтобы предотвратить утечку данных. Поскольку об обновлении должна заботиться внешняя логика, динамическое ОЗУ требует более сложного сопряжения, чем статическое, хотя этот недостаток компенсируется большим объемом.

Поскольку динамическому ОЗУ нужен только 1 транзистор **и** 1 конденсатор на бит (статическому ОЗУ требуется в лучшем случае 6 транзисторов на бит), динамическое ОЗУ имеет очень высокую плотность записи (много битов на одну микросхему). По этой причине основная память почти всегда строится на основе динамических ОЗУ. Однако динамические ОЗУ работают очень медленно (время доступа занимает десятки наносекунд). Таким образом, сочетание кэш-памяти на основе статического ОЗУ и основной памяти на основе динамического ОЗУ соединяет в себе преимущества обоих устройств.

Существует несколько типов динамических ОЗУ. Самый древний тип, который все еще используется, — **FPM (Fast Page Mode — быстрый постраничный**

режим). Это ОЗУ представляет собой матрицу битов. Аппаратное обеспечение представляет адрес строки, а затем — адреса столбцов (мы описывали этот процесс, когда говорили об устройстве памяти, показанном на рис. 3.30, *б).*

FPM постепенно замещается **EDO**[1] **(Extended Data Output — память с расширенными возможностями вывода),** которая позволяет обращаться к памяти еще до того, как закончилось предыдущее обращение. Такой конвейерный режим не ускоряет доступ к памяти, но зато увеличивает пропускную способность, выдавая больше слов в секунду.

И FPM, и EDO являются асинхронными. В отличие от них так называемое **синхронное динамическое ОЗУ** управляется одним синхронизирующим сигналом. Данное устройство представляет собой гибрид статического и динамического ОЗУ. Синхронное динамическое ОЗУ часто используется при производстве кэш-памяти большого объема. Возможно, данная технология в будущем станет наиболее предпочтительной и в изготовлении основной памяти.

ОЗУ — не единственный тип микросхем памяти. Во многих случаях данные должны сохраняться, даже если питание отключено (например, если речь идет об игрушках, различных приборах и машинах). Более того, после установки ни программы, ни данные не должны изменяться. Эти требования привели к появлению **ПЗУ (постоянных запоминающих устройств),** которые не позволяют изменять и стирать хранящуюся в них информацию (ни умышленно, ни случайно). Данные записываются в ПЗУ в процессе производства. Для этого изготавливается трафарет с определенным набором битов, который накладывается на фоточувствительный материал, а затем открытые (или закрытые) части поверхности вытравливаются. Единственный способ изменить программу в ПЗУ — поменять целую микросхему.

ПЗУ стоят гораздо дешевле ОЗУ, если заказывать их большими партиями, чтобы оплатить расходы на изготовление трафарета. Однако они не допускают изменений после выпуска с производства, а между подачей заказа на ПЗУ и его выполнением может пройти несколько недель. Чтобы компаниям было проще разрабатывать новые устройства, основанные на ПЗУ, были выпущены **программируемые ПЗУ.** В отличие от обычных ПЗУ, их можно программировать в условиях эксплуатации, что позволяет сократить время выполнения заказа. Многие программируемые ПЗУ содержат массив крошечных плавких перемычек. Можно пережечь определенную перемычку, если выбрать нужную строку и нужный столбец, а затем приложить высокое напряжение к определенному выводу микросхемы.

Следующая разработка этой линии — **стираемое программируемое ПЗУ,** которое можно не только программировать в условиях эксплуатации, но и стирать с него информацию. Если кварцевое окно в данном ПЗУ подвергать воздействию сильного ультрафиолетового света в течение 15 минут, все биты установятся на 1. Если нужно сделать много изменений во время одного этапа проектирования, стираемые ПЗУ гораздо экономичнее, чем обычные программируемые ПЗУ, поскольку их можно использовать многократно. Стираемые программируемые ПЗУ обычно устроены так же, как статические ОЗУ. Например, микросхема 27C040 имеет структуру, которая показана на рис. 3.30, а, а такая структура типична для статического ОЗУ.

[1] Динамическая память типа EDO вытеснила обычную динамическую память, работающую в режиме FPM, в середине 90-х годов. — *Примеч. научи, ред.*

Следующий этап — электронно-перепрограммируемое **ПЗУ,** с которого можно стирать информацию, прилагая к нему импульсы, и которое не нужно для этого помещать в специальную камеру, чтобы подвергнуть воздействию ультрафиолетовых лучей. Кроме того, чтобы перепрограммировать данное устройство, его не нужно вставлять в специальный аппарат для программирования, в отличие от стираемого программируемого ПЗУ, Но с другой стороны, самые большие электронно-перепрограммируемые ПЗУ в 64 раза меньше обычных стираемых ПЗУ, и работают они в два раза медленнее. Электронно-перепрограммируемые ПЗУ не могут конкурировать с динамическими и статическими ОЗУ, поскольку они работают в 10 раз медленнее, их емкость в 100 раз меньше и они стоят гораздо дороже. Они используются только в тех ситуациях, когда необходимо сохранение информации при выключении питания.

Более современный тип электронно-перепрограммируемого ПЗУ — **флэш-память.** В отличие от стираемого ПЗУ, которое стирается под воздействием ультрафиолетовых лучей, и от электронно-программируемого ПЗУ, которое стирается по байтам, флэш-память стирается и записывается блоками. Как и любое электронно-перепрограммируемое ПЗУ, флэш-память можно стирать, не вынимая ее из микросхемы. Многие изготовители производят небольшие печатные платы, содержащие десятки мегабайтов флэш-памяти. Они используются для хранения изображений в цифровых камерах и для других целей. Возможно, когда-нибудь флэш-память вытеснит диски, что будет грандиозным шагом вперед, учитывая время доступа в 100 не. Основной технической проблемой в данный момент является то, что флэш-память изнашивается после 10 000 стираний, а диски могут служить годами независимо от того, сколько раз они перезаписывались. Краткое описание различных типов памяти дано в табл. 3.2.

Таблица 3.2. Характеристики различных видов памяти

Тип запоминающего устройства	Категория	Стирание записи	Изменение информации по байтам	Энергозависимость	Применение
Статическое ОЗУ (SRAM)	Чтение/запись	Электрическое	**Да**	**Да**	Кэш-память второго уровня
Динамическое ОЗУ(DRAM)	Чтение/запись	Электрическое	**Да**	**Да**	Основная память
ПЗУ (ROM)	Только чтение	Невозможно	**Нет**	**Нет**	Устройства большого размера
Программируемое ПЗУ (PROM)	Только чтение	Невозможно	**Нет**	**Нет**	Устройства небольшого размера
Стираемое программируемое nЗУ(EPROM)	Преимущественно чтение	Ультрафиолетовый свет	Нет	**Нет**	Моделирование устройств
Электронно-перепрограммируемое ПЗУ (EEPROM)	Преимущественно чтение	Электрическое	**Да**	**Нет**	Моделирование устройств
Флэш-память (Flash)	Чтение/запись	Электрическое	Нет	Нет	Цифровые камеры

Микросхемы процессоров и шины

Поскольку нам уже известна некоторая информация о МИС, СИС и микросхемах памяти, то мы можем сложить все составные части вместе и изучать целые системы. В этом разделе сначала мы рассмотрим процессоры на цифровом логическом уровне, включая цоколевку (то есть значение сигналов на различных выводах). Поскольку центральные процессоры тесно связаны с шинами, которые они используют, мы также кратко изложим основные принципы разработки шин. В следующих разделах мы подробно опишем примеры центральных процессоров и шин для них.

Микросхемы процессоров

Все современные процессоры помещаются на одной микросхеме. Это делает вполне определенным их взаимодействие с остальными частями системы. Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Одни выводы передают сигналы от центрального процессора, другие принимают сигналы от других компонентов, третьи делают и то и другое. Изучив функции всех выводов, мы сможем узнать, как процессор взаимодействует с памятью и устройствами ввода-вывода на цифровом логическом уровне.

Выводы микросхемы центрального процессора можно подразделить на три типа: адресные, информационные и управляющие. Эти выводы связаны с соответствующими выводами на микросхемах памяти и микросхемах устройств ввода-вывода через набор параллельных проводов (так называемую шину). Чтобы вызвать команду, центральный процессор сначала посылает в память адрес этой команды по адресным выводам. Затем он запускает одну или несколько линий управления, чтобы сообщить памяти, что ему нужно, например, прочитать слово. Память выдает ответ, помещая требуемое слово на информационные выводы процессора и посылая сигнал о том, что это сделано. Когда центральный процессор получает данный сигнал, он принимает слово и выполняет вызванную команду.

Команда может требовать чтения или записи слов, содержащих данные. В этом случае весь процесс повторяется для каждого дополнительного слова. Как происходит процесс чтения и записи, мы подробно рассмотрим ниже. Важно понимать, что центральный процессор обменивается информацией с памятью и устройствами ввода-вывода, подавая сигналы на выводы и принимая сигналы на входы. Другого способа обмена информацией не существует.

Число адресных выводов и число информационных выводов — два ключевых параметра, которые определяют производительность процессора. Микросхема, содержащая m адресных выводов, может обращаться к 2^m ячейкам памяти. Обычно m равно 16, 20, 32 или 64. Микросхема, содержащая n информационных выводов, может считывать или записывать n-битное слово за одну операцию. Обычно n равно 8,16,32,36 или 64. Центральному процессору с 8 информационными выводами понадобится 4 операции, чтобы считать 32-битное слово, тогда как процессор, имеющий 32 информационных вывода, может сделать ту же работу в одну

операцию. Следовательно, микросхема с 32 информационными выводами работает гораздо быстрее, но и стоит гораздо дороже.

Кроме адресных и информационных выводов каждый процессор содержит выводы управления. Выводы управления регулируют и синхронизируют поток данных к процессору и от него, а также выполняют другие разнообразные функции. Все процессоры содержат выводы для питания (обычно +3,3 В или +5 В), «земли» и синхронизирующего сигнала (меандра). Остальные выводы разнятся от процессора к процессору. Тем не менее выводы управления можно разделить на несколько основных категорий:

1. Управление шиной.
2. Прерывание.
3. Арбитраж шины.
4. Состояние.
5. Разное.

Ниже мы кратко опишем каждую из этих категорий. Когда мы будем рассматривать микросхемы Pentium II, UltraSPARC II и picojava II, мы дадим более подробную информацию. Схема типичного центрального процессора, в котором используются эти типы сигналов, изображена на рис. 3.31.

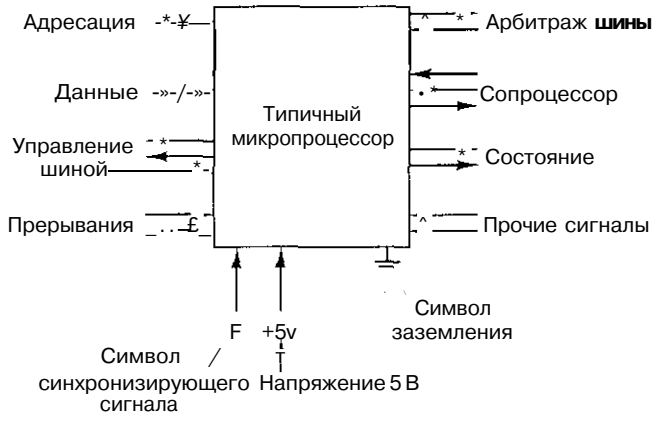


Рис. 3.31. Цоколевка типичного центрального процессора. Стрелочки указывают входные и выходные сигналы. Короткие диагональные линии указывают на наличие нескольких выводов. Для конкретных процессоров будет дано число этих выводов

Выводы управления шиной по большей части представляют собой выходы из центрального процессора в шину (и следовательно, входы в микросхемы памяти и микросхемы устройств ввода-вывода). Они сообщают, что процессор хочет считать информацию из памяти, или записать информацию в память, или сделать что-нибудь еще.

Выводы прерывания — это входы из устройств ввода-вывода в процессор. В большинстве систем процессор может дать сигнал устройству ввода-вывода начать операцию, а затем приступить к какому-нибудь другому действию, пока устройство

ввода-вывода выполняет свою работу. Когда устройство ввода-вывода заканчивает свою работу, контроллер ввода-вывода посылает сигнал на один из выводов прерывания, чтобы прервать работу процессора и заставить его обслуживать устройство ввода-вывода (например, проверять ошибки ввода-вывода). Некоторые процессоры содержат выходной вывод, чтобы подтверждать получение сигнала прерывания.

Выводы разрешения конфликтов в шине нужны для того, чтобы регулировать поток информации в шине, то есть не допускать таких ситуаций, когда два устройства пытаются воспользоваться шиной одновременно. В целях разрешения конфликтов центральный процессор считается устройством.

Некоторые центральные процессоры могут работать с различными сопроцессорами (например, с графическими процессорами, процессорами с плавающей точкой и т. п.). Чтобы обеспечить обмен информации между процессором и сопроцессором, нужны специальные выводы для передачи сигналов.

Кроме этих выводов у некоторых процессоров есть различные дополнительные выводы. Одни из них выдают или принимают информацию о состоянии, другие нужны для перезагрузки компьютера, а третьи — для обеспечения совместимости со старыми микросхемами устройств ввода-вывода,

Шины

Шина — это группа проводников, соединяющих различные устройства. Шины можно разделить на группы в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устройствами ввода-вывода. Каждый тип шины обладает определенными свойствами, и к каждому из них предъявляются определенные требования. В этом и следующих разделах мы сосредоточимся на шинах, которые связывают центральный процессор с памятью и устройствами ввода-вывода. В следующей главе мы подробно рассмотрим внутренние шины процессора.

Первые персональные компьютеры имели одну внешнюю шину, которая называлась системной **шиной.** Она состояла из нескольких медных проводов (от 50 до 100), которые встраивались в материнскую плату. На материнской плате находились разъемы на одинаковых расстояниях друг от друга для микросхем памяти и устройств ввода-вывода. Современные персональные компьютеры обычно содержат специальную шину между центральным процессором и памятью и по крайней мере еще одну шину для устройств ввода-вывода. На рис. 3.32 изображена система с одной шиной памяти и одной шиной ввода-вывода.

В литературе шины обычно изображаются в виде жирных стрелок, как показано на этом рисунке. Разница между жирной и нежирной стрелкой небольшая. Когда все биты одного типа, например адресные или информационные, рисуется обычная стрелка. Когда включаются адресные линии, линии данных и управления, используется жирная стрелка.

Хотя разработчики процессоров могут использовать любой тип шины для микросхемы, должны быть введены четкие правила о том, как работает шина, и все

устройства, связанные с шиной, должны подчиняться этим правилам, чтобы платы, которые выпускаются третьими лицами, подходили к системной шине. Эти правила называются протоколом **шины.** Кроме того, должны существовать определенные технические требования, чтобы платы от третьих производителей подходили к каркасу для печатных плат и имели разъемы, соответствующие материнской плате механически и с точки зрения мощностей, синхронизации и т. д.

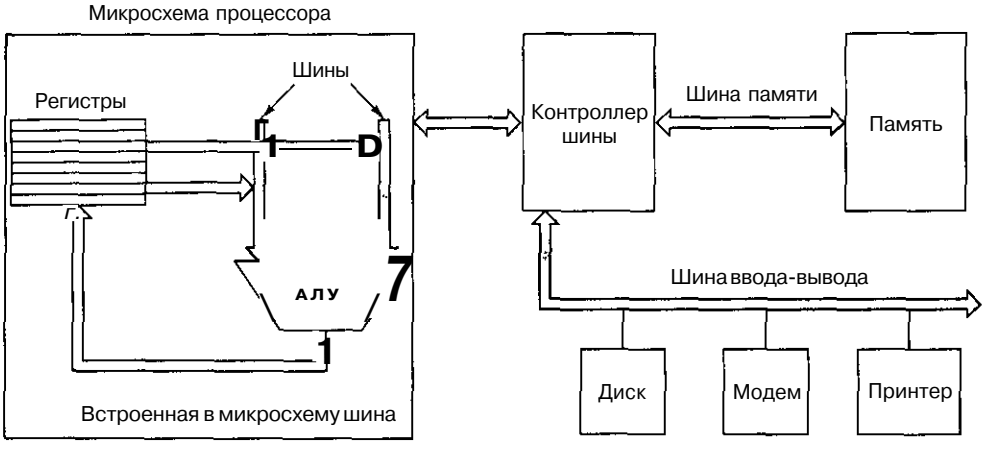


Рис. 3.32. Компьютерная система с несколькими шинами

Существует ряд широко используемых в компьютерном мире шин. Приведем несколько примеров: Omnibus (PDP-8), Unibus (PDP-11), IBM PC (PC/XT), ISA (PC/AT), EISA (80386), MicroChannel (PC/2), PCI (различные персональные компьютеры), SCSI (различные персональные компьютеры и рабочие станции), Nubus (Macintosh), Universal Serial Bus (современные персональные компьютеры), FireWire (бытовая электроника), VME (оборудование в кабинетах физики) и Cataac (физика высоких энергий). Может быть, все стало бы намного проще, если бы все шины, кроме одной, исчезли с поверхности Земли (или кроме двух). К сожалению, стандартизация в этой области кажется маловероятной, и уже вложено слишком много средств во все эти несовместимые системы.

Давайте начнем с того, как работают шины. Некоторые устройства, связанные с шиной, являются активными и могут инициировать передачу информации по шине, тогда как другие являются пассивными и ждут запросов. Активное устройство называется **задающим устройством,** пассивное — **подчиненным устройством.** Когда центральный процессор требует от контроллера диска считать или записать блок информации, центральный процессор действует как задающее устройство, а контроллер диска — как подчиненное устройство. Контроллер диска может действовать как задающее устройство, когда он командует памяти принять слова, которые считал с диска. Несколько типичных комбинаций задающего и подчиненного устройств указаны в табл. 3.3. Память ни при каких обстоятельствах не может быть задающим устройством.

Таблица 3.3. Примеры задающих и подчиненных устройств

Задающее устройство	Подчиненное устройство	Пример
Центральный процессор	Память	Вызов команд и данных
Центральный процессор	Устройство ввода-вывода	Инициализация передачи данных
Центральный процессор	Сопроцессор	Передача команды от процессора к сопроцессору
Устройство ввода-вывода	Память	ПДП (прямой доступ к памяти)
Сопроцессор	Центральный процессор	Вызов сопроцессором операндов из центрального процессора

Двоичные сигналы, которые выдают устройства компьютера, часто недостаточно интенсивны, чтобы активизировать шину, особенно если она достаточно длинная и если к ней подсоединено много устройств. По этой причине большинство задающих устройств шины обычно связаны с ней через микросхему, которая называется **драйвером шины,** по существу являющуюся двоичным усилителем. Сходным образом большинство подчиненных устройств связаны с шиной **приемником шины.** Для устройств, которые могут быть и задающим, и подчиненным устройством, используется **приемопередатчик шины.** Эти микросхемы взаимодействия с шиной часто являются устройствами с тремя состояниями, что дает им возможность отсоединяться, когда они не нужны. Иногда они подключаются через **открытый коллектор,** что дает сходный эффект. Когда одно **или** несколько устройств на открытом коллекторе требуют доступа к шине в одно и то же время, результатом является булева операция ИЛИ над всеми этими сигналами. Такое соглашение называется монтажным **ИЛИ.** В большинстве шин одни линии являются устройствами с тремя состояниями, а другие, которым требуется свойство монтажного ИЛИ, — открытым коллектором.

Как и процессор, шина имеет адресные **линии,** информационные линии и линии управления. Тем не менее между выводами процессора и сигналами шины может и не быть взаимно однозначного соответствия. Например, некоторые процессоры содержат три вывода, которые выдают сигнал чтения из памяти или записи в память, или чтения устройства ввода-вывода, или записи на устройство ввода-вывода, или какой-либо другой операции. Обычная шина может содержать одну линию для чтения из памяти, вторую линию для записи в память, третью — для чтения устройства ввода-вывода, четвертую — для записи на устройство ввода-вывода и т. д. Микросхема-декодер должна тогда связывать данный процессор с такой шиной, чтобы преобразовывать 3-битный кодированный сигнал в отдельные сигналы, которые могут управлять линиями шины.

Разработка шин и принципы действия шин — это достаточно сложные вопросы и по этому поводу написан ряд книг [128, 135,136]. Принципиальными вопросами в разработке являются ширина шины, синхронизация шины, арбитраж шины и функционирование шины. Все эти параметры существенно влияют на скорость и пропускную способность шины. В следующих четырех разделах мы рассмотрим каждый из них.

Ширина шины

Ширина шины — самый очевидный параметр при разработке. Чем больше адресных линий содержит шина, тем к большему объему памяти может обращаться процессор. Если шина содержит n адресных линий, тогда процессор может использовать ее для обращения к 2" различным ячейкам памяти. Для памяти большой емкости необходимо много адресных линий. Это звучит достаточно просто.

Проблема заключается в том, что для широких шин требуется больше проводов, чем для узких. Они занимают больше физического пространства (например, на материнской плате), и для них нужны разъемы большего размера. Все эти факторы делают шину дорогостоящей. Следовательно, необходим компромисс между максимальным размером памяти и стоимостью системы. Система с шиной, содержащей 64 адресные линии, и памятью в 2^{32} байт будет стоить дороже, чем система с шиной, содержащей 32 адресные линии, и такой же памятью в 2^{32} байт. Дальнейшее расширение не бесплатное.

Многие разработчики систем недальновидны, что приводит к неприятным последствиям. Первая модель IBM PC содержала процессор 8088 и 20-битную адресную шину (рис. 3.33, *а*). Шина позволяла обращаться к 1 Мбайт памяти.

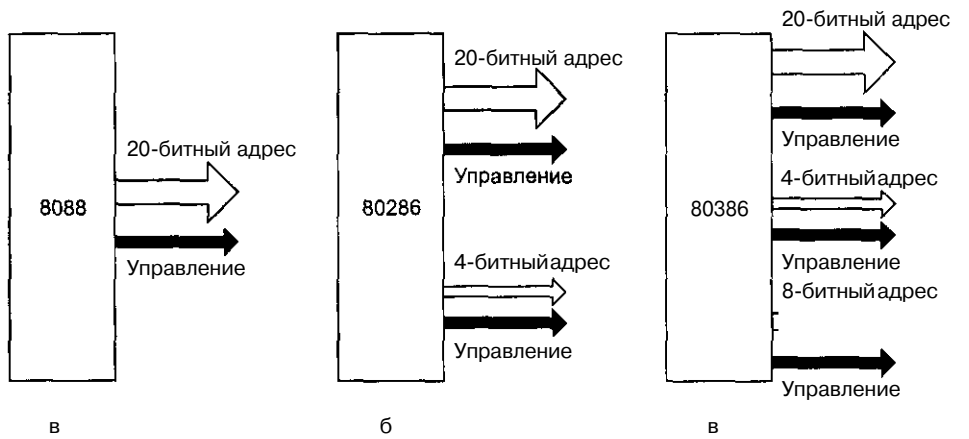


Рис. 3.33. Расширение адресной шины с течением времени

Когда появился следующий процессор (80286), Intel решил увеличить адресное пространство до 16 Мбайт, поэтому пришлось добавить еще 4 линии (не нарушая изначальные 20 по причинам совместимости с более старыми версиями), как показано на рис. 3.33, *б*. К сожалению, пришлось также добавить линии управления для новых адресных линий. Когда появился процессор 80386, было добавлено еще 8 адресных линий и, естественно, несколько линий управления, как показано на рис. 3.33, *в*. В результате получилась шина EISA. Однако было бы лучше, если бы с самого начала имелось 32 линии.

С течением времени увеличивается не только число адресных линий, но и число информационных линий. Хотя это происходит по несколько другой причине. Можно увеличить пропускную способность шины двумя способами: сократить время цикла шины (сделать большее количество передач в секунду) или увели-

чить ширину шины данных (то есть увеличить количество битов за одну передачу). Можно повысить скорость работы шины, но сделать это довольно сложно, поскольку сигналы на разных линиях передаются с разной скоростью (это явление называется **перекосом шины).** Чем быстрее работает шина, тем больше перекос.

При увеличении скорости работы шины возникает еще одна проблема: в этом случае она не будет совместимой с более старыми версиями. Старые платы, разработанные для более медленной шины, не могут работать с новой. Такая ситуация невыгодна для владельцев и производителей старых плат. Поэтому обычно для увеличения производительности просто добавляются новые линии, как показано на рис. 3.33. Как вы понимаете, в этом тоже есть свои недостатки. IBM PC и его последователи, например, начали с 8 информационных линий, затем перешли к 16, а затем к 32, и все это в одной и той же шине.

Чтобы обойти эту проблему, разработчики иногда отдают предпочтение мультиплексной **шине.** В этой шине нет разделения на адресные и информационные линии. В ней может быть, например, 32 линии и для адресов, и для данных. Сначала эти линии используются для адресов. Затем они используются для данных. Чтобы записать информацию в память, нужно сначала передавать в память адрес, а затем данные. В случае с отдельными линиями адреса и данные могут передаваться вместе. Объединение линий сокращает ширину и стоимость шины, но система работает при этом медленнее. Поэтому разработчикам приходится взвешивать все за и против, прежде чем сделать выбор.

Синхронизация шины

Шины можно разделить на две категории в зависимости от их синхронизации. **Синхронная шина** содержит линию, которая запускается кварцевым генератором. Сигнал на этой линии представляет собой меандр с частотой обычно от 5 до 100 МГц. Любое действие шины занимает целое число так называемых **циклов шины. Асинхронная шина** не содержит задающего генератора. Циклы шины могут быть любой требуемой длины и необязательно одинаковы по отношению ко всем парам устройств. Ниже мы рассмотрим каждый тип шины отдельно.

Синхронные шины

В качестве примера того, как работает асинхронная шина, рассмотрим временную диаграмму на рис. 3.34. В этом примере мы будем использовать задающий генератор на 40 МГц, который дает цикл шины в 25 не. Хотя может показаться, что шина работает медленно по сравнению с процессорами на 500 МГц и выше, не многие современные шины работают быстрее. Например, шина ISA (она встроена во все персональные компьютеры с процессором Intel) работает с частотой 8,33 МГц, и даже популярная шина PCI — с частотой 33 МГц или 66 МГц. Причины такой низкой скорости современных шин были даны выше: такие технические проблемы, как перекос шины и требование совместимости.

В нашем примере мы предполагаем, что считывание информации из памяти занимает 40 не с того момента, как адрес стал постоянным. Как мы скоро увидим, понадобится три цикла шины, чтобы считать одно слово. Первый цикл начинается

на нарастающем фронте отрезка T_b а третий заканчивается на нарастающем фронте отрезка T_3, как показано на рис. 3.34. Отметим, что ни один из нарастающих и задних фронтов не нарисован вертикально, потому что ни один электрический сигнал не может изменять свое значение за нулевое время. В нашем примере мы предполагаем, что для изменения сигнала требуется 1 не. Генератор и линии ADDRESS, DATA, MREQ, RD, WAIT показаны в том же масштабе времени.

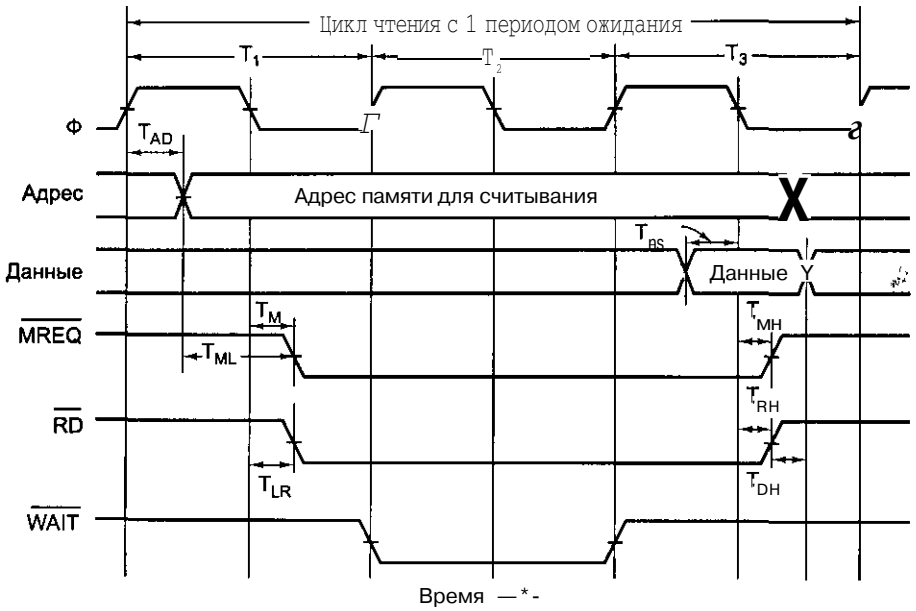


Рис. 3.34. Временная диаграмма процесса считывания на синхронной шине

Начало Ti определяется нарастающим фронтом генератора. За часть времени Ti центральный процессор помещает адрес нужного слово на адресные линии. Поскольку адрес представляет собой не одно значение (в отличие от генератора), мы не можем показать его в виде одной линии на схеме. Вместо этого мы показали его в виде двух линий с пересечениями там, где этот адрес меняется. Серый цвет на схеме показывает, что в этот момент не важно, какое значение принял сигнал. Используя то же соглашение, мы видим, что содержание линий данных не имеет значения до отрезка T_3.

После того как у адресных линий появляется возможность приобрести новое значение, устанавливаются сигналы MREQ и RD. Первый указывает, что осуществляется доступ к памяти, а не к устройству ввода-вывода, а второй — что осуществляется чтение, а не запись. Поскольку считывание информации из памяти занимает 40 не после того, как адрес стал постоянным (часть первого цикла), память не может передать требуемые данные за период T_2. Чтобы центральный процессор не ожидал поступления данных, память устанавливает линию WAIT в начале отрезка T_a. Это действие вводит периоды ожидания (дополнительные циклы шины), до тех пор пока память не сбросит сигнал WAIT. В нашем примере вводится один период ожидания (T_2), поскольку память работает слишком медленно. В начале

Тз, когда есть уверенность в том, что память получит данные в течение текущего цикла, сигнал WAIT сбрасывается.

Во время первой половины T_3 память помещает данные на информационные линии. На заднем фронте T_3 центральный процессор стробирует (то есть считывает) информационные линии, сохраняя их значения во внутреннем регистре. Считав данные, центральный процессор сбрасывает сигналы $\overline{MREQ}$. и $\overline{RD}$. В случае необходимости на следующем нарастающем фронте может начаться еще один цикл памяти.

Далее проясняется значение восьми символов на временной диаграмме (см. рис. 3.34 и табл. 3.4). T_{AD}, например, — это временной интервал между нарастающим фронтом Т(и установкой адресных линий. В соответствии с требованиями синхронизации $T_{AD} \leq 11$ не. Значит, производитель процессора 1арантирует, что во время любого цикла считывания центральный процессор будет выдавать требуемый адрес в пределах 11 не от середины нарастающего фронта TV

Таблица 3.4. Некоторые временные характеристики процесса считывания на синхронной шине

Символ	Значение	Минимум, не	Максимум, не
TAD	Задержка выдачи адреса		11
T_{ML}	Промежуток между стабилизацией адреса и установкой сигнала $\overline{MREQ}$	6	
$T_м$	Промежуток между задним фронтом синхронизирующего сигнала в цикле Ti и установкой сигнала $\overline{MREQ}$		8
TR_L	Промежуток между задним фронтом синхронизирующего сигнала в цикле T_t и установкой сигнала $\overline{RD}$		8
Tos	Период передачи данных до заднего фронта синхронизирующего сигнала	5	
$T_{мн}$	Промежуток между задним фронтом синхронизирующего сигнала в цикле Тз и сбросом сигнала $\overline{MREQ}$		8
Тдн	Промежуток между задним фронтом синхронизирующего сигнала в цикле Тэ и сбросом сигнала $\overline{RD}$		8
Тон	Период продолжения передачи данных с момента сброса сигнала $\overline{RD}$	О	

Условия синхронизации также требуют, чтобы данные поступали на информационные линии по крайней мере за 5 не (T_Ds) до заднего фронта T_3, чтобы дать данным время установиться до того, как процессор стробирует их. Сочетание ограничений на T_AD И T_DS означает, что в худшем случае в распоряжении памяти будет только 62,5-11-5=46,5 не с момента появления адреса и до момента, когда нужно выдавать данные. Поскольку достаточно 40 не, память даже в самом худшем случае может всегда ответить за период T_3. Если памяти для считывания требуется 50 не, то необходимо ввести второй период ожидания, и тогда память ответит в течение T_3.

Требования синхронизации гарантируют, что адрес будет установлен по крайней мере за 6 не до того, как появится сигнал $\overline{\text{MREQ}}$. Это время может быть важно в том случае, если $\overline{\text{MREQ}}$ запускает выбор элемента памяти, поскольку некоторые типы памяти требуют некоторого времени на установку адреса до выбора элемента памяти. Ясно, что разработчику системы не следует выбирать микросхему памяти, на установку которой нужно 10 не.

Ограничения на $T_м$ и $T_{R|}$. означают, что $\overline{\text{WREQ}}$ и $\overline{\text{RD}}$ будут установлены в пределах 8 не от заднего фронта $T_ь$ В худшем случае у микросхемы памяти после установки $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ останется всего 25+25-8-5=37 не на передачу данных по шине. Это ограничение дополнительно по отношению к интервалу в 40 не и не зависит от него.

Тмн и TRH определяют, сколько времени требуется на отмену сигналов MREQ и $\overline{\text{RD}}$ после того, как данные стробированы. Наконец, $T_{о}$н определяет, сколько времени память должна держать данные на шине после снятия сигнала КП. В нашем примере при данном процессоре память может удалить данные с шины, как только сбрасывается сигнал RT); при других процессорах, однако, данные могут сохраняться еще некоторое время.

Необходимо подчеркнуть, что наш пример представляет собой сильно упрощенную версию реальных временных ограничений. В действительности должно определяться гораздо больше таких ограничений. Тем не менее этот пример наглядно демонстрирует, как работает синхронная шина.

Отметим, что сигналы управления могут задаваться или с помощью низкого, или с помощью высокого напряжения. Что является более удобным в каждом конкретном случае, должен решать разработчик, хотя, по существу, выбор произволен.

Асинхронные шины

Хотя достаточно удобно использовать синхронные шины благодаря дискретным временным интервалам, здесь все же есть некоторые проблемы. Например, если процессор и память способны закончить передачу за 3,1 цикла, они вынуждены продлить ее до 4,0 циклов, поскольку неполные циклы запрещены.

Еще хуже то, что если однажды был выбран определенный цикл шины и в соответствии с ним были разработаны память и карты ввода-вывода, то в будущем трудно делать технологические усовершенствования. Например, предположим, что через несколько лет после выпуска системы, изображенной на рис. 3.34, появилась новая память с временем доступа не 40, а 20 не. Это избавило бы нас от периода ожидания и увеличило скорость работы машины. Теперь представим, что появилась память с временем доступа 10 не. При этом улучшения производительности уже не будет, поскольку в данной разработке минимальное время для чтения — 2 цикла.

Если синхронная шина соединяет ряд устройств, одни из которых работают быстро, а другие медленно, шина подстраивается под самое медленное устройство, а более быстрые не могут использовать свой полный потенциал.

По этой причине были разработаны асинхронные шины, то есть шины без задающего генератора, как показано на рис. 3.35. Здесь ничего не привязывается к генератору. Когда задающее устройство устанавливает адрес, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и любой

другой требуемый сигнал, он выдает специальный сигнал, который мы будем называть MSYN (Master SYNchronization). Когда подчиненное устройство получает этот сигнал, оно начинает выполнять свою работу настолько быстро, насколькс это возможно. Когда работа закончена, устройство выдает сигнал SSYN (Slavf SYNchronization).

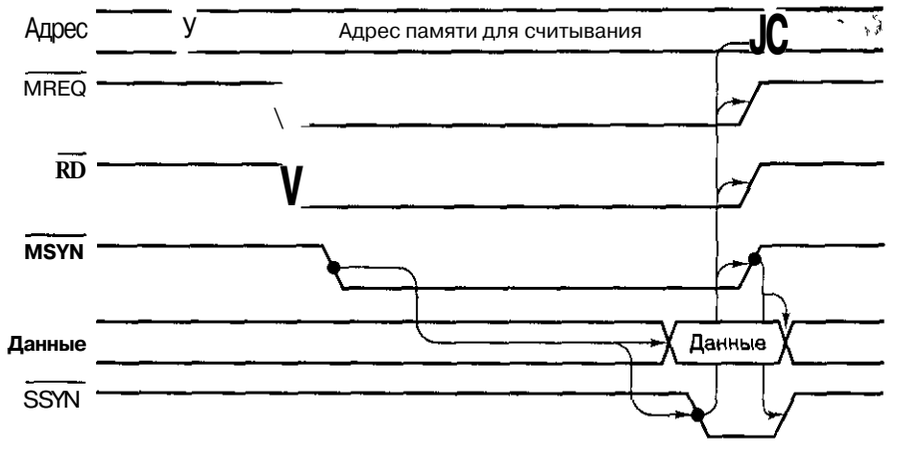


Рис. 3.35. Работа асинхронной шины

Сигнал SSYN означает для задающего устройства, что данные доступны. *Oi* фиксирует их, а затем отключает адресные линии вместе с MREQ, RD и M$YN Отмена сигнала MSYN означает для подчиненного устройства, что цикл закончен поэтому устройство отменяет сигнал SSYN, и все возвращается к первоначальному состоянию, когда все сигналы отменены.

Стрелочки на временных диаграммах асинхронных шин (а иногда и синхронных шин) показывают причину и следствие какого-либо действия (рис. 3.35) Установка сигнала MSYN приводит к запуску информационных линий, а также ь установке сигнала SSYN. Установка сигнала SSYN, в свою очередь, вызывает отключение адресных линий, MREQ, RD и MSYN. Наконец, отключение MSYN вызывает отключение SSYN, и на этом процесс считывания заканчивается.

Набор таких взаимообусловленных сигналов называется **полным квитированием.** Здесь, в сущности, наблюдается 4 события:

1. Установка сигнала MSYN.
2. Установка сигнала SSVN в ответ на сигнал MSYN.
3. Отмена сигнала MSYN в ответ на сигнал SSYN.
4. Отмена сигнала SSYN в ответ на отмену сигнала MSYN.

Следует уяснить, что взаимообусловленность сигналов не зависит от синхронизации. Каждое событие вызывается предыдущим событием, а не импульсами генератора. Если какая-то пара двух устройств (задающего и подчиненного) работает медленно, это никак не повлияет на следующую пару устройств, которая работает гораздо быстрее.

Преимущества асинхронной шины очевидны, но в действительности большинство шин являются синхронными. Дело в том, что синхронную систему построить проще, чем асинхронную. Центральный процессор просто выдает сигналы, а память просто реагирует на них. Здесь нет никакой причинно-следственной связи, но если компоненты выбраны удачно, все будет работать и без квитирования. Кроме того, в разработку синхронных шин сделано очень много вложений.

Арбитраж шины

До этого момента мы неявно предполагали, что существует только одно задающее устройство шины — центральный процессор, В действительности микросхемы ввода-вывода могут становиться задающим устройством при считывании информации из памяти и записи информации в память. Кроме того, они могут вызывать прерывания. Сопроцессоры также могут становиться задающим устройством шины. Возникает вопрос: «Что происходит, когда задающим устройством шины могут стать два или несколько устройств одновременно?** Чтобы предотвратить хаос, который может при этом возникнуть, нужен специальный механизм — так называемый **арбитраж шины.**

Механизмы арбитража могут быть централизованными или децентрализованными. Рассмотрим сначала централизованный арбитраж. Простой пример централизованного арбитража показан на рис. 3.36, *а.* В данном примере один арбитр шины определяет, чья очередь следующая. Часто бывает, что арбитр встроен в микросхему процессора, но иногда требуется отдельная микросхема. Шина содержит одну линию запроса (монтажное ИЛИ), которая может запускаться одним или несколькими устройствами в любое время. Арбитр не может определить, сколько устройств запрашивают шину. Он может определять только наличие или отсутствие запросов.

Когда арбитр видит запрос шины, он запускает линию предоставления шины. Эта линия последовательно связывает все устройства ввода-вывода (как в елочной гирлянде). Когда физически ближайшее к арбитру устройство воспринимает сигнал предоставления шины, оно проверяет, нет ли запроса шины. Если запрос есть, устройство пользуется шиной, но не распространяет сигнал предоставления дальше по линии. Если запроса нет, устройство передает сигнал предоставления шины следующему устройству. Это устройство тоже проверяет, есть ли запрос, и действует соответствующим образом в зависимости от наличия или отсутствия запроса. Передача сигнала предоставления шины продолжается до тех пор, пока какое-нибудь устройство не воспользуется предоставленной шиной. Такая система называется **системой последовательного опроса.** При этом приоритеты устройств зависят от того, насколько близко они находятся к арбитру. Ближайшее к арбитру устройство обладает главным приоритетом.

Чтобы обойти такую систему, в которой приоритеты зависят от расстояния от арбитра, в некоторых шинах устраивается несколько уровней приоритета. На каждом уровне приоритета есть линия запроса шины и линия предоставления шины. На рис. 3.36, *б* изображено 2 уровня (хотя в действительности шины обычно содержат 4, 8 или 16 уровней). Каждое устройство связано с одним из уровней

запроса шины, причем, чем выше уровень приоритета, тем больше устройств привязано к этому уровню. На рис. 3.36, *б* можно видеть, что устройства 1, 2 и 4 используют приоритет 1, а устройства 3 и 5 — приоритет 2,

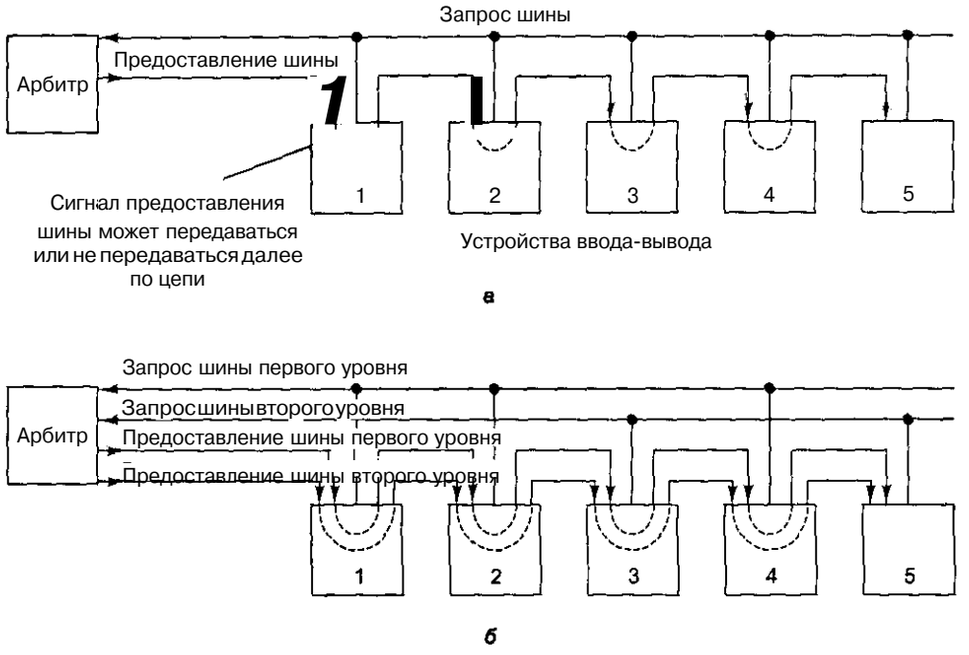


Рис. 3.36. Одноуровневый централизованный арбитраж шины с использованием системы последовательного опроса (а); двухуровневый централизованный арбитраж (б)

Если одновременно запрашивается несколько уровней приоритета, арбитр предоставляет шину самому высокому уровню. Среди устройств одинакового приоритета используется система последовательного опроса. На рис. 3.36, обидно, что в случае конфликта устройство 2 «побеждает» устройство 4, а устройство 4 «побеждает» устройство 3. Устройство 5 имеет низший приоритет, поскольку оно находится в самом конце самого нижнего уровня.

Линия предоставления шины второго уровня необязательно должна последовательно связывать устройства 1 и 2, поскольку они не могут посылать на нее запросы. Однако гораздо проще провести все линии предоставления шины через все устройства, чем соединять устройства особым образом в зависимости от их приоритетов.

Некоторые арбитры содержат третью линию, которая запускается, как только устройство принимает сигнал предоставления шины, и берет шину в свое распоряжение. Как только запускается эта линия подтверждения приема, линии запроса и предоставления шины могут быть отключены. В результате другие устройства могут запрашивать шину, пока первое устройство использует ее. К тому моменту, когда закончится текущая передача, следующее задающее устройство уже будет выбрано. Это устройство может начать работу, как только отключается линия

подтверждения приема. С этого момента начинается следующий арбитраж. Такая структура требует наличия дополнительной линии и большего количества логических схем в каждом устройстве, но зато при этом циклы шины используются рациональнее.

В системах, где память связана с главной шиной, центральный процессор должен завершать работу со всеми устройствами ввода-вывода практически на каждом цикле шины. Чтобы решить эту проблему, можно предоставить центральному процессору самый низкий приоритет. При этом шина будет предоставляться процессору только в том случае, если она не нужна ни одному другому устройству. Центральный процессор всегда может подождать, а устройства ввода-вывода должны получить доступ к шине как можно быстрее, чтобы не потерять данные. Диски, вращающиеся с высокой скоростью, тоже не могут ждать. Во многих современных компьютерах память помещается на одну шину, а устройства ввода-вывода — на другую, поэтому им не приходится завершать работу, чтобы предоставить доступ к шине.

Возможен также децентрализованный арбитраж шины. Например, компьютер может содержать 16 приоритетных линий запроса шины. Когда устройству нужна шина, оно запускает свою линию запроса. Все устройства контролируют все линии запроса, поэтому в конце каждого цикла шины каждое устройство может определить, обладает ли оно в данный момент высшим приоритетом и, следовательно, разрешено ли линии пользоваться шиной в следующем цикле. Такой метод требует наличия большего количества линий, но зато не требует затрат на арбитра. Он также ограничивает число устройств числом линий запроса.

При другом типе децентрализованного арбитража используется только три линии независимо от того, сколько устройств имеется в наличии (рис. 3.37). Первая линия — монтажное ИЛИ. Она используется для запроса шины. Вторая линия называется BUSY. Она запускается текущим задающим устройством шины. Третья линия используется для арбитража шины. Она последовательно соединяет все устройства. Начало цепи связано с источником питания с напряжением 5 В.

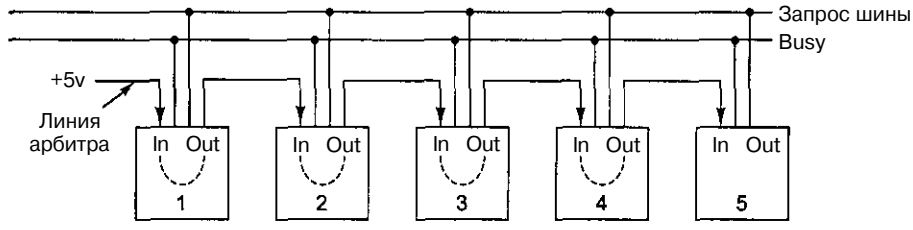


Рис. 3.37. Децентрализованный арбитраж шины

Когда шина не требуется ни одному из устройств, линия арбитра передает сигнал всем устройствам. Чтобы получить доступ к шине, устройство сначала проверяет, свободна ли шина, и установлен ли сигнал арбитра IN. Если сигнал IN не установлен, устройство не может стать задающим устройством шины. В этом случае оно сбрасывает сигнал OUT. Если сигнал IN установлен, устройство также сбрасывает сигнал OUT, в результате чего следующее устройство не получает сигнал IN и, в свою очередь, сбрасывает сигнал OUT. Следовательно, все следующие по цепи устройства не получают сигнал IN и сбрасывают сигнал OUT. В результа-

те остается только одно устройство, у которого сигнал IN установлен, а сигнал OUT сброшен. Оно становится задающим устройством шины, запускает линию BUSY и сигнал OUT и начинает передачу данных.

Немного поразмыслив, можно обнаружить, что из всех устройств, которым нужна шина, доступ к шине получает самое левое. Такая система сходна с системой последовательного опроса, только в данном случае нет арбитра, поэтому она стоит дешевле и работает быстрее. К тому же не возникает проблем со сбоями арбитра.

Принципы работы шины

До этого момента мы обсуждали только обычные циклы шины, когда задающее устройство (обычно центральный процессор) считывает информацию из подчиненного устройства (обычно из памяти) или записывает в него информацию. Однако существует еще несколько типов циклов шины. Давайте рассмотрим некоторые из них.

Обычно за раз передается одно слово. При использовании кэш-памяти желательно сразу вызывать всю строку кэш-памяти (то есть 16 последовательных 32-битных слов). Часто передача блоками может быть более эффективна, чем такая последовательная передача информации. Когда начинается чтение блока, задающее устройство сообщает подчиненному устройству, сколько слов нужно передать (например, помещая общее число слов на информационные линии в период Ti). Вместо того чтобы выдать в ответ одно слово, задающее устройство выдает одно слово в течение каждого цикла до тех пор, пока не будет передано требуемое количество слов. На рис. 3.38 изображена такая же схема, как и на рис. 3.34, только здесь появился дополнительный сигнал BLOCK, который указывает, что запрашивается передача блока. В данном примере считывание блока из 4 слов занимает 6 циклов вместо 12.

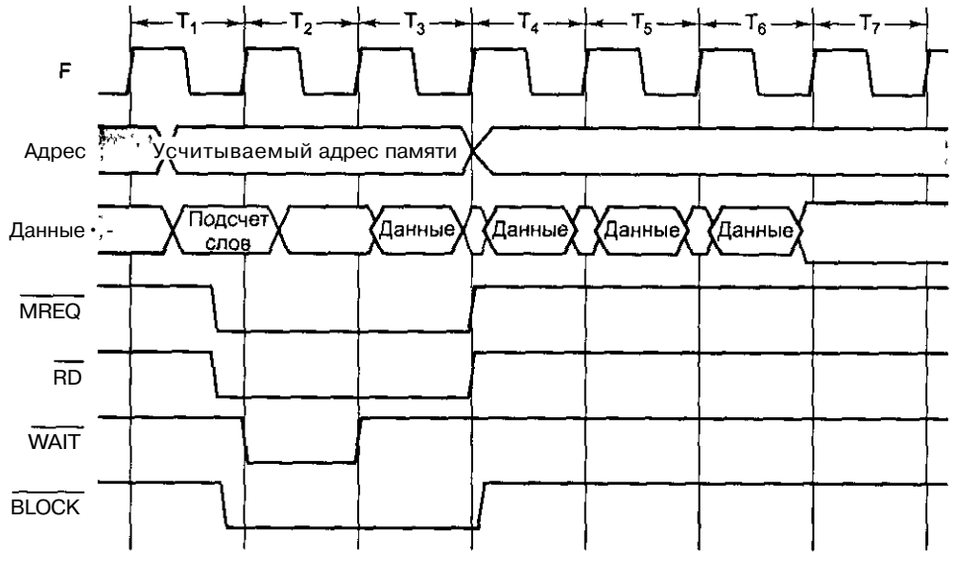


Рис. 3.38. Передача блока данных

Существуют также другие типы циклов шины. Например, если речь идет о системах с двумя или несколькими центральными процессорами на одной шине, нужно быть уверенным, что в конкретный момент только один центральный процессор может использовать определенную структуру данных в памяти. Чтобы упорядочить этот процесс, в памяти должна содержаться переменная, которая принимает значение 0, когда центральный процессор использует структуру данных, и 1, когда структура данных не используется. Если центральному процессору нужно получить доступ к структуре данных, он должен считать переменную, и если она равна О, придать ей значение 1. Проблема заключается в том, что два центральных процессора могут считать переменную на последовательных циклах шины. Если каждый процессор видит, что переменная равна 0, а затем каждый процессор меняет значение переменной на 1, как будто только он один использует эту структуру данных, то такая последовательность событий ведет к хаосу.

Чтобы предотвратить такую ситуацию, в многопроцессорных системах предусмотрен специальный цикл шины, который дает возможность любому процессору считать слово из памяти, проверить и изменить его, а затем записать обратно в память; весь этот процесс происходит без освобождения шины. Такой цикл не дает возможности другим центральным процессорам использовать шину и, следовательно, мешать работе первого процессора.

Еще один важный цикл шины — цикл для осуществления прерываний. Когда центральный процессор командует устройству ввода-вывода произвести какое-то действие, он ожидает прерывания после завершения работы. Для сигнала прерывания нужна шина.

Поскольку может сложиться ситуация, когда несколько устройств одновременно хотят произвести прерывание, здесь имеют место те же проблемы разрешения конфликтных ситуаций, что и в обычных циклах шины. Чтобы избежать таких проблем, нужно каждому устройству приписать определенный приоритет и использовать централизованный арбитр для распределения приоритетов. Существует стандартный контроллер прерываний, который широко используется. В компьютерах IBM PC и последующих моделях применяется микросхема Intel 8259A. Она изображена на рис. 3.39.

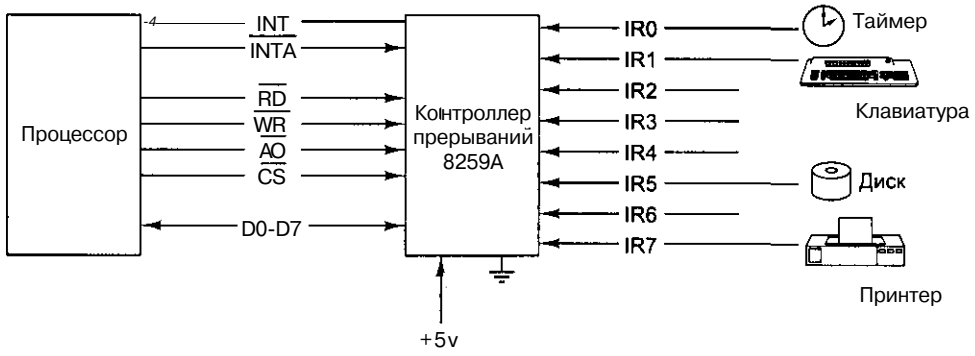


Рис. 3.39. Контроллер прерывания 8259A

До восьми контроллеров ввода-вывода могут быть непосредственно связаны с восемью входами IRx (Interrupt Request — запрос прерывания) микросхемы 8259A Когда любое из этих устройств хочет произвести прерывание, оно запускает свою линию входа Если активизируется один или несколько входов, контроллер 8259A выдает сигнал INT (INTerrupt — прерывание), который подается на соответствующий вход центрального процессора Когда центральный процессор способен произвести прерывание, он посылает микросхеме 8259A импульс через вывод INTA (INTerrupt Acknowledge — подтверждение прерывания) В этот момент микросхема 8259A должна определить, на какой именно вход поступил сигнал прерывания Для этого она помещает номер входа на информационную шину Эта операция требует наличия особого цикла шины Центральный процессор использует этот номер для обращения в таблицу указателей, которую называют таблицей **векторов прерывания,** чтобы найти адрес процедуры, производящей соответствующее прерывание

Микросхема 8259A содержит несколько регистров, которые центральный процессор может считывать и записывать, используя обычные циклы шины и выводы RD (ReaD — чтение), $\overline{WR}$ (WRite — запись), CS (Chip Select — выбор элемента памяти) и Xfl Когда программное обеспечение обработало прерывание и готово получить следующее, оно записывает специальный код в один из регистров, который вызывает сброс сигнала INT микросхемой 8259A, если не появляется другая задержка прерывания Регистры также могут записываться для того, чтобы ввести микросхему 8259A в один из нескольких режимов, и для выполнения некоторых других функций

Когда присутствует более восьми устройств ввода-вывода, микросхемы 8259A могут быть соединены каскадно В самой экстремальной ситуации все восемь входов могут быть связаны с выходами еще восьми микросхем 8259A, соединяя до 64 устройств ввода-вывода в двухступенчатую систему прерывания Микросхема 8259A содержит несколько выводов для управления каскадированием, но мы их опустили ради простоты

Хотя мы никоим образом не исчерпали все вопросы разработки шин, материал, изложенный выше, дает достаточно информации для общего понимания принципов работы шины и того, как центральный процессор взаимодействует с шиной А теперь мы перейдем от общего к частному и рассмотрим несколько конкретных примеров процессоров и их шин

Примеры центральных процессоров

В этом разделе мы рассмотрим процессоры Pentium II, Ultra SPARC II и picojava на уровне аппаратного обеспечения

Pentium II

Pentium II — прямой потомок процессора 8088, который использовался в первой модели IBM PC Хотя Pentium II очень сильно отличается от процессора 8088 (первый содержит 7,5 млн транзисторов, а второй — всего 29 000), он полностью

совместим с 8088 и может выполнять программы, написанные для 8088 (не говоря уже о программах для всех процессоров, появившихся между Pentium II и 8088).

С точки зрения программного обеспечения, Pentium II представляет собой 32-разрядную машину. Он содержит ту же архитектуру системы команд, что и процессоры 80386, 80486, Pentium и Pentium Pro, включая те же регистры, те же команды и такую же встроенную систему с плавающей точкой стандарта IEEE 754.

С точки зрения аппаратного обеспечения, Pentium II представляет собой нечто большее, поскольку он может обращаться к 64 Гбайт физической памяти и передавать данные в память и из памяти блоками по 64 бита. Программист не видит этих передач по 64 бита, но такая машина работает быстрее, чем 32-разрядная.

На микроархитектурном уровне Pentium II представляет собой Pentium Pro с командами MMX. Команды вызываются из памяти заранее и разбиваются на микрооперации. Эти микрооперации хранятся в буфере, и как только одна из них получает необходимые ресурсы для выполнения, она может начаться. Если в одном цикле может начинаться несколько микроопераций, Pentium II является суперскалярной машиной.

Pentium II имеет двухуровневую кэш-память. Кэш-память первого уровня содержит 16 Кбайт для команд и 16 Кбайт для данных, а смежная кэш-память второго уровня — еще 512 Кбайт. Строка кэш-памяти состоит из 32 байт. Тактовая частота кэш-памяти второго уровня в два раза меньше тактовой частоты центрального процессора. Тактовая частота центрального процессора — 233 МГц и выше.

В системах с процессором Pentium II используются две внешние шины, обе они синхронные. Шина памяти используется для доступа к главному динамическому ОЗУ; шина PCI используется для сообщения с устройствами ввода-вывода. Иногда к шине PCI подсоединяется **унаследованная** (то есть прежняя) шина, чтобы можно было подключать старые периферические устройства.

Система Pentium II может содержать один или два центральных процессора, которые разделяют общую память. В системе с двумя процессорами может возникнуть одна неприятная ситуация. Слово, считанное в одну из микросхем кэш-памяти и измененное там, может не записаться обратно в память, и если второй процессор попытается считать это слово, он получит неправильное значение. Чтобы предотвратить такую ситуацию, существуют специальные системы поддержки.

Pentium II существенно отличается от своих предшественников компоновкой. Все процессоры, начиная с 8088 и заканчивая Pentium Pro, были обычными микросхемами с выводами по бокам или снизу, которые вставлялись в разъемы. Микропроцессор Pentium II представляет собой так называемый **SEC (Single Edge Cartridge — картридж с однорядным расположением контактов).** Как видно из рис. 3.40, этот картридж представляет собой довольно большую пластиковую коробку, содержащую центральный процессор, двухуровневую кэш-память и торцевой соединитель для передачи сигналов. Он содержит 242 контакта.

Хотя у Intel были все основания перейти к такой модели, она повлекла за собой проблему, которую компания не могла предвидеть. Многие покупатели имеют привычку разбирать компьютер и искать микросхему процессора. Однако в первых системах Pentium II покупатели не могли найти процессор и громко жаловались («У моего компьютера нет процессора!»). Компания Intel разрешила эту проблему, приклеив изображение процессора (голограмму) на следующие модели SEC.

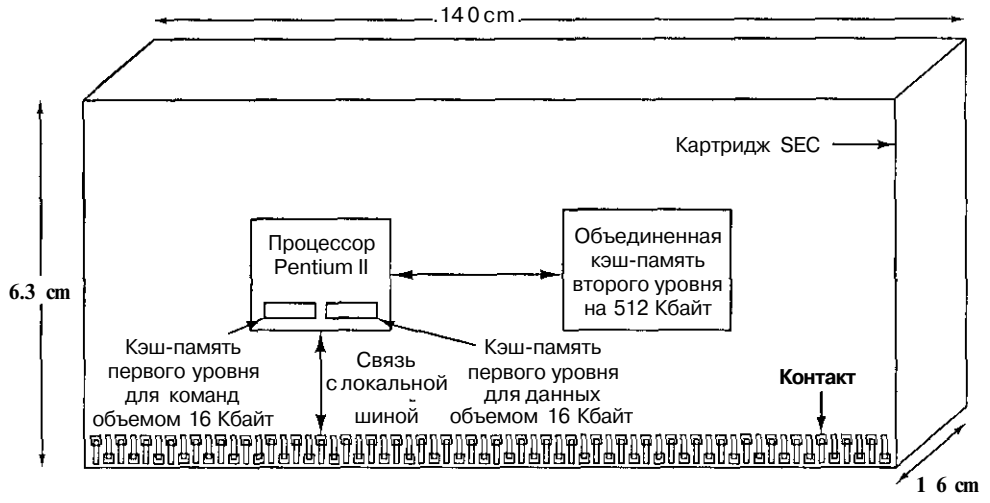


Рис. 3.40. Компоновка SEC

Главная проблема, связанная с процессором Pentium II, — управление режимом электропитания. Количество выделяемого тепла зависит от тактовой частоты, а выделяемая мощность при этом разнится от 30 до 50 Вт. Это огромная цифра для компьютерной микросхемы. Чтобы получить представление, что такое 50 Вт, поднесите руку к электрической лампочке в 50 Вт, которая включена уже некоторое время (только не дотрагивайтесь до нее). По этой причине SEC снабжен радиатором, чтобы рассеивать накопившееся тепло. Когда Pentium II перестанет быть рабочим процессором, его можно будет использовать в качестве нагревателя.

В соответствии с законами физики все, что выделяет большое количества тепла, должно потреблять большое количество энергии. В случае с портативным компьютером, который работает от батареи с ограниченным зарядом, потребление большого количества энергии нежелательно. Чтобы решить эту проблему, компания Intel нашла способ вводить центральный процессор в режим пониженного энергоснабжения (состояние «сна»), если он не выполняет никаких действий, и вообще отключать его (вводить в состояние «глубокого сна»), если есть вероятность, что он не будет выполнять никаких действий некоторое время. В последнем случае значения кэш-памяти и регистров сохраняются, а тактовый генератор и все внутренние блоки отключаются. Видит ли Pentium II сны во время «глубокого сна», науке пока не известно.

Цоколевка процессора Pentium II

Из 242 контактов картриджа SEC 170 используются для сигналов, 27 для питания (с различной мощностью), 35 для «земли» и еще 10 остаются на будущее. Для некоторых логических сигналов используется два и более выводов (например, для запроса адреса памяти), поэтому существует только 53 типа выводов. Цоколевка в несколько упрощенном виде представлена на рис. 3.41. С левой стороны рисунка показано 6 основных групп сигналов шины памяти; с правой стороны расположены прочие сигналы. Заглавными буквами обозначены названия самих сигналов,

а строчными — общие названия для групп связанных сигналов (в последнем случае только первая буква заглавная).

Компания Intel использует одно соглашение, которое важно понимать. Поскольку микросхемы разрабатываются с использованием компьютеров, нужно каким-то образом представлять названия сигналов в виде текста ASCII. Использовать черту над названиями сигналов, запускаемых низким напряжением, слишком сложно, вместо этого компания Intel помещает после названия сигнала знак #. Например, вместо обозначения BPRĪ используется BPRI#. Как видно из рисунка, большинство сигналов Pentium II запускаются низким напряжением.

Давайте рассмотрим различные типы сигналов. Начнем с сигналов шины. Первая группа сигналов используется для запроса шины (то есть для арбитража). Сигнал BPRI# позволяет устройству с высоким приоритетом получить доступ к шине раньше других устройств. Сигнал LOCK# позволяет центральному процессору не предоставлять остальным устройствам доступа к шине, пока работа не будет закончена.

Центральный процессор или другое задающее устройство шины может производить запрос на доступ к шине, используя следующую группу сигналов. Адреса состоят из 36 бит, но три последних бита должны всегда быть равны 0, и следовательно, они не имеют собственных выводов, поэтому A# содержит только 33 вывода. Все передачи состоят из 8 байтов. Поскольку адрес содержит 36 бит, максимальный объем памяти составляет 2^{36}, то есть 64 Гбайт.

Когда адрес передается на шину, устанавливается сигнал ADS#. Этот сигнал сообщает целевому объекту (например, памяти), что задействованы адресные линии. На линиях REQ# запускается цикл шины определенного типа (например, считывание слова или запись блока). Два сигнала четности нужны для проверки A#, а третий — для проверки ADS# и REQ#. Подчиненное устройство использует пять специальных линий для сообщения об ошибках четности. Эти же линии используются всеми устройствами для сообщения о каких-либо других ошибках.

Группа сигналов для отслеживания адресов, по которым происходило изменение данных, используется в многопроцессорных системах. Эти сигналы позволяют процессору обнаружить, есть ли слово, которое ему нужно, в кэш-памяти другого процессора. Как процессор Pentium II отслеживает эти адреса, мы рассмотрим в главе 8.

Ответные сигналы передаются от подчиненного к задающему устройству. Сигнал RS# содержит код состояния. Сигнал TRDY# показывает, что подчиненное устройство (целевой объект) готово принять данные от задающего устройства. Эти сигналы также проверяются на четность.

Последняя группа сигналов нужна для передачи данных. Сигнал D# используется, чтобы поместить 8 байтов данных на шину. Когда они туда помещаются, выдается сигнал DRDY# (сигнал наличия данных в шине). Он сообщает устройствам, что шина в данный момент занята.

Сигнал RESET# нужен для перезагрузки процессора в случае сбоя. Pentium II может осуществлять прерывания тем же способом, что и процессор 8088 (это требуется в целях совместимости), или использовать новую систему прерывания с устройством **APIC (Advanced Programmable Interrupt Controller — встроенный контроллер прерываний).**

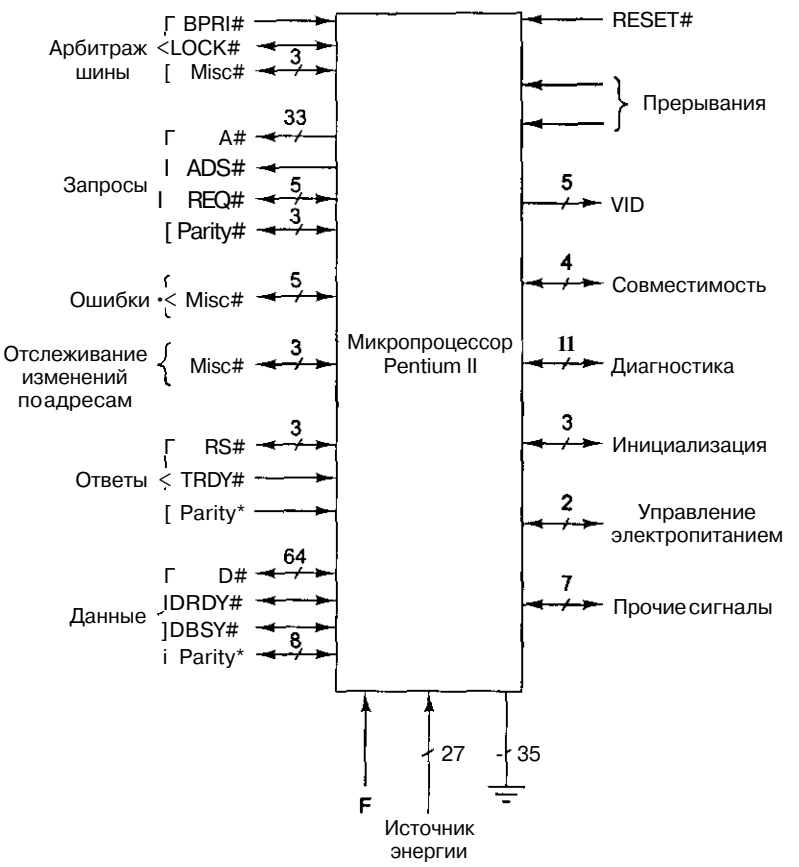


Рис. 3.41. Цоколевка процессора Pentium II. Официальные названия отдельных сигналов приведены заглавными буквами. Строчными буквами даны общие названия групп связанных сигналов или описания сигналов

Pentium II может работать при разном напряжении. Сигналы VID используются для автоматического выбора напряжения источника питания. Сигналы совместимости нужны для работы с устройствами более старых моделей, которые изначально предназначены для процессора 8088. Группа сигналов диагностики содержит сигналы для проверки и отладки системы в соответствии со стандартом IEEE 1149.1 JTAG. Сигналы инициализации связаны с загрузкой системы. Сигналы управления режимом электропитания позволяют процессору входить в состояние «сна» или «глубокого сна». Наконец, оставшаяся группа содержит сигналы разного рода. Сюда относится, например, сигнал, который выдается центральным процессором, если его внутренняя температура превышает 130°C (266°F). Хотя если температура центрального процессора превышает 130°C, он уже, вероятно, мечтает о выходе на пенсию и хочет служить в качестве нагревателя.

Конвейерный режим шины памяти процессора Pentium U

Современные процессоры, например Pentium II, работают гораздо быстрее современных динамических ОЗУ. Чтобы процессор не простаивал, необходима макси-

мально возможная производительность памяти. По этой причине шина памяти процессора Pentium II работает в конвейерном режиме, при этом в шине происходит одновременно 8 операций. Понятие конвейера мы рассматривали в главе 2, когда говорили о конвейерных процессорах. Отметим, что память тоже может быть конвейерной.

Обращения процессора к памяти, которые называются транзакциями, имеют 6 стадий:

1. Фаза арбитража шины.
2. Фаза запроса.
3. Фаза сообщения об ошибке.
4. Фаза проверки на наличие нужного слова в другом процессоре.
5. Фаза ответа.
6. Фаза передачи данных.

Наличие всех шести фаз необязательно. На фазе арбитража шины определяется, какое из задающих устройств будет следующим. На фазе запроса на шину передается адрес. На фазе сообщения об ошибке подчиненное устройство передает сигнал об ошибке четности в адресе или о наличии каких-либо других неполадок. На следующей фазе центральный процессор проверяет, нет ли нужного ему слова в другом процессоре. Эта стадия нужна только в многопроцессорных системах. В следующей фазе задающее устройство узнает, где взять необходимые данные. На последней стадии осуществляется передача данных.

В системе с процессором Pentium II на каждой стадии используются определенные сигналы, отличные от сигналов других стадий, поэтому каждая из них не зависит от остальных. Шесть групп необходимых сигналов показаны в левой части рис. 3.41. Например, один из процессоров может пытаться получить доступ к шине, используя сигналы арбитража. Как только процессор получает право на доступ к шине, он освобождает эти линии шины и занимает линии запроса. Тем временем другой процессор или какое-нибудь устройство ввода-вывода может войти в фазу арбитража шины и т. д. На рис. 3.42 показано, как осуществляется одновременно несколько транзакций.

Фаза арбитража шины на рис. 3.42 не показана, поскольку она не всегда нужна. Например, если устройство, обладающее в данный момент шиной (часто это центральный процессор), хочет произвести еще одну транзакцию, ему не требуется заново получать доступ к шине. Ему нужно запрашивать шину заново только в том случае, если он уступает ее другому устройству. Транзакции 1 и 2 обычные: пять фаз за пять циклов шины. Во время транзакции 3 вводится более длительная фаза передачи данных (поскольку, например, требуется передать целый блок или нужно ввести режим ожидания). Вследствие этого транзакция 4 не может начать фазу передачи данных сразу после стадии ответа Стадия передачи данных начинается только после того, как исчезнет сигнал DBSY#. Фаза ответа в транзакции 5 также может занимать несколько циклов шины, что задерживает транзакцию 6. Наконец, мы видим, что в транзакции 7 также происходит задержка, поскольку она уже появилась ранее. В действительности же маловероятно, что центральный процессор будет пытаться начать новую транзакцию на каждом цикле шины, поэтому простои не такие уж длительные.

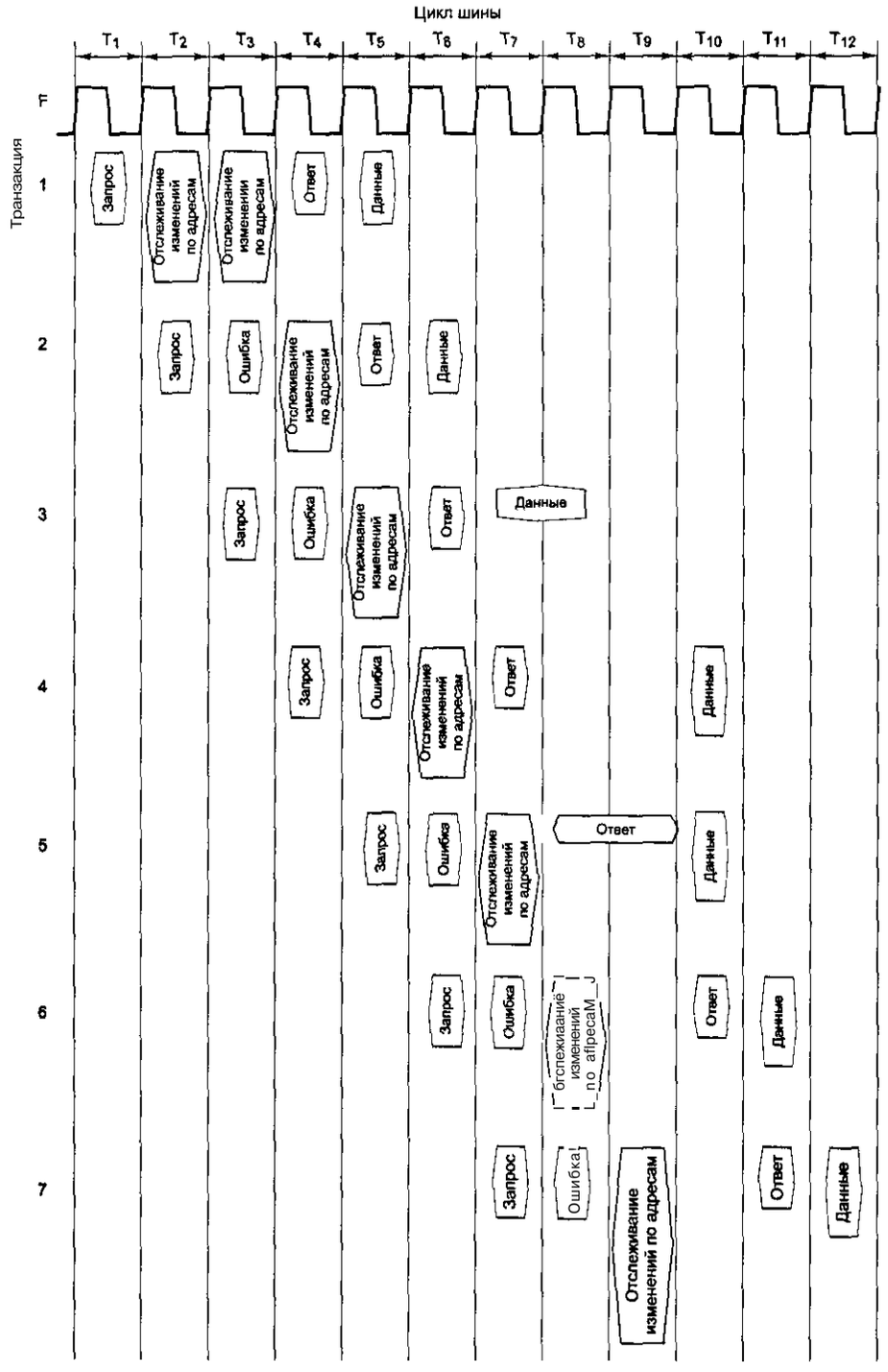


Рис. 3. Конвейерный режим шины памяти в системе с отслеживанием

UltraSPARC II

В качестве второго примера процессора возьмем семейство UltraSPARC (производитель — компания Sun). Семейство UltraSPARC — это серия 64-разрядных процессоров SPARC. Эти процессоры полностью соответствуют архитектуре Version 9 SPARC, которая также подходит для 64-разрядных процессоров. Они используются в рабочих станциях и серверах Sun, а также во многих других системах. Семейство включает в себя процессоры UltraSPARC I, UltraSPARC II и UltraSPARC III, которые имеют сходную архитектуру, но различаются датой выпуска и тактовой частотой. Ниже мы будем говорить о процессоре UltraSPARC II, поскольку нам нужен конкретный пример, но изложенная информация по большей части имеет силу и для других типов UltraSPARC.

UltraSPARC II представляет собой машину типа RISC. Он полностью совместим с 32-разрядным SPARC V8. Единственное, чем UltraSPARC II отличается от SPARC V9, — это наличием команд VIS, которые разработаны для графических приложений, кодировки MPEG в реальном времени и т. п.

Процессор UltraSPARC II был разработан для создания 4-узловых мультипроцессоров с разделенной памятью без добавления внешних схем, а также для создания более крупных мультипроцессоров с минимальным добавлением внешних схем. Иными словами, в каждую микросхему UltraSPARC II включены связующие элементы, необходимые для построения мультипроцессора.

В отличие от структуры Pentium II SEC, процессор UltraSPARC II представляет собой относительно большую самостоятельную микросхему, содержащую 5,4 млн транзисторов. Микросхема содержит 787 выводов, расположенных снизу, как показано на рис. 3.43. Такое большое число выводов объясняется, с одной стороны, использованием 64 битов для адресов и 128 битов для данных. С другой стороны, это объясняется особенностями работы кэш-памяти. Кроме того, многие выводы являются резервными. Число 787 было выбрано для того, чтобы промышленность могла производить стандартные модули. Компании, вероятно, считают простое число выводов счастливым.

Процессор UltraSPARC II содержит 2 внутренних блока кэш-памяти: 16 Кбайт для команд и 16 Кбайт для данных. Как и у Pentium II, здесь вне кристалла процессора расположена кэш-память второго уровня, но, в отличие от Pentium II, процессор UltraSPARC II не упакован в один картридж с кэш-памятью второго уровня, поэтому разработчики вправе выбирать любые микросхемы для кэш-памяти второго уровня.

Решение объединить кэш-память второго уровня с процессором или разделить ее с процессором обусловлено выбором между различными техническими преимуществами, а также особенностями компаний Intel и Sun. Внешняя кэш-память более гибкая (кэш-память процессора UltraSPARC II можно расширить с 512 Кбайт до 16 Мбайт; кэш-память процессора Pentium II имеет фиксированный объем 512 Кбайт), но при этом она работает медленнее из-за того, что расположена дальше от процессора. Для обращения к внешней кэш-памяти требуется больше сигналов (у картриджа SEC нет контактов для связи с кэш-памятью, поскольку в данном случае кэш-память встроена прямо в картридж), но среди 787 выводов процессора UltraSPARC II обязательно должны быть выводы для управления кэш-памятью.

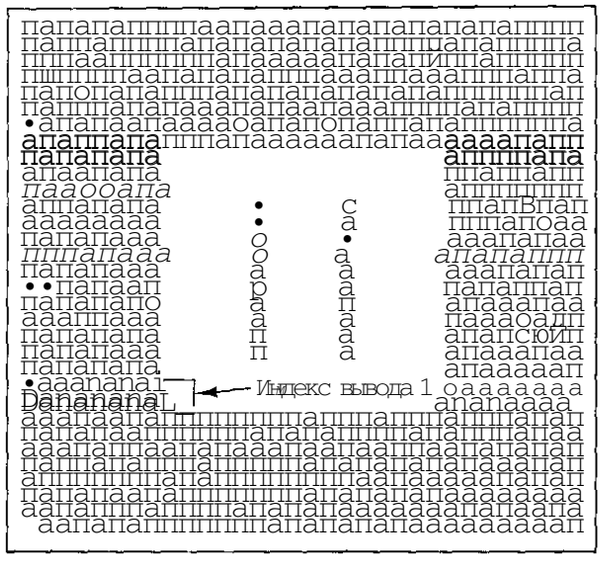


Рис. 3.43. Микросхема процессора UltraSPARC II

Что касается производственных особенностей, компания Intel является поставщиком полупроводниковых приборов, поэтому у нее есть возможность разрабатывать и выпускать собственные микросхемы кэш-памяти второго уровня и связывать их с центральным процессором через собственный интерфейс с высокими техническими характеристиками. Компания Sun, напротив, является производителем компьютеров, а не микросхем. Она иногда разрабатывает собственные микросхемы (например, UltraSPARC II), но поручает их производство предприятиям, выпускающим полупроводниковые приборы (например, Texas Instruments и Fujitsu). Иногда компания Sun предпочитает использовать микросхемы, имеющиеся в продаже. Статические ОЗУ для кэш-памяти второго уровня можно приобрести у различных производителей, поэтому у компании Sun не было особой необходимости разрабатывать собственные ОЗУ. А если ОЗУ не разрабатывается специально, то нужно устанавливать кэш-память второго уровня отдельно от центрального процессора.

Большинство рабочих станций Sun содержат синхронную шину на 25 МГц, которая называется **Sbus.** К этой шине могут подсоединяться устройства ввода-вывода. Однако шина Sbus работает слишком медленно и не подходит для памяти, поэтому компания Sun придумала другой механизм для соединения процессоров UltraSPARC II с памятью: **UPA (Ultra Port Architecture — высокоскоростной пакетный коммутатор).** UPA может воплощаться в виде шины, переключателя или сочетания того и другого. В различных рабочих станциях и серверах используются различные реализации UPA. Реализация UPA никак не зависит от процессора, поскольку интерфейс с UPA точно определен и процессор должен поддерживать (и поддерживает) именно этот интерфейс.

На рис. 3.44 мы видим ядро системы UltraSPARC II: центральный процессор, интерфейс UPA и кэш-память второго уровня (2 статических ОЗУ). На рисунке

также изображена микросхема **UDB II (UltraSPARC II Data Buffer II. Data Buffer — буфер данных),** функции которой мы обсудим ниже. Когда процессору нужно слово из памяти, сначала он обращается к кэш-памяти первого уровня. Если он находит слово, он продолжает выполнять операции с полной скоростью. Если он не находит слово в кэш-памяти первого уровня, он обращается к кэш-памяти второго уровня.

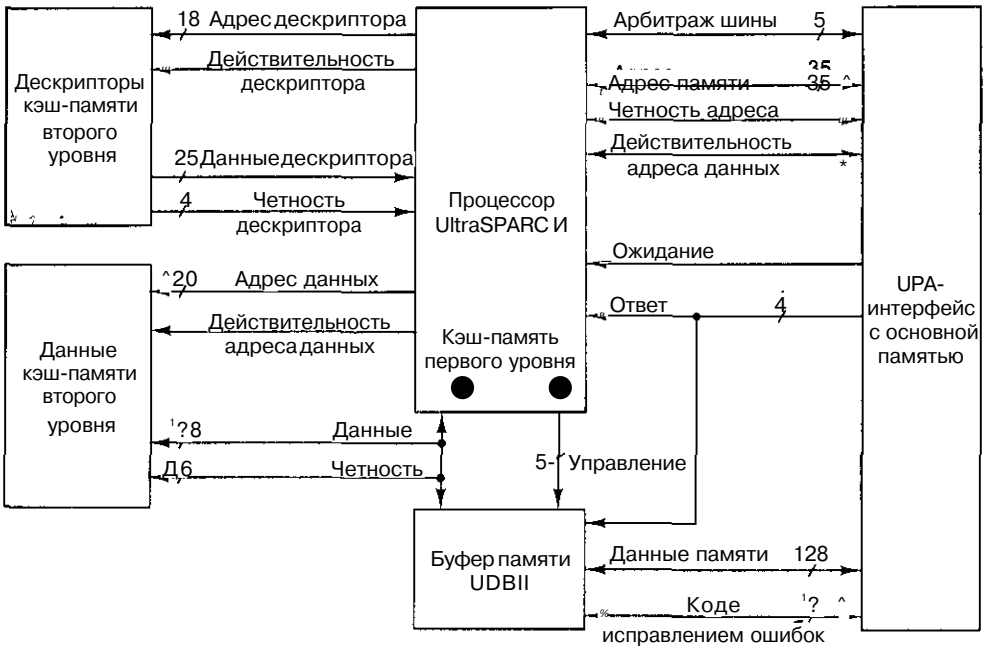


Рис. 3.44. Основная структура системы UltraSPARC II

Хотя мы в главе 4 будем подробно обсуждать работу кэш-памяти, все-таки стоит сказать здесь несколько слов об этом. Вся основная память подразделяется на строки кэш-памяти (блоки) по 64 байта. В кэш-памяти первого уровня находятся 256 наиболее часто используемых строк команд и 256 наиболее часто используемых строк данных. В кэш-памяти второго уровня содержатся строки, которые не поместились в кэш-память первого уровня. Кэш-память второго уровня содержит линии данных и команд вперемешку. Они хранятся в статическом ОЗУ, которое на рис. 3.44 обозначено прямоугольником с надписью «Данные кэш-памяти второго уровня». Система должна следить за тем, какие строки находятся в кэш-памяти второго уровня. Эта информация хранится во втором статическом ОЗУ, обозначенном на рис. 3.44 «Дескрипторы кэш-памяти второго уровня».

В случае отсутствия нужной строки в кэш-памяти первого уровня центральный процессор посылает идентификатор строки, которую он ищет (адрес дескриптора), в кэш-память второго уровня. Ответ (данные дескриптора) предоставляет центральному процессору информацию о том, есть ли нужная строка в кэш-памяти второго уровня. Если строка есть, центральный процессор получает ее. Передача данных осуществляется по 16 байтов, поэтому для пересылки целой строки в кэш-память первого уровня требуется 4 цикла.

Если требуемой строки нет в кэш-памяти второго уровня, ее нужно вызвать из основной памяти через интерфейс UPA. UPA в системе UltraSPARC II управляется централизованным контроллером. Туда поступают адресные сигналы и сигналы управления от центрального процессора (или процессоров, если их больше чем один). Чтобы получить доступ к памяти, центральный процессор должен сначала получить разрешение воспользоваться шиной. Когда шина предоставляется процессору, он получает сигнал с адресных выводов, определяет тип запроса и передает сигнал по нужному адресному выводу. (Эти выводы двунаправлены, поскольку другим процессорам в системе UltraSPARC II нужен доступ к отдаленным блокам кэш-памяти.) Адрес и тип цикла шины передаются на адресные выводы за два цикла, причем в первом цикле выдается строка, а во втором — столбец, как мы видели на рис. 3.30.

В ожидании результатов центральный процессор вполне может заниматься другой работой. Например, отсутствие нужной команды в кэш-памяти вовсе не мешает выполнению одной или нескольких команд, которые уже вызваны, и каждая из которых может обращаться к данным не из кэш-памяти. Таким образом, сразу несколько транзакций к UPA могут ожидать выполнения. Система UPA может справляться с двумя независимыми потоками транзакций (обычно это чтение и запись), каждый поток проходит с несколькими задержками. Задача централизованного контроллера — следить за всем этим и производить обращения к памяти в наиболее рациональном порядке.

Данные из памяти могут поступать блоками по 8 байтов. Они содержат 16-битный код с исправлением ошибок для большей надежности. Можно запрашивать весь блок кэш-памяти, 8 байтов или даже меньше. Все входные данные поступают в буфер UDB и хранятся там. Буфер UDB нужен для того, чтобы дать возможность центральному процессору и памяти работать асинхронно. Например, если центральному процессору необходимо записать слово или строку кэш-памяти в основную память, он может не ждать доступа к UPA, а сразу записать данные в буфер UDB, который доставит их в память позднее. UDB также генерирует код с исправлением ошибок. Отметим, что описание процессоров UltraSPARC II и Pentium II в этой книге сильно упрощено. Тем не менее мы изложили основную суть их работы.

PicoJava II

Pentium II и UltraSPARC II — процессоры с высокой производительностью, которые были разработаны для построения быстрых персональных компьютеров и рабочих станций. Существуют и другие компьютеры: так называемые встроенные системы. Именно их мы и рассмотрим кратко в этом разделе.

Не будет преувеличением сказать, что практически любое электронное устройство стоимостью более 100 долларов содержит встроенный компьютер. Телевизоры, сотовые телефоны, электронные записные книжки, микроволновые печи, видеокамеры, видеомагнитофоны, лазерные принтеры, охранные сигнализации, слуховые аппараты, электронные игры и многие другие устройства (их можно перечислять до бесконечности) управляются компьютером. При этом упор делается не на высокую производительность, а на низкую стоимость встроенного компьютера, что приводит к несколько другому соотношению преимуществ и недостатков по сравнению с процессорами, которые мы обсуждали до сих пор.

Традиционно встроенные процессоры программировались на языке ассемблер, но так как с течением времени приборы усложнялись и последствия сбоев программного обеспечения становились более серьезными, появились другие подходы. Особенно удобно использовать в качестве языка программирования для встроенных систем язык Java, поскольку он относительно прост и программы занимают мало места. К достоинствам также можно отнести независимость базовых программных средств. Однако у этого языка есть и недостатки. Во-первых, чтобы использовать язык Java во встроенных системах, требуется большой интерпретатор для выполнения кода JVM. (Программу на языке Java в код JVM преобразует специальный компилятор.) Во-вторых, процесс интерпретации занимает много времени.

Чтобы разрешить эту проблему, Sun и другие компании разработали процессоры со встроенным набором команд JVM. При таком подходе сочетаются и простота использования языка Java, и мобильность, и небольшой размер бинарного кода JVM, порождаемого компилятором, и высокая скорость выполнения операций, которая достигается благодаря особенностям аппаратного обеспечения. В этом разделе мы рассмотрим один из процессоров, который был разработан специально для встроенных систем.

Речь идет о процессоре picojava II, который составляет основу микросхемы microjava 701. Микросхема была разработана компанией Sun, но другие компании также имеют право использовать эту разработку. Это однокристальный процессор с двумя интерфейсами шины: один из них предназначен для шины памяти шириной в 64 бита, а другой — для шины PCI, как показано на рис. 3.45. Как Pentium II и UltraSPARC II, данный процессор может содержать кэш-память первого уровня (до 16 Кбайт для команд и до 16 Кбайт для данных). Но, в отличие от этих двух процессоров, он не имеет кэш-памяти второго уровня, поскольку низкая стоимость является ключевым параметром при разработке встроенных систем. Ниже мы рассмотрим микросхему microjava II 701. Она небольшого размера: содержит всего 2 млн транзисторов плюс еще 1,5 млн для кэш-памяти.

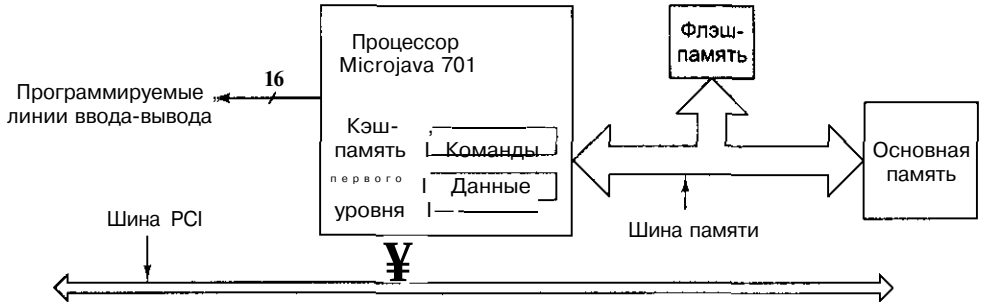


Рис. 3.45. Система microjava H 701

На рис. 3.45 видны три особенности микросхемы. Во-первых, в микросхеме microjava 701 используется шина PCI (на частоте 33 МГц или 66 МГц). Эта шина была разработана компанией Intel для использования в системах Pentium, но она подходит и для других процессоров. Преимущество шины PCI состоит в том, что она является стандартной, и поэтому не нужно каждый раз разрабатывать новую

шину. Кроме того, существует огромное количество сменных плат для этой шины. Хотя платы **PCI** и не играют большой роли при создании сотовых телефонов, они могут пригодиться для различных устройств большого размера (например, web-TV).

Во-вторых, система microjava II701 обычно содержит флэш-память. Дело в том, что в прибор должна быть встроена если не вся программа, то по крайней мере ее большая часть. Флэш-память хорошо подходит для хранения программы, поэтому полезно иметь соответствующий интерфейс. Другая микросхема (на рисунке она не показана), которую можно добавить к системе, содержит последовательные **и** параллельные интерфейсы ввода-вывода.

В-третьих, microjava 701 содержит 16 программируемых линий ввода-вывода, которые можно связать с кнопками, переключателями и лампочками прибора. Например, у микроволновой печи обычно есть клавишная панель с цифрами и несколько дополнительных кнопок, которые можно было бы соединить непосредственно с процессором. Наличие программируемых линий ввода-вывода на процессоре исключает необходимость использования программируемых контроллеров ввода-вывода, что делает прибор проще и дешевле. В микросхему microjava 701 встроено также три программируемых тактовых генератора, которые могут быть полезны, приборы часто работают в реальном времени.

Микросхема microjava 701 выпускается в стандартном корпусе **BGA (Ball Grid Array — корпус с выводами в виде сетки крошечных шариков).** Он содержит 316 выводов. Из них 59 выводов связаны с шиной PCI. Ниже в этой главе мы рассмотрим шину PCI подробно. Еще 123 вывода предназначены для шины памяти, среди них есть 64 двунаправленных вывода для передачи данных, а также отдельные адресные выводы. Остальные выводы используются для управления (7), синхронизирующих импульсов (3), прерываний (11), проверки (10), ввода-вывода(16). Некоторые из оставшихся выводов используются для питания и «земли», а остальные вообще не используются. Другие производители процессора picojava II вправе выбирать иную шину, компоновку и т. д.

У данной микросхемы есть много других особенностей. Она, например, может переходить в режим ожидания (чтобы экономить заряд батарейки), она содержит встроенный контроллер прерывания, она также имеет полную поддержку для стандарта тестирования IEEE 1149.1 JTAG.

Примеры шин

Шины соединяют компьютерную систему в одно целое. В этом разделе мы рассмотрим несколько примеров шин: шину ISA, шину PCI и Universal Serial Bus (универсальную последовательную шину). Шина ISA представляет собой небольшое расширение первоначальной шины IBM PC. По соображениям совместимости она все еще используется во всех персональных компьютерах Intel'. Однако такие компьютеры всегда содержат еще одну шину, которая работает быстрее, чем шина ISA.

[1] Шина ISA не используется в современных компьютерах. Уже несколько лет компания Intel настоятельно рекомендует разработчикам компьютеров не использовать эту шину. — *Примеч. научи, ред.*

Это шина PCI. Она шире, чем ISA, и функционирует с более высокой тактовой частотой. Шина USB обычно используется в качестве шины ввода-вывода для периферийных устройств малого быстродействия (например, мыши и клавиатуры). В следующих разделах мы рассмотрим каждую из этих шин по очереди.

Шина ISA

Шина IBM PC была неофициальным стандартом систем с процессором 8088, поскольку практически все производители клонов скопировали ее, чтобы иметь возможность использовать в своих системах платы ввода-вывода от различных поставщиков. Шина содержала 62 сигнальные линии, из них 20 для адреса ячейки памяти, 8 для данных и по одной для сигналов считывания информации из памяти, записи информации в память, считывания с устройства ввода-вывода и записи на устройство ввода-вывода. Имелись и сигналы для запроса прерываний и их разрешения, а также для прямого доступа к памяти. Шина была очень примитивной.

Шина IBM PC встраивалась в материнскую плату персонального компьютера. На плате было несколько разъемов, расположенных на расстоянии 2 см друг от друга. В разъемы вставлялись различные платы. На платах имелись позолоченные выводы (по 31 с каждой стороны), которые физически подходили под разъемы. Через них осуществлялся электрический контакт с разъемом.

Когда компания IBM разрабатывала компьютер PC/AT с процессором 80286, она столкнулась с некоторыми трудностями. Если бы компания разработала совершенно новую 16-битную шину, многие потенциальные покупатели не стали бы приобретать этот компьютер, поскольку ни одна из сменных плат, выпускаемых другими компаниями, не подошла бы к новой машине. С другой стороны, если оставить старую шину, то новый процессор не сможет реализовать все свои возможности (например, возможность обращаться к 16 Мбайт памяти и передавать 16-битные слова).

В результате было принято решение расширить старую шину. Сменные платы персональных компьютеров содержали краевой разъем (62 контакта), но этот краевой разъем проходил не по всей длине платы. Поэтому на плате поместили еще один краевой разъем, смежный с главным. Кроме того, схемы PC/AT были разработаны таким образом, чтобы можно было подсоединять платы обоих типов. На рис. 3.46 изображена шина PC/AT.

Второй краевой разъем шины PC/AT содержит 36 линий. Из них 31 предназначена для дополнительных адресных линий, информационных линий, линий прерывания, каналов ПДП (прямого доступа к памяти), а также для питания и «земли». Остальные связаны с различиями между 8-битными и 16-битными передачами.

Когда компания IBM выпустила серию компьютеров PS/2, пришло время начать разработку шины заново. С одной стороны, это решение было обусловлено чисто техническими причинами (шина PC к тому времени уже устарела). Но с другой стороны, оно было вызвано желанием воспрепятствовать компаниям, выпускавшим клоны, которые в то время заполонили компьютерный рынок. Поэтому компьютеры PS/2 с высокой и средней производительностью были оснащены абсолютно новой шиной MCA (MicroChannel Architecture), которая была защищена патентами.

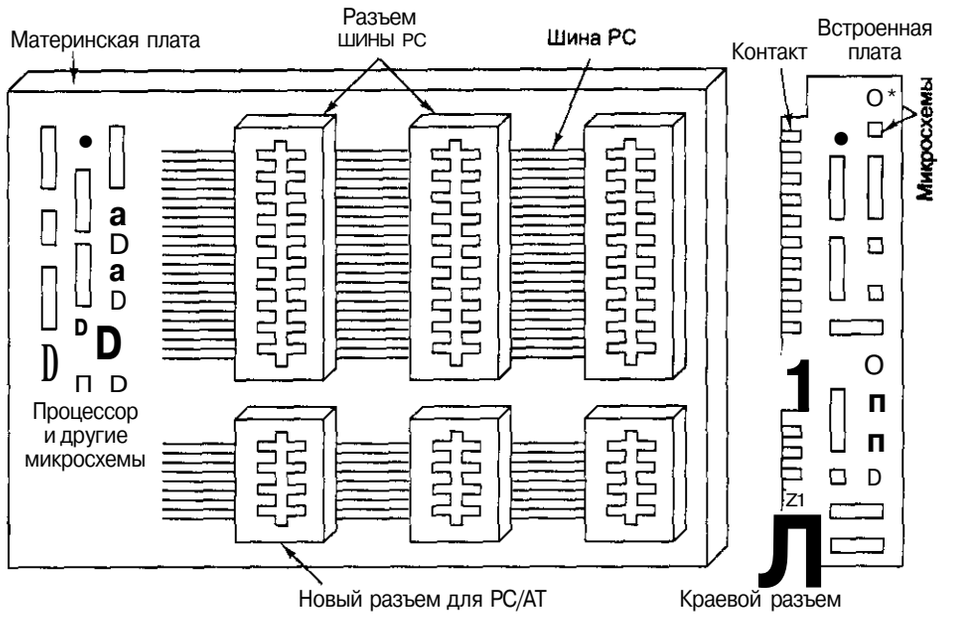


Рис. 3.46. Шина PC/AT состоит из двух компонентов' старой и новой шины

Компьютерная промышленность отреагировала на такой шаг введением своего собственного стандарта, шины **ISA (Industry Standard Architecture — стандартная промышленная архитектура),** которая, по существу, представляет собой шину PC/AT, работающую при частоте 8,33 МГц. Преимущество такого подхода состоит в том, что при этом сохраняется совместимость с существующими машинами и платами. Отметим, что в основе этого стандарта лежит шина, разработанная компанией IBM. IBM когда-то необдуманно предоставила права на производство этой шины многим компаниям, чтобы как можно больше производителей имели возможность выпускать платы для компьютеров IBM. Однако впоследствии компании IBM пришлось об этом сильно пожалеть. Эта шина до сих пор используется во всех персональных компьютерах с процессором Intel, хотя обычно кроме нее там есть еще одна или несколько других шин. Исчерпывающее описание шины ISA можно найти в книге [127].

Позднее шина ISA была расширена до 32 разрядов. У нее появились некоторые новые особенности (например, возможность параллельной обработки). Такая шина называлась **EISA (Extended Industry Standard Architecture — расширенная архитектура промышленного стандарта).** Для нее было разработано несколько плат.

Шина PCI

В первых компьютерах IBM PC большинство приложений имели дело с текстами. Постепенно с появлением Windows вошли в употребление графические интерфейсы пользователя. Ни одно из этих приложений не давало большой нагрузки на шину ISA. Однако с течением времени появилось множество различных приложе-

ний, в том числе игр, для которых потребовалось полноэкранное видеоизображение, и ситуация коренным образом изменилась.

Давайте произведем небольшое вычисление. Рассмотрим монитор 1024x768 для цветного движущегося изображения (3 байта/пиксел). Одно экранное изображение содержит 2,25 Мбайт данных. Для показа плавных движений требуется 30 кадров в секунду, и следовательно, скорость передачи данных должна быть 67,5 Мбайт/с. В действительности дело обстоит гораздо хуже, поскольку чтобы передать изображение, данные должны перейти с жесткого диска, компакт-диска или DVD-диска через шину в память. Затем данные должны поступить в графический адаптер (тоже через шину). Таким образом, пропускная способность шины должна быть 135 Мбайт/с, и это только для передачи видеоизображения. Но в компьютере есть еще центральный процессор и другие устройства, которые тоже должны пользоваться шиной, поэтому пропускная способность должна быть еще выше.

Максимальная частота передачи данных шины ISA — 8,33 МГц. Она способна передавать два байта за цикл, поэтому ее максимальная пропускная способность составляет 16,7 Мбайт/с. Шина EISA может передавать 4 байта за цикл. Ее пропускная способность достигает 33,3 Мбайт/с. Ясно, что ни одна из них совершенно не соответствует тому, что требуется для полноэкранного видео.

В 1990 году компания Intel разработала новую шину с гораздо более высокой пропускной способностью, чем у шины EISA. Эту шину назвали **PCI (Peripheral Component Interconnect— взаимодействие периферийных компонентов).** Компания Intel запатентовала шину PCI и сделала все патенты всеобщим достоянием, так что любая компания могла производить периферические устройства для этой шины без каких-либо выплат за право пользования патентом. Компания Intel также сформировала промышленный консорциум Special Interest Group, который должен был заниматься дальнейшими усовершенствованиями шины PCI. Все эти действия привели к тому, что шина PCI стала чрезвычайно популярной. Фактически в каждом компьютере Intel (начиная с Pentium), а также во многих других компьютерах содержится шина PCI. Даже компания Sun выпустила версию UltraSPARC, в которой используется шина PCI (это компьютер UltraSPARC Ш). Подробно шина PCI описывается в книгах [128,136].

Первая шина PCI передавала 32 бита за цикл и работала с частотой 33 МГц (время цикла 30 не), общая пропускная способность составляла 133 Мбайт/с. В 1993 году появилась шина PCI 2.0, а в 1995 году - PCI 2.1. Шина PCI 2.2 подходит и для портативных компьютеров (где требуется экономия заряда батареи). Шина PCI работает с частотой 66 МГц, способна передавать 64 бита за цикл, а ее общая пропускная способность составляет 528 Мбайт/с. При такой производительности полноэкранное видеоизображение вполне достижимо (предполагается, что диск и другие устройства системы справляются со своей работой). Во всяком случае, шина PCI не будет ограничивать производительность системы.

Хотя 528 Мбайт/с — достаточно высокая скорость передачи данных, все же здесь есть некоторые проблемы. Во-первых, этого не достаточно для шины памяти. Во-вторых, эта шина не совместима со всеми старыми картами ISA. По этой причине компания Intel решила разрабатывать компьютеры с тремя и более шинами, как показано на рис. 3.47. Здесь мы видим, что центральный процессор может обмениваться информацией с основной памятью через специальную шину памяти и что

шину ISA можно связать с шиной PCI. Такая архитектура используется фактически во всех компьютерах Pentium II, поскольку она удовлетворяет всем требованиям.

Ключевыми компонентами данной архитектуры являются мосты между шинами (эти микросхемы *выпускает* компания *Intel — отсюда такой* интерес к проекту). Мост PCI связывает центральный процессор, память и шину PCI. Мост ISA связывает шину PCI с шиной ISA, а также поддерживает один или два диска IDE. Практически все системы Pentium II выпускаются с одним или несколькими свободными слотами PCI для подключения дополнительных высокоскоростных периферийных устройств и с одним или несколькими слотами ISA для подключения низкоскоростных периферийных устройств.

Преимущество системы, изображенной на рис. 3.47, состоит в том, что шина между центральным процессором и памятью имеет высокую пропускную способность, шина PCI также обладает высокой пропускной способностью и хорошо подходит для связи с быстрыми периферийными устройствами (SCSI-дисками, графическими адаптерами и т. п.), и при этом еще могут использоваться старые платы ISA. На рисунке также изображена шина USB, которую мы будем обсуждать ниже в этой главе.

Мы проиллюстрировали систему с одной шиной PCI и одной шиной ISA. На практике может использоваться и по несколько шин каждого типа. Существуют специальные мосты, которые связывают две шины PCI, поэтому в больших системах может содержаться несколько отдельных шин PCI (2 и более). В системе также может быть несколько мостов (2 и более), которые связывают шину PCI и шину ISA, что дает возможность использовать несколько шин ISA.

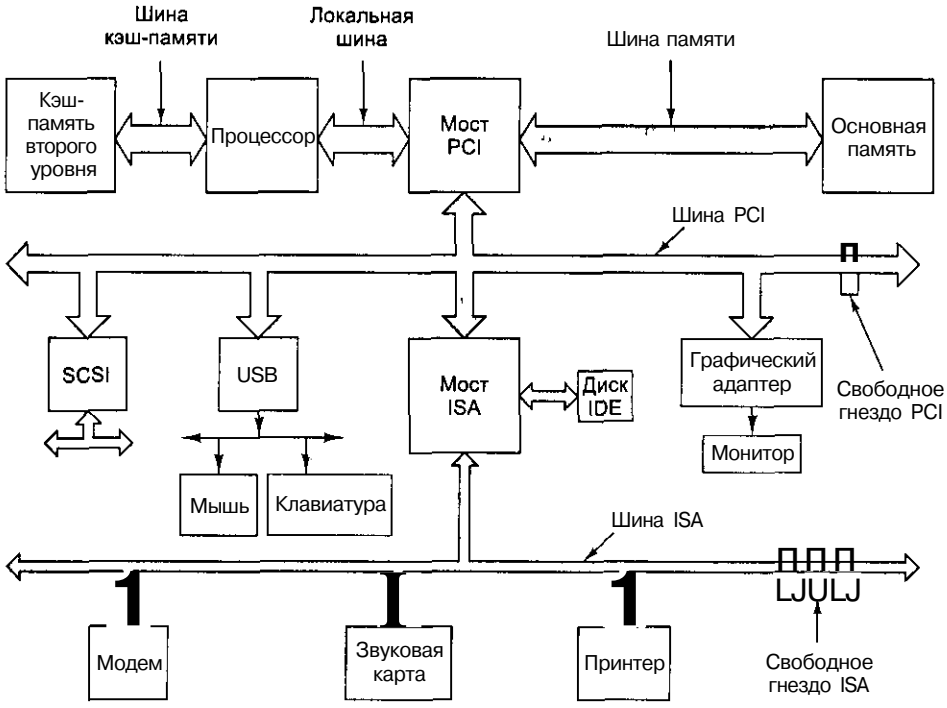


Рис. 3.47. Архитектура типичной системы Pentium И. Чем толще стрелка, обозначающая шину, тем выше пропускная способность этой шины

Было бы неплохо, если бы существовал только один тип плат PCI. К сожалению, это не так. Платы различаются по потребляемой мощности, разрядности и синхронизации. Старые компьютеры обычно используют напряжение 5 В, а новые — 3,3 В, поэтому шина PCI поддерживает и то и другое напряжение. Коннекторы одни и те же (они отличаются только двумя кусочками пластмассы, которые предназначены для того, чтобы невозможно было вставить плату на 5 В в шину PCI на 3,3 В и наоборот). К счастью, существуют и универсальные платы, которые поддерживают оба напряжения и которые можно вставить в любой слот. Платы различаются не только по мощности, но и по разрядности. Существует два типа плат: 32-битные и 64-битные. 32-битные платы содержат 120 выводов; 64-битные платы содержат те же 120 выводов плюс 64 дополнительных вывода (аналогично тому, как шина IBM PC была расширена до 16 битов, см. рис. 3.46). Шина PCI, поддерживающая 64-битные платы, может поддерживать и 32-битные, но обратное не верно. Наконец, шины PCI и соответствующие платы могут работать с частотой или 33 МГц, или 66 МГц. В обоих случаях контакты идентичны. Различие состоит в том, что один из выводов связывается либо с источником питания, либо с «землей».

Шины PCI являются синхронными, как и все шины PC, восходящие к первой модели IBM PC. Все транзакции в шине PCI осуществляются между задающим и подчиненным устройствами. Чтобы не увеличивать число выводов на плате, адресные и информационные линии объединяются. При этом достаточно 64 выводов для всей совокупности адресных и информационных сигналов, даже если PCI работает с 64-битными адресами и 64-битными данными.

Объединенные адресные и информационные выводы функционируют следующим образом. При операции считывания во время цикла 1 задающее устройство передает адрес на шину. Во время цикла 2 задающее устройство удаляет адрес и шина реверсируется таким образом, чтобы подчиненное устройство могло ее использовать. Во время цикла 3 подчиненное устройство выдает запрашиваемые данные. При операциях записи шине не нужно переключаться, поскольку задающее устройство помещает на нее и адрес, и данные. Тем не менее минимальная транзакция занимает три цикла. Если подчиненное устройство не может дать ответ в течение трех циклов, то вводится режим ожидания. Допускаются пересылки блоков неограниченного размера, а также некоторые другие типы циклов шины.

Арбитраж шины PCI

Чтобы передать по шине PCI какой-нибудь сигнал, устройство сначала должно получить к ней доступ. Шина PCI управляется централизованным арбитром, как показано на рис. 3.48. В большинстве случаев арбитр шины встраивается в один из мостов между шинами. От каждого устройства PCI к арбитру тянутся две специальные линии. Одна из них (REQ#) используется для запроса шины, а вторая (GNT#) — для получения разрешения на доступ к шине.

Чтобы сделать запрос на доступ к шине, устройство PCI (в том числе и центральный процессор) устанавливает сигнал REQ# и ждет, пока арбитр не выдаст сигнал GNT#. Если арбитр выдал сигнал GNT#, то устройство может использо-

вать шину в следующем цикле. Алгоритм, которым руководствуется арбитр, не зависит от технических характеристик шины PCI. Допустим арбитраж по кругу, по приоритету и другие схемы арбитража. Хороший арбитр должен быть справедлив, чтобы не заставлять некоторые устройства ждать целую вечность.

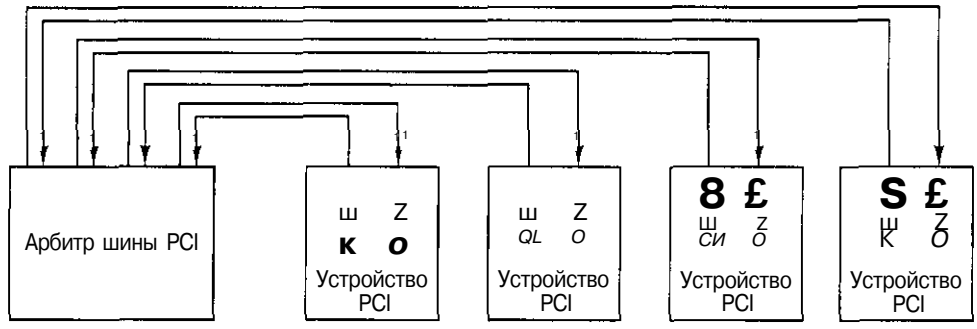


Рис. 3.48. В шине PCI используется централизованный арбитр

Шина предоставляется для одной транзакции, хотя продолжительность этой транзакции теоретически произвольна. Если устройству нужно совершить вторую транзакцию и ни одно другое устройство не запрашивает шину, оно может занять шину снова, хотя обычно между транзакциями нужно вставлять пустой цикл. Однако при особых обстоятельствах (при отсутствии конкуренции на доступ к шине) устройство может совершать последовательные транзакции без пустых циклов между ними. Если задающее устройство осуществляет очень длительную передачу, а какое-нибудь другое устройство выдало запрос на доступ к шине, арбитр может сбросить линию GNT#. Предполагается, что задающее устройство следит за линией GNT#. Если линия сбрасывается, устройство должно освободить шину в следующем цикле. Такая система позволяет осуществлять очень длинные передачи (что весьма рационально) при отсутствии конкуренции на доступ к шине, однако при этом она быстро реагирует на запросы шины, поступающие от других устройств.

Сигналы шины PCI

Шина PCI содержит ряд обязательных сигналов (табл. 3.5) и ряд факультативных сигналов (табл. 3.6). Оставшиеся выводы используются для питания, «земли» и разнообразных связанных сигналов. В столбцах «Задающее устройство» и «Подчиненное устройство» указывается, какое из устройств устанавливает сигнал при обычной транзакции. Если сигнал выдается другим устройством (например, CLK), оба столбца остаются пустыми.

Теперь давайте рассмотрим каждый сигнал шины PCI отдельно. Начнем с обязательных (32-битных) сигналов, а затем перейдем к факультативным (64-битным). Сигнал CLK запускает шину. Большинство сигналов совпадают с ним во времени. В отличие от шины ISA, в шине PCI транзакция начинается на заднем фронте сигнала CLK, то есть не в начале цикла, а в середине.

Таблица 3.5. Обязательные сигналы шины PCI

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
CLK	1			Тактовый генератор (33 МГц или 66 МГц)
AD	**32**	x	x	Объединенные адресные и информационные линии
PAR	**1**	x		Бит четности для адреса или данных
C/BE#	**4**	x		1} команда шине 2) битовый массив, который показывает, какие байты из слова нужно считать (или записать)
FRAME*	1	x		Указывает, что установлены сигналы AD И C/BE
IRDY#	1	x		При чтении: задающее устройство готово принять данные; при записи: данные находятся в шине
IDS EL	1	x		Считывание пространства конфигураций
DEVSEL#	1		x	Подчиненное устройство распознало свой адрес и ждет сигнала
TRDY#	1		x	При чтении: данные находятся на линиях AD; при записи: подчиненное устройство готово принять данные
STOP#	1		x	Подчиненное устройство требует немедленно прервать текущую транзакцию
PERR#	1			Обнаружена ошибка четности данных
SERR#	1			Обнаружена ошибка четности адреса или системная ошибка
REQ#	1			Арбитраж шины: запрос на доступ к шине
GNT#	1			Арбитраж шины: предоставление шины
RST#	1			Перезагрузка системы и всех устройств

Сигналы AD (их 32) нужны для адресов и данных (для передач по 32 бита). Обычно адрес устанавливается во время первого цикла, а данные — во время третьего. Сигнал PAR — это бит четности для сигнала AD. Сигнал C/BE# выполняет две функции. Во время первого цикла он содержит команду (считать одно слово, считать блок и т. п.). Во время второго цикла он содержит массив из 4 битов, который показывает, какие байты 32-битного слова действительны. Используя сигнал C/BE#, можно считывать 1, 2 или 3 байта из слова, а также все слово целиком.

Сигнал FRAME# устанавливается задающим устройством, чтобы начать транзакцию. Этот сигнал сообщает подчиненному устройству, что адрес и команды в данный момент действительны. При чтении одновременно с сигналом FRAME# устанавливается сигнал IRDY#. Он сообщает, что задающее устройство готово принять данные. При записи сигнал IRDY# устанавливается позже, когда данные уже находятся в шине.

Сигнал **IDSEL** связан с тем, что у каждого устройства PCI должно быть пространство конфигураций на 256 байтов, которое другие устройства могут считывать (установив сигнал IDSEL). Это пространство конфигураций содержит характеристики устройства. В некоторых операционных системах структура Plug-and-Play (Режим автоматического конфигурирования) использует это пространство конфигураций, чтобы распознать, какие устройства подключены к шине.

А теперь рассмотрим сигналы, которые устанавливаются подчиненным устройством. Сигнал DEVSEL# означает, что подчиненное устройство распознало свой адрес на линиях AD и готово участвовать в транзакции. Если сигнал DEVSEL# не поступает в течение определенного промежутка времени, задающее устройство предполагает, что подчиненное устройство, к которому направлено обращение, либо отсутствует, либо неисправно.

Следующий сигнал — TRDY#. Его подчиненное устройство устанавливает при чтении, чтобы сообщить, что данные находятся на линиях AD, и при записи, чтобы сообщить, что оно готово принять данные.

Следующие три сигнала нужны для сообщения об ошибках. Один из них, сигнал STOP#, устанавливается подчиненным устройством, если произошла какая-нибудь неполадка и нужно прервать текущую транзакцию. Следующий сигнал, PERR#, используется для сообщения об ошибке четности в данных в предыдущем цикле. Для чтения этот сигнал устанавливается задающим устройством, для записи — подчиненным устройством. Необходимые действия должно предпринимать устройство, получившее этот сигнал. Наконец, сигнал SERR# нужен для сообщения об адресных и системных ошибках.

Таблица 3.6. Факультативные сигналы шины **PCI**

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
REQ64#	1	×		Запрос на осуществление 64-битной транзакции
ACK64#	1		×	Разрешение 64-битной транзакции
AD	32	×		Дополнительные 32 бита адреса или данных
PAR64	1	×		Проверка четности для дополнительных 32 битов адреса или данных
C/BE#	4	×		Дополнительные 4 бита для указания, какие байты из слова нужно считать (или записать)
LOCK	1	×		В многопроцессорных системах: блокировка шины при осуществлении транзакции одним из процессоров
SBO#	1			Обращение к кэш-памяти другого процессора
SDONE	1			Отслеживание адресов, по которым произошли изменения, завершено.
INTx	4			Запрос прерывания
JTAG	5			Сигналы тестирования IEEE 1149.1 JTAG
M66EN	1			Сигнал связывается с источником питания или с «землей» (66 МГц или 33 МГц)

Сигналы REQ# и GNT# предназначены для арбитража шины. Они устанавливаются не тем устройством, которое является задающим в данный момент, а тем, которому нужно стать задающим. Последний обязательный сигнал, RST#, используется для перезагрузки системы, которая происходит, либо если пользователь нажмет кнопку RESET, либо если какое-нибудь системное устройство обнаружит фатальную ошибку. После установки этого сигнала компьютер перезагружается.

Перейдем к факультативным сигналам, большинство из которых связано с расширением разрядности с 32 до 64 битов. Сигналы REQ64# и ACK 64# позволяют задающему устройству попросить разрешение осуществить 64-битную транзакцию, а подчиненному устройству принять эту транзакцию. Сигналы AD, PAR64 и C/BE# являются расширениями соответствующих 32-битных сигналов.

Следующие три сигнала не связаны с противопоставлением 32 бита — 64 бита. Они имеют отношение к многопроцессорным системам. Не все платы PCI поддерживают такие системы, поэтому эти сигналы являются факультативными. Сигнал LOCK позволяет блокировать шину для параллельных транзакций. Следующие два сигнала связаны с отслеживанием всех адресов, по которым происходит изменение данных. Подобное отслеживание необходимо для того, чтобы сохранить непротиворечивость кэш-памяти различных процессоров.

Сигналы INT* нужны для запроса прерываний. Плата PCI может содержать до четырех логических устройств, каждое из которых имеет собственную линию запроса прерывания. Сигналы JTAG предназначены для процедуры тестирования IEEE 1149.1 JTAG. Наконец, сигнал M66EN связывается либо с источником питания, либо с «землей», что определяет тактовую частоту. Она не должна меняться во время работы системы.

Транзакции шины PCI

Шина PCI в действительности очень проста. Чтобы лучше понять это, рассмотрим временную диаграмму на рис. 3.49. Здесь мы видим транзакцию чтения, за ней следует пустой цикл и транзакция записи, которая осуществляется тем же задающим устройством.

Во время цикла Ti на заднем фронте синхронизирующего сигнала задающее устройство помещает адрес на линии AD и команду на линии C/BE#. Затем задающее устройство устанавливает сигнал FRAME#, чтобы начать транзакцию.

Во время цикла T_2 задающее устройство переключает шину, чтобы подчиненное устройство могло воспользоваться ею во время цикла T_3. Задающее устройство также изменяет сигнал C/BE#, чтобы указать, какие байты в слове ему нужно считать.

Во время цикла T_3 подчиненное устройство устанавливает сигнал DEVSEL#. Этот сигнал сообщает задающему устройству, что подчиненное устройство получило адрес и собирается ответить. Подчиненное устройство также помещает данные на линии AD и выдает сигнал TRDY#, который сообщает задающему устройству о данном действии. Если подчиненное устройство не может ответить быстро, оно не снимает сигнал DEVSEL#, который сообщает о его присутствии, но при этом не устанавливает сигнал TRDY# до тех пор, пока не сможет передать данные. При такой процедуре вводится один или несколько периодов ожидания.

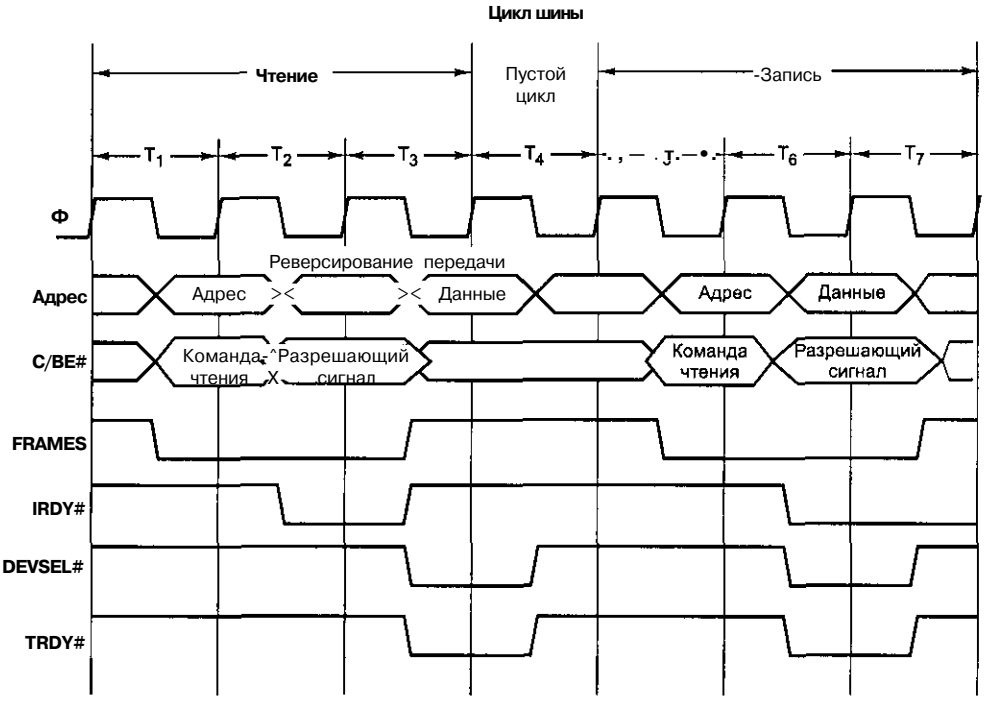


Рис. 3.49. Примеры 32-битных транзакций в шине PCI. Во время первых трех циклов происходит операция чтения, затем идет пустой цикл, а следующие три цикла — операция записи

В нашем примере (часто это бывает и в действительности) следующий цикл пустой. Мы видим, что в цикле T_5 то же самое задающее устройство инициирует процесс записи. Сначала оно, как обычно, помещает адрес и команду на шину. В следующем цикле оно выдает данные. Поскольку линиями AD управляет одно и то же устройство, цикл реверсирования передачи не требуется. В цикле T_7 память принимает данные.

Шина USB

Шина PCI очень хорошо подходит для подсоединения высокоскоростных периферических устройств, но использовать интерфейс PCI для низкоскоростных устройств ввода-вывода (например, мыши и клавиатуры) было бы слишком дорого. Изначально каждое стандартное устройство ввода-вывода подсоединялось к компьютеру особым образом, при этом для добавления новых устройств использовались свободные слоты ISA и PCI. К сожалению, такая схема имеет некоторые недостатки. Например, каждое новое устройство ввода-вывода часто снабжено собственной платой ISA или PCI. Пользователь при этом должен сам установить переключатели и перемычки на плате и увериться, что установленные параметры не конфликтуют с другими платами. Затем пользователь должен открыть системный блок, аккуратно вставить плату, закрыть системный блок, а затем включить компьютер. Для многих этот процесс очень сложен и часто приводит к ошибкам.

Кроме того, число слотов ISA и PCI очень мало (обычно их два или три). Платы Plug and Play исключают установку переключателей, но пользователь все равно должен открывать компьютер и вставлять туда плату. К тому же количество слотов шины ограничено.

В середине 90-х годов представители семи компаний (Compaq, DEC, IBM, Intel, Microsoft, NEC и Nothern Telecom) собрались вместе, чтобы разработать шину, оптимально подходящую для подсоединения низкоскоростных устройств. Потом к ним примкнули сотни других компаний. Результатом их работы стала шина **USB (Universal Serial Bus — универсальная последовательная шина),** которая сейчас широко используется в персональных компьютерах. Она подробно описана в книгах [7,144].

Некоторые требования, изначально составляющие основу проекта:

1. Пользователи не должны устанавливать переключатели и перемычки на платах и устройствах.
2. Пользователи не должны открывать компьютер, чтобы установить новые устройства ввода-вывода.
3. Должен существовать только один тип кабеля, подходящий для подсоединения всех устройств.
4. Устройства ввода-вывода должны получать питание через кабель.
5. Необходима возможность подсоединения к одному компьютеру до 127 устройств.
6. Система должна поддерживать устройства реального времени (например, звук, телефон).
7. Должна быть возможность устанавливать устройства во время работы компьютера.
8. Должна отсутствовать необходимость перезагружать компьютер после установки нового устройства.
9. Производство новой шины и устройств ввода-вывода для нее не должно требовать больших затрат.

Шина USB удовлетворяет всем этим условиям. Она разработана для низкоскоростных устройств (клавиатур, мышей, фотоаппаратов, сканеров, цифровых телефонов и т. д.). Общая пропускная способность шины составляет 1,5 Мбайт/с. Этого достаточно для большинства таких устройств. Предел был выбран для того, чтобы снизить стоимость шины.

Шина USB состоит из центрального хаба[1], который вставляется в разъем главной шины (см. рис. 3.47). Этот центральный хаб (часто называемый корневым концентратором) содержит разъемы для кабелей, которые могут подсоединяться к устройствам ввода-вывода или к дополнительным хабам, чтобы обеспечить большее количество разъемов. Таким образом, топология шины USB представляет собой дерево с корнем в центральном хабе, который находится внутри компьютера. Коннекторы кабеля со стороны устройства отличаются от коннекторов со стороны хаба, чтобы пользователь случайно не подсоединил кабель другой стороной.

[1] От англ. *hub* - концентратор. — *Примеч. пер.*

Кабель состоит из четырех проводов: два из них предназначены для передачи данных, один — для источника питания (+5 В) и один — для «земли». Система передает 0 изменением напряжения, а 1 — отсутствием изменения напряжения, поэтому длинная последовательность нулей порождает поток регулярных импульсов.

Когда подсоединяется новое устройство ввода-вывода, центральный хаб (концентратор) распознает это и прерывает работу операционной системы. Затем операционная система запрашивает новое устройство, что оно собой представляет и какая пропускная способность шины для него требуется. Если операционная система решает, что для этого устройства пропускной способности достаточно, она приписывает ему уникальный адрес (1-127) и загружает этот адрес и другую информацию в регистры конфигурации внутри устройства. Таким образом, новые устройства могут подсоединяться «на лету», при этом пользователю не нужно устанавливать новые платы ISA или PCI. Неинициализированные платы начинаются с адреса 0, поэтому к ним можно обращаться. Многие устройства снабжены встроенными сетевыми концентраторами для дополнительных устройств. Например, монитор может содержать два хаба для правой и левой колонок.

Шина USB представляет собой ряд каналов от центрального хаба к устройствам ввода-вывода. Каждое устройство может разбить свой канал максимум на 16 подканалов для различных типов данных (например, аудио и видео). В каждом канале или подканале данные перемещаются от центрального концентратора к устройству или обратно. Между двумя устройствами ввода-вывода обмена информацией не происходит.

Ровно через каждую миллисекунду (±0,05 мс) центральный концентратор передает новый кадр, чтобы синхронизировать все устройства во времени. Кадр состоит из пакетов, первый из которых передается от концентратора к устройству. Следующие пакеты кадра могут передаваться в том же направлении, а могут и в противоположном (от устройства к хабу). На р№\ 3.50 показаны четыре последовательных кадра.

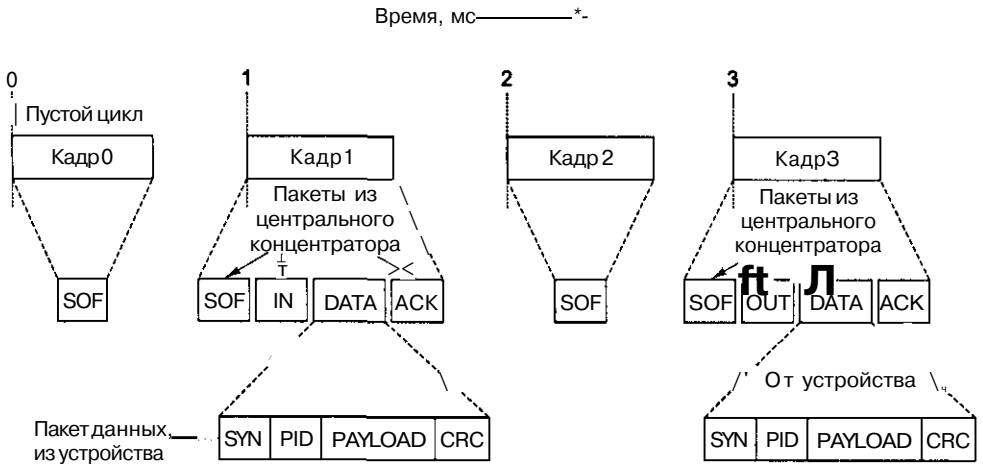


Рис. 3.50. Центральный концентратор шины USB передает кадры каждую миллисекунду

Рассмотрим рис. 3.50. В кадрах 0 и 2 не происходит никаких действий, поэтому в них содержится только пакет SOF (Start of Frame — начало кадра). Этот пакет всегда посылается всем устройствам. Кадр 1 — упорядоченный опрос (например, сканеру посылается запрос на передачу битов сканированного им изображения). Кадр 3 состоит из отсылки данных какому-нибудь устройству (например, принтеру).

Шина USB поддерживает 4 типа кадров: кадры управления, изохронные кадры, кадры передачи больших массивов данных и кадры прерывания. Кадры управления используются для конфигурации устройств, передачи команд устройствам и запросов об их состоянии. Изохронные кадры предназначены для устройств реального времени (микрофонов, акустических систем и телефонов), которые должны принимать и посылать данные через равные временные интервалы. Задержки хорошо прогнозируются, но в случае ошибки такие устройства не производят повторной передачи. Кадры следующего типа используются для передач большого объема от устройств и к устройствам без требований реального времени (например, принтеров). Наконец, кадры последнего типа нужны для того, чтобы осуществлять прерывания, поскольку шина USB не поддерживает прерывания. Например, вместо того чтобы вызывать прерывание всякий раз, когда происходит нажатие клавиши, операционная система может вызывать прерывания каждые 50 млс и «собирать» все задержанные нажатия клавиш.

Кадр состоит из одного или нескольких пакетов. Пакеты могут посылаться в обоих направлениях. Существует четыре типа пакетов: маркеры, пакеты данных, пакеты квитирования и специальные пакеты. Маркеры передаются от концентратора к устройству и предназначены для управления системой. Пакеты SOF, IN и OUT на рис. 3.50 — маркеры. Пакет SOF (Start of Frame — начало кадра) является первым в любом кадре и показывает начало кадра. Если никаких действий выполнять не нужно, пакет SOF единственный в кадре. Пакет IN — это запрос. Этот пакет требует, чтобы устройство выдало определенные данные. Поля в пакете IN содержат информацию, какой именно канал запрашивается, и таким образом устройство определяет, какие именно данные выдавать (если оно обращается с несколькими потоками данных). Пакет OUT объявляет, что далее последует передача данных для устройства. Последний тип маркера, SETUP (он не показан на рисунке), используется для конфигурации.

Кроме маркеров существует еще три типа пакетов. Это пакеты DATA (используются для передачи 64 байтов информации в обоих направлениях), пакеты квитирования и специальные пакеты. Формат пакета данных показан на рис. 3.50. Он состоит из 8-битного поля синхронизации, 8-битного указателя типа пакета (PID), полезной нагрузки и 16-битного **CRC (Cyclic Redunduncy Code — циклический избыточный код)** для обнаружения ошибок. Есть три типа пакетов квитирования: ACK (предыдущий пакет данных был принят правильно), NAC (найдена ошибка CRC) и STALL (подождите, пожалуйста, я сейчас занят).

А теперь давайте снова посмотрим на рис. 3.50. Центральный концентратор должен отсылать новый кадр каждую миллисекунду, даже если не происходит никаких действий. Кадры 0 и 2 содержат только один пакет SOF, что говорит о том, что ничего не происходит. Кадр 1 представляет собой опрос, поэтому он начинается с пакетов SOF и IN, которые передаются от компьютера к устройству ввода-вывода, а затем следует пакет DATA от устройства к компьютеру. Пакет ACK сообщает

устройству, что данные были получены без ошибок. В случае ошибки устройство получает пакет NACK, после чего данные передаются заново (отметим, что изохронные данные повторно не передаются) Кадр 3 похож по структуре на кадр 1, но в нем поток данных направлен от компьютера к устройству.

Средства сопряжения

Обычная компьютерная система малого или среднего размера состоит из микросхемы процессора, микросхем памяти и нескольких контроллеров ввода-вывода. Все эти микросхемы соединены шиной Мы уже рассмотрели память, центральные процессоры и шины. Теперь настало время изучить микросхемы ввода-вывода. Именно через эти микросхемы компьютер обменивается информацией с внешними устройствами.

Микросхемы ввода-вывода

В настоящее время существует множество различных микросхем ввода-вывода. Новые микросхемы появляются постоянно Из наиболее распространенных можно назвать UART, USART, контроллеры CRT (CRT — электронно-лучевая трубка), дисковые контроллеры и **PIO. UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик)** — это микросхема, которая может считывать байт из шины данных и передавать этот байт по битам на линию последовательной передачи к терминалу или получать данные от терминала. Скорость работы микросхем UART различна: от 50 до 19 200 бит/с; ширина знака от 5 до 8 битов; 1,1,5 или 2 стоповых бита. Микросхема может обеспечивать проверку на четность или на нечетность, контроль по четности может также отсутствовать, все это находится под управлением программ Микросхема **USART (Universal Synchronous Asynchronous Receiver Transmitter— универсальный синхронно-асинхронный приемопередатчик)** может осуществлять синхронную передачу, используя ряд протоколов. Она также выполняет все функции UART. Поскольку микросхемы UART мы уже рассматривали в главе 2, сейчас в качестве примера микросхемы ввода-вывода мы возьмем параллельный интерфейс.

Микросхемы PIO

Типичным примером микросхемы **PIO (Parallel Input/Output — параллельный ввод-вывод)** является Intel 8255A (рис. 3.51). Она содержит 24 линии ввода-вывода и может сопрягаться с любыми устройствами, совместимыми с TTL-схемами (например, клавиатурами, переключателями, индикаторами, принтерами) Программа центрального процессора может записать 0 или 1 на любую линию или считать входное состояние любой линии, обеспечивая высокую гибкость Микросхема PIO часто заменяет целую плату с микросхемами МИС и СИС (особенно во встроенных системах).

Центральный процессор может конфигурировать микросхему 8255A различными способами, загружая регистры состояния микросхемы, и мы остановимся на

некоторых наиболее простых режимах работы. Можно представить данную микросхему в виде трех 8-битных портов A, B и C. С каждым портом связан 8-битный регистр. Чтобы установить линии на порт, центральный процессор записывает 8-битное число в соответствующий регистр, и это 8-битное число появляется на выходных линиях и остается там до тех пор, пока регистр не будет перезаписан. Чтобы использовать порт для входа, центральный процессор просто считывает соответствующий регистр.

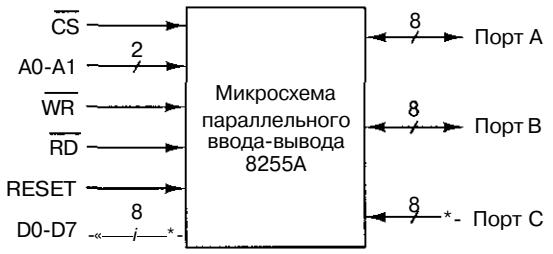


Рис. 3.51. Микросхема 8255A

Другие режимы работы предусматривают квитирование связи с внешними устройствами. Например, чтобы передать данные устройству, микросхема 8255A может представить данные на порт вывода и подождать, пока устройство не выдаст сигнал о том, что данные получены и можно посылать еще. В данную микросхему включены необходимые логические схемы для фиксирования таких импульсов и передачи их центральному процессору.

Из рис. 3.51 мы видим, что помимо 24 выводов для трех портов микросхема 8255A содержит восемь линий, непосредственно связанных с шиной данных, линию выбора элемента памяти, линии чтения и записи, две адресные линии и линию для переустановки микросхемы. Две адресные линии выбирают один из четырех внутренних регистров, три из которых соответствуют портам A, B и C. Четвертый регистр — регистр состояния. Он определяет, какие порты используются для входа, а какие для выхода, а также выполняет некоторые другие функции. Обычно две адресные линии соединяются с двумя младшими битами адресной шины.

Декодирование адреса

До настоящего момента мы не останавливались подробно на том, как происходит выбор микросхемы памяти или устройства ввода-вывода. Пришло время это узнать. Рассмотрим простой 16-битный встроенный компьютер, состоящий из центрального процессора, стираемого программируемого ПЗУ объемом 2Kx8 байт для хранения программы, ОЗУ объемом 2Kx8 байт для хранения данных и микросхемы PIO. Такая небольшая система может встраиваться в дешевую игрушку или простой прибор. Вместо стираемого программируемого ПЗУ может использоваться обычное ПЗУ.

Микросхема PIO может быть выбрана одним из двух способов: как устройство ввода-вывода или как часть памяти. Если микросхема нам нужна в качестве

устройства ввода-вывода, мы должны выбрать ее, используя внешнюю линию шины, которая показывает, что мы обращаемся к устройству ввода-вывода, а не к памяти. Если мы применяем другой подход, так называемый **ввод-вывод с распределением памяти,** мы должны присвоить микросхеме 4 байта памяти для трех портов и регистра управления. Наш выбор в какой-то степени произволен. Мы выбираем ввод-вывод с распределением памяти, поскольку этот метод наглядно иллюстрирует некоторые интересные проблемы сопряжения.

Стираемому программируемому ПЗУ требуется 2 К адресного пространства, ОЗУ требуется также 2 К адресного пространства, а микросхеме РЮ нужно 4 байта. Поскольку в нашем примере адресное пространство составляет 64 К, мы должны выбрать, где поместить данные три устройства. Один из возможных вариантов показан на рис. 3.52. Стираемое программируемое ПЗУ занимает адреса до 2 К, ОЗУ занимает адреса от 32 К до 34 К, а РЮ — 4 старших байта адресного пространства, от 65532 до 65535. С точки зрения программиста не важно, какие именно адреса использовать, однако для сопряжения это имеет большое значение. Если бы мы обращались к РЮ через пространство ввода-вывода, нам не потребовались бы адреса памяти (зато понадобились бы четыре адреса пространства ввода-вывода).

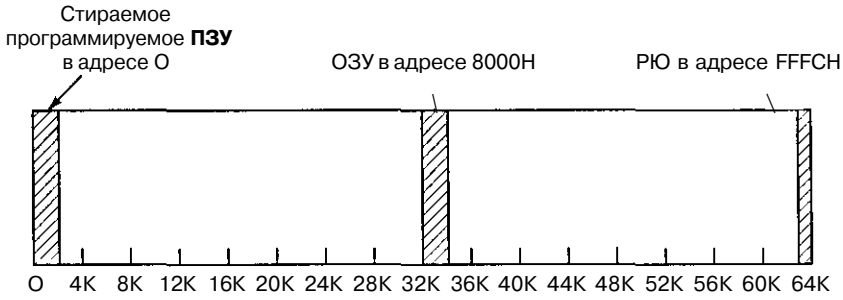


Рис. 3.52. Расположение стираемого ПЗУ, ОЗУ и РЮ на адресном пространстве в 64 К

При таком распределении адресов (рис. 3.52) стираемое ПЗУ нужно выбирать с помощью 16-битного адреса памяти ОООООхххххххххх (в двоичной системе). Другими словами, любой адрес, у которого пять старших битов равны 0, попадает в область памяти до 2 К и, следовательно, в стираемое ПЗУ. Таким образом, сигнал выбора стираемого ПЗУ можно связать с 5-разрядным компаратором, у которого один из входов всегда будет соединен с 00000.

Чтобы достичь того же результата, лучше было бы использовать пятивходовый вентиль ИЛИ, у которого пять входов связаны с адресными линиями от А11 до А15. Выходной сигнал будет равен 0 тогда и только тогда, когда все пять линий равны 0. В этом случае устанавливается сигнал US. К сожалению, в стандартных сериях МИС не существует пятивходовых вентилей ИЛИ. Однако мы можем использовать восьмивходовый вентиль НЕ-ИЛИ. Заземлив три входа и инвертировав выход, мы можем получить нужный нам сигнал (рис. 3.53, *а).* Схемы МИС стоят очень дешево, поэтому неэффективное использование одной из них вполне допустимо. По соглашению неиспользуемые входы на схемах не показываются.

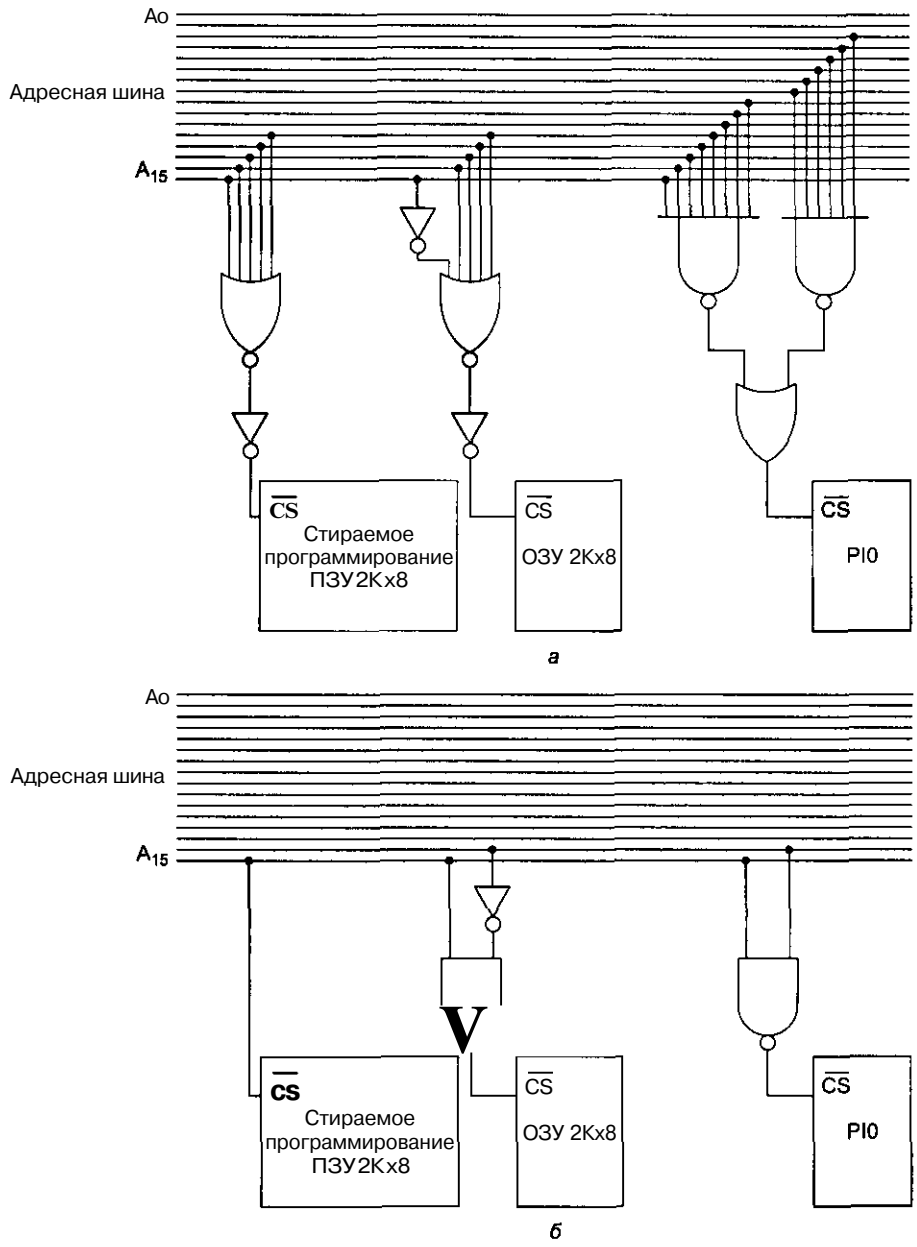


Рис. 3.53. Полное декодирование адреса (а); частичное декодирование адреса (б)

Тот же принцип можно применить и для ОЗУ. Однако ОЗУ должно отвечать на бинарные адреса типа ЮООООхххххххххх, поэтому необходим дополнительный инвертор (он показан на схеме). Декодирование адреса микросхемы PIO несколько сложнее, поскольку данная микросхема выбирается с помощью 4 адресов типа 1111111111111хх. Один из возможных вариантов схемы, которая устанавливает

сигнал CS только в том случае, если на адресной шине появляется адрес данного типа, показан на рис. 3.53. Здесь используются два восьмивходовых вентиля НЕ-И, которые соединяются с вентилем ИЛИ. Чтобы сконструировать схему декодирования адреса, изображенную на рис. 3.53, а, требуется шесть микросхем МИС: четыре восьмивходовые микросхемы, вентиль ИЛИ и микросхема с тремя инверторами.

Если компьютер состоит только из центрального процессора, двух микросхем памяти и PIO, мы можем сильно упростить процесс декодирования адреса. Дело в том, что у всех адресов стираемого ПЗУ и только у адресов стираемого ПЗУ старший разряд A15 всегда равен 0. Следовательно, мы можем просто связать сигнал C5 с линией A15, как показано на рис. 3.53, *б*.

Теперь решение поместить ОЗУ в адрес 8000H кажется не таким уж произвольным. Отметим, что в ОЗУ попадают адреса типа Юxxxxxxxxxxxxxx, поэтому для декодирования достаточно двух битов. Точно так же, любой адрес, начинающийся с 11, является адресом PIO. Полная логика декодирования состоит из двух вентилей НЕ-И и инвертора. Поскольку инвертор можно сделать из вентиля НЕ-И, связав два входа вместе, одного счетверенного вентиля НЕ-И более чем достаточно.

Логика декодирования адреса, показанная на рис. 3.53, *б,* называется **частичным декодированием адреса,** поскольку в данном случае полные адреса не используются. При таком декодировании считывание из адресов 0001000000000000, 0001100000000000 и 0010000000000000 будет давать один и тот же результат. В действительности любой адрес в нижней половине адресного пространства будет выбирать стираемое ПЗУ. Поскольку дополнительные адреса не используются, в этом нет ничего ужасного, но при разработке компьютера, который будет расширяться в будущем (в случае с игрушками это маловероятно), следует избегать частичного декодирования, поскольку оно сильно ограничивает адресное пространство.

Можно применять и другую технологию декодирования адреса — технологию с использованием декодера (см. рис. 3.12). Связав три входа с тремя адресными линиями самых старших разрядов, мы получаем восемь выходов, которые соответствуют адресам в первом отрезке 8 K, втором отрезке 8 K и т. д. В компьютере, содержащем 8 микросхем ОЗУ по 8 Kx8 байт, полное декодирование осуществляет одна такая микросхема. Если компьютер содержит 8 микросхем памяти по 2 Kx8 байт, для декодирования также достаточно одного декодера, при условии что каждая микросхема памяти занимает отдельный участок адресного пространства в 8 K. (Вспомните наше замечание о том, что расположение микросхем памяти и устройств ввода-вывода внутри адресного пространства имеет значение.)

Краткое содержание главы

Компьютеры конструируются из интегральных схем, содержащих крошечные переключатели, которые называются вентилями. Обычно используются вентили И, ИЛИ, НЕ-И, НЕ-ИЛИ и НЕ. Комбинируя отдельные вентили, можно построить простые схемы.

Более сложными схемами являются мультиплексоры, демультиплексоры, кодеры, декодеры, схемы сдвига и АЛУ. С помощью программируемой логической матрицы можно запрограммировать произвольные булевы функции. Если требу-

ется много булевых функций, программируемые логические матрицы обычно более эффективны, чем другие средства. Законы булевой алгебры используются для преобразования схем из одной формы в другую. Во многих случаях таким способом можно произвести более экономичные схемы.

Арифметические действия в компьютерах осуществляются сумматорами. Одноразрядный полный сумматор можно сконструировать из двух полусумматоров. Чтобы построить сумматор для многоразрядных слов, полные сумматоры нужно соединить таким образом, чтобы выход переноса каждого сумматора передавался его левому соседу.

Статическая память состоит из защелок и триггеров, каждый из которых может хранить один бит информации. Их можно объединять и получать восьмиразрядные триггеры и защелки либо законченную память для хранения слов. Существуют различные типы памяти: ОЗУ, ПЗУ, программируемое ПЗУ, стираемое ПЗУ, электронно-перепрограммируемое ПЗУ и флэш-память. Статическое ОЗУ не нужно обновлять: оно хранит информацию, пока включен компьютер. Динамическое ОЗУ, напротив, нужно периодически обновлять, чтобы предотвратить утечку информации.

Компоненты компьютерной системы соединяются шинами. Большинство выводов обычного центрального процессора (хотя не все) запускают одну линию шины. Линии шины можно подразделить на адресные, информационные и линии управления. Синхронные шины запускаются задающим генератором. В асинхронных шинах для согласования работы задающего и подчиненного устройств используется система полного квитирования.

Pentium II представляет собой пример современного процессора. Системы с таким процессором включают в себя шину памяти, шину PCI, шину ISA и шину USB. Шина PCI может передавать за один раз 64 бита информации с частотой 66 МГц. Этого вполне достаточно практически для всех периферических устройств, но не для памяти.

Переключатели, индикаторы, принтеры и многие другие устройства ввода-вывода можно связать с компьютером, используя микросхемы ввода-вывода (например, 8255A). Эти микросхемы по желанию можно сделать частью пространства ввода-вывода или частью пространства памяти. Выбор микросхемы может происходить с помощью полного или частичного декодирования адреса в зависимости от того, какие задачи выполняет компьютер.

Вопросы и задания

1. Логик заезжает в закусочную и говорит: «Дайте мне, пожалуйста, гамбургер или хот-дог и картофель фри». К несчастью, повар не закончил и шести классов и не знает (да и не хочет знать), какая из двух логических операций, «и» или «или», имеет приоритет над другой. Он считает, что в данном случае допустима любая интерпретация. А какие из нижеперечисленных интерпретаций этого высказывания действительно допустимы? (Отметим, что «или» означает «исключающее ИЛИ»).

1. Только гамбургер.
2. Только хот-дог.
3. Только картофель фри.
4. Хот-дог и картофель фри.
5. Гамбургер и картофель фри.
6. Хот-дог и гамбургер.
7. Все три.
8. Ничего — логик голодает, потому что он слишком умный.

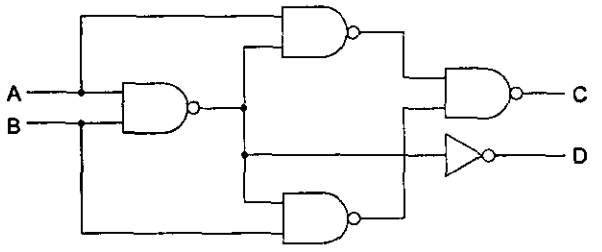
2. Миссионер, заблудившийся в Южной Калифорнии, остановился на развилке дороги. Он знает, что в этом районе обитают две мотоциклетные банды. Одна из них всегда говорит правду, а другая всегда лжет. Он хочет узнать, какая дорога ведет в Диснейленд. Какой вопрос он должен задать?
3. Существует 4 булевы функции от одной переменной и 16 функций от двух переменных. Сколько существует функций от трех переменных? А от и переменных?
4. Используя таблицу истинности, покажите, что P=(P И £))ИЛИ(РИ НЕ Q).
5. Покажите, как можно воплотить функцию И, используя два вентиля НЕ-И.
6. Используя закон Де Моргана, найдите дополнение от АБ.
7. Используя мультиплексор с тремя переменными, изображенный на рис. 3.11, реализуйте функцию, значение которой равно 1 тогда и только тогда, когда нечетное число входных сигналов равно 1.
8. Мультиплексор с тремя переменными, изображенный на рис. 3.11, в действительности способен вычислять произвольную функцию от четырех логических переменных. Опишите, как это происходит, и нарисуйте логическую схему для функции, которая принимает значение 0, если слово, соответствующее строке таблицы истинности, содержит четное число букв, и 1, если оно содержит нечетное число букв (например, 0000 = нуль = четыре буквы —> 0; 0010 - два - три буквы —> 1; 0111 = семь = четыре буквы -» 0; 1101 = тринадцать = десять букв —> 0). *Подсказка:* назовем четвертую входную переменную D. Тогда восемь входных линий можно связать с V_{cc}, «землей», D или $\overline{D}$.

9 Нарисуйте логическую схему 2-разрядного демультиплексора, у которого сигнал на единственной входной линии направляется к одной из четырех выходных линий в зависимости от значений двух линий управления.

10 Нарисуйте логическую схему 2-разрядного кодера, который содержит 4 входные и 2 выходные линии. Ровно одна из входных линий всегда равна 1. Двухразрядное двоичное число на 2 выходных линиях показывает, какая именно входная линия равна 1.

11. Перерисуйте программируемую логическую матрицу, изображенную на рис. 3.14. Покажите, как на ней можно реализовать логическую функцию большинства (см. рис. 3.3). Обязательно покажите, какие из потенциально возможных связей используются в первой и второй матрице.

12. Что делает данная схема?



13. Обычная схема СИС представляет собой 4-разрядный сумматор. Четыре такие микросхемы можно связать вместе и получить 16-разрядный сумматор. Как вы думаете, сколько выводов должен содержать каждый 4-разрядный сумматор? Почему?

14. п-разрядный сумматор можно получить путем каскадирования п полных сумматоров, причем перенос в стадию i, который мы будем обозначать Ci, получается из результата вычислений на стадии i-i. Перенос в стадию 0 (Co) равен 0. Если вычисление суммы и переноса составляет на каждой стадии T не, то перенос в стадию i будет вычислен только через iT не после начала суммирования. При большом п до вычисления переноса в последнюю стадию может пройти очень много времени. Разработайте сумматор, который работает быстрее. *Подсказка:* каждый перенос C, можно выразить через операнды (биты) A,-i и B,.ь так же как и перенос C,_i. Используя это соответствие, можно выразить C, как функцию от входов на стадии от 0 до i-1, так что все переносы можно будет генерировать одновременно.

15. Если все вентили на рис. 3.18имеютзадержку на прохождение сигнала 10 не, а все прочие задержки не учитываются, сколько потребуется времени (минимум) для получения достоверного выходного сигнала?

16. АЛУ, изображенное на рис. 3.19, способно выполнять сложение 8-разрядных двоичных чисел. Может ли оно выполнять вычитание двоичных чисел? Если да, то объясните, как. Если нет, преобразуйте схему таким образом, чтобы она могла вычитать.

17. Иногда бывает нужно, чтобы 8-разрядное АЛУ (см., например, рис. 3.19) выдавало на выходе константу - 1 . Предложите два различных способа того, как это можно сделать. Для каждого способа определите значения шести сигналов управления.

18. 16-разрядное АЛУ конструируется из 16 одноразрядных АЛУ, каждое из которых тратит на суммирование 10 не. Если задержка на прохождение сигнала от одного АЛУ к другому составляет 1 не, сколько времени потребуется для получения конечного результата?

19. Каково состояние покоя входов S и R SR-защелки, построенной из двух вентилей НЕ-И?

20. Схема на рис. 3.25 представляет собой триггер, который запускается на нарастающем фронте синхронизирующего сигнала. Преобразуйте эту схему

так, чтобы получить триггер, который запускается на заднем фронте синхронизирующего сигнала.

21. Вы консультируете неопытных производителей микросхем МИС. Один из ваших клиентов предложил выпустить микросхему, содержащую четыре D-триггера, каждый из которых имеет выходы Q и $\overline{Q}$ по требованию потенциального важного покупателя. В данном проекте все 4 синхронизирующих сигнала объединены (также по требованию). Входов предварительной установки и очистки у схемы нет. Ваша задача — дать профессиональную оценку этой разработки.
22. В памяти 4x3, изображенной на рис. 3.28, используется 22 вентиля И и три вентиля ИЛИ Сколько потребуется вентилей каждого из двух типов, если схему расширить до размеров 256x8?
23. С увеличением объема памяти, помещаемой на одну микросхему, число выводов, необходимых для обращения к этой памяти, также увеличивается. Иметь большое количество адресных выводов на микросхеме довольно неудобно. Придумайте способ обращения к 2" словам памяти при наличии меньшего количества выводов, чем п.
24. В компьютере с 32-битной шиной данных используются динамические ОЗУ размером 1 Mxl. Каков минимальный объем памяти (в байтах), который может содержаться в этом компьютере?
25. Посмотрите на временную диаграмму на рис. 3.34. Предположим, что вы замедлили задающий генератор до периода в 40 не вместо 25 не, но временные ограничения сохранились без изменений. Сколько времени в худшем случае будет у памяти на то, чтобы передать данные па шину во время T_3 после того, как был установлен сигнал $\overline{MREQ}$?
26. Снова посмотрите на рис. 3 34. Предположим, что тактовый генератор работает с частотой 40 МГц, а T_{AD} возросло до 16 не. Можно ли при этом продолжать использовать микросхемы памяти на 40 не?
27. В табл. 3.4 показано, что T_{ML} ДОЛЖНО быть по крайней мере 6 не. Можете ли вы представить микросхему, у которой этот показатель отрицательный? Другими словами, может ли процессор устанавливать сигнал $\overline{MREQ}$ до того, как адрес стал стабильным? Объясните почему.
28. Предположим, что передача блока, показанная на рис. 3.38, была произведена на шине с рисунка 3.34. Насколько больше получается пропускная способность при передаче блока по сравнению с отдельными передачами (для длинных блоков)? А теперь предположите, что ширина шины составляет 32 бита вместо 8 битов. Каков будет ваш ответ теперь?
29. Посмотрите на рис. 3.35. Обозначьте время перехода адресных линий как Тд[и T_{A2} время перехода линии $\overline{MREQ}$ как T_{MREQ1} И T_{MREQ2} И Т. Д. Напишите все неравенства, подразумеваемые при полном квитировании
30. Большинство 32-битных шин допускают считывание и запись по 16 битов. Существуют ли какие-нибудь варианты, где именно поместить данные? Аргументируйте.

31. Многие процессоры содержат особый тип цикла шины для подтверждения прерывания. Зачем это нужно?
32. Компьютеру PC/AT, работающему с частотой 10 МГц, требуется 4 цикла, чтобы считать слово. Какую часть пропускной способности шины потребляет процессор?
33. 32-битный процессор с адресными линиями A2-A31 требует, чтобы все ссылки к ячейкам памяти были выровнены. Это значит, что центральный процессор должен обращаться только к словам, состоящим из 4, 8,12 и т. д. байтов (число байтов кратно 4), и к полусловам, состоящим из четного числа байтов. Байты могут располагаться где угодно. Сколько существует допустимых комбинаций считываний из памяти и сколько требуется выводов, чтобы их выразить? Дайте два ответа.
34. Почему процессор Pentium II не может работать с 32-битной шиной PCI без потери функциональных возможностей? Ведь другие компьютеры с 64-битной шиной могут осуществлять передачи по 32,16 и даже 8 битов.
35. Предположим, что центральный процессор содержит кэш-память первого и второго уровня со временем доступа 5 не и 10 не соответственно. Время доступа к основной памяти составляет 50 не. Если 20% от всех обращений к памяти приходится на долю кэш-памяти первого уровня, а 60% — на долю кэш-памяти второго уровня, то каково среднее время доступа?
36. Возможно ли, чтобы небольшая встроенная система picojava II содержала микросхему 8255A?
37. Вычислите пропускную способность шины, необходимую для отображения на мониторе VGA (640x480) цветного фильма (30 кадров/с), Предполагается, что данные должны проходить по шине дважды: один раз от компакт-диска к памяти, а второй раз от памяти к монитору.
38. Как вы думаете, какой сигнал процессора Pentium II запускает линию FRAME#HaimmePCI?
39. Какие из сигналов, показанных на рис. 3.49, не являются обязательными для протокола шины?
40. Компьютеру на выполнение каждой команды требуется два цикла шины: один для вызова команды, а второй для вызова данных. Каждый цикл шины занимает 250 не, а выполнение каждой команды занимает 500 не (время обработки не принимается в расчет). В компьютере имеется диск. Каждая дорожка этого диска состоит из 16 секторов по 512 байтов. Время обращения диска составляет 8,192 миллисекунд. На сколько процентов снижается скорость работы компьютера во время передачи ПДП (прямой доступ к памяти), если каждая передача ПДП занимает один цикл шины? Рассмотрите два случая: для 8-битных передач и для 16-битных передач по шине.
41. Максимальная полезная нагрузка пакета данных, передаваемого по шине USB, составляет 1023 байта. Если предположить, что устройство может посылать только один пакет данных за кадр, какова максимальная пропускная способность для одного изохронного устройства?

42. Посмотрите на рис. 3.53, *б*. Что получится, если к вентилю НЕ-И, который выбирает микросхему PIO, добавить третью входную линию, связанную cA13?

43. Напишите программу, которая имитирует работу матрицы размером mxn, состоящей из двухвходовых вентилей НЕ-И. Эта схема (она помещается на микросхему) содержит] входных выводов и к выходных выводов. Значения j, к, m и п обрабатываются в процессе компиляции. Программа считывает таблицу монтажных соединений, каждое из соединений определяет вход и выход. Входом может быть либо один из j входных выводов, либо выход какого-нибудь вентиля НЕ-И. Выходом может быть либо один из к выходных выводов, либо вход в какой-нибудь вентиль НЕ-И. Неиспользованные входы принимают значение логической 1. После считывания таблицы соединений программа должна напечатать выход для каждого из 2^j возможных входов. Подобные вентильные матрицы широко используются при нанесении на микросхему схем, разрабатываемых по техническим заданиям заказчика, поскольку большая часть этой работы (имеется в виду нанесение вентильной матрицы на микросхему) не зависит от того, какая это будет схема. Для каждой разработки имеет значение только выбор монтажных соединений.

AA. Напишите программу, которая на входе получает два произвольных логических выражения и проверяет, представляют ли они одну и ту же функцию. Входной язык должен включать отдельные буквы (логические переменные), операнды И, ИЛИ и НЕ и скобки Каждое выражение должно помещаться на одну входную линию. Программа вычисляет таблицы истинности для обеих функций и сравнивает их.

45. Напишите программу, которая получает на входе ряд логических выражений и строит матрицы 24x50 и 50x6, которые нужны для реализации этих выражений в программируемой логической матрице, изображенной на рис. 3.14. Входной язык такой же, как в предыдущем задании. Распечатайте эти матрицы на строчном печатающем устройстве.

Глава 4
Микроархитектурный уровень

Над цифровым логическим уровнем находится микроархитектурный уровень. Его задача — интерпретация команд уровня 2 (уровня архитектуры команд), как показано на рис. 1.2. Строение микроархитектурного уровня зависит от того, каков уровень архитектуры команд, а также от стоимости и предназначения компьютера. В настоящее время уровень архитектуры команд часто содержит простые команды, которые выполняются за один цикл (таковы, в частности, системы RISC). В других системах (например, в системах Pentium II) на этом уровне имеются более сложные команды; выполнение одной такой команды занимает несколько циклов. Чтобы выполнить команду, нужно найти операнды в памяти, считать их и записать полученные результаты обратно в память. Управление уровнем команд со сложными командами отличается от управления уровнем команд с простыми командами, так как в первом случае выполнение одной команды требует определенной последовательности операций.

Пример микроархитектуры

В идеале мы должны были сначала описать общие принципы разработки микроархитектурного уровня. К сожалению, таких общих принципов не существует. Каждая разработка индивидуальна. По этой причине мы просто подробно рассмотрим конкретный пример. В качестве примера мы выбрали подмножество виртуальной машины Java, как мы и обещали в главе 1. Это подмножество содержит только команды с целыми числами, поэтому мы назвали ее **IJVM (Integer JVM; integer** — целое число). Полную структуру JVM мы рассмотрим в главе 5.

Начнем с описания микроархитектуры, на основе которой мы воплотим IJVM. Система IJVM содержит несколько довольно сложных команд. Подобные архитектуры часто реализуются с помощью микропрограммирования, как уже было сказано в главе 1. Хотя структура IJVM несложная, она послужит отправной точкой в описании основных принципов управления командами и последовательности их выполнения.

Наша микроархитектура содержит микропрограмму (в ПЗУ), которая должна вызывать, декодировать и выполнять команды IJVM. Мы не можем использовать для этой микропрограммы интерпретатор JVM, разработанный компанией Sun,

поскольку нам нужна крошечная микропрограмма, которая запускает отдельные вентили аппаратного обеспечения. Интерпретатор JVM компании Sun был написан на языке C, чтобы обеспечить мобильность программного обеспечения. Этот интерпретатор не может управлять аппаратным обеспечением на таком детализированном уровне, который нам нужен. Поскольку реальное аппаратное обеспечение состоит только из компонентов, описанных в главе 3, то теоретически после изучения этой главы читатель сможет пойти в магазин, купить огромное количество транзисторов и сконструировать машину IJVM. Тому, кто успешно выполнит эту задачу, будет предоставлен дополнительный кредит (а также полное психиатрическое обследование).

Разработку данной микроархитектуры удобно считать проблемой программирования, при этом каждая команда уровня архитектуры команд — функция, вызываемая основной программой. В данном случае основная программа довольно проста. Она представляет собой бесконечный цикл. Сначала программа определяет, какую функцию нужно выполнить, затем вызывает эту функцию, а затем начинает все снова.

Микропрограмма содержит набор переменных, к которым имеют доступ все функции. Этот набор переменных называется **состоянием** компьютера. Каждая функция изменяет по крайней мере несколько переменных, формируя при этом состояние. Например, счетчик команд — это часть состояния. Он указывает местонахождение функции (то есть команды уровня архитектуры команд), которую нужно выполнить следующей. Во время выполнения каждой команды счетчик команд указывает на следующую команду.

Команды IJVM очень короткие. Каждая команда состоит из нескольких полей, обычно одного или двух, каждое из которых выполняет определенную функцию. Первое поле является **кодом операции.** Этот код определяет тип команды и сообщает, что это, например, команда сложения или команда ветвления, или еще какая-нибудь команда. Многие команды содержат дополнительное поле, которое определяет тип операнда. Например, команды, которые имеют доступ к локальным переменным, должны иметь специальное поле, чтобы определить, какая это переменная.

Такая модель выполнения команды, называемая иногда **циклом выборка-исполнение,** полезна для теории, а также может служить основой воплощения уровня архитектуры команд со сложными командами (например, IJVM). Ниже мы опишем, как работает эта модель, что собой представляет микроархитектура и как ею управляют микрокоманды, каждая из которых занимает тракт данных на один цикл. Полный список команд формирует микропрограмму, которая будет рассмотрена очень подробно.

Тракт данных

Тракт данных — это часть центрального процессора, состоящая из АЛУ (арифметико-логического устройства) и его входов и выходов. Тракт данных нашей микроархитектуры показан на рис. 4.1. Хотя этот тракт данных и был оптимизирован для интерпретации программ IJVM, он схож с трактами данных большинства компьютеров. Он содержит ряд 32-разрядных регистров, которым мы приписали символические названия (например, PC, SP, MDR). Хотя некоторые из этих названий

нам знакомы, важно понимать, что эти регистры доступны только на микроархитектурном уровне (для микропрограммы). Им даны такие названия, поскольку они обычно содержат значения, соответствующие переменным с аналогичными названиями на уровне архитектуры команд Содержание большинства регистров передается на шину B. Выходной сигнал АЛУ запускает схему сдвига, а затем шину C Значение из шины C может записываться в один или несколько регистров одновременно Шину A мы введем позже, а пока представим, что ее нет

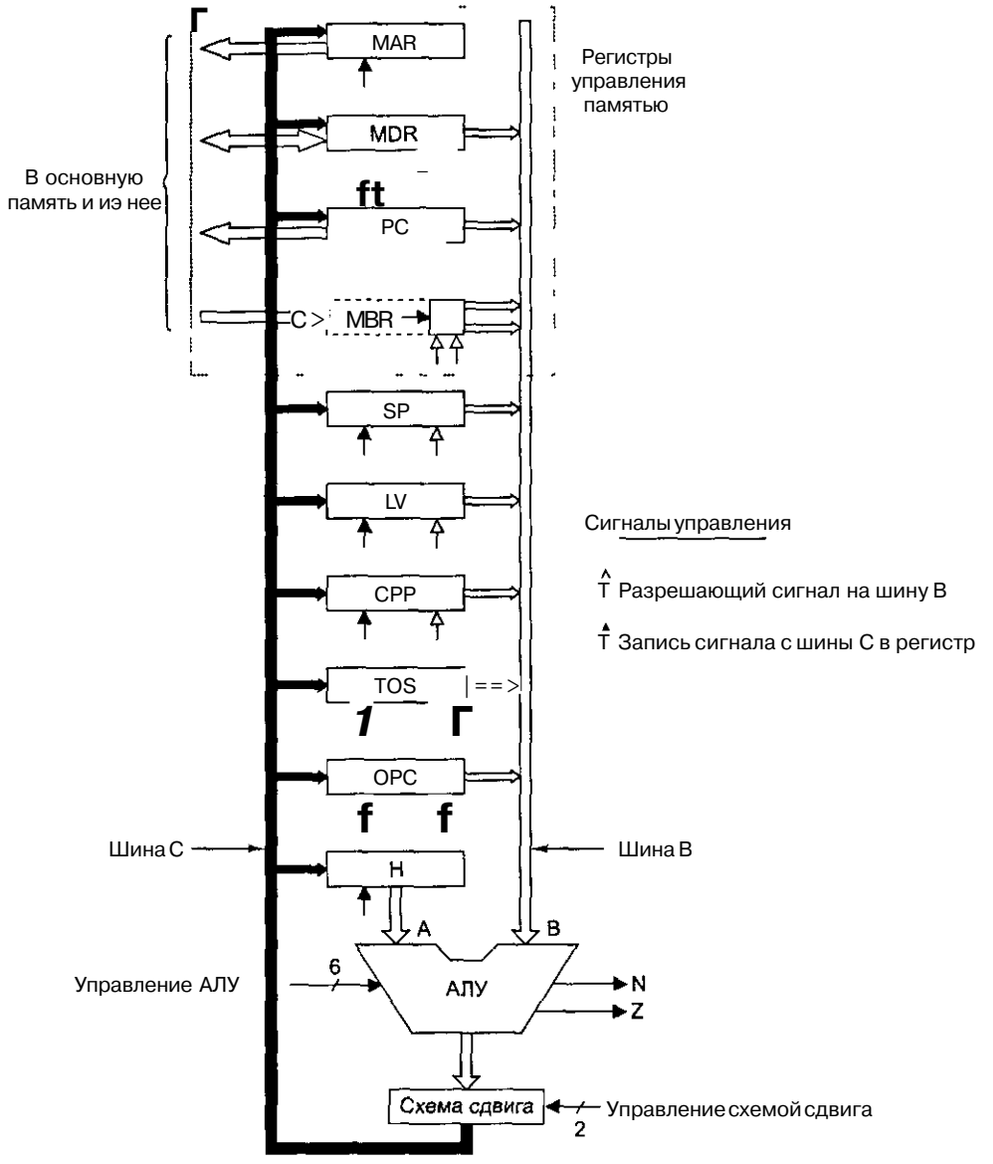


Рис. 4 . 1 . Тракт данных микроархитектуры, рассматриваемой в этой главе

Данное АЛУ идентично тому, которое изображено на рис. 3.18 и 3.19. Его функционирование зависит от линий управления. На рис. 4.1 перечеркнутая стрелочка с цифрой 6 сверху указывает на наличие шести линий управления АЛУ. Из них F_u и р!служат для определения операции, ENA и ENB — для разрешения входных сигналов А и В соответственно, 1NVA — для инверсии левого входа и INC — для прибавления единицы к результату. Однако не все 64 комбинации значений на линиях управления могут быть полезны.

Некоторые комбинации показаны в табл. 4.1. Не все из этих функций нужны для IJVM, но многие из них могут пригодиться для полной JVM. В большинстве случаев существует несколько возможностей для достижения одного и того же результата. В данной таблице знак «+» означает арифметический плюс, а знак «-» — арифметический минус, поэтому -А означает дополнение А.

Таблица 4.1. Некоторые комбинации сигналов АЛУ и соответствующие им функции

F_0	F_1	ENA	ENB	INVA	INC	Функция
0	1	1	0	0	0	A
0	1	0	1	0	0	**B**
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	**0**	$\bar{B}$
1	1	1	1	**0**	0	A+B
1	1	1	1	**0**	1	A+B+1
1	1	1	0	0	1	A+1
1	1	**0**	1	0	1	B+1
1	1	1	**1**	1	1	B-A
1	1	0	1	1	0	B-1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A И B
0	1	1	1	0	0	A ИЛИ B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	**0**	-1

АЛУ, изображенное на рис. 4.1, содержит два входа для данных: левый вход (А) и правый вход (В). С левым входом связан регистр временного хранения Н. С правым входом связана шина В, в которую могут поступать значения из одного из девяти источников, что показано с помощью девяти серых стрелок, примыкающих к шине. Существует и другая разработка АЛУ с двумя полноразрядными шинами, и мы рассмотрим ее чуть позже в этой главе.

В регистр Н может поступать функция АЛУ, которая проходит через правый вход (из шины В) к выходу АЛУ Одна из таких функций — сложение входных сигналов АЛУ, только при этом сигнал ENA отрицается, и левый вход получает значение 0. Если к значению шины В прибавить 0, это значение не изменится. Затем результат проходит через схему сдвига (также без изменений) и сохраняется в регистре Н.

Существует еще две линии управления, которые используются независимо от остальных. Они служат для управления выходом АЛУ. Линия SLL8 (Shift Left Logical — логический сдвиг влево) сдвигает число влево на 1 байт, заполняя 8 самых младших двоичных разрядов нулями; линия SRA1 (Shift Right Arithmetic — арифметический сдвиг вправо) число вправо на 1 бит, оставляя самый старший двоичный разряд без изменений.

Можно считать и записать один и тот же регистр за один цикл. Для этого, например, нужно поместить значение SP на шину B, закрыть левый вход АЛУ, установить сигнал INC и сохранить полученный результат в регистре SP, увеличив таким образом его значение на 1 (см. восьмую строку табл. 4.1). Если один и тот же регистр может считываться и записываться за один цикл, то как при этом предотвратить появление ненужных данных? Дело в том, что процессы чтения и записи проходят в разных частях цикла. Когда в качестве правого входа АЛУ выбирается один из регистров, его значение помещается на шину B в начале цикла и хранится там на протяжении всего цикла. Затем АЛУ выполняет свою работу и производит результат, который через схему сдвига поступает на шину C. Незадолго до конца цикла, когда значения выходных сигналов АЛУ и схемы сдвига стабилизировались, содержание шины C передается в один или несколько регистров. Одним из этих регистров вполне может быть тот, от которого поступил сигнал на шину B. Точная синхронизация тракта данных делает возможным считывание и запись одного и того же регистра за один цикл. Об этом речь пойдет ниже.

Синхронизация тракта данных

Как происходит синхронизация этих действий, показано на рис. 4.2. Здесь в начале каждого цикла генерируется короткий импульс. Он может выдаваться задающим генератором, как показано на рис. 3.20, *в*. На заднем фронте импульса устанавливаются биты, которые будут запускать все вентили. Этот процесс занимает определенный отрезок времени Aw. Затем выбирается регистр, и его значение передается на шину B. На это требуется время Дх. Затем АЛУ и схема сдвига начинают оперировать поступившими к ним данными. После промежутка Ду выходные сигналы АЛУ и схемы сдвига стабилизируются. В течение следующего отрезка Дг результаты проходят по шине C к регистрам, куда они загружаются на нарастающем фронте следующего импульса. Загрузка должна запускаться фронтом сигнала и осуществляться мгновенно, так что даже в случае изменений каких-либо входных регистров изменения в шине C будут происходить только после полной загрузки регистров. На нарастающем фронте импульса регистр, запускающий шину B, приостанавливает свою работу и ждет следующего цикла. На рис. 4.2 упомянуты регистры MPC и MIR, а также память. Их предназначение мы обсудим чуть позже.

Важно осознавать, что хотя в тракте данных нет никаких запоминающих элементов, для прохождения сигнала по нему требуется определенное время. Изменение значения на шине B вызывает изменения на шине C не сразу, а только через некоторое время (это объясняется задержками на каждом шаге). Следовательно, даже если один из входных регистров изменяется, новое значение будет сохранено

в регистре задолго до того, как старое (и уже неправильное) значение этого регистра, помещенное на шину B, сможет достичь АЛУ.

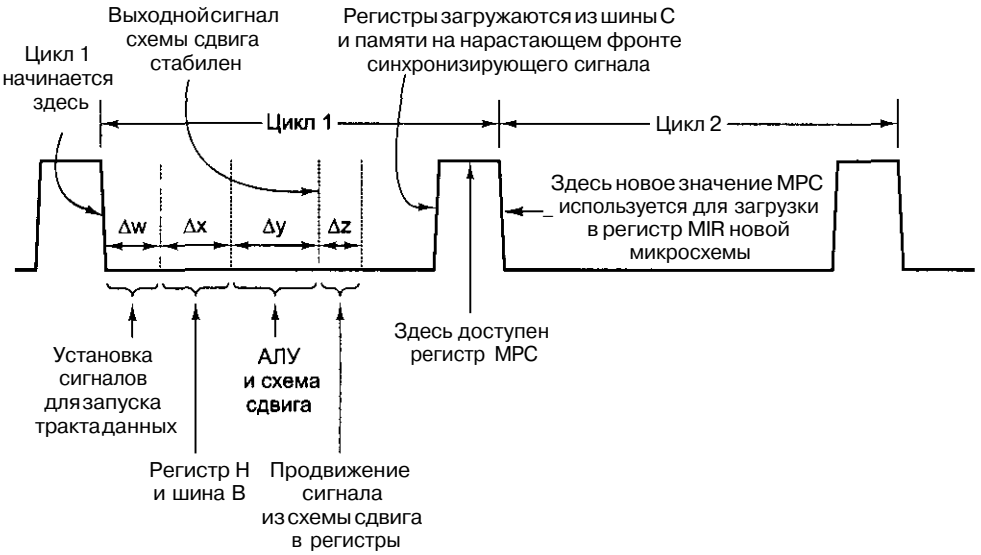


Рис. 4.2. Временная диаграмма цикла тракта данных

Для такой разработки требуется жесткая синхронизация и довольно длинный цикл; должно быть известно минимальное время прохождения сигнала через АЛУ; регистры должны загружаться из шины C очень быстро. Если подойти к этому вопросу с особым вниманием и осторожностью, можно сделать так, чтобы тракт данных функционировал правильно.

Цикл тракта данных можно разбить на подциклы. Начало подцикла 1 запускается задним фронтом синхронизирующего сигнала. Ниже показано, что происходит во время каждого из подциклов. В скобках приводится длина подцикла.

1. Устанавливаются сигналы управления (Aw).
2. Значения регистров загружаются на шину B (Ах).
3. Происходит работа АЛУ и схемы сдвига (Ду).
4. Результаты проходят по шине C обратно к регистрам (Дг).

На нарастающем фронте следующего цикла результаты сохраняются в регистрах.

Никаких внешних сигналов, указывающих на начало и конец подцикла и сообщающих АЛУ, когда нужно начинать работу и когда нужно передавать результаты на шину C, нет. В действительности АЛУ и схема сдвига работают постоянно. Однако их входные сигналы недействительны в течение периода Aw+Дх. Точно так же их выходные сигналы недействительны в течение периода Aw+ Дх+Ду. Единственными внешними сигналами, управляющими трактом данных, являются задний фронт синхронизирующего сигнала, с которого начинается цикл тракта данных, и нарастающий фронт синхронизирующего сигнала, который загружает регистры из шины C. Границы подциклов определяются толоко временем прохождения сигнала, поэтому разработчики тракта данных должны все очень четко рассчитать.

Работа памяти

Наша машина может взаимодействовать с памятью двумя способами: через порт с пословной адресацией (32-битный) и через порт с байтовой адресацией (8-битный). Порт с пословной адресацией управляется двумя регистрами; **MAR (Memory Address Register — регистр адреса ячейки памяти)** и **MDR (Memory Data Register — информационный регистр памяти),** которые показаны на рис. 4.1. Порт с байтовой адресацией управляется регистром PC, который записывает 1 байт в 8 младших разрядов регистра **MBR (Memory Buffer Register — буферный регистр памяти).** Этот порт может считывать данные из памяти, но не может их записывать в память.

Каждый из этих регистров, а также все остальные регистры, изображенные на рис. 4.1, запускаются одним из **сигналов управления.** Белая стрелка под регистром указывает на сигнал управления, который разрешает передавать выходной сигнал регистра на шину B. Регистр MAR не связан с шиной B, поэтому у него нет сигнала разрешения. У регистра H этого сигнала тоже нет, так как он является единственным возможным левым входом АЛУ **и** поэтому всегда разрешен.

Черная стрелка под регистром указывает на сигнал управления, который записывает (то есть загружает) регистр из шины C. Поскольку регистр MBR не может загружаться из шины C, у него нет сигнала записи (но зато есть два сигнала разрешения, о которых речь пойдет ниже). Чтобы инициировать процесс считывания из памяти или записи в память, нужно загрузить соответствующие регистры памяти, а затем передать памяти сигнал чтения или записи (он не показан на рис. 4.1).

Регистр **MAR** содержит адреса слов, таким образом, значения 0,1,2 и т. д. указывают на последовательные слова. Регистр PC содержит адреса байтов, таким образом, значения 0,1,2 и т. д. указывают на последовательные байты. Если значение 2 поместить в регистр PC и начать процесс чтения, то из памяти считается байт 2, который затем будет записан в 8 младших разрядов регистра MBR. Если значение 2 поместить в регистр MAR и начать процесс чтения, то из памяти считаются байты 8-11 (то есть слово 2), которые затем будут записаны в регистр MDR.

Для чего потребовалось два регистра с разной адресацией? Дело в том, что регистры MAR и PC будут использоваться для обращения к двум разным частям памяти, а зачем это нужно, станет ясно чуть позже. А пока достаточно сказать, что регистры MAR и MDR используются для чтения и записи слов данных на уровне архитектуры команд, а регистры PC и MBR — для считывания программы уровня архитектуры команд, которая состоит из потока байтов. Во всех остальных регистрах, содержащих адреса, применяется принцип пословной адресации, как и в MAR.

В действительности существует только одна память: с байтовой адресацией. Как же регистр MAR обращается к словам, если намять состоит из байтов? Когда значение регистра MAR помещается на адресную шину, 32 бита этого значения не попадают точно на 32 адресные линии (с 0 по 31). Вместо этого бит 0 соединяется с адресной линией 2, бит 1 — с адресной линией 3 и т. д. Два старших бита не учитываются, поскольку они нужны только для адресов свыше 2^{32}, а такие адреса недопустимы в нашей машине на 4 Гбайт. Когда значение MAR равно 1, на шину помещается адрес 4; когда значение MAR равно 2, на шину помещается адрес 8 **и** т. д. Распределение битов регистра MAR по адресным линиям показано на рис. 4.3.

Рис. 4.3. Распределение битов регистра MAR в адресной шине

Как уже было сказано выше, данные, считанные из памяти через 8-битный порт, сохраняются в 8-битном регистре MBR. Этот регистр может быть скопирован на шину B двумя способами: со знаком и без знака. Когда требуется значение без знака, 32-битное слово, помещаемое на шину B, содержит значение MBR в младших 8 битах и нули в остальных 24 битах. Значения без знака нужны для индексирования таблиц или для получения целого 16-битного числа из двух последовательных байтов (без знака) в потоке команд.

Другой способ превращения 8-битного регистра MBR в 32-битное слово — рассматривать его как значение со знаком между -128 и +127 и использовать это значение для порождения 32-битного слова с тем же самым численным значением. Это преобразование делается путем дублирования знакового бита (самого левого бита) регистра MBR в верхние 24 битовые позиции шины B. Такой процесс называется расширением по знаку или знаковым расширением Если выбран данный параметр, то либо все старшие 24 бита примут значение 0, либо все они примут значение 1, в зависимости от того, каков самый левый бит регистра MBR: 0 или 1.

В какое именно 32-битное значение (со знаком или без знака) превратится 8-битное значение регистра MBR, определяется тем, какой из двух сигналов управления (две белые стрелки под регистром MBR на рис. 4.1) установлен. Прямоугольник, обозначенный на рисунке пунктиром, показывает способность 8-битного регистра MBR действовать в качестве источника 32-битных слов для шины B.

Микрокоманды

Для управления трактом данных, изображенным на рис. 4.1, нам нужно 29 сигналов Их можно разделить на пять функциональных групп'

- 9 сигналов для записи данных из шины C в регистры.
- 9 сигналов для разрешения передачи регистров на шину B и в АЛУ.
- 8 сигналов для управления АЛУ и схемой сдвига.
- 2 сигнала, которые указывают, что нужно осуществить чтение или запись через регистры MAR/MDR (на рисунке они не показаны)
- 1 сигнал, который указывает, что нужно осуществить вызов из памяти через регистры PC/MBR (на рисунке также не показан).

Значения этих 29 сигналов управления определяют операции для одного цикла тракта данных. Цикл состоит из передачи значений регистров на шину B, прохождения этих сигналов через АЛУ и схему сдвига, передачи полученных результатов на шину C и записи их в нужный регистр (регистры). Кроме того, если установлен сигнал считывания данных, то в конце цикла после загрузки регистра MAR начинается работа памяти. Данные из памяти помещаются в MBR или MDR в конце *следующего* цикла, а использоваться эти данные могут в цикле, который идет *после* него. Другими словами, если считывание из памяти через любой из портов начинается в конце цикла к, то полученные данные еще не могут использоваться в цикле к+1 (только в цикле к+2 и позже).

Этот процесс объясняется на рис. 4.2. Сигналы управления памятью выдаются только после загрузки регистров MAR и PC, которая происходит на нарастающем фронте синхронизирующего сигнала незадолго до конца цикла 1. Мы предположим, что память помещает результаты на шину памяти в течение одного цикла, поэтому регистры MBR и (или) MDR могут загружаться на следующем нарастающем фронте вместе с другими регистрами.

Другими словами, мы загружаем регистр MAR в конце цикла тракта данных и запускаем память сразу после этого. Следовательно, мы не можем ожидать, что результаты считывания будут в регистре MDR в начале следующего цикла, особенно если длительность импульса небольшая. Этого времени будет недостаточно. Поэтому между началом считывания из памяти и использованием этого результата должен помещаться один цикл. Однако во время этого цикла может выполняться не только передача слова из памяти, но и другие операции.

Предположение о том, что работа памяти занимает один цикл, эквивалентно предположению, что количество успешных обращений в кэш-память составляет 100%. Подобное предположение никогда не может быть истинным, но мы не будем здесь рассказывать о циклах памяти переменной длины, поскольку это не входит в задачи данной книги.

Так как регистры MBR и MDR загружаются на нарастающем фронте синхронизирующего сигнала вместе с другими регистрами, они могут считывать во время циклов, в течение которых осуществляется передача нового слова из памяти. Они возвращают старые значения, поскольку прошло еще недостаточно времени для того, чтобы поменять их на новые. Здесь нет никакой двусмысленности: до тех пор пока новые значения не загрузятся в регистры MBR и MDR на нарастающем фронте сигнала, предыдущие значения находятся там и могут использоваться. Отметим, что считывания могут проходить одно за другим, то есть в двух последовательных циклах (поскольку сам процесс считывания занимает только один цикл). Кроме того, обе памяти могут действовать в одно и то же время. Однако попытка чтения и записи одного и того же байта одновременно приводит к неопределенным результатам.

Выходной сигнал шины C можно записывать сразу в несколько регистров, однако нежелательно передавать значения более одного регистра на шину B. Немного расширив схемотехнику, мы можем сократить количество битов, необходимых для выбора одного из возможных источников для запуска шины B. Существует только

9 входных регистров, которые могут запустить шину B (регистры MBR со знаком и без знака учитываются отдельно) Следовательно, мы можем закодировать информацию для шины B в 4 бита и использовать декодер для порождения 16 сигналов управления, 7 из которых не нужны. У разработчиков коммерческих моделей, возможно, было бы большое желание избавиться от одного из регистров, чтобы обойтись 3 битами. Однако мы как ученые предпочитаем иметь один лишний бит, но при этом получить более ясную и простую разработку.

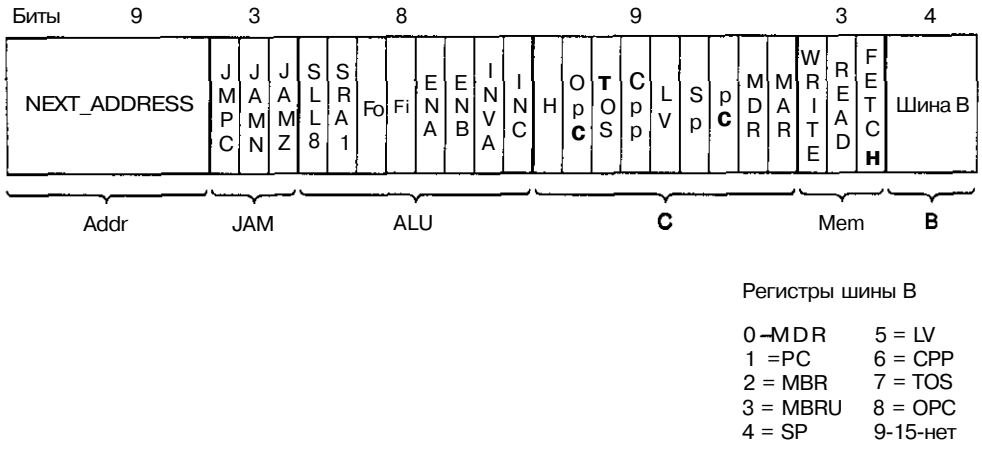


Рис. 4.4. Формат микрокоманды для Мю-1

Теперь мы можем управлять трактом данных с помощью 9+4+8+2+1=24 сигналов, следовательно, нам требуется 24 бита. Однако эти 24 бита управляют трактом данных только в течение одного цикла. Задача управления — определить, что нужно делать в следующем цикле. Чтобы включить это в разработку контроллера, мы создадим формат для описания операций, которые нужно выполнить, используя 24 бита управления и два дополнительных поля поле NEXT_ADDRESS (следующий адрес) и поле JAM. Содержание каждого из этих полей мы обсудим позже. На рис. 4.4 изображен один из возможных форматов. Он разделен на следующие 6 групп, содержащие 36 сигналов'

- Addr — содержит адрес следующей потенциальной микрокоманды.
- JAM — определяет, как выбирается следующая микрокоманда.
- ALU — функции АЛУ и схемы сдвига.
- C — выбирает, какие регистры записываются из шины C.
- Mem — функции памяти.

4 B — выбирает источник для шины B (как он кодируется, было показано выше)

Порядок групп в принципе произволен, хотя мы долго и тщательно его подбирали, чтобы избежать пересечений на рис. 4 5. Подобные пересечения на диаграммах часто соответствуют пересечениям проводов на микросхемах. Они сильно затрудняют разработку и их лучше сводить к минимуму.

Управление микрокомандами: Mic-1

До сих пор мы рассказывали о том, как происходит управление трактом данных, но мы еще не касались того, каким образом решается, какой именно сигнал управления и на каком цикле должен запускаться. Для этого существует **контроллер последовательности,** который отвечает за последовательность операций, необходимых для выполнения одной команды.

Контроллер последовательности в каждом цикле должен выдавать следующую информацию:

1. Состояние каждого сигнала управления в системе.
2. Адрес микрокоманды, которая будет выполняться следующей.

Рисунок 4.5 представляет собой подробную диаграмму полной микроархитектуры нашей машины, которую мы назовем **Mic-1.** На первый взгляд она может показаться внушительной, но тем не менее ее нужно подробно изучить. Если вы разберетесь во всех прямоугольниках и линиях, изображенных на этом рисунке, вам легче будет понять структуру микроархитектурного уровня. Диаграмма состоит из двух частей: тракта данных (слева), который мы уже подробно обсудили, и блока управления (справа), который мы рассмотрим сейчас.

Самой большой и самой важной частью блока управления является **управляющая память.** Удобно рассматривать ее как память, в которой хранится полная микропрограмма, хотя иногда она реализуется в виде набора логических вентилей. Мы будем называть ее управляющей памятью, чтобы не путать с основной памятью, доступ к которой осуществляется через регистры MBR и MDR. Функционально управляющая память представляет собой память, которая содержит микрокоманды вместо обычных команд. В нашем примере она содержит 512 слов, каждое из которых состоит из одной 32-битной микрокоманды с форматом, изображенным на рис. 4.4. В действительности не все эти слова нужны, но по ряду причин нам требуются адреса для 512 отдельных слов.

Управляющая память отличается от основной памяти тем, что команды, хранящиеся в основной памяти, выполняются в порядке адресов (за исключением ветвлений), а микрокоманды — нет. Увеличение счетчика команд в листинге 2.1 означает, что команда, которая будет выполняться после текущей, — это команда, которая идет вслед за текущей в памяти. Микропрограммы должны обладать большей гибкостью (поскольку последовательности микрокоманд обычно короткие), поэтому они не обладают этим свойством. Вместо этого каждая микрокоманда сама указывает на следующую микрокоманду.

Поскольку управляющая память функционально представляет собой ПЗУ, ей нужен собственный адресный регистр и собственный регистр данных. Ей не требуются сигналы чтения и записи, поскольку здесь постоянно происходит процесс считывания. Мы назовем адресный регистр управляющей памяти **MPC (Microprogram Counter — микропрограммный счетчик).** Название не очень подходящее, поскольку микропрограммы не упорядочены явным образом и понятие счетчика тут неуместно, но мы не можем пойти против традиций. Регистр данных мы назовем **MIR (Microinstruction Register — регистр микрокоманд).** Он содержит текущую микрокоманду, биты которой запускают сигналы управления, влияющие на работу тракта данных.

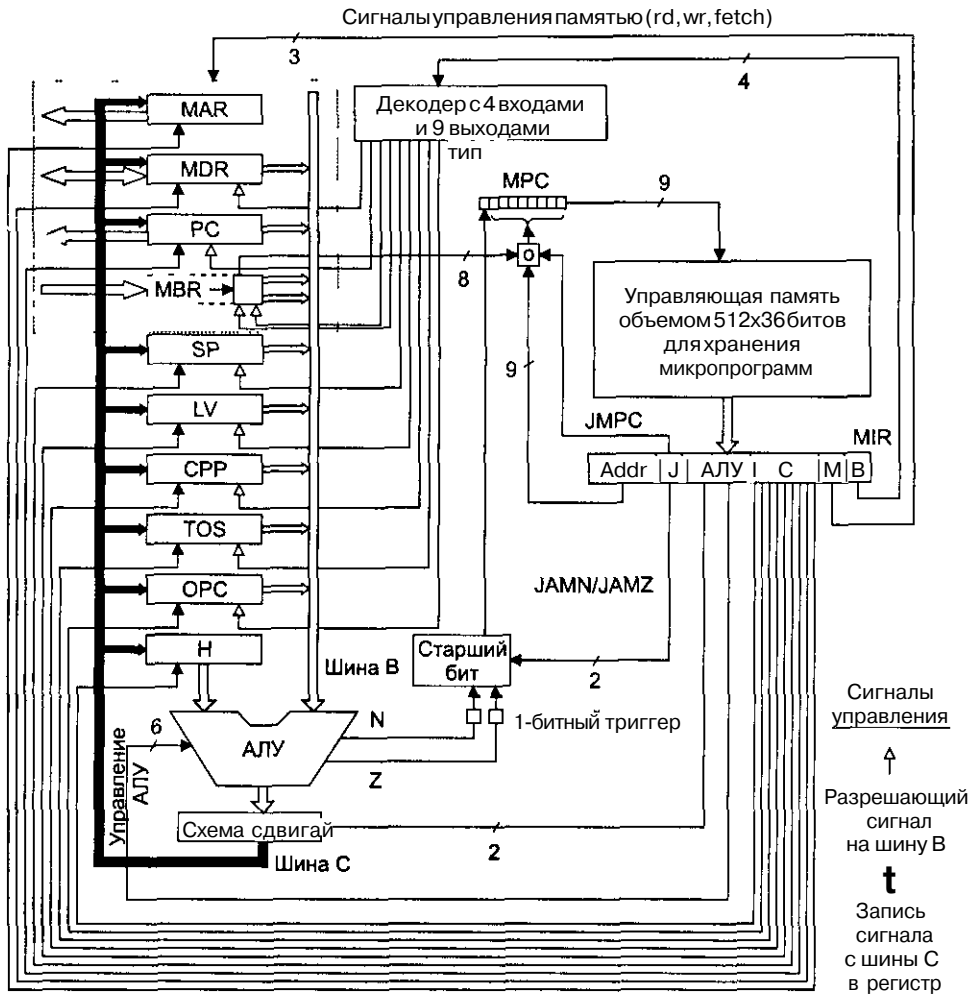


Рис. 4.5. Полная диаграмма микроархитектуры Mic-1

Регистр MIR, изображенный на рис 4 5, содержит те же шесть групп сигналов, которые показаны на рис 4.4. Группы Addr и J (то же, что JAM) контролируют выбор следующей микрокоманды Мы обсудим их чуть позже Группа ALU содержит 8 битов, которые выбирают функцию АЛУ и запускают схему сдвига Биты C загружают отдельные регистры из шины C Сигналы M управляют работой памяти

Наконец, последние 4 бита запускают декодер, который определяет, значение какого регистра будет передано на шину B В данном случае мы выбрали декодер, который содержит 4 входа и 16 выходов, хотя имеется всего 9 разных регистров В более проработанной модели мог бы использоваться декодер, имеющий 4 входа и 9 выходов Мы используем стандартную схему, чтобы не разрабатывать свою собственную Использовать стандартную схему гораздо проще, и кроме того, вы сможете избежать ошибок Ваша собственная микросхема займет меньше места,

но на ее разработку потребуется довольно длительное время, к тому же вы можете построить ее неправильно.

Схема, изображенная на рис. 4.5, работает следующим образом. В начале каждого цикла (задний фронт синхронизирующего сигнала на рис. 4.2) в регистр MIR загружается слово из управляющей памяти, которая на рисунке отмечена буквами MPC. Загрузка регистра MIR занимает период Да;, то есть первый подцикл (см. рис. 4.2).

Когда микрокоманда попадает в MIR, в тракт данных поступают различные сигналы. Значение определенного регистра помещается на шину B, а АЛУ узнает, какую операцию нужно выполнять. Все это происходит во время второго подцикла. После периода *Aw+Ax* входные сигналы АЛУ стабилизируются.

После периода Дг/ стабилизируются сигналы АЛУ N и Z и выходной сигнал схемы сдвига. Затем значения N и Z сохраняются в двух 1-битных триггерах. Эти биты, как и все регистры, которые загружаются из шины C и из памяти, сохраняются на нарастающем фронте синхронизирующего сигнала, ближе к концу цикла тракта данных. Выходной сигнал АЛУ не сохраняется, а просто передается в схему сдвига. Работа АЛУ и схемы сдвига происходит во время подцикла 3.

После следующего интервала, Дг, выходной сигнал схемы сдвига, пройдя через шину C, достигает регистров. Регистры загружаются в конце цикла на нарастающем фронте синхронизирующего сигнала (см. рис. 4.2). Во время подцикла 4 происходит загрузка регистров и триггеров N и Z. Он завершается сразу после нарастающего фронта, когда все значения сохранены, результаты предыдущих операций памяти доступны и регистр MPC загружен. Этот процесс продолжается снова и снова, пока вы не устанете и не выключите компьютер.

Микропрограмме приходится не только управлять трактом данных, но и определять, какая микрокоманда должна быть выполнена следующей, поскольку они не упорядочены в управляющей памяти. Вычисление адреса следующей микрокоманды начинается после загрузки регистра MIR. Сначала в регистр MPC копируется 9-битное поле NEXT_ADDRESS (следующий адрес). Пока происходит копирование, проверяется поле JAM Если оно содержит значение 000, то ничего больше делать не нужно; когда копирование поля NEXT_ADDRESS завершится, регистр MPC укажет на следующую микрокоманду.

Если один или несколько бит в поле JAM равны 1, то требуются еще некоторые действия. Если бит JAMN равен 1, то триггер N соединяется через схему ИЛИ со старшим битом регистра MPC. Если бит JAMZ равен 1, то триггер Z соединяется через схему ИЛИ со старшим битом регистра MPC. Если оба бита равны 1, они оба соединяются через схему ИЛИ с тем же битом А теперь объясним, зачем нужны триггеры N и Z. Дело в том, что после нарастающего фронта сигнала (и вплоть до заднего фронта) шина B больше не запускается, поэтому выходные сигналы АЛУ уже не могут считаться правильными. Сохранение флагов состояния *АЛУ* в регистрах N и Z делает правильные значения стабильными и доступными для вычисления регистра MPC, независимо от того, что происходит вокруг АЛУ.

На рис. 4.5 схема, которая выполняет это вычисление, называется «старший бит». Она вычисляет следующую булеву функцию:

F = (0AMZ И *Z)* ИЛИ QAMN И N)) ИЛИ NEXT_ADDRESS[8]

Отметим, что в любом случае регистр MPC может принять только одно из двух возмол.чых значений:

1. Значение NEXT_ADDRESS.
2. Значение NEXT_ADDRESS со старшим битом, соединенным с логической единицей операцией ИЛИ.

Других значений не существует. Если старший бит значения NEXT_ADDRESS уже равен 1, нет смысла использовать JAMN или JAMZ.

Отметим, что если все биты JAM равны 0, то адрес следующей команды — просто 9-битный номер в поле NEXT_ADDRESS- Если JAMN или JAMZ равны 1, то существует два потенциально возможных адреса следующей микрокоманды: NEXT_ ADDRESS и NEXT_ADDRESS, соединенный операцией ИЛИ с 0x100 (предполагается, что NEXT_ADDRESS≤OxFF). (Отметим, что Ox указывает, что число, следующее за ним, дается в шестнадцатеричной системе счисления). Это проиллюстрировано рис. 4.6. Текущая микрокоманда с адресом 0x75 содержит поле NEXT_ADDRESS=0x92, причем бит JAMZ установлен на 1. Следовательно, следующий адрес микрокоманды зависит от значения бита Z, сохраненного при предыдущей операции АЛУ. Если бит Z равен 0, то следующая микрокоманда имеет адрес 0x92. Если бит Z равен 1, то следующая микрокоманда имеет адрес 0x192.

Третий бит в поле JAM — JMPC. Если он установлен, то 8 битов регистра MBR поразрядно связываются операцией ИЛИ с 8 младшими битами поля NEXT_ ADDRESS из текущей микрокоманды. Результат отправляется в регистр MPC. На рис. 4.5 значком «ИЛИ» обозначена схема, которая выполняет операцию ИЛИ над MBR и NEXT_ADDRESS, если бит JMPC равен 1, и просто отправляет NEXT_ADDRESS в регистр MPC, если бит JMPC равен 0. Если JMPC равен 1, то младшие 8 битов поля NEXT_ADDRESS равны 0. Старший бит может быть 0 или 1, поэтому значение поля NEXT_ADDRESS обычно 0x000 или 0x100. Почему иногда используется 0x000, а иногда — 0x100, мы обсудим позже.

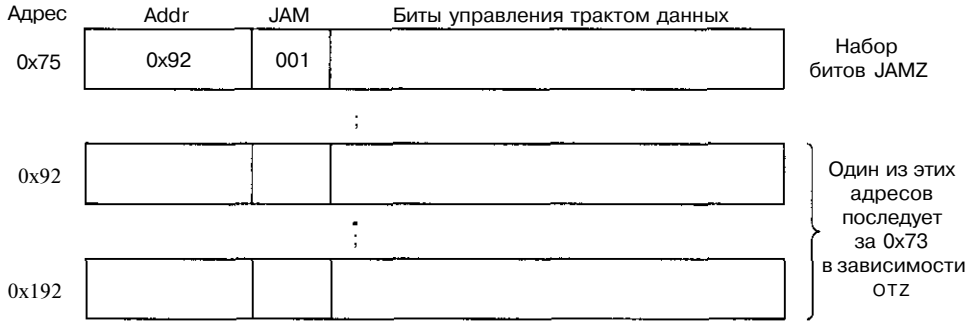


Рис. 4.6. Микрокоманда с битом JAMZ, равным 1, указывает на две потенциальные последующие микрокоманды

Возможность выполнять операцию ИЛИ над MBR и NEXT_ADDRESS и сохранять результат в регистре MPC позволяет реализовывать межуровневые переходы. Отметим, что по битам, находящимся в регистре MBR, можно определить любой адрес из 256 возможных. Регистр MBR содержит код операции, поэтому использование JMPC приведет к единственно возможному выбору следующей

микрокоманды. Этот метод позволяет осуществлять быстрый переход у функции, соответствующей вызванному коду операции.

Для того чтобы продолжить чтение этой главы, очень важно понимать принципы синхронизации машины, поэтому повторим их еще раз. Синхронизирующий сигнал делится на подциклы, хотя внешние изменения этого сигнала происходят только на заднем фронте, с которого начинается цикл, и на нарастающем фронте, который загружает регистры и триггеры N и Z. Посмотрите еще раз на рис. 4.2.

Во время подцикла 1, который инициируется задним фронтом сигнала, адрес, находящийся в данный момент в регистре MPC, загружается в регистр MIR. Во время подцикла 2 регистр MIR выдает сигналы и в шину B загружается выбранный регистр. Во время подцикла 3 происходит работа АЛУ и схемы сдвига. Во время подцикла 4 стабилизируются значения шины C, шин памяти и АЛУ. На нарастающем фронте сигнала загружаются регистры из шины C, загружаются триггеры N и Z, а регистры MBR и MDR получают результаты работы памяти, начавшейся в конце предыдущего цикла (если эти результаты вообще имеются). Как только регистр MBR получает свое значение, загружается регистр MPC. Это происходит где-то в середине отрезка между нарастающим и задним фронтами, но уже после загрузки MBR/MDR. Он может загружаться уровнем сигнала (но не фронтом сигнала) либо загружаться через фиксированный отрезок времени после нарастающего фронта. Все это означает, что регистр MPC не получает своего значения до тех пор, пока не будут готовы регистры MBR, N и Z, от которых он зависит. На заднем фронте сигнала, когда начинается новый цикл, регистр MPC может обращаться к памяти.

Отметим, что каждый цикл является самодостаточным. В каждом цикле определяется, значение какого регистра должно поступать на шину B, что должны делать АЛУ и схема сдвига, куда нужно сохранить значение шины C, и, наконец, каким должно быть следующее значение регистра MPC.

Следует сделать еще одно замечание по поводу рис. 4.5. До сих пор мы считали MPC регистром, который состоит из 9 битов и загружается на высоком уровне сигнала. В действительности этот регистр вообще не нужен. Все его входные сигналы можно непосредственно связать с управляющей памятью. Поскольку они имеются в управляющей памяти на заднем фронте синхронизирующего сигнала, когда выбирается и считывается регистр MIR, этого достаточно. Их не нужно хранить в регистре MPC. По этой причине MPC может быть реализован в виде **виртуального регистра,** который представляет собой просто место скопления сигналов и похож скорее на коммутационное поле, чем на настоящий регистр. Если MPC сделать виртуальным регистром, то процедура синхронизации сильно упрощается: теперь события происходят только на нарастающем фронте и заднем фронте сигнала. Но если вам проще считать MPC реальным регистром, то такой подход тоже вполне допустим.

Пример архитектуры команд: IJVM

Чтобы продолжить описание нашего примера, введем уровень набора команд, которые должна интерпретировать микропрограмма машины IJVM (см. рис. 4.5). Для удобства уровень архитектуры команд мы иногда будем называть **макроархитек-**

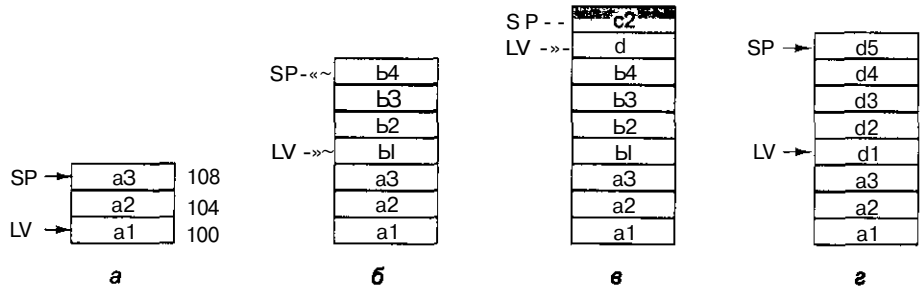
турой, чтобы противопоставить его микроархитектуре. Однако перед тем как приступать к описанию IJVM, мы немного отвлечемся.

Стек

Во всех языках программирования есть понятие процедур с локальными переменными Эти переменные доступны во время выполнения процедуры, но перестают быть доступными после окончания процедуры. Возникает вопрос: где должны храниться такие переменные?

К сожалению, предоставить каждой переменной абсолютный адрес в памяти невозможно. Проблема заключается в том, что процедура может вызывать себя сама. Мы рассмотрим такие рекурсивные процедуры в главе 5. А пока достаточно сказать, что если процедура вызывается дважды, то хранить ее переменные под конкретными адресами в памяти нельзя, поскольку второй вызов нарушит результаты первого.

Вместо этого используется другая стратегия. Для переменных резервируется особая область памяти, которая называется **стеком,** но отдельные переменные не получают в нем абсолютных адресов. Какой-либо регистр, скажем, LV, указывает на базовый адрес локальных переменных для текущей процедуры. Рассмотрим рис. 4.7, *а.* В данном случае вызывается процедура А с локальными переменными *а1,а2и а3,* и для этих переменных резервируется участок памяти, начинающийся с адреса, который указывается регистром LV. Другой регистр, SP, указывает на старшее слово локальных переменных процедуры А. Если значение регистра LV равно 100, а слова состоят из 4 байтов, то значение SP будет 108. Для обращения к переменной нужно вычислить ее смещение от адреса LV. Структура данных между LV и SP (включая оба указанных слова) называется **фреймом локальных переменных.**



Рис, 4.7. Стек для хранения локальных переменных во время процедуры *А* (а); после того как процедура *А* вызывает процедуру *В* (б), после того как процедура *В* вызывает процедуру С (в); после того как процедуры С и в прекращаются, а процедура *А* вызывает процедуру *D* (г)

А теперь давайте посмотрим, что происходит, если процедура *А* вызывает другую процедуру, *В.* Где должны храниться 4 локальные переменные процедуры *В (Ы, Ь2, Ь3, Ь4)?* Ответ: в стеке, расположенном над стеком для процедуры *Л,* как показано на рис. 4.7, *6.* Отметим, что после вызова процедуры регистр LV указывает уже на локальные переменные процедуры *В.* Обращаться к локальным переменным процедуры *В* можно по их сдвигу от LV. Если процедура *В* вызывает про-

цедуру *C,* то регистры LV и SP снова переопределяются и указывают на местонахождение локальных переменных процедуры *C,* как показано на рис. 4.7, *в.*

Когда процедура C завершается, *B* снова активизируется и стек возвращается в прежнее состояние (см. рис. 4.7, *б),* так что LV теперь указывает на локальные переменные процедуры *B.* Когда процедура *B* завершается, стек переходит в исходное состояние (см. рис. 4.7, *а).* При любых условиях LV указывает на базовый адрес стекового фрейма для текущей процедуры, а SP — на верхнее слово этого фрейма.

Предположим, что процедура *A* вызывает процедуру *D,* которая содержит 5 локальных переменных. Соответствующий стек показан на рис. 4.7, *г.* Локальные переменные процедуры *D* используют участок памяти процедуры *B* и часть стека процедуры *C.* В памяти с такой организацией размещаются только текущие процедуры. Когда процедура завершена, отведенный для нее участок памяти освобождается.

Но стек используется не только для хранения локальных переменных, а также и для хранения операндов во время вычисления арифметических выражений. Такой стек называется **стеком операндов.** Предположим, что перед вызовом процедуры *B* процедура Л должна произвести следующее вычисление:

a1=a2+a3

Чтобы вычислить эту сумму, можно поместить *a2* в стек, как показано на рис. 4.8, *а.* Тогда значение регистра SP увеличится на число, равное количеству байтов в слове (скажем, на 4), и будет указывать на адрес первого операнда. Затем в стек помещается переменная *a3,* как показано на рис. 4.8, *б.* Отметим, что названия процедур и переменных выбираются пользователем, а названия регистров и кодов операций встроены. Названия процедур и переменных мы выделяем в тексте курсивом.

Теперь можно произвести вычисление, выполнив команду, которая выталкивает два слова из стека, складывает их и помещает результат обратно в стек, как показано на рис. 4.8, *в.* После этого верхнее слово можно вытолкнуть из стека и поместить его в локальную переменную *a1,* как показано на рис. 4.8, *г.*

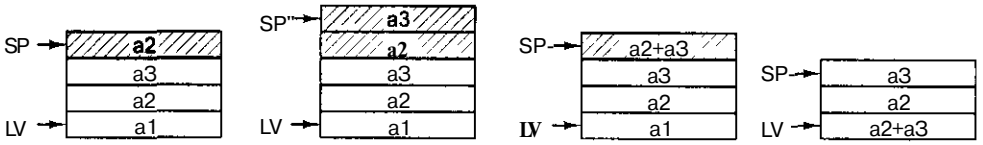


Рис. 4.8. Использование стека операндов для арифметических действий

Фреймы локальных переменных и стеки операндов могут смешиваться. Например, когда вызывается функция f при вычислении выражения $x^2+f(x)$, часть этого выражения (x^2) может находиться в стеке операндов. Результат вычисления функции остается в стеке над x^2, чтобы следующая команда сложила их.

Следует упомянуть, что все машины используют стек для хранения локальных переменных, но не все используют его для операндов. В большинстве машин нет стека операндов, но у JVM и IJVM он есть. Стековые операции мы рассмотрим подробно в главе 5.

Модель памяти IJVM

А теперь мы можем рассмотреть архитектуру **IJVM.** Она состоит из памяти, которую можно рассматривать либо как массив из 4 294 967 296 байтов (4 Гбайт), либо как массив из 1 073 741 824 слов, каждое из которых содержит 4 байта. В отличие от большинства архитектур команд, виртуальная машина Java не совершает обращений к памяти, видимых на уровне команд, но здесь существует несколько неявных адресов, которые составляют основу для указателя. Команды IJVM могут обращаться к памяти только через эти указатели. Определены следующие области памяти:

1. *Набор констант.* Эта область состоит из констант, цепочек и указателей на другие области памяти, на которые можно делать ссылку. Данная область загружается в тот момент, когда программа загружается из памяти, и после этого не меняется. Существует неявный регистр CPP (Constant Pool Pointer — указатель набора констант), который содержит адрес первого слова набора констант.
2. *Фрейм локальных переменных.* Эта область предназначена для хранения переменных во время выполнения процедуры. Она называется **фреймом локальных переменных.** В начале этого фрейма располагаются параметры (или аргументы) вызванной процедуры. Фрейм локальных переменных не включает в себя стек операндов. Он помещается отдельно. Исходя из соображений производительности, мы поместили стек операндов прямо над фреймом локальных переменных. Существует неявный регистр, который содержит адрес первой переменной фрейма. Мы назовем этот регистр LV (Local Variable — локальная переменная). Параметры вызванной процедуры хранятся в начале фрейма локальных переменных.
3. *Стек операндов.* Стек операндов не должен превышать определенный размер, который заранее вычисляется компилятором Java. Пространство стека операндов располагается прямо над фреймом локальных переменных, как показано на рис. 4.9. В данном случае стек операндов удобно считать частью фрейма локальных переменных. В любом случае существует виртуальный регистр, который содержит адрес верхнего слова стека. Отметим, что в отличие от регистров CPP и LV этот указатель меняется во время выполнения процедуры, поскольку операнды помещаются в стек и выталкиваются из него.
4. *Область процедур.* Наконец, существует область памяти, в которой содержится программа. Есть виртуальный регистр, содержащий адрес команды, которая будет вызвана следующей. Этот указатель называется счетчиком команд, или PC (Program Counter). В отличие от других участков памяти, область процедуры представляет собой массив байтов.

Следует сделать одно примечание по поводу указателей. Регистры CPP, LV и SP указывают на *слова,* а не на *байты,* и смещения происходят на определенное число слов. Например, LV, LV+1 и LV+2 указывают на первые три слова из фрейма локальных переменных, а LV, LV+4 и LV+8 — на слова, расположенные на расстоянии четырех слов (16 байтов) друг от друга.

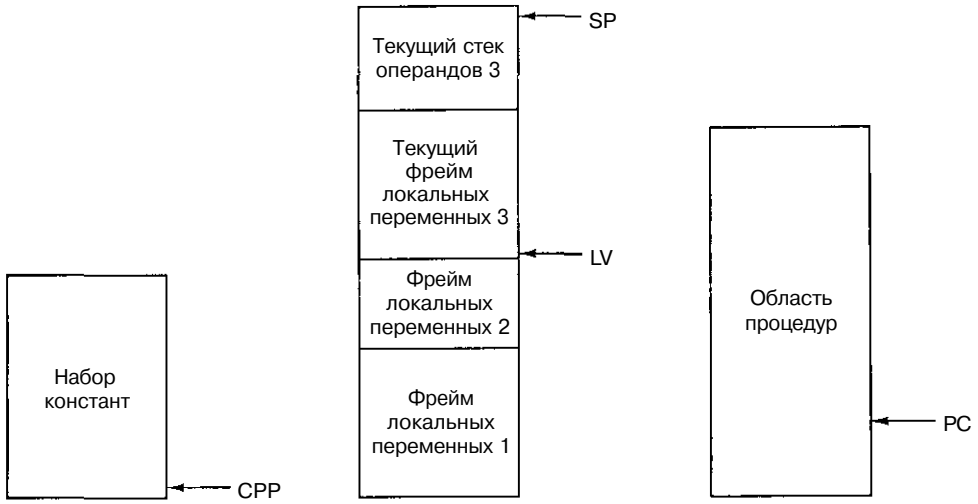


Рис. 4.9. Области памяти IJVM

Регистр PC, напротив, содержит адреса байтов, и изменение этого значения означает увеличение на определенное количество байтов, а не слов. Обращение к памяти регистром PC отличается от обращений других регистров, поэтому в машине Mic-1 и предусмотрен специальный порт памяти для PC. Запомните, что его размер составляет всего один байт. Если увеличить PC на единицу и начать процесс чтения, то это приведет к вызову следующего *байта*. Если увеличить SP на единицу и начать процесс чтения, то это приведет к вызову следующего *слова*.

Набор команд IJVM

Набор команд IJVM приведен в табл. 4.2. Каждая команда состоит из кода операции и иногда из операнда (например, смещения адреса или константы). В первом столбце приводится шестнадцатеричный код команды. Во втором столбце дается мнемоника языка ассемблера. В третьем столбце описывается предназначение команды.

Команды нужны для того, чтобы помещать слова из различных источников в стек. Источники — это набор констант (LDC_W), фрейм локальных переменных (ILOAD) и сама команда (BIPUSH). Переменную можно также вытолкнуть из стека и сохранить ее во фрейме локальных переменных (ISTORE). Над двумя верхними словами стека можно совершать две арифметические (IADD и ISUB) и две логические операции (IAND и I0R). При выполнении любой арифметической или логической операции два слова выталкиваются из стека, а результат помещается обратно в стек. Существует 4 команды перехода: одна для безусловного перехода (GOTO), а три другие для условных переходов (IFEQ, IFLT и IF_ICMPEQ). Все эти команды изменяют значение PC на размер их смещения, который следует за кодом операции в команде. Операнд смещения состоит из 16 битов. Он прибавляется к адресу кода операции. Существуют также команды для перестановки двух верхних слов стека (SWAP), дублирования верхнего слова (DUP) и удаления верхнего слова (POP).

Таблица 4.2. Набор команд IJVM. Размер операндов byte, const и varnum — 1 байт. Размер операндов disp, index и offset — 2 байта

Число	Мнемоника	Примечание
0x10	BIPUSHbyfe	Помещает байт в стек
0x59	DUP	Копирует верхнее слова стека и помещает его в стек
0xA7	GOTO *offset*	Безусловный переход
0x60	IADD	Выталкивает два слова из стека; помещает в стек их сумму
0x7E	IAND	Выталкивает два слова из стека; помещает в стек результат логического умножения (операция И)
0x99	IFEQoffsef	Выталкивает слово из стека и совершает переход, если оно равно нулю
0x9B	IFLT *offset*	Выталкивает слово из стека и совершает переход, если оно меньше нуля
0x9F	IFJCMPEQ *offset*	Выталкивает два слова из стека; совершает переход, если они равны
0x84	IINC *varnum const*	Прибавляет константу к локальной переменной
0x15	\LOAD *varnum*	Помещает локальную переменную в стек
0xB6	INVOKEVIRTUAL *disp*	Вызывает процедуру
0x80	IOR	Выталкивает два слова из стека; помещает в стек результат логического сложения {операция ИЛИ)
OxAC	IRETURN	Выдает результат выполнения процедуры (целое число)
0x36	ISTORE *varnum*	Выталкивает слово из стека и запоминает его во фрейме локальных переменных
0x64	ISUB	Выталкивает два слова из стека; помещает в стек их разность
0x13	*LDCJN index*	Берет константу из набора констант и помещает ее в стек
0x00	NOP	Не производит никаких действий
0x57	POP	Удаляет верхнее слово стека
0x5F	SWAP	Переставляет два верхних слова стека
0xC4	WIDE	Префиксная команда; следующая команда содержит 16-битный индекс

Некоторые команды имеют сложный формат, допускающий краткую форму для часто используемых версий. Из всех механизмов, которые JVM применяет для этого, в IJVM мы включили два. В одном случае мы пропустили краткую форму в пользу более традиционной. В другом случае мы показываем, как префиксная команда WIDE может использоваться для изменения следующей команды.

Наконец, существует команда для вызова другой процедуры (INVOKEVIRTUAL) и команда для выхода из текущей процедуры и возвращения к процедуре, из которой она была вызвана. Из-за сложности механизма мы немного упростили определение. Ограничение состоит в том, что, в отличие от языка Java, в нашем примере процедура может вызывать только такую процедуру, которая находится внутри нее. Это ограничение сильно искажает язык Java, но зато позволяет представить более простой механизм, избегая требования размещать процедуру динамически. (Если вы не знакомы с объектно-ориентированным программированием, вы можете пропустить это предложение. Мы просто превратили язык Java из объектно-

ориентированного в обычный, такой как C или Pascal.) На всех компьютерах, кроме JVM, адрес процедуры, которую нужно вызвать, непосредственно определяется командой CALL, поэтому наш подход скорее правило, чем исключение.

Механизм вызова процедуры состоит в следующем. Сначала вызывающая программа помещает в стек указатель на объект, который нужно вызвать. На рис. 4.10, *а* этот указатель обозначен буквами OBJREF. Затем вызывающая программа помещает в стек параметры процедуры (в данном примере *Параметр 1, Параметр 2* и *Параметр 3).* После этого выполняется команда INVOKEVIRTUAL.

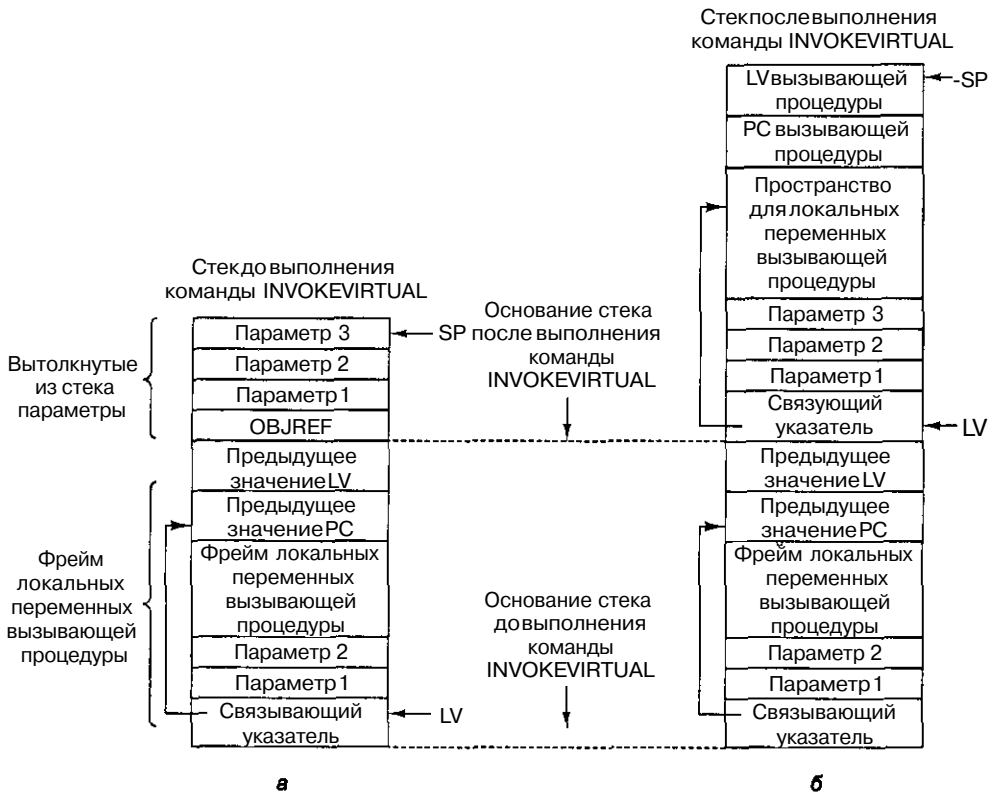


Рис. 4.10. Память до выполнения команды INVOKEVIRTUAL (а); память после выполнения этой команды (б)

Команда INVOKEVIRTUAL включает в себя относительный адрес *(disp).* Он указывает на позицию в наборе констант. В этой позиции содержится начальный адрес вызываемой процедуры, которая хранится в области процедур. Первые 4 байта в области процедур содержат специальные данные. Первые два байта представляют собой целое 16-битное число, указывающее на количество параметров данной процедуры (сами параметры были ранее помещены в стек). В данном случае OBJREF считается параметром: параметром 0. Это 16-битное целое число вместе со значением SP дает адрес OBJREF. Отметим, что регистр LV указывает на OBJREF, а не

на первый реальный параметр. Выбор, на что указывает LV, в какой-то степени произволен.

Следующие два байта в области процедур представляют еще одно 16-битное целое число, указывающее размер области локальных переменных для вызываемой процедуры. Дело в том, что для данной процедуры предоставляется новый стек, который размещается прямо над фреймом локальных переменных, для этого и нужно это число. Наконец, пятый байт в области процедур содержит код первой операции, которую нужно выполнить.

Ниже описывается, что происходит перед вызовом процедуры (см. также рис. 4.10). Два байта без знака, которые следуют за кодом операции, используются для индексирования таблицы констант (первый байт — это старший байт). Команда вычисляет базовый адрес нового фрейма локальных переменных. Для этого из указателя стека вычитается число параметров, а LV устанавливается на OBJREF. В OBJREF хранится адрес ячейки, в которой находится старое значение'PC. Этот адрес вычисляется следующим образом. К размеру фрейма локальных переменных (параметры + локальные переменные) прибавляется адрес, содержащийся в регистре LV. Сразу над адресом, в котором должно быть сохранено старое значение PC, находится адрес, в котором должно быть сохранено старое значение LV. Над этим адресом начинается стек для новой вызванной процедуры. SP указывает на старое значение LV, адрес которого находится сразу под первой пустой ячейкой стека. Помните, что SP всегда указывает на верхнее слово в стеке. Если стек пуст, то SP указывает на адрес, который находится непосредственно под стеком, поскольку стек заполняется снизу вверх.

И наконец, для выполнения команды INVOKEVIRTUAL нужно сделать так, чтобы PC указывал на пятый байт в кодовом пространстве процедуры.

Команда IRETURN противоположна команде INVOKEVIRTUAL (рис. 4.11). Она освобождает пространство, используемое процедурой. Она также возвращает стек в предыдущее состояние, за исключением того, что: 1) OBJREF и все параметры удаляются из стека; 2) возвращенное значение помещается в стек, туда, где раньше находился OBJREF. Чтобы восстановить прежнее состояние, команда IRETURN должна вернуть прежние значения указателей PC и LV. Для этого она обращается к связующему указателю (это слово, определяемое текущим значением LV). В этом месте, где изначально находился параметр OBJREF, команда OBJREF сохранила адрес, содержащий старое значение PC. Это слово и слово над ним извлекаются, чтобы восстановить старые значения PC и LV соответственно. Возвращенное значение, которое хранится на самой вершине стека завершающейся процедуры, копируется туда, где изначально находился OBJREF, и теперь SP указывает на этот адрес. И тогда управление переходит к команде, которая следует сразу за INVOKEVIRTUAL.

До сих пор у нашей машины не было никаких команд ввода-вывода. Мы и не собираемся их вводить. В нашем примере, как и в виртуальной машине Java, они не нужны, и в описании JVM никогда не упоминаются процессы ввода-вывода. Считается, что машина без ввода-вывода более надежна. (Чтение и запись осуществляются в JVM путем вызова специальных процедур.)

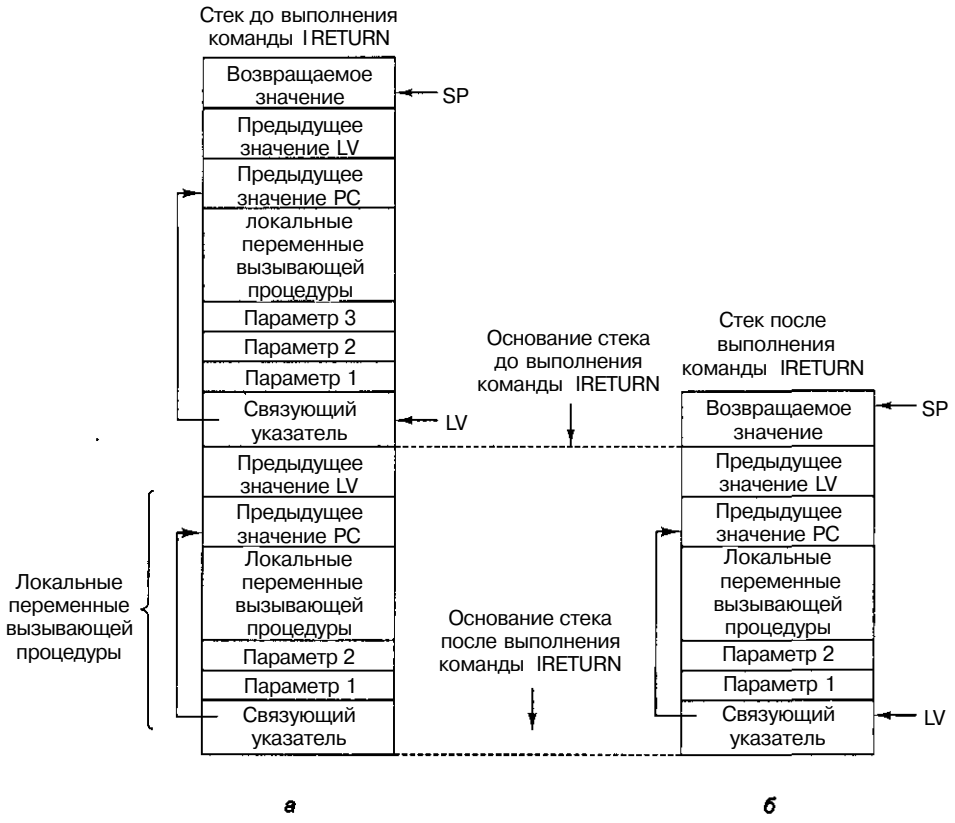


Рис. 4.11. Память до выполнения команды IRETURN (а); память после выполнения этой команды (б)

Компиляция Java для IJVM

А теперь рассмотрим, как язык Java связывается с машиной IJVM. В листинге 4.1 представлен небольшой фрагмент программы на языке Java. Компилятор Java должен был переделать эту программу в программу на языке ассемблер IJVM. Эта программа приведена в листинге 4.2. Цифры с 1 по 15 в левой части листинга, а также комментарии за значком «//» не являются частью самой программы. Они даны для наглядности и просто облегчают понимание. Затем ассемблер Java транслировал бы ее в программу в двоичном коде. Эта программа приведена в листинге 4.3. (В действительности компилятор Java сразу производит двоичную программу.) В данном примере i — локальная переменная 1, j — локальная переменная 2, а k — локальная переменная 3.

Листинг 4.1. Фрагмент программы на языке Java

```
i=j+k;
if(i==3)
    k=0;
else
    j=j-i;
```

Листинг 4.2. Программа на языке ассемблер Java

```
 1      ILOAD j        //i=J+k
 2      ILOAD k
 3      IADD
 4      ISTORE 1
 5      I LOAD 1       //if (1-3)
 6      BIPUSH 3
 7      IFJCMPEQ LI
 8      ILOAD j        //J-J-l
 9      BIPUSH 1
10      ISUB
11      ISTORE j
12      GOTO L2
13  LI  BIPUSH 0       //k-0
14      ISTORE k
15  L2
```

Листинг 4.3. Программа IJVM в шестнадцатеричном коде

```
0x15  0x02
0x15  0x03
0x60
0x36  0x01
0x15  0x01
0x10  0x03
0x9F 0x00 OxOD
0x15  0x02
0x10  0x01
0x64
Ox36  0x02
0xA7  0x00  0x07
0x10  0x00
0x36  0x03
```

Скомпилированная программа проста. Сначала j и к помещаются в стек, складываются, а результат сохраняется в i Затем i и константа 3 помещаются в стек и сравниваются Если они равны, то совершается условный переход к *L1,* где к получает значение 0 Если они не равны, то выполняется часть программы после IF_ICMPEQ После этого осуществляется переход к *L2,* где сливаются части el se и then

Стек операндов для программы IJVM, приведенной в листинге 4.2, изображен на рис. 4 12 До начала выполнения программы стек пуст, что показывает горизонтальная черта над цифрой 0 После выполнения первой команды I LOAD j помещается в стек (См на рисунке прямоугольник над цифрой 1) Цифра 1 означает, что выполнена команда 1 После выполнения второй команды ILOAD в стеке оказываются уже два слова, как показано в прямоугольнике над цифрой 2 После выполнения команды IADD в стеке остается только одно слово, которое представляет собой сумму j+k. Когда верхнее слово выталкивается из стека и сохраняется в i, стек снова становится пустым

Команда 5 (ILOAD) начинает оператор if. Эта команда помещает i в стек Затем идет константа 3 (в команде 6) После сравнения стек снова становится пустым (7) Команда 8 является началом фрагмента el se Он продолжается вплоть до команды 12, когда совершается переход к метке *L2.*

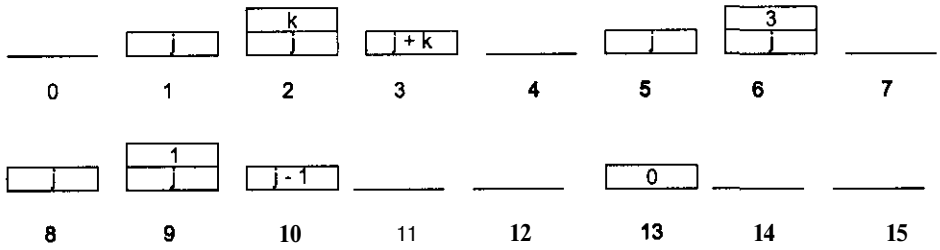


Рис. 4.12. Состояние стека после выполнения каждой команды в программе, приведенной в листинге 4.2

Пример реализации микроархитектуры

Мы подробно описали, что такое микроархитектура и макроархитектура. Осталось осуществить реализацию. Другими словами, нам предстоит узнать, что собой представляет и как работает программа микроархитектурного уровня, интерпретирующая команды макроархитектуры. Прежде чем ответить на эти вопросы, мы должны изложить систему обозначений, которую мы будем использовать для описания.

Микрокоманды и их запись

В принципе мы могли бы описать работу управляющей памяти с помощью двоичной системы счисления, по 36 битов в слове. Но гораздо удобнее ввести систему обозначений, с помощью которой можно передать суть рассматриваемых вопросов, и при этом не вдаваться в ненужные подробности. Важно понимать, что язык, который мы выбираем, предназначен для того, чтобы проиллюстрировать основные принципы работы программы, а не для того, чтобы использовать его в новых проектах. Если бы нашей целью было практическое применение языка, мы бы ввели совсем другую запись, чтобы максимально повысить гибкость программы. При этом была бы очень важна проблема выбора адресов, поскольку адреса в памяти не упорядочены. Насколько эффективным будет выбор адресов, зависит от способностей разработчика. Поэтому мы введем простой символический язык, который полностью описывает каждую операцию, но не объясняет полностью, как определяются все адреса.

Наша система обозначений показывает все действия, которые происходят на одной линии за один цикл. Теоретически для описания этих операций мы могли бы использовать язык высокого уровня. Однако контроль циклов очень важен, поскольку это дает возможность выполнять несколько операций одновременно. Кроме того, такой контроль необходим для того, чтобы можно было проанализировать каждый цикл, понять все операции и проверить их. Если целью разработки является повышение скорости и производительности, то имеет значение каждый цикл. При практической реализации в программу включается множество различных приемов для экономии циклов. В такой экономии есть большая выгода: четырехцикловая команда, которую можно сократить на два цикла, будет после этого выполняться в два раза быстрее. И такое повышение скорости достигается каждый раз, когда мы выполняем эту команду.

Один из возможных подходов — просто выдать список сигналов, которые должны активизироваться в каждом цикле. Предположим, что в одном цикле мы хотим

увеличить значение SP на единицу. Мы также хотим инициировать операцию чтения и хотим, чтобы следующая команда находилась в управляющей памяти в ячейке 122. Тогда мы могли бы написать:

```
ReadRegister=SP, ALIMNC, WSP, Read, NEXT_ADDRESS=122
```

Здесь WSP значит «записать регистр SP». Эта запись полная, но она сложна для понимания. Вместо этого мы соединим эти операции и передадим в записи результат действий:

```
SP-SP+1, rd
```

Назовем наш микроассемблер высокого уровня «MAL» (Micro Assembly Language — микроассемблер). По-французски «MAL» значит «болезнь» — это то, что с вами случится, если вы будете писать слишком большие программы на этом языке. Язык MAL разработан для того, чтобы продемонстрировать основные характеристики микроархитектуры. Во время каждого цикла могут записываться любые регистры, но обычно записывается только один. Значение только одного регистра может передаваться на шину B и в АЛУ. На шине A может быть +1, 0, -1 и регистр H. Следовательно, для обозначения определенной операции мы можем использовать простой оператор присваивания, как в языке Java. Например, чтобы копировать регистр SP в регистр MDR, мы можем написать:

```
MDR=SP
```

Чтобы показать, что мы используем какую-либо функцию АЛУ, мы можем написать, например:

```
MDR-H+SP
```

Эта строка означает, что значение регистра H складывается со значением регистра SP и результат записывается в регистр MDR. Операция сложения коммутативна (это значит, что порядок операндов не имеет значения), поэтому данное выше выражение можно записать в виде:

```
MDR=SP+H
```

и при этом породить ту же 36-битную микрокоманду, хотя, строго говоря, H является левым операндом АЛУ.

Мы должны использовать только допустимые операции. Самые важные операции приведены в табл. 4.3, где SOURCE — значение любого из регистров MDR, PC, MBR, MBRU, SP, LV, CPP, TOS и OPC (MBRU (MBR Unsigned) - это значение регистра MBR без знака). Все эти регистры могут выступать в качестве источников значений для АЛУ (они поступают в АЛУ через шину B). Сходным образом DEST может обозначать любой из следующих регистров: MAR, MDR, PC, SP, LV, CPP, TOS, OPC и H. Любой из этих регистров может быть пунктом назначения для выходного сигнала АЛУ, который передается к регистрам по шине C. Многие, казалось бы, разумные утверждения недопустимы. Например, выражение

```
MDR=SP+MDR
```

выглядит вполне корректно, но эту операцию нельзя выполнить в тракте данных, изображенном на рис. 4.5, за один цикл. Такое ограничение существует, поскольку для операции сложения (в отличие от увеличения или уменьшения на 1) один из операндов должен быть значением регистра H. Точно так же, выражение

```
H=H-MDR
```

могло бы пригодиться, но оно невозможно, поскольку единственным возможным источником вычитаемого является регистр H. Ассемблер должен отбрасывать выражения, которые кажутся пригодными, но в действительности недопустимы.

В нашей системе записи допускается использование нескольких операторов присваивания. Например, чтобы прибавить 1 к регистру SP и сохранить полученное значение в регистрах SP и MDR, нужно записать следующее:

```
SP=MDR=SP+1
```

Для обозначения процессов считывания из памяти и записи в память слов по 4 байта мы будем вставлять в микрокоманду слова rd и wr. Для вызова байта через 1-байтный порт используется команда fetch. Операции присваивания и операции взаимодействия с памятью могут происходить в одном и том же цикле. То, что происходит в одном цикле, записывается в одну строку.

Чтобы избежать путаницы, напомним еще раз, что Mic-1 может обращаться к памяти двумя способами. При чтении и записи 4-байтных слов данных используются регистры MAR/MDR. Эти процессы показываются в микрокомандах словами rd и wr соответственно. При чтении 1-байтных кодов операций из потока команд используются регистры PC/MBR. В микрокоманде это показывается словом fetch. Оба типа операций взаимодействия с памятью могут происходить одновременно.

Однако один и тот же регистр не может получать значение из памяти и тракта данных в одном и том же цикле. Рассмотрим кусок программы

```
MAR=SP. rd
MDR-H
```

В результате выполнения первой микрокоманды значение из памяти приписывается регистру MDR в конце второй микрокоманды. Однако вторая микрокоманда в то же самое время приписывает другое значение регистру MDR. Эти две операции присваивания конфликтуют, поскольку результаты не определены.

Таблица 4.3. Все допустимые операции. Любую из перечисленных операций можно расширить, добавив «<<8», что означает сдвиг результата влево на 1 байт. Например, часто используется операция H = MBR << 8

DEST=H
DEST=SOURCE
DEST=$\overline{H}$
DEST=$\overline{SOURCE}$
DEST=H+SOURCE
DEST=H+SOURCE+1
DEST=H+1
DEST=SOURCE+1
DEST=SOURCE-H
DEST=SOURCE-1
DEST= -H
DEST=H И SOURCE
DEST=H ИЛИ SOURCE
DEST=O
DESTM
DEST=-1

Помните, что в каждой микрокоманде должен явно показываться адрес следующей микрокоманды Однако часто бывает так, что микрокоманда вызывается только одной другой микрокомандой, а именно той микрокомандой, которая находится в строке над ней. Чтобы упростить работу программиста, микроассемблер обычно приписывает адрес каждой микрокоманде (порядок адресов может и не соответствовать последовательности микрокоманд в управляющей памяти) и заполняет поле NEXT_ADDRESS, так что последовательность выполнения микрокоманд соответствует последовательности строк микропрограммы.

Однако программисту иногда нужно совершить переход, условный или безусловный. Запись безусловных переходов проста:

```
goto label
```

Такая запись может включаться в любую микрокоманду В ней явным образом указывается имя следующей микрокоманды. Например, очень часто последовательность микрокоманд заканчивается возвращением к первой команде основного цикла, поэтому последняя команда в каждой такой последовательности содержит запись

```
goto Main1
```

Отметим, что в тракте данных происходят обычные операции даже во время выполнения микрокоманд, которые содержат goto. В любой микрокоманде есть поле NEXT_ADDRESS. Команда goto сообщает ассемблеру, что в это поле вместо адреса микрокоманды, записанной в следующей строке, нужно поместить особое значение. В принципе каждая строка должна содержать запись goto, но если нужный адрес — это адрес микрокоманды, записанной в следующей строке, goto может опускаться для удобства.

Для условных переходов нам требуется другая запись Помните, что JAMN и JAMZ используют биты N и Z соответственно. Например, иногда нужно проверить, не равно ли значение регистра 0. Для этого можно было бы пропустить это значение через АЛУ, сохранив его после этого в том же регистре. Тогда мы бы написали:

```
TOS=TOS
```

Запись выглядит забавно, но выполняет необходимые действия (устанавливает триггер Z и записывает значение в регистре TOS). В целях удобочитаемости микропрограммы мы расширили язык MAL, добавив два новых воображаемых регистра N и Z, которым можно присваивать значения. Например, строка

```
Z=TOS
```

пропускает значение регистра TOS через АЛУ, устанавливая триггер Z (и N), но при этом не сохраняет значение ни в одном из регистров. Использование регистра Z или N в качестве пункта назначения показывает микроассемблеру, что нужно установить все биты в поле C (см. рис. 4.4) на 0. Тракт данных проходит обычный цикл, выполняются все обычные допустимые операции, но ни один из регистров не записывается. Не важно, где находится пункт назначения' в регистре N или в регистре Z. Микрокоманды, которые при этом порождает микроассемблер, одинаковы. Программисты, выбравшие не тот регистр, в наказание будут неделю работать на первом компьютере IBM PC с частотой 4,77 МГц.

Чтобы микроассемблер установил бит JAMZ, нужно написать следующее:

```
if(Z) goto LI, else goto L2
```

Поскольку аппаратное обеспечение требует, чтобы 8 младших битов этих адресов совпадали, задача микроассемблера состоит в том, чтобы присвоить им такие адреса. С другой стороны, *L2* может находиться в любом из младших 256 слов управляющей памяти, поэтому микроассемблер без труда найдет подходящую пару.

Часто эти два утверждения сочетаются. Например,

```
Z=TOS. if(Z) goto LI. else goto L2
```

В результате такой записи MAL породит микрокоманду, в которой значение регистра TOS пропускается через АЛУ, но при этом нигде не сохраняется, так что это значение устанавливает бит Z. Сразу после загрузки из АЛУ бит Z соединяется со старшим битом регистра MPC через схему ИЛИ, вследствие чего адрес следующей микрокоманды будет вызван или из *L2,* или из *ЫЛ.* Значение регистра MPC стабилизируется, и он сможет использовать его для вызова следующей микрокоманды.

Наконец, нам нужна специальная запись, чтобы использовать бит JMPC:

```
goto (MBR OR value)
```

Эта запись сообщает микроассемблеру, что нужно использовать *value* (значение) для поля NEXT^ADDRESS и установить бит JMPC, так чтобы MBR соединялся через схему ИЛИ с регистром MPC вместе со значением NEXT_ADDRESS. Если *value* равно 0, достаточно написать:

```
goto (MBR)
```

Отметим, что только 8 младших битов регистра MBR соединяются с регистром MPC (см. рис. 4.5), поэтому вопрос о знаковом расширении тут не возникает. Также отметим, что используется то значение MBR, которое доступно в конце текущего цикла.

Реализация IJVM с использованием Mic-1

Сейчас мы уже дошли до того момента, когда можно соединить все части вместе. В табл. 4.4-приводится микропрограмма, которая работает на микроархитектуре Mic-1 и интерпретирует IJVM. Программа очень короткая — всего 112 микрокоманд. Таблица состоит из трех столбцов. В первом столбце записано символическое обозначение микрокоманды, во втором — сама микрокоманда, а в третьем — комментарий. Как мы уже говорили, последовательность микрокоманд не обязательно соответствует последовательности адресов в управляющей памяти.

Выбор названий большинства регистров, изображенных на рис. 4.1, должен стать очевидным. Регистры CPP (Constant Pool Pointer — указатель набора констант), LV (Local Variable pointer — указатель фрейма локальных переменных) и SP (Stack Pointer — указатель стека) содержат указатели адресов набора констант, фрейма локальных переменных и верхнего элемента в стеке соответственно, а регистр PC (Program Counter — счетчик команд) содержит адрес байта, который нужно вызвать следующим из потока команд. Регистр MBR (Memory Buffer Register — буферный регистр памяти) — это 1-байтовый регистр, который содержит байты потока команд, поступающих из памяти для интерпретации. TOS и OPC — дополнительные регистры. Они будут описаны ниже.

В определенные моменты в каждом из этих регистров обязательно находится определенное значение. Однако каждый из них также может использоваться в качестве временного регистра в случае необходимости. В начале и конце каждой команды регистр TOS (Top Of Stack register — регистр вершины стека) содержит значение адреса памяти, на который указывает SP. Это значение избыточно, поскольку его всегда можно считать из памяти, но если хранить это значение в регистре, то обращение к памяти не требуется. Для некоторых команд использование регистра TOS, напротив, влечет за собой *больше* обращений к памяти. Например, команда POP отбрасывает верхнее слово стека и, следовательно, должна вызвать новое значение вершины стека из памяти и записать его в регистр TOS.

OPC — временный регистр. У него нет определенного заданного назначения. В нем, например, может храниться адрес кода операции для команды перехода, пока значение PC увеличивается, чтобы получить доступ к параметрам. Он также используется в качестве временного регистра в командах условного перехода.

Как и все интерпретаторы, микропрограмма, приведенная в табл. *А1,* включает в себя основной цикл, который вызывает, декодирует и выполняет команды интерпретируемой программы (в данном случае команды IJVM). Основной цикл начинается со строки Mainl, а именно с инварианта (утверждения), что в регистр PC уже загружен адрес ячейки памяти, в которой содержится код операции. Более того, этот код операции уже вызван из памяти в регистр MBR. Когда мы вернемся к этой ячейке, мы должны быть уверены, что значение PC уже обновлено и указывает на код следующей операции, а сам код операции уже вызван из памяти в MBR.

Такая последовательность действий имеет место в начале каждой команды, поэтому важно сделать ее как можно более короткой. Разрабатывая аппаратное и программное обеспечение микроархитектуры Mic-1, мы смогли сократить основной цикл до одной микрокоманды. Каждый раз, когда выполняется эта микрокоманда, код операции, которую нужно выполнить, уже находится в регистре MBR. Эта микрокоманда, во-первых, осуществляет переход к микрокоду, который выполняет данную операцию, а во-вторых, вызывает следующий после кода операции байт, который может быть либо операндом, либо кодом операции.

А теперь мы можем объяснить главную причину, почему в каждой микрокоманде явным образом указывается следующая микрокоманда и почему последовательность команд может и не соответствовать порядку их расположения в памяти. Все адреса управляющей памяти, соответствующие кодам операций, должны быть зарезервированы для первого слова интерпретатора соответствующей команды. Так, из табл. 4.2 мы видим, что программа, которая интерпретирует команду POP, начинается в ячейке 0x57, а программа, которая интерпретирует команду DUP, начинается в ячейке 0x59. (Как язык MAL узнает, что команду POP нужно поместить в ячейку 0x57, — одна из загадок Вселенной. Предположительно, где-то существует файл, который сообщает ему об этом.)

К сожалению, программа, интерпретирующая команду POP, включает в себя три микрокоманды, поэтому если их расположить в памяти последовательно, то эта программа смешается с началом команды DUP. Поскольку все адреса управляющей памяти, которые соответствуют кодам операций, зарезервированы, то все микрокоманды, идущие после первой микрокоманды в каждой последовательности, должны размещаться в промежутках между зарезервированными адресами. По этой

причине происходит очень много «скачков», и было бы нерационально каждый раз вставлять микрокоманду перехода, чтобы перепрыгнуть от одной последовательности адресов к другой.

Чтобы понять, как работает интерпретатор, предположим, что регистр MBR содержит значение 0x60, то есть код операции IADD (см. табл. 4.2). В основном цикле, который состоит из одной микрокоманды, выполняется следующее:

1. Значение регистра PC увеличивается, и теперь он содержит адрес первого байта после кода операции.
2. Начинается передача следующего байта в регистр MBR. Этот байт понадобится рано или поздно либо в качестве операнда текущей команды IJVM, либо в качестве кода следующей операции (как в случае с командой IADD, у которой нет операндов).
3. Совершается переход к адресу, который содержался в регистре MBR в начале цикла Main 1. Номер адреса равен значению кода операции, которая выполняется в данный момент. Этот адрес был помещен туда предыдущей микрокомандой. Отметим, что значение, которое вызывается из памяти во время этой микрокоманды, не играет никакой роли в межуровневом переходе.

Здесь начинается вызов следующего байта, поэтому он будет доступен уже к концу третьей микрокоманды. В этот момент он, может быть, и не нужен, но он в любом случае когда-нибудь понадобится, поэтому не будет никакого вреда в том, что вызов начнется именно здесь.

Таблица 4.4. Микропрограмма для Mic-1

Микрокоманда	Операции	Комментарий
Maini	PC=PC+1; fetch; goto(MBR)	MBR содержит код операции; получение следующего байта; отсылка
пор!	goto Maini	Ничего не происходит
iaddi	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
iadd2	H=TOS	H = вершина стека
iadd3	MDR=TOS=MDR+H;wr; goto Main 1	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека
isubi	MAR=SP=SP-1.rd	Чтение слова, идущего после верхнего слова стека
isub2	H=TOS	H = вершина стека
isub3	MDR=TOS=MDR-H;wr; goto Maini	Вычитание; запись результата в вершину стека
iandi	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
iand2	H=TOS	H = вершина стека
iand3	MDR=TOS=MDRHH,wr; goto Maini	Операция И; запись результата в вершину стека
ior1 стека	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова
ior2	H=TOS	H = вершина стека
ior3	M0P=TO5=МОРИЛИН; wr;gotoMaini	Операция ИЛИ; запись результата в вершину стека
du,oI	MAR=SP=SP+1	Увеличение SP на 1 и копирование результата в регистр MAR

Микрокоманда	Операции	Комментарий
dup2	MDR=TOS; wr; goto Maini	Запись нового слова в стек
pop1	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока считается из памяти новое значение регистра TOS
pop3	TOS^MDR; goto Maini	Копирование нового слова в регистр TOS
swapi	MAR=SP=SP-1;rd	Установка регистра MAR на значение SP-1; чтение второго слова из стека
swap2	MAR=SP	Установка регистра MAR на верхнее слово стека
swap3	**H=MDR;** wr	Сохранение значения TOS в регистре H; запись второго слова в вершину стека
swap4	MDR=TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR=SP-1;wr	Установка регистра MAR на значение SP-1; запись второго слова в стек
swap6	TOS-H; goto Maini	Обновление TOS
bipushi	SP=MAR=SP+1	MBR = байт, который нужно поместить в стек
bipush2	PC=PC+1; fetch	Увеличение PC на 1; вызов кода следующей операции
bipush3	MDR=TOS=MBR;wr; goto Maini	Добавление к байту дополнительного знакового разряда и запись значения в стек
iloadi	H=LV	MBR содержит индекс; копирование значения LVBH
iload2	MAR=MBRU+H;rd	MAR = адрес локальной переменной, которую нужно поместить в стек
iload3	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload4	PC=PC+1; fetch; wr	Увеличение значения PC на 1, вызов кода следующей операции, запись вершины стека
iload5	TOS=MDR;gotoMam1	Обновление TOS
istore1	H=LV	MBR содержит индекс; копирование значения LVBH
istore2	MAR=MBRU+H	MAR = адрес локальной переменной, в которую нужно сохранить слово из стека
istore3	MDR=TOS; wr	Копирование значения TOS в регистр **MDR;** запись слова
istore4	SP=MAR=SP-1;rd	Чтение из стека второго слова сверху
istore5	PC=PC+1; fetch	Увеличение PC на 1; вызов следующего кода операции
istore6	TOS=MDR; goto Main 1	Обновление TOS
widei	PC=PC+1;fetch, goto(MBR ИЛИ 0x100)	Межуровневый переход к старшим адресам
widejloadi	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
wide_iload2	H=MBRU«8	H = первый байт индекса, сдвинутый влево на 8 битов
wide_iload3	H=мвриилин	H = 16-битный индекс локальной переменной
wide_iload4	MAR=LV+H;rd; goto iload3	**MAR** = адрес локальной переменной, которую нужно записать в стек

*продолжение^*

Таблица 4.4 *(продолжение)*

Микрокоманда	Операции	Комментарий
widejstorei	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
wide_istore2	H=MBRU«8	H = первый байт индекса, сдвинутый влево на 8 битов
wide_istore3	H=MBRU ИЛИ H	H = 16-битный индекс локальной переменной
wide_istore4	MAR=LV+H; rd; goto istore3	MAR = адрес локальной переменной, в которую нужно записать слово из стека
ldc_w1	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
ldc_w2	H=MBRU«8	H = первый байт индекса, сдвинутый влево на 8 битов
ldc_w3	H^MBИИИЛИН	H = 16-битный индекс константы в наборе констант
ldc_w4	MAR=H+CPP; rd; goto iload3	MAR = адрес константы в наборе констант
iincl	H=LV	MBR содержит индекс; копирование значения LVBH
iinc2	MAR=MBRU+H;rd	Копирование суммы значения LV и индекса в регистр MAR; чтение переменной
iinc3	PC=PC+1; fetch	Вызов константы
iinc4	H=MDR	Копирование переменной в регистр H
iinc5	PC=PC+1; fetch	Вызов следующего кода операции
iinc6	MDR=MBR+H;wr, goto Maini	Запись суммы в регистр MDR; обновление переменной
gotoi	OPC=PC-1	Сохранение адреса кода операции
goto2	PC=PC+1; fetch	MBR = первый байт смещения; вызов второго байта
goto3	H=MBR«8	Сдвиг первого байта влево на 8 битов и сохранение его в регистре H
goto4	H=MBRU ИЛИ H	H =16-битное смещение перехода
goto5	PC=OPC+H; fetch	Суммирование смещения и OPC
goto6	goto Maini	Ожидание вызова следующего кода операции
iflt1	MAR=SP=SP-1;rd	Чтение второго сверху слова в стеке
iflt2	OPC=TOS	Временное сохранение TOS в OPC
iflt3	TOS=MDR	Запись новой вершины стека в TOS
mt4	N=OPC; if(N) goto T; else goto F	Переход в бит N
ifeqi	MAR=SP=SP~1;rd	Чтение второго сверху слова е стеке
ifeq2	OPC=TOS	Временное сохранение TOS в OPC
ifeq3	TOS=MDR	Запись новой вершины стека в TOS
ifeq4	ZOPC;if(Z)gotoT; else goto F	Переход в бит Z
ifjcmpeqi	MAR=SP=SP-1;rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR=SP=SP-1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H=MDR; rd	Копирование второго слова из стека в регистр H

Микрокоманда	Операции	Комментарий
if_icmpeq4	OPC=TOS	Временное сохранение TOS в OPC
if_icmpeq5	TOS=MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z=OPC-H, if(Z)gotoT, else goto F	Если два верхних слова равны, осуществляется переход к T, если они не равны, осуществляется переход к F
T	OPOPC-1; fetch; goto goto2	То же, что gotoi, нужно для адреса целевого объекта
F	PC=PC+1	Пропуск первого байта смещения
F2	PC=PC+1; fetch	PC указывает на следующий код операции
F3	goto Mam1	Ожидание вызова кода операции
invoke_virtual!	PC=PC+1, fetch	MBR = первый байт индекса; увеличение PC на 1, вызов второго байта
invoke_virtual	H=MBRU«8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
mvoke_virtual3	H=MBRU ИЛИ H	H = смещение указателя процедуры от регистра CPP
invoke_virtual4	MAR=CPP+H, rd	Вызов указателя процедуры из набора констант
mvoke_virtual5	OPC=PC+1	Временное сохранение значения PC в регистре OPC
invoke_virtual6	PC=MDR, fetch	Регистр PC указывает на новую процедуру, вызов числа параметров
mvoke_virtual7	PC=PC+1; fetch	Вызов второго байта числа параметров
mvoke_virtual8	H=MBRU«8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
invoke_virtual9	H=MBRU ИЛИ H	H = число параметров
invoke_virtual!0	PC=PC+1, fetch	Вызов первого байта размера области локальных переменных
invoke_virtual11	TOS=SP-H	TOS = адрес OBJREF-1
mvoke_virtual12	TOS=MAR=TOS+1	TOS = адрес OBJREF {новое значение LV)
invoke_virtual 13	PC=PC+1, fetch	Вызов второго байта размера области локальных переменных
mvoke_virtual14	H=MBRU<<8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
invoke_virtual15	H=MBRU ИЛИ H	H = размер области локальных переменных
mvoke_virtual16	MDR=SP+H+1;wr	Перезапись OBJREF со связующим указателем
invoke_virtual17	MAR=SP=MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invoke_virtual18	MDR=OPC, wr	Сохранение старого значения PC над локальными переменными
invoke_virtual 19	MAR=SP=SP+1	SP указывает на ячейку, в которой хранится старое значение LV
mvoke_virtual20	MDR=LV, wr	Сохранение старого значения LV над сохраненным значением PC
invoke_virtual21	PC=PC+1, fetch	Вызов первого кода операции новой процедуры
invoke_virtual22	LV=TOS, gotoMami	Установка значения LV на первый адрес фрейма локальных переменных
ireturni	MAR=SP=LV; rd	Переустановка регистров SP и MAR для вызова связующего указателя

продолжение!

Таблица 4.4 *{продолжение}*

Микрокоманда	Операции	Комментарий
ireturn2		Процесс считывания
ireturn3	LV=MAR=MDR; rd	Установка регистра LV на связующий указатель; вызов старого значения PC
ireturn4	MAR=LV+1	Установка регистра MAR на чтение старого значения LV
ireturn5	PC=MDR; rd; fetch	Восстановление PC; вызов следующего кода операции
ireturn6	MAR=SP	Установка MAR на запись TOS
ireturn7	LV=MDR	Восстановление LV
ireturn8	MDR=TOS; wr; goto Mainl	Сохранение результата в изначальной вершине стека

Если все разряды байта в регистре MBR равны 0 (это код операции для команды NOP), то следующей будет микрокоманда nopl, которая вызывается из ячейки 0. Поскольку эта команда не производит никаких операций, она просто совершает переход к началу основного цикла, где повторяется та же последовательность действий, но уже с новым кодом операции в MBR.

Еще раз подчеркнем, что микрокоманды, приведенные в табл. 4.4, не расположены в памяти последовательно и что микрокоманда Mainl находится вовсе не в ячейке с адресом 0 (поскольку в этой ячейке должна находиться микрокоманда nopl). Задача микроассемблера — поместить каждую команду в подходящую ячейку и связать их в короткие последовательности, используя поле NEXT_ADDRESS. Каждая последовательность начинается с адреса, который соответствует номерному значению кода операции (например, команда POP начинается с адреса 0x57), но остальные части последовательности могут находиться в любых ячейках управляющей памяти, и эти ячейки не обязательно идут подряд.

А теперь рассмотрим команду IADD. Она начинается с микрокоманды iaddl. Требуется выполнить следующие действия:

1. Значение регистра TOS уже есть, но из памяти нужно вызвать второе слово стека.
2. Значение регистра TOS нужно прибавить ко второму слову стека, вызванному из памяти.
3. Результат, который помещается в стек, должен быть сохранен в памяти и в регистре TOS.

Для того чтобы вызвать операнд из памяти, необходимо уменьшить значение указателя стека и записать его в регистр MAR. Отметим, что этот адрес будет использоваться для последующей записи. Более того, поскольку эта ячейка памяти будет новой вершиной стека, данное значение должно быть присвоено регистру SP. Следовательно, определить новое значение SP и MAR, уменьшить значение SP на 1 и записать его в оба регистра можно за одну операцию.

Все эти действия выполняются в первом цикле (iaddl). Здесь же инициируется операция чтения. Кроме того, регистр MPC получает значение из поля NEXT ADDRESS микрокоманды iaddl. Это адрес микрокоманды iadd2. Затем iadd2 счи-

тывается из управляющей памяти. Во втором цикле, пока происходит считывание операнда из памяти, мы копируем верхнее слово стека из TOS в H, где оно будет доступно для сложения, когда процесс считывания завершится

В начале третьего цикла (iadd3) MDR содержит второе слагаемое, вызванное из памяти. В этом цикле оно прибавляется к значению регистра H, а результат сохраняется обратно в регистры MDR и TOS. Кроме того, начинается операция записи, в процессе которой новое верхнее слово стека сохраняется в памяти. В этом цикле команда goto приписывает адрес Mainl регистру MPC, таким образом, мы возвращаемся к исходному пункту и можем начать выполнение следующей операции

Если следующий код операции, который содержится в данный момент в регистре MBR, равен 0x64 (ISUB), то повторяется практически таже последовательность действий После выполнения Mainl управление передается микрокоманде с адресом 0x64 (1 subl) За этой микрокомандой следуют i sub2, i sub3, а затем снова Mai nl Единственное различие между этой и предыдущей последовательностью состоит в том, что в цикле isub3 содержание регистра H не прибавляется к значению MDR, а вычитается из него

Команда IAND идентична командам IADD и ISUB, только в данном случае два верхних слова стека подвергаются логическому умножению (операция И), а не складываются и не вычитаются Нечто подобное происходит и во время выполнения команды I0R

Если код операции соответствует командам OUP, POP или SWAP, то нужно использовать стек Команда DUP дублирует верхнее слово стека Поскольку значение этого слова уже находится в регистре TOS, нужно просто увеличить SP на 1 Теперь регистр SP указывает на новый адрес В эту новую ячейку и записывается значение регистра TOS Команда POP тоже достаточно проста нужно только уменьшить значение SP на 1, чтобы отбросить верхнее слово стека Однако теперь необходимо считать новое верхнее слово стека из памяти и записать его в регистр TOS. Наконец, команда SWAP меняет местами значения двух ячеек памяти, а именно два верхних слова стека Регистр TOS уже содержит одно из этих значений, поэтому считывать его (значение) из памяти не нужно Подробнее мы обсудим эту команду немного позже.

Команда BIPUSH сложнее предыдущих, поскольку за кодом операции следует байт, как показано на рис 4 13 Этот байт представляет собой целое число со знаком Этот байт, который уже был передан в регистр MBR во время микрокоманды Mainl, нужно расширить до 32 битов (знаковое расширение) и скопировать его в регистр MDR. Наконец, значение SP увеличивается на 1 и копируется в MAR, что позволяет записать операнд на вершину стека. Этот операнд также должен копироваться в регистр TOS Отметим, что значение регистра PC должно увеличиваться на 1, чтобы в микрокоманде Mainl следующий код операции уже имелся в наличии.

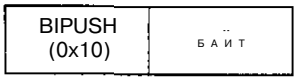


Рис. 4.13. Формат команды BIPUSH

Теперь рассмотрим команду ILOAD В этой команде за кодом операции также следует байт (рис. 4.14, *а),* но этот байт представляет собой индекс (без знака),

используемый для того, чтобы найти в пространстве локальных переменных слово, которое нужно поместить в стек. Поскольку здесь имеется всего 1 байт, можно различать только 2^8=256 слов, а именно первые 256 слов пространства локальных переменных. Для выполнения команды ILOAD требуется и процесс чтения (чтобы вызвать слово), и процесс записи (чтобы поместить его в стек). Чтобы определить адрес для считывания, нужно прибавить смещение, которое хранится в регистре MBR (буферном регистре памяти), к содержимому регистра LV. Доступ к регистрам MBR и LV можно получить только через шину B, поэтому сначала значение LV копируется в регистр H (в цикле iloadl), а затем прибавляется значение MBR. Результат суммирования копируется в регистр MAR, и начинается процесс чтения (в цикле iload2).

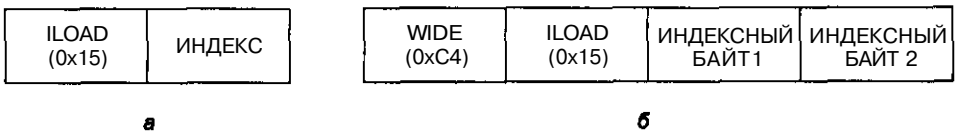


Рис. **4.14.** Команда ILOAD с однобайтным индексом (а); команда WIDE ILOAD с двубайтным индексом (б)

Однако здесь регистр MBR используется не совсем так, как в команде BIPUSH, где байт расширен по знаку. В случае с индексом смещение всегда положительно, поэтому байт смещения должен быть целым числом без знака (в отличие от BIPUSH, где байт представляет собой 8-битное целое число со знаком). Интерфейс между регистром MBR и шиной B разработан таким образом, чтобы обе операции были возможны. В случае с BIPUSH (где байт — 8-битное целое число со знаком) самый левый бит значения MBR копируется в 24 старших бита шины B. В случае с ILOAD (где байт — 8-битное целое число без знака) 24 старших бита шины B заполняются нулями. Два специальных сигнала помогают определить, какую из этих двух операций нужно выполнить (см. рис. 4.5). В микропрограмме слово MBR указывает на байт со знаком (как в команде bipush3), а MBRU — на байт без знака (как в команде iload2).

Пока ожидается поступление операнда из памяти (во время iload3), значение регистра SP увеличивается на 1 для записи новой вершины стека. Это значение также копируется в регистр MAR (это требуется для записи операнда в стек). Затем значение PC снова увеличивается на 1, чтобы вызвать следующий код операции (микрокоманда iload4). Наконец, значение MDR копируется в регистр TOS, чтобы показать новое верхнее слово стека (микрокоманда iload5).

Команда ISTORE противоположна команде ILOAD (из стека выталкивается верхнее слово и сохраняется в ячейке памяти, адрес которой равен сумме значения регистра LV и индекса данной команды). В данном случае используется такой же формат, как и в команде ILOAD (рис. 4.14, й), только здесь код операции не 0x15, а 0x36. Поскольку верхнее слово стека уже известно (оно находится в регистре TOS), его можно сразу сохранить в памяти. Однако новое верхнее слово стека все же необходимо вызвать из памяти, поэтому требуется и операция чтения, и операция записи, хотя их можно выполнять в любом порядке (или даже одновременно, если бы это было возможно).

Команды ILOAD и ISTORE имеют доступ только к первым 256 локальным переменным. Хотя для большинства программ этого пространства будет достаточно, все же нужно иметь возможность обращаться к любой локальной переменной, в какой бы части фрейма она не находилась. Чтобы обеспечить такую возможность, машина IJVM использует то же средство, что и JVM: специальный код операции WIDE (так называемый префиксный байт), за которым следует код операции ILOAD или ISTORE. Когда встречается такая последовательность, формат команды ILOAD или ISTORE меняется, и за кодом операции идет не 8-битный, а 16-битный индекс, как показано на рис. 4,14, *б*.

Команда WIDE декодируется обычным способом. Сначала происходит переход к микрокоманде widel, которая обрабатывает код операции WIDE. Хотя код операции, который нужно расширить, уже присутствует в регистре MBR, микрокоманда widel вызывает первый байт после кода операции, поскольку этого требует логика микропрограммы. Затем совершается еще один межуровневыи переход, но на этот раз для перехода используется байт, который следует за WIDE. Но поскольку команда WIDE ILOAD требует набора микрокоманд, отличного от ILOAD, а команда WIDE ISTORE требует набора микрокоманд, отличного от ISTORE, и т. д., при осуществлении межуровневого перехода нельзя использовать в качестве целевого адреса код операции.

Вместо этого микрокоманда wi del подвергает логическому сложению адрес Ox 100 и код операции, поместив его в регистр MPC. В результате интерпретация WIDE LOAD начинается с адреса 0x115 (а не 0x15), интерпретация WIDE ISTORE — с адреса 0x136 (а не 0x36) и т. д. Таким образом, каждый код операции WIDE начинается с адреса, который в управляющей памяти на 256 (то есть Ox 100) слов выше, чем соответствующий обычный код операции. Начальная последовательность микрокоманд для ILOAD и WIDE ILOAD показана на рис. 4.15.

Команда WIDE ILOAD отличается от обычной команды ILOAD только тем, что индекс в ней состоит из двух индексных байтов. Слияние и последующее суммирование этих байтов должно происходить по стадиям, при этом сначала первый индексный байт сдвигается влево на 8 битов и копируется в H. Поскольку индекс — целое число без знака, то здесь используется регистр MBRU (24 старших бита заполняются нулями). Затем прибавляется второй байт индекса (операция сложения идентична слиянию, поскольку младший байт регистра H в данный момент равен 0), при этом гарантируется, что между байтами не будет переноса. Результат снова сохраняется в регистре H. С этого момента происходят те же действия, что и в стандартной команде ILOAD. Вместо того чтобы дублировать последние команды ILOAD (от iload3 до iload5), мы просто совершили переход от wide_iload4 к iload3. Отметим, что во время выполнения этой команды значение PC должно увеличиваться на 1 дважды, чтобы в конце этот регистр указывал на следующий код операции. Команда ILOAD увеличивает значение один раз; последовательность команд WIDE_ILOAD также увеличивает это значение один раз.

Такая же ситуация имеет место при выполнении WIDEISTORE. После выполнения первых четырех микрокоманд (от wi de_i storel до wi de_i store4) последовательность действий та же, что и в команде ISTORE после первых двух микрокоманд, поэтому мы совершаем переход от wide_istore4 к istore3.

Далее мы рассмотрим команду LDC_W. Существует два отличия этой команды от ILOAD. Во-первых, она содержит 16-битное смещение без знака (как и расширенная версия ILOAD), а во-вторых, эта команда индексируется из регистра CPP, а не из

LV, поскольку она считывает значение из набора констант, а не из фрейма локальных переменных. (Существует еще и краткая форма этой команды — LDC, но мы не стали включать ее в машину IJVM, поскольку полная форма содержит в себе все варианты краткой формы, хотя при этом занимает 3 байта вместо 2.)

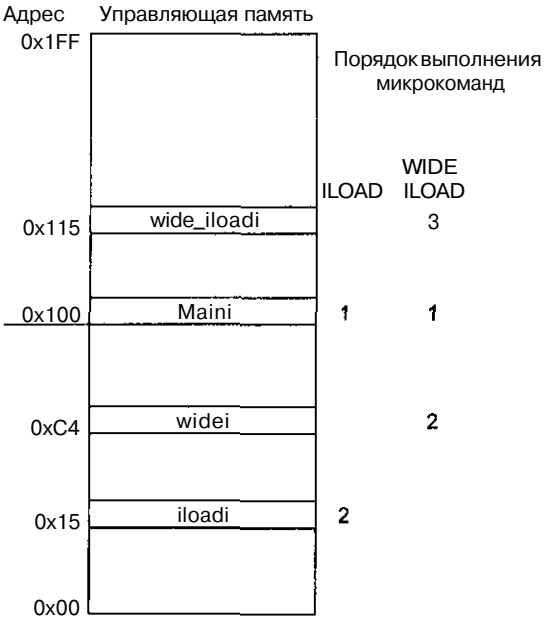


Рис. 4.15. Начало последовательности микрокоманд для ILOAD и WIDE ILOAD. Адреса приводятся в качестве примера

Команда IINC — единственная команда кроме ISTORE, которая может изменять локальную переменную. Она включает в себя два операнда по одному байту, как показано на рис. 4.16.

Рис. 4.16. Команда IINC содержит два поля операндов

Поле индекса нужно для того, чтобы определить смещение от начала фрейма локальных переменных. Команда считывает эту переменную, увеличивает ее на константу (константа содержится во втором поле) и помещает результат обратно в ту же ячейку памяти. Отметим, что константа является 8-битным числом со знаком в промежутке от -128 до +127. В машине JVM есть расширенная версия этой команды, в которой длина каждого операнда составляет два байта.

Рассмотрим первую команду перехода: GOTO. Эта команда изменяет значение регистра PC таким образом, чтобы следующая команда IJVM находилась в ячейке памяти с адресом, который вычисляется путем прибавления 16-битного смещения (со знаком) к адресу кода операции GOTO. Сложность здесь в том, что смещение связано с тем значением, которое содержится в регистре PC в начале декодирова-

ния команды, а не тем, которое содержится в том же регистре после вызова двух байтов смещения.

Чтобы лучше это понять, посмотрите на рис. 4.17, *а*. Здесь изображена ситуация, которая имеет место в начале цикла Mainl. Код операции уже находится в регистре MBR, но значение PC еще не увеличилось. На рис. 4.17, 6 мы видим ситуацию в начале цикла gotol. В данном случае значение PC уже увеличено на 1 и первый байт смещения уже передан в MBR. В следующей микрокоманде (рис. 4.17, *в)* старое значение PC, которое указывает на код операции, сохраняется в регистре OPC. Это значение нам нужно, поскольку именно от него, а не от текущего значения PC, зависит смещение команды GOTO. И именно для этого предназначен регистр OPC.

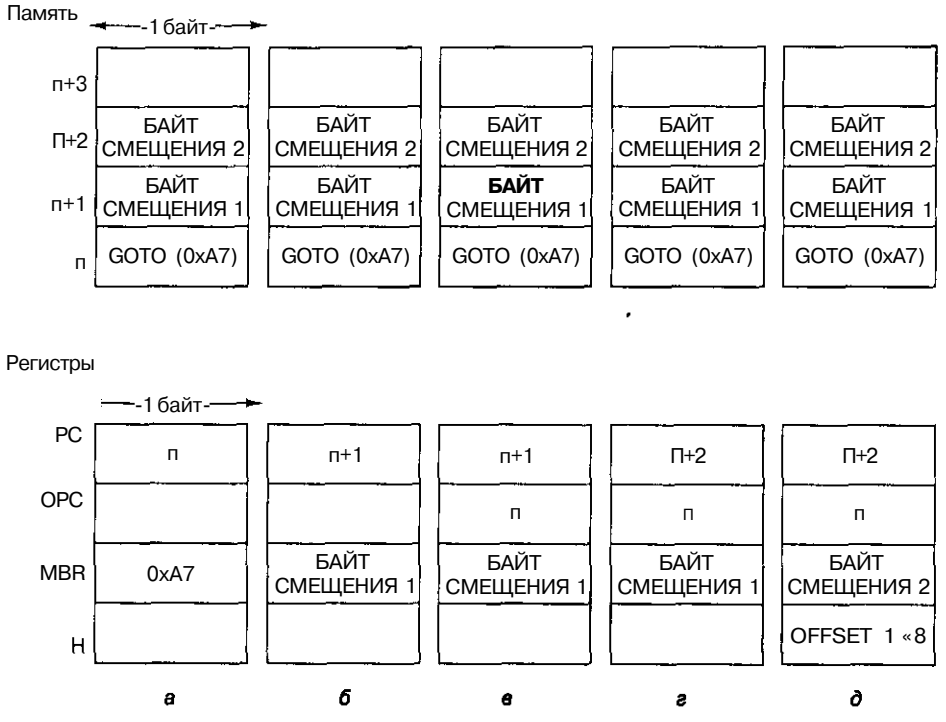


Рис. 4.17. Ситуация вначале выполнения различных микрокоманд- Mainl (а); gotol (б); goto2 (в); goto3 (г); goto4 (д)

Микрокоманда goto2 начинает вызов второго байта смещения, что приводит к ситуации, показанной на рис. 4.17, г (микрокоманда goto3). После того как первый байт смещения сдвигается влево на 8 битов и копируется в регистр H, мы переходим к микрокоманде goto4 (см. рис. 4.17, *д)*. Теперь у нас первый байт смещения, сдвинутый влево, находится в регистре H, второй байт смещения — в регистре MBR, а основа смещения — в регистре OPC. В микрокоманде goto5 путем прибавления полного 16-битного смещения к основе смещения мы получаем новый адрес, который помещается в регистр PC. Отметим, что в goto4 вместо MBR мы используем MBRU, поскольку нам не нужно знаковое расширение второго байта. 16-битное смещение строится путем логического сложения (операция ИЛИ) двух половинок. Нако-

нец, поскольку программа перед переходом к Mainl требует, чтобы в MBR был помещен следующий код операции, мы должны вызвать этот код. Последний цикл, goto6, нужен для того, чтобы вовремя помесить данные из памяти в регистр MBR.

Смещения, которые используются в команде goto, представляют собой 16-битные значения со знаком в промежутке от -32768 до +32767. Это значит, что переходы на более дальние расстояния невозможны. Это свойство можно рассматривать либо как дефект, либо как особенность машины IJVM (а также JVM). Те, кто считает это дефектом, скажут, что машина JVM не должна ограничивать возможности программирования. Те, кто считает это особенностью, скажут, что работа многих программистов продвинулась бы кардинальным образом, если бы им в ночных кошмарах снилось следующее сообщение компилятора:

```
Program is too big and hairy. You must rewrite it. Compilation aborted. (Программа
слишком длинная и сложная. Вы должны переписать ее. Компиляция прекращена.)
```

К сожалению (это наша точка зрения), это сообщение появляется только в том случае, если выражение el se или then превышает 32 Кбайт, что составляет, по крайней мере, 50 страниц текста на языке Java.

А теперь рассмотрим три команды условного перехода: IFLT, IFEQ и IFICMPEQ. Первые две выталкивают верхнее слово из стека и совершают переход в том случае, если это слово меньше 0 или равно 0 соответственно. Команда IFICMPEQ берет два верхних слова из стека и совершает переход, если они равны. Во всех трех случаях необходимо считывать новое верхнее слово стека и помещать его в регистр TOS.

Эти три команды сходны. Сначала операнд или операнды помещаются в регистры, затем в TOS записывается новое верхнее слово стека, и, наконец, происходит сравнение и осуществляется переход. Сначала рассмотрим IFLT. Слово, которое нужно проверить, уже находится в регистре TOS, но поскольку команда IFLT выталкивает слово из стека, нужно считать из памяти новую вершину стека и сохранить ее в регистре TOS. Процесс считывания начинается в микрокоманде iflt1. Во время iflt2 слово, которое нужно проверить, сохраняется в регистре OPC, поэтому новое значение можно сразу помесить в регистр TOS, и при этом предыдущее значение не пропадет. В цикле iflt3 новое верхнее слово стека, которое уже находится в MDR, копируется в регистр TOS. Наконец, в цикле i f114 проверяемое слово (в данный момент оно находится в регистре OPC) пропускается через АЛУ без сохранения результата, после чего проверяется бит N. Если после проверки условие подтверждается, микрокоманда осуществляет переход к T, а если не подтверждается — то к F.

Если условие подтверждается, то происходят те же действия, что и в начале команды GOTO, и далее осуществляется переход к goto2. Если условие не подтверждается, необходима короткая последовательность микрокоманд (F, F2 и F3), чтобы пропустить оставшуюся часть команды (смещение), возвратиться к Mainl и перейти к следующей команде.

Команда IFEQ аналогична команде IFLT, только вместо битаЫ используется бит Z. В обоих случаях ассемблер должен убедиться, что адреса микрокоманд F и T различаются только крайним левым битом.

Команда IF_ICMPEQ в целом сходна с командой IFLT, только здесь нужно считывать еще и второй операнд. Второй операнд сохраняется в регистре H во время цикла i f_i cmpeq3, где начинается чтение нового верхнего слова стека. Текущее верхнее слово стека сохраняется в OPC, а новое загружается в регистр TOS. Наконец, микрокоманда if_icmpeq6 аналогична ifeq4.

Теперь рассмотрим команды INVOKEVIRTUAL и IRETURN. Как было описано в разделе «Набор команд IJVM», они служат для вызова процедуры и выхода из нее. Команда INVOKEVIRTUAL представляет собой последовательность из 22 микрокоманд. Это самая сложная команда машины IJVM. Последовательность действий при выполнении этой команды показана на рис. 4.10. 16-битное смещение используется для того, чтобы определить адрес процедуры, которую нужно вызвать. Номер адреса процедуры находится в наборе констант. Следует помнить, что первые 4 байта каждой процедуры — не команды. Это два 16-битных указателя. Первый из них выдает число параметров (включая OBJREF — см. рис. 4.10), а второй — размер области локальных переменных (в словах). Эти поля вызываются через 8-битный порт и объединяются таким же образом, как 16-битное смещение в одной команде.

Затем требуется специальная информация для восстановления предыдущего состояния машины — адрес начала прежней области локальных переменных и старое значение регистра PC. Они сохранены непосредственно над областью локальных переменных под новым стеком. Наконец, вызывается следующий код операции, значение регистра PC увеличивается, происходит переход к циклу Mainl и начинается выполнение следующей команды.

IRETURN — простая команда без операндов. Эта команда просто обращается к первому слову области локальных переменных, чтобы извлечь информацию для возвращения к прежнему состоянию. Затем она восстанавливает предыдущие значения регистров SP, LV и PC и копирует результат выполнения процедуры из нового стека в предыдущий стек, как показано на рис. 4.11.

Разработка микроархитектурного уровня

При разработке микроархитектурного уровня (как и при разработке других уровней) постоянно приходится идти на компромисс. У компьютера есть много важных характеристик: скорость, цена, надежность, простота использования, количество потребляемой энергии, физические размеры. При разработке центрального процессора очень важную роль играет выбор между высокой скоростью и низкой стоимостью. В этом разделе мы подробно рассмотрим данную проблему, покажем преимущества и недостатки каждого из вариантов, а также узнаем, какой производительности можно достичь, какова при этом будет стоимость компьютера и насколько сложным будет аппаратное обеспечение.

Скорость и стоимость

С развитием технологий скорость работы компьютеров стремительно растет. Существует три основных подхода, которые позволяют увеличить скорость выполнения операций:

1. Сокращение количеств циклов, необходимых для выполнения команды.
2. Упрощение организации машины таким образом, чтобы можно было сделать цикл короче.
3. Выполнение нескольких операций одновременно.

Первые два подхода очевидны, но существует огромное количество различных вариантов разработки, которые могут очень сильно повлиять на число циклов, период или (что бывает чаще всего) и то и другое вместе. В этом разделе мы приведем пример того, как кодирование и декодирование операции могут подействовать на цикл.

Число циклов, необходимых для выполнения набора операций, называется **длиной пути.** Иногда длину пути можно уменьшить с помощью дополнительного аппаратного обеспечения. Например, если к регистру PC добавить инкрементор (по сути, это сумматор, у которого один из входов постоянно связан с единицей), то нам больше не придется использовать для этого АЛУ, и следовательно, количество циклов сократится. Однако такой подход не настолько эффективен, как хотелось бы. Часто в том же цикле, в котором значение PC увеличивается на 1, происходит еще и операция чтения, и следующая команда в любом случае не может начаться раньше, поскольку она зависит от данных, которые должны поступить из памяти.

Для сокращения числа циклов, необходимых для вызова команды, требуется нечто большее, чем простое добавление схемы, которая увеличивает PC на 1. Чтобы повысить скорость вызова команды, нужно применить третью технологию — параллельное выполнение команд. Весьма эффективно отделение схем для вызова команд (8-битного порта памяти и регистров PC и MBR), если этот блок сделать функционально независимым от основного тракта данных. Таким образом, он может сам вызывать следующий код операции или операнд. Возможно, он даже будет работать асинхронно относительно другой части процессора и вызывать одну или несколько команд заранее.

Один из наиболее трудоемких процессов при выполнении команд — вызов двубайтного смещения и сохранение его в регистре H для подготовки к сложению (например, при переходе к PC±n байтов). Одно из возможных решений — увеличить порт памяти до 16 битов, но это сильно усложняет операцию, поскольку требуемые 16 битов могут перекрывать границы слова, поэтому даже считывание из памяти 32 битов за один раз не обязательно приведет к вызову обоих нужных нам байтов.

Одновременное выполнение нескольких операций — самый продуктивный подход. Он дает возможность очень сильно повысить скорость работы компьютера. Даже простое перекрытие вызова и выполнения команды чрезвычайно эффективно. При более сложных технологиях допустимо одновременное выполнение нескольких команд. Вообще говоря, эта идея является основой проектов современных компьютеров. Ниже мы обсудим некоторые технические приемы, позволяющие воплотить этот подход.

На одной чаше весов находится скорость, на другой — стоимость. Стоимость можно измерять различными способами, но точное определение стоимости дать очень трудно. В те времена, когда процессоры конструировались из дискретных компонентов, достаточно было подсчитать общее число этих компонентов. В настоящее время процессор целиком помещается на одну микросхему, но большие и более сложные микросхемы стоят гораздо дороже, чем более простые микросхемы небольшого размера. Можно подсчитать отдельные компоненты (транзисторы, вентили, функциональные блоки), но обычно это число не так важно, как размер контактного участка, необходимого для интегральной схемы. Чем больше участок, тем

больше микросхема. И стоимость микросхемы растет гораздо быстрее, чем занимаемое ею пространство. По этой причине разработчики часто измеряют стоимость в единицах, применимых к «недвижимости», то есть с точки зрения пространства, которое требуется для микросхемы (предполагаем, что площадь поверхности измеряется в пикоакрах).

В истории компьютерной промышленности одной из наиболее тщательно проработанных микросхем является двоичный сумматор. Были реализованы тысячи проектов, и самые быстрые двоичные сумматоры очень сильно превышают по скорости самые медленные. Естественно, высокоскоростные сумматоры гораздо сложнее низкоскоростных. Специалистам по разработке систем приходится выбирать определенное соотношение скорости и занимаемого пространства.

Сумматор — не единственный компонент, допускающий различные варианты разработки. Практически любой компонент системы может быть спроектирован таким образом, что *он* будет функционировать с более высокой или с более низкой скоростью, при этом, естественно, стоимость разных моделей будет различаться. Главной задачей разработчика является определение тех компонентов системы, усовершенствование которых может максимально повлиять на скорость работы компьютера. Интересно отметить, что если какой-нибудь компонент заменить более быстрым, это не обязательно повлечет за собой повышение общей производительности. В следующих разделах мы рассмотрим некоторые вопросы разработки и возможные соотношения цены и скорости.

Одним из ключевых факторов в определении скорости работы генератора синхронизирующего сигнала является количество действий, которые должны быть сделаны за один цикл. Очевидно, чем больше действий должно быть сделано, тем длиннее цикл. Однако все не так просто, ведь аппаратное обеспечение способно выполнять некоторые операции параллельно, поэтому в действительности длина цикла зависит от количества *последовательных* операций в одном цикле.

Должен также учитываться объем выполняемого декодирования. Посмотрите на рис. 4.5. Вспомните, что в АЛУ может передаваться значение одного из девяти регистров, и чтобы определить, какой именно регистр нужно выбрать, требуется всего 4 бита в микрокоманде. К сожалению, такая экономия дорого обходится. Схема декодера вносит дополнительную задержку в работу компьютера. Это значит, что какой бы регистр мы не выбрали, он получит команду немного позже и передаст свое содержимое на шину B немного позже. Следовательно, АЛУ получает входные сигналы и выдает результат также с небольшой задержкой. Соответственно, этот результат поступает на шину C для записи в один из регистров тоже немного позже. Поскольку эта задержка часто является фактором, который определяет длину цикла, это значит, что генератор синхронизирующего сигнала не может функционировать с такой скоростью и весь компьютер должен работать немного медленнее. Таким образом, существует определенная зависимость между скоростью и ценой. Если сократить каждое слово управляющей памяти на 5 битов, это приведет к снижению скорости работы генератора. Инженер при разработке компьютера должен принимать во внимание его предназначение, чтобы сделать правильный выбор. В компьютере с высокой производительностью использовать декодер не рекомендуется, а вот для дешевой машины он вполне подойдет.

Сокращение длины пути

Микроархитектура Mic-1 имеет относительно простую структуру и работает довольно быстро, хотя эти две характеристики очень трудно совместить. В общем случае простые машины не являются высокоскоростными, а высокоскоростные машины довольно сложны. Процессор Mic-1 использует минимум аппаратного обеспечения; 10 регистров, простое АЛУ (см. рис. 3.18), продублированное 32 раза, декодер, схему сдвига, управляющую память и некоторые связующие элементы. Для построения всей системы требуется менее 5000 транзисторов, управляющая память (ПЗУ) и основная память (ОЗУ).

Мы уже показали, как можно воплотить IJVM с помощью микропрограммы, используя небольшое количество аппаратного обеспечения. Теперь рассмотрим альтернативные варианты. Сначала мы изучим, какими способами можно снизить количество микрокоманд в одной команде (то есть каким образом можно сократить длину пути), а затем перейдем к другим подходам.

Слияние цикла интерпретатора с микропрограммой

В микроархитектуре Mic-1 основной цикл состоит из микрокоманды, которая должна выполняться в начале каждой команды IJVM. В некоторых случаях возможно ее перекрытие предыдущей командой. В каком-то смысле эта идея уже получила свое воплощение. Вспомните, что во время цикла Mainl код следующей операции уже находится в регистре MBR. Этот код операции был вызван или во зремя предыдущего основного цикла (если у предыдущей команды не было операндов), или во время выполнения предыдущей команды.

Эту идею можно развивать и дальше. В некоторых случаях основной цикл можно свести к нулю. Это происходит следующим образом. Рассмотрим каждую последовательность микрокоманд, которая завершается переходом к Mainl. Каждый раз основной цикл может добавляться в конце этой последовательности (а не в начале следующей), при этом межуровневый переход дублируется много раз (но всегда с одним и тем же набором целевых объектов). В некоторых случаях микрокоманда Mic-1 может сливаться с предыдущими микрокомандами, поскольку эти команды не всегда полностью используются.

В табл. 4.5 приведена последовательность микрокоманд для команды POP. Основной цикл идет перед каждой командой и после каждой команды, в таблице этот цикл показан только после команды POP. Отметим, что выполнение этой команды занимает 4 цикла: три цикла специальных микрокоманд для команды POP и один основной цикл.

Таблица 4.5. Новая микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR=SP=SP-1; rd	Считывание второго сверху слова в стеке
pop2		Ожидание, пока из памяти считается новое значение TOS
pop3	TOS=MDR; goto Mainl	Копирование нового слова в регистр TOS
Mainl	PC=PC+1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта; переход

В табл. 4.6 последовательность сокращена до трех команд за счет того, что в цикле pop2 АЛУ не используется. Отметим, что в конце этой последовательности сразу осуществляется переход к коду следующей команды, поэтому требуется всего 3 цикла. Этот небольшой трюк позволяет сократить время выполнения следующей микрокоманды на один цикл, поэтому, например, последующая команда IADD сокращается с четырех циклов до трех. Это эквивалентно повышению частоты синхронизирующего сигнала с 250 МГц (каждая микрокоманда по 4 не) до 333 МГц (каждая микрокоманда по 3 не).

Таблица 4.6. Усовершенствованная микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR>SP=SP-1; rd	Считывание второго сверху слова *в* стеке
Maini pop	PC=PC+1; fetch	Регистр MBR содержит код операции, вызов следующего байта
pop3	TOS=MDR, goto(MBR)	Копирование нового слова в регистр TOS; переход к коду операции

Команда POP очень хорошо подходит для такой переработки, поскольку она содержит цикл, в котором АЛУ не используется, а основной цикл требует АЛУ. Таким образом, чтобы сократить длину команды на одну микрокоманду, нужно в этой команде найти цикл, где АЛУ не используется. Такие циклы встречаются нечасто, но все-таки встречаются, поэтому установка цикла Mai nl в конце каждой последовательности микрокоманд вполне целесообразна. Для этого требуется всего лишь небольшая управляющая память. Итак, мы получили первый способ сокращения длины пути:

помещение основного цикла в конце каждой последовательности микрокоманд.

Трехшинная архитектура

Что еще можно сделать, чтобы сократить длину пути? Можно подвести к АЛУ две полные входные шины, А и В, и следовательно, всего получится три шины. Все (или, по крайней мере, большинство регистров) должны иметь доступ к обеим входным шинам. Преимущество такой системы состоит в том, что есть возможность складывать любой регистр с любым другим регистром за один цикл. Чтобы увидеть, насколько продуктивен такой подход, рассмотрим реализацию команды ILOAD (табл. 4 7).

Таблица 4.7. Микропрограмма для выполнения команды ILOAD

Микрокоманда	Операции	Комментарий
iloadi	H=LV	MBR содержит индекс, копирование LV в H
iload2	MAR=MBRU+H, rd	MAR= адрес локальной переменной, которую нужно поместить в стек
iload3	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека, подготовка к записи
iload4	PC=PC+1; fetch; wr	Увеличение PC на 1, вызов следующего кода операции, запись вершины стека
iload5	TOS=MDR;gotoMain1	Обновление TOS
Mam1	PC=PC+1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта, переход

Мы видим, что в микрокоманде iloadl значение LV копируется в регистр H. Это нужно только для того, чтобы сложить H с MBRU в микрокоманде iload2. В разработке с двумя шинами нет возможности складывать два произвольных регистра, поэтому один из них сначала нужно копировать в регистр H. В трехшинной архитектуре мы можем сэкономить один цикл, как показано в табл. 4.8. Мы добавили основной цикл к команде ILOAD, но при этом длина пути не увеличилась и не уменьшилась. Однако дополнительная шина сокращает общее время выполнения команды с шести циклов до пяти циклов. Теперь мы знаем второй способ сокращения длины пути:

переходотдвухшинной к трехшиннопархитектуре.

Таблица 4.8. Микропрограмма для выполнения команды ILOAD при наличии трехшинной архитектуры

Микрокоманда	Операции	Комментарий
Iloadl	MAR=MBRU+LV; rd	MAR= адрес локальной переменной, которую нужно поместить в стек
Iload2	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека; подготовка к записи
Iloacl3	PC=PC+1; fetch; wr	Увеличение PC на 1; вызов следующего кода операции; запись вершины стека
!load4	TOS=MDR	Обновление TOS
IloadS	PC=PC+1; fetch; goto(MBR)	Регистр MBR уже содержит код операции; вызов индексного байта

Блоквыборкикоманд

Оба эти способа стоит использовать, но чтобы достичь существенного продвижения, требуется нечто более радикальное. Давайте вернемся чуть-чуть назад и рассмотрим обычные составные части любой команды: поле вызова и поле декодирования. Отметим, что в каждой команде могут происходить следующие операции:

1. Значение PC пропускается через АЛУ и увеличивается на 1.
2. PC используется для вызова следующего байта в потоке команд.
3. Операнды считываются из памяти.
4. Операнды записываются в память.
5. АЛУ выполняет вычисление, и результаты сохраняются в памяти.

Если команда содержит дополнительные поля (для операндов), каждое поле должно вызываться эксплицитно по одному байту. Поле можно использовать только после того, как эти байты будут объединены. При вызове и компоновке поля АЛУ должно для каждого байта увеличивать PC на единицу, а затем объединять получившийся индекс или смещение. Когда помимо выполнения основной работы команды приходится вызывать и объединять поля этой команды, АЛУ используется практически в каждом цикле.

Чтобы объединить основной цикл с какой-нибудь микрокомандой, нужно освободить АЛУ от некоторых таких задач. Для этого можно ввести второе АЛУ, хотя работа полного АЛУ в большинстве случаев не потребуется. Отметим, что

АЛУ часто применяется для копирования значения из одного регистра в другой. Эти циклы можно убрать, если ввести дополнительные тракты данных, которые не проходят через АЛУ. Полезно будет, например, создать тракт от TOS к MDR или от MDR к TOS, поскольку верхнее слово стека часто копируется из одного регистра в другой.

В микроархитектуре Mic-1 с АЛУ можно снять большую часть нагрузки, если создать независимый блок для вызова и обработки команд. Этот блок, который называется **блоком выборки команд,** может независимо от АЛУ увеличивать значение PC на 1 и вызывать байты из потока байтов до того, как они понадобятся. Этот блок содержит инкрементор, который по строению гораздо проще, чем полный сумматор. Разовьем эту идею. Блок выборки команд может также объединять 8-битные и 16-битные операнды, чтобы они могли использоваться сразу, как только они стали нужны. Это можно осуществить, по крайней мере, двумя способами:

1. Блок выборки команд может интерпретировать каждый код операции, определять, сколько дополнительных полей нужно вызвать, и собирать их в регистр, который будет использоваться основным операционным блоком.
2. Блок выборки команд может постоянно предоставлять следующие 8- или 16-битные куски информации независимо от того, имеет это смысл или нет. Тогда основной операционный блок может запрашивать любые данные, которые ему требуются.

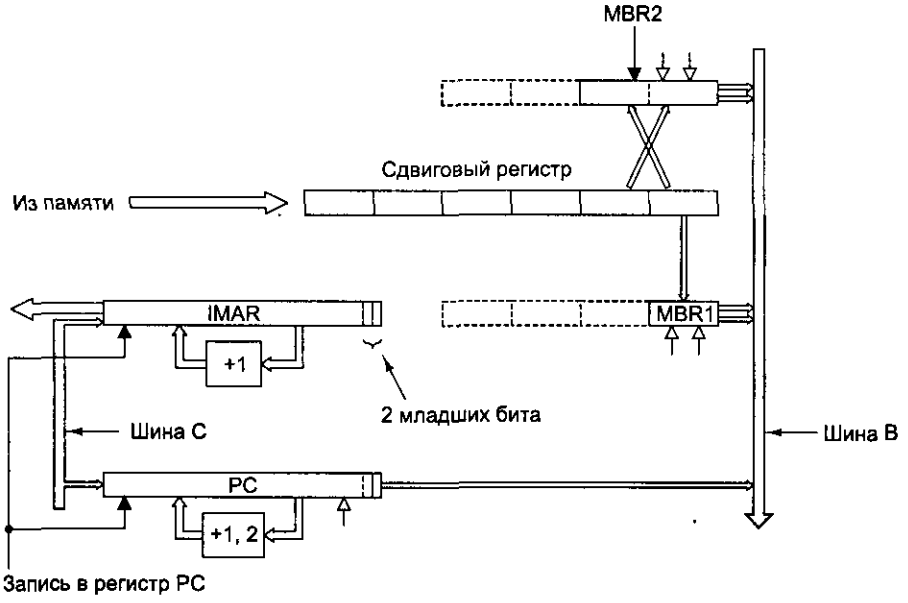


Рис. 4.18. Блок выборки команд для микроархитектуры Мю-1

На рис. 4.18 показан второй способ реализации. Вместо одного 8-разрядного регистра MBR (буферного регистра памяти) здесь есть два регистра MBR: 8-разрядный MBR1 и 16-разрядный MBR2. Блок выборки команд следит за самым последним байтом или байтами, которые поступили в основной операционный блок.

Кроме того, он передает следующий байт в регистр MBR, как и в архитектуре Mic-1, только в данном случае он автоматически определяет, когда значение регистра считано, вызывает следующий байт и сразу загружает его в регистр MBR1. Как и в микроархитектуре Mic-1, он имеет два интерфейса с шиной B: MBR1 и MBR1U. Первый получает знаковое расширение до 32 битов, второй дополнен нулями.

Регистр MBR2 функционирует точно так же, но содержит следующие 2 байта. Он имеет два интерфейса с шиной B: MBR2 и MBR2U, первый из которых расширен по знаку, а второй дополнен до 32 битов нулями.

Блок выборки команд отвечает за вызов байтов. Для этого он использует стандартный 4-байтный порт памяти, вызывая полные 4-байтные слова заранее и загружая следующие байты в сдвиговый регистр, который выдает их по одному или по два за раз в том порядке, в котором они вызываются из памяти. Задача сдвигового регистра — сохранить последовательность поступающих байтов для загрузки в регистры MBR1 и MBR2.

MBR1 всегда содержит самый старший байт сдвигового регистра, а MBR2 — 2 старших байта (старший байт — левый байт), которые формируют 16-битное целое число (см. рис. 4.14, *б)*. Два байта в регистре MBR2 могут быть из различных слов памяти, поскольку команды IJVM никак не связаны с границами слов.

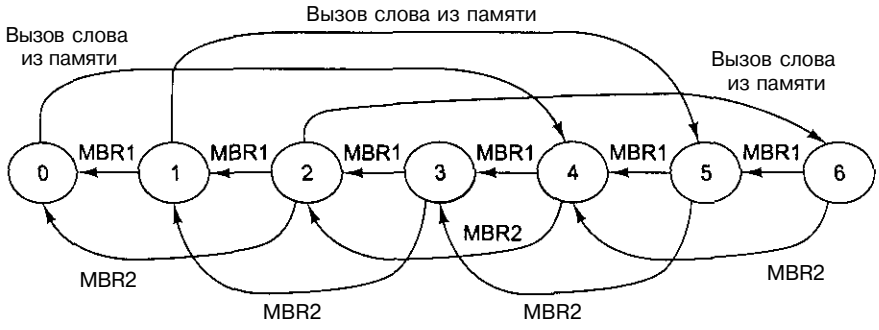
Всякий раз, когда считывается регистр MBR1, значение сдвигового регистра сдвигается вправо на 1 байт. Всякий раз, когда считывается регистр MBR2, значение сдвигового регистра сдвигается вправо на 2 байта. Затем в регистры MBR1 и MBR2 загружается самый старший байт и пара самых старших байтов соответственно. Если в данный момент в сдвиговом регистре осталось достаточно места для целого слова, блок выборки команд начинает цикл обращения к памяти, чтобы считать следующее слово. Мы предполагаем, что когда считывается любой из регистров MBR, он перезагружается к началу следующего цикла, поэтому новое значение можно считывать уже в последующих циклах.

Блок выборки команд может быть смоделирован в виде **автомата с конечным числом состояний** (или **конечного автомата),** как показано на рис. 4.19. Во всех конечных автоматах есть **состояния** (на рисунке это кружочки) и **переходы** (это дуги от одного состояния к другому). Каждое состояние — это одна из возможных ситуаций, в которой может находиться конечный автомат. Данный конечный автомат имеет семь состояний, которые соответствуют семи состояниям сдвигового регистра, показанного на рис. 4.18. Семь состояний соответствуют количеству байтов, которые находятся в данный момент в регистре (от 0 до 6 включительно).

Каждая дуга репрезентирует событие, которое может произойти. В нашем конечном автомате возможны три различных события. Первое событие — чтение одного байта из регистра MBR1. Оно активизирует сдвиговый регистр, самый правый байт в нем убирается, и осуществляется переход в другое состояние (меньшее на 1). Второе событие — чтение двух байтов из регистра MBR2. При этом осуществляется переход в состояние, меньшее на 2 (например, из состояния 2 в состояние 0 или из состояния 5 в состояние 3). Оба эти перехода вызывают перезагрузку регистров MBR1 и MBR2. Когда конечный автомат переходит в состояния 0,1 или 2, начинается процесс обращения к памяти, чтобы вызвать новое слово (предпо-

лагается, что память уже не занята считыванием слова). При поступлении слова номер состояния увеличивается на 4.

Чтобы функционировать правильно, схема выборки команд должна блокироваться в том случае, если от нее требуют произвести какие-то действия, которые она выполнить не может (например, передать значение в MBR2, когда в сдвиговом регистре находится только 1 байт, а память все еще занята вызовом нового слова). Кроме того, блок выборки команд не может выполнять несколько операций одновременно, поэтому вся поступающая информация должна передаваться последовательно. Наконец, всякий раз, когда изменяется значение PC, блок выборки команд должен обновляться. Все эти детали усложняют работу блока. Однако многие устройства аппаратного обеспечения конструируются в виде конечных автоматов.



События
MBR1: Чтение регистра MBR1
MBR2: Чтение регистра MBR2
Вызов слова из памяти: это событие означает считывание слова из памяти и помещение 4 байтов в сдвиговый регистр

Рис. 4.19. Автомат с конечным числом состояний для реализации блока выборки команд

Блок выборки команд имеет свой собственный регистр адреса ячейки памяти (IMAR), который используется для обращения к памяти, когда нужно вызвать новое слово. У этого регистра есть специальный инкрементор, поэтому основному АЛУ не требуется прибавлять 1 к значению PC для вызова следующего слова. Блок выборки команд должен контролировать шину C, чтобы каждый раз при загрузке регистра PC новое значение PC также копировалось в IMAR. Поскольку новое значение в регистре PC может быть и не на границе слова, блок выборки команд должен вызвать нужное слово и скорректировать значение сдвигового регистра соответствующим образом.

Основной операционный блок записывает значение в PC только в том случае, если необходимо изменить характер последовательности байтов. Это происходит в команде перехода, в команде INVOKEVIRTUAL и команде IRETURN.

Поскольку микропрограмма больше не увеличивает PC явным образом при вызове кода операции, блок выборки команд должен обновлять PC сам. Как это происходит? Блок выборки команд способен распознать, что байт из потока команд получен, то есть, что значения регистров MBR1 и MBR2 (или их вариантов

без знака) уже считаны. С регистром PC связан отдельный инкрементор, который увеличивает значение на 1 или на 2 в зависимости от того, сколько байтов получено. Таким образом, регистр PC всегда содержит адрес первого еще не полученного байта. В начале каждой команды в регистре MBR находится адрес кода операции этой команды.

Отметим, что существует два разных инкрементора, которые выполняют разные функции. Регистр PC считает *байты* и увеличивает значение на 1 или на 2. Регистр IMAR считает *слова* и увеличивает значение только на 1 (для 4 новых байтов). Как и MAR, регистр IMAR соединен с адресной шиной, при этом бит 0 регистра IMAR связан с адресной линией 2 и т. д. (перекос шины), чтобы осуществлять переход от адреса слова к адресу байта.

Мы скоро увидим, что если нет необходимости увеличивать значение PC в основном цикле, это приводит к большому выигрышу, поскольку обычно в микрокоманде, в которой происходит увеличение PC, кроме этого больше ничего не происходит. Если эту команду устранить, длина пути сократится. Однако для увеличения скорости работы машины требуется больше аппаратного обеспечения. Таким образом, мы пришли к третьему способу сокращения длины пути:

командыизпамятидолжнывызыватьсяспециализированнымфункциональным блоком.

Микроархитектура с упреждающей выборкой команд из памяти: Mic-2

Блок выборки команд может сильно сократить длину пути средней команды. Во-первых, он полностью устраняет основной цикл, поскольку в конце каждой команды просто стразу осуществляется переход к следующей команде. Во-вторых, АЛУ не нужно увеличивать значение PC. В-третьих, блок выборки команд сокращает длину пути всякий раз, когда вычисляется 16-битный индекс или смещение, поскольку он объединяет 16-битное значение и сразу передает его в АЛУ в виде 32-битного значения, избегая необходимости производить объединение в регистре H. На рис. 4.20 показана микроархитектура Mic-2, которая представляет собой усовершенствованную версию Mic-1, к которой добавлен блок выборки команд, изображенный на рис. 4.18. Микропрограмма для усовершенствованной машины приведена в табл. 4.9.

Чтобы продемонстрировать, как работает Mic-2, рассмотрим команду IADD. Она берет второе слово из стека и выполняет сложение, как и раньше, но только сейчас ей не нужно осуществлять переход к Mainl после завершения операции, чтобы увеличить значение PC и перейти к следующей микрокоманде. Когда блок выборки команд распознает, что в цикле iadd3 произошло обращение к регистру MBR1, его внутренний сдвиговый регистр сдвигает все вправо и перезагружает MBR1 и MBR2. Он также осуществляет переход в состояние, которое на 1 ниже текущего. Если новое состояние — это состояние 2, блок выборки команд начинает вызов слова из памяти. Все это происходит в аппаратном обеспечении. Микропрограмма ничего не должна делать. Именно поэтому команду IADD можно сократить с пяти до трех микрокоманд.

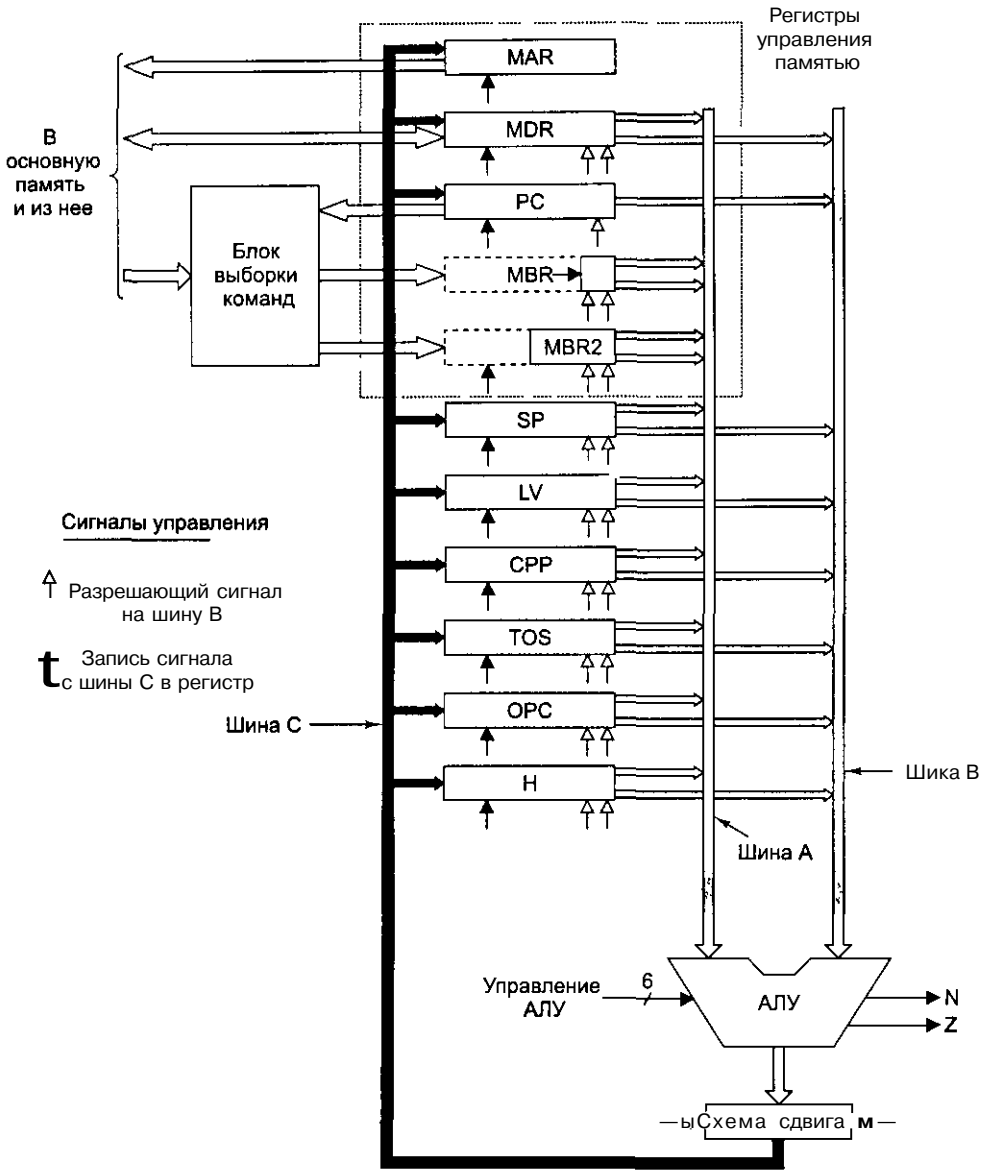


Рис. 4.20. Тракт данных для Мю-2

Таблица 4.9. Микропрограмма для Mic-2

Микрокоманда	Операции	Комментарий
пор1	goto (MBR)	Переход к следующей команде
iadd1	MAR=SP=SP-1; rd	Чтение слова, идущего после верхнего слова стека
iadd2	H=TOS	H = вершина стека

продолжение &

Таблица 4.9 *{продолжение}*

Микрокоманда	Операции	Комментарий
iadd3	MDR=TOS=MDR+H; wr;goto(MBR1)	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека
isubi	MAR=SP=SP-I;rd	Чтение слова, идущего после верхнего слова стека
isub2	H=TOS	H = вершина стека
isub3	MDR=TOS=MDR-H; wr; goto(MBRI)	Вычитание TOS из вызванного значения TOS-1
iandi	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
land2	H=TOS	H = вершина стека
iand3	MDR=TOS=MDRHH; wr;goto{MBR1)	Логическое умножение вызванного значения TOS-1 и TOS (операция И)
ior1	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
ior2	H=TOS	H = вершина стека
ior3	MDR=TOS=MDRHnM H;wr;goto(MBR1)	Логическое сложение вызванного значения TOS-1 и TOS (операция ИЛИ)
dup1	MAR=SP=SP+1	Увеличение SP на 1 и копирование результата в регистр MAR
dup2	MDR=TOS; wr; goto (MBR1)	Запись нового слова в стек
pop1	MAR=SP=SP-1;rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока закончится процесс чтения
pop3	TOS=MDR;goto(MBR1)	Копирование нового слова в регистр TOS
swapi	MAR=SP-1;rd	Чтение второго слова из стека; установка регистра MAR на значение SP
swap2	MAR=SP	Подготовка к записи нового второго слова стека
swap3	H=MDR; wr	Сохранение нового значения TOS; запись второго слова стека
swap4	MDR=TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR=SP-1;wr	Запись прежнего значения TOS на второе место в стеке
swap6	TOS=H;goto(MBR1)	Обновление TOS
bipush!	SP=MAR=SP+1	Установка регистра MAR для записи в новую вершину стека
bipush2	MDR=TOS=MBR1; wr;goto(MBR1)	Обновление стека в регистре TOS и памяти
iloadi	MAR=LV+MBR1U;rd	Перемещение значения LV с индексом в регистр MAR; чтение операнда
iload2	MAR=SP=SP+1	Увеличение SP на 1; перемещение нового значения SP в регистр MAR
iload3	TOS=MDR;wr; goto{MBR1)	Обновление стека в регистре TOS и памяти
istore1	MAR=LV+MBR1U	Установка регистра MAR на значение LV+индекс
istore2	MDR=TOS;wr	Копирование значения TOS для сохранения в памяти
istore3	MAR=SP=SP-1;rd	Уменьшение SP на 1; чтение нового значения TOS
istore4		Машина ждет окончания процесса чтения
istore5	TOS=MDR;goto(MBR1I,	Обновление TOS

Микрокоманда	Операции	Комментарий
widei	goto{MBR1 ИЛИ 0x100)	Следующий адрес —0x100 ИЛИ код операции
widejloadi	MAR=LV+MBR2U, rd, goto iload2	То же, что iloadi, но с использованием 2-байтного индекса
widejstorei	MAR=LV+MBR2U, goto istore2	То же, что istorei, но с использованием 2-байтного индекса
tdc_w1	MAR=CPP+MBR2U, rd, goto iload2	То же, что widejload 1, но индексирование осуществляется из регистра CPP
iind	MAR=LV+MBR1U,rd	Установка регистра MAR на значение LV+индекс, чтение этого значения
nnc2	H=MBR1	Присваивание регистру H константы
нпс3	MDR=MDR+H,wr, goto(MBRI)	Увеличение на константу и обновление
gotoi	H=PC-1	Копирование значения PC в регистр H
goto2	PC=H+MBR2	Прибавление смещения и обновление PC
goto3		Машина ждет, пока блок выборки команд вызовет новый код операции
goto4	goto(MBRI)	Переход к следующей команде
rflti	MAR=SP=SP-1,rd	Чтение второго слерху слова в стеке
rflt2	OPC=TOS	Временное сохранение TOS в OPC
iflt3	TOS=MDR	Запись новой вершины стека в TOS
iflt4	N=OPC,if(N)gotoT, else goto F	Переход в бит N
ifeqi	MAR=SP=SP-1,rd	Чтение второго сверху слова в стеке
ifeq2	OPC=TOS	Временное сохранение TOS в OPC
ifeq3	TOS=MDR	Запись новой вершины стека в TOS
ifeq4	Z=OPC, if{Z)gotoT; else goto F	Переход в бит Z
ifjcmpeql	MAR=SP=SP-1,rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR=SP=SP-1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H=MDR, rd	Копирование второго слова из стека в регистр H
if_icmpeq4	OPC=TOS	Временное сохранение TOS в OPC
if_icmpeq5	TOS=MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z=H-OPC,if(Z)gotoT, else goto F	Если два верхних слова равны, осуществляется переход к T, если они не равны, осуществляется переход к F
T	H=PC-1,gotogoto2	Тоже, чтодоШ!
F	H=MBR2	Игнорирование байтов, находящихся в регистре MBR2
F2	goto(MBRI)	
invoke_virtuaM	MAR=CPP+MBR2U, rd	Помещение адреса указателя процедуры в регистр MAR
invoke_virtua!2	OPC=PC	Сохранение значения PC в регистре OPC
invoke_virtual3	PC=MDR	Установка PC на первый байт кода процедуры
invoke_virtual4	TOS=SP-MBR2U	TOS = адрес OBJREF-1
invoke_virtual5	TOS=MAR=H=TOS+1	TOS = адрес OBJREF

*продочжение^*

Таблица 4.9 *(продолжение)*

Микрокоманда	Операции	Комментарий
invoke_virtual6	MDR=SP+MBR2U+1;wr	Перезапись OBJREF со связующим указателем
invoke_virtual7	MAR=SP=MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invoke_virtual8	MDR-OPC; wr	Подготовка к сохранению старого значения PC
invoke_virtua!9	MAR=SP=SP+1	Увеличение SP на 1; теперь SP указывает на ячейку, в которой хранится старое значение LV
invoke_virtual10	MDR=LV; wr	Сохранение старого значения LV
invoke_virtual11	LV=TOS; goto (MBR1)	Установка значения LV на нулевой параметр
ireturni	MAR=SP=LV;rd	Переустановка регистров SP и MAR для чтения связующего указателя
ireturn2		Процесс считывания связующего указателя
ireturn3	LV=MAR=MDR; rd	Установка регистров LV и MAR на связующий указатель; чтение старого значения PC
ireturn4	MAR=LV+1	Установка регистра MAR на старое значение LV; чтение старого значения LV
ireturn5	PC-MDR; rd	Восстановление PC
ireturn6	MAR=SP	
ireturn7	LV=MDR	Восстановление LV
ireturn8	MDR-TOS; wr; goto(MBRI)	Сохранение результата в изначальной вершине стека

Микроархитектура Mic-2 совершенствует некоторые команды в большей степени, чем другие. Команда LDCW сокращается с 9 до 3 микрокоманд, и следовательно, время выполнение команды уменьшается в три раза. Несколько по-другому дело обстоит с командой SWAP: изначально там было 8 команд, а стало 6. Для общей производительности компьютера играет роль сокращение наиболее часто повторяющихся команд. Это команды ILOAD (было 6, сейчас 3), IADD (было 4, сейчас 3) и IFJCMPEQ (было 13, сейчас 10 для случая, если слова равны; было 10, сейчас 8 для случая, если слова не равны). Чтобы вычислить, насколько улучшилась производительность, можно проверить эффективность системы по эталонному тесту, но и без этого ясно, что здесь наблюдается огромный выигрыш в скорости.

Конвейерная архитектура: Mic-3

Ясно, что Mic-2 — это усовершенствованная микроархитектура Mic-1. Она работает быстрее и использует меньше управляющей памяти, хотя стоимость блока выборки команд, несомненно, превышает ту сумму, которая выигрывается за счет сокращения пространства при уменьшении управляющей памяти. Таким образом, машина Mic-2 работает значительно быстрее при минимальном росте стоимости. Давайте посмотрим, можно ли еще больше повысить скорость.

А что, если попробовать уменьшить время цикла? В значительной степени время цикла определяется базовой технологией. Чем меньше транзисторы и чем меньше физическое расстояние между ними, тем быстрее может работать задающий генератор. Б технологии, которую мы рассматриваем, время, затрачиваемое на

прохождение через тракт данных, фиксировано (по крайней мере, с нашей точки зрения). Тем не менее у нас есть некоторая свобода действий, и далее мы используем ее в полной мере.

Еще один вариант усовершенствования — ввести в машину больше параллелизма. В данный момент микроархитектура Mic-2 выполняет большинство операций последовательно. Она помещает значения регистров на шины, ждет, пока АЛУ и схема сдвига обработают их, а затем записывает результаты обратно в регистры. Если не учитывать работу блока выборки команд, никакого параллелизма здесь нет. Внедрение дополнительных средств параллельной обработки дает нам большие возможности.

Как мы уже говорили, длительность цикла определяется временем, необходимым для прохождения сигнала через тракт данных. На рис. 4.2 показано распределение этой задержки между различными компонентами во время каждого цикла. В цикле тракта данных есть три основных компонента:

1. Время, которое требуется на передачу значений выбранных регистров в шины A и B.
2. Время, которое требуется на работу АЛУ и схемы сдвига.
3. Время, которое требуется на передачу полученных значений обратно в регистры и сохранение этих значений.

На рис. 4.21 показана новая трехшинная архитектура с блоком выборки команд и тремя дополнительными защелками (регистрами), каждая из которых расположена в середине каждой шины. Эти регистры записываются в каждом цикле. Они делят тракт данных на отдельные части, которые теперь могут функционировать независимо друг от друга. Мы будем называть такую архитектуру **конвейерной моделью** или **Mic-3.**

Зачем нужны эти три дополнительных регистра? Ведь теперь для прохождения сигнала через тракт данных требуется три цикла: один — для загрузки регистров A и B, второй — для запуска АЛУ и схемы сдвига и загрузки регистра C, и третий — для сохранения значения регистра-защелки C обратно в нужные регистры. Мы что, сумасшедшие? *(Подсказка:* нет). Существует целых две причины введения дополнительных регистров:

1. Мы можем повысить тактовую частоту, поскольку максимальная задержка теперь стала короче.
2. Во время каждого цикла мы можем использовать все части тракта данных.

После разбиения тракта данных на три части максимальная задержка прохождения сигнала уменьшается, и в результате тактовая частота может повыситься. Предположим, что если разбить цикл тракта данных на три примерно равных интервала, тактовая частота увеличится втрое. (На самом деле это не так, поскольку мы добавили в тракт данных еще два регистра, но в качестве первого приближения это допустимо.)

Поскольку мы предполагаем, что все операции чтения из памяти и записи в память выполняются с использованием кэш-памяти первого уровня и эта кэш-память построена из того же материала, что и регистры, то следовательно, операция с памятью занимает один цикл. На практике, однако, этого не так легко достичь.

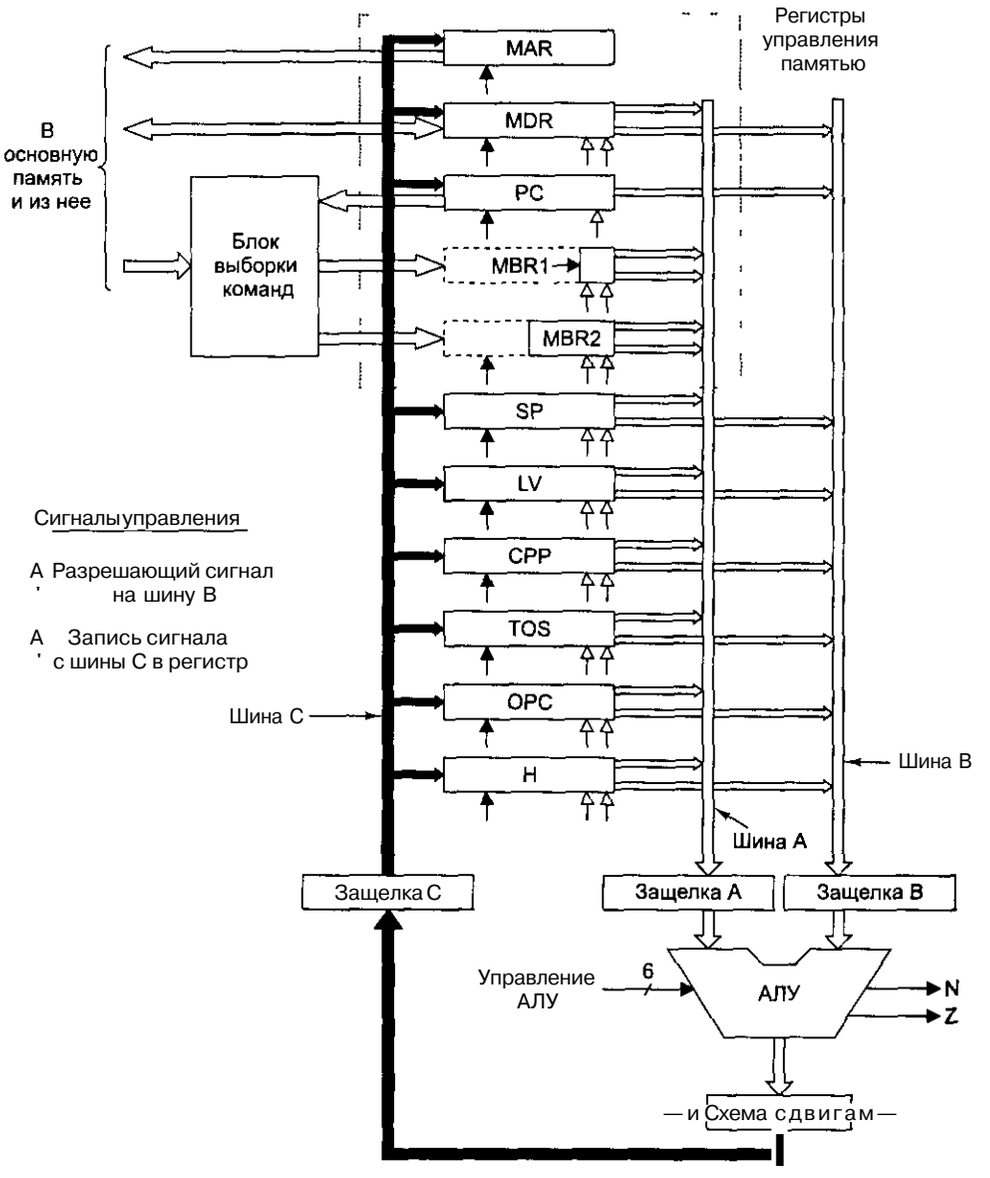


Рис. 4.21, Тракт данных с тремя шинами в микроархитектуре Mic-3

Второй пункт связан с общей производительностью, а не со скоростью выполнения отдельной команды В микроархитектуре Mic-2 во время первой и третьей части каждого цикла АЛУ простаивает Если разделить тракт данных на три части, то появляется возможность использовать АЛУ в каждом цикле, вследствие чего производительность машины увеличивается в три раза

А теперь посмотрим, как работает тракт данных Mic-3. Перед тем как начать, нужно ввести определенные обозначения для защелок Проще всего назвать за-

щелки A, B и C и считать их регистрами, подразумевая ограничения тракта данных. В таблице 4.10 приведен кусок программы для микроархитектуры Mic-2 (реализация команды SWAP).

Таблица 4.10. Программа Mic-2 для команды SWAP

Микрокоманда	Операции	Комментарий
swapi	MAR=SP-1;rd	Чтение второго слова из стека; установка MAR на SP
swap2	MAR=SP	Подготовка к записи нового второго слова
swap3	H=MDR;wr	Сохранение нового значения TOS; запись второго слова в стек
swap4	MDR=TOS	Копирование старого значения TOS в регистр MDR
swap5	MAR=SP-1;wr	Запись старого значения TOS на второе место в стеке
swap6	TOS=H;goto(IVIBR1)	Обновление TOS

Давайте перепишем эту последовательность для Mic-3. Следует помнить, что теперь работа тракта данных занимает три цикла: один — для загрузки регистров A и B, второй — для выполнения операции и загрузки регистра C и третий — для записи результатов в регистры. Каждый из этих участков будем называть **микрошагом.**

Реализация команды SWAP для Mic-3 показана в табл. 4.11. В цикле 1 мы начинаем микрокоманду swapl, копируя значение SP в регистр B. Не имеет никакого значения, что происходит в регистре A, поскольку чтобы отнять 1 из B, ENA (сигнал разрешения A) блокируется (см. табл. 4.1). Для простоты мы не показываем присваивания, которые не используются. В цикле 2 мы производим операцию вычитания. В цикле 3 результат сохраняется в регистре MAR, и после этого, в конце третьего цикла, начинается процесс чтения. Поскольку чтение из памяти занимает один цикл, закончится он только в конце четвертого цикла. Это показано присваиванием значения регистру MDR в цикле 4. Значение из MDR можно считывать не раньше пятого цикла.

Таблица 4.11. Реализация команды SWAP для Mic-3

	Swapl	Swap2	Swap3	Swap4	Swap5	Swap6
Цикл	MAR=SP-1;rd	MAR=SP	H=MDR,wr	MDR-TOS	MAR-SP-1 ;wr	TOS=H;goto(MBR1)
1	B=SP					
2	C=B-1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B-1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS-C
11						goto(MBRI)

А теперь вернемся к циклу 2. Мы можем разбить микрокоманду swap2 на микрошаги и начать их выполнение. В цикле 2 мы копируем значение SP в регистр B, затем пропускаем значение через АЛУ в цикле 3 и, наконец, сохраняем его в регистре MAR в цикле 4. Пока все хорошо. Должно быть ясно, что если мы сможем начинать новую микрокоманду в каждом цикле, скорость работы машины увеличится в три раза. Такое повышение скорости происходит за счет того, что машина Mic-3 производит в три раза больше циклов в секунду, чем Mic-2. Фактически мы построили конвейерный процессор.

К сожалению, мы наткнулись на преграду в цикле 3. Мы бы рады начать микрокоманду swap3, но эта микрокоманда сначала пропускает значение MDR через АЛУ, а значение MDR не будет получено из памяти до начала цикла 5. Ситуация, когда следующий микрошаг не может начаться, потому что перед этим нужно получить результат выполнения предыдущего микрошага, называется **реальной взаимозависимостью** или **RAW-взаимозависимостью (Read After Write — чтение после записи).** В такой ситуации требуется считать значение регистра, которое еще не записано. Единственное разумное решение в данном случае — отложить начало микрокоманды swap3 до того момента, когда значение MDR станет доступным, то есть до пятого цикла. Ожидание нужного значения называется **простаиванием.** После этого мы можем начинать выполнение микрокоманд в каждом цикле, поскольку таких ситуаций больше не возникает, хотя имеется пограничная ситуация: микрокоманда swap6 считывает значение регистра H в цикле, который следует сразу после записи этого регистра в микрокоманде swap3. Если бы значение этого регистра считывалось в микрокоманде swap5, машине пришлось бы простаивать один цикл.

Хотя программа Mic-3 занимает больше циклов, чем программа Mic-2, она работает гораздо быстрее. Если время цикла микроархитектуры Mic-3 составляет ДТ наносекунд, то для выполнения команды SWAP машине Mic-3 требуется 11ДТ не, а машине Mic-2 нужно 6 циклов по ЗДТ не каждый, то есть всего 18ДТ не. Конвейеризация увеличивает скорость работы компьютера, даже несмотря на то, что один раз приходится простаивать из-за явления взаимозависимости.

Конвейеризация является ключевой технологией во всех современных процессорах, поэтому важно хорошо понимать эту технологию. На рис. 4.22 графически проиллюстрирована конвейеризация тракта данных, изображенного на рис. 4.21. В первой колонке демонстрируется, что происходит во время цикла 1, вторая колонка представляет цикл 2 и т. д. (предполагается, что простаиваний нет). Закрашенная область на рисунке для цикла 1 и команды 1 означает, что блок выборки команд занят вызовом команды 1. В цикле 2 значения регистров, вызванных командой 1, загружаются в A и B, а в это время блок выборки команд занимается вызовом команды 2. Все это также показано с помощью закрашенных серым прямоугольников.

Во время цикла 3 команда 1 использует АЛУ **и** схему сдвига, регистры A **и** B загружаются для команды 2, а команда 3 вызывается. Наконец, во время цикла 4 работают все 4 команды одновременно. Сохраняются результаты выполнения команды 1, АЛУ выполняет вычисления для команды 2, регистры A и B загружаются для команды 3, а команда 4 вызывается.

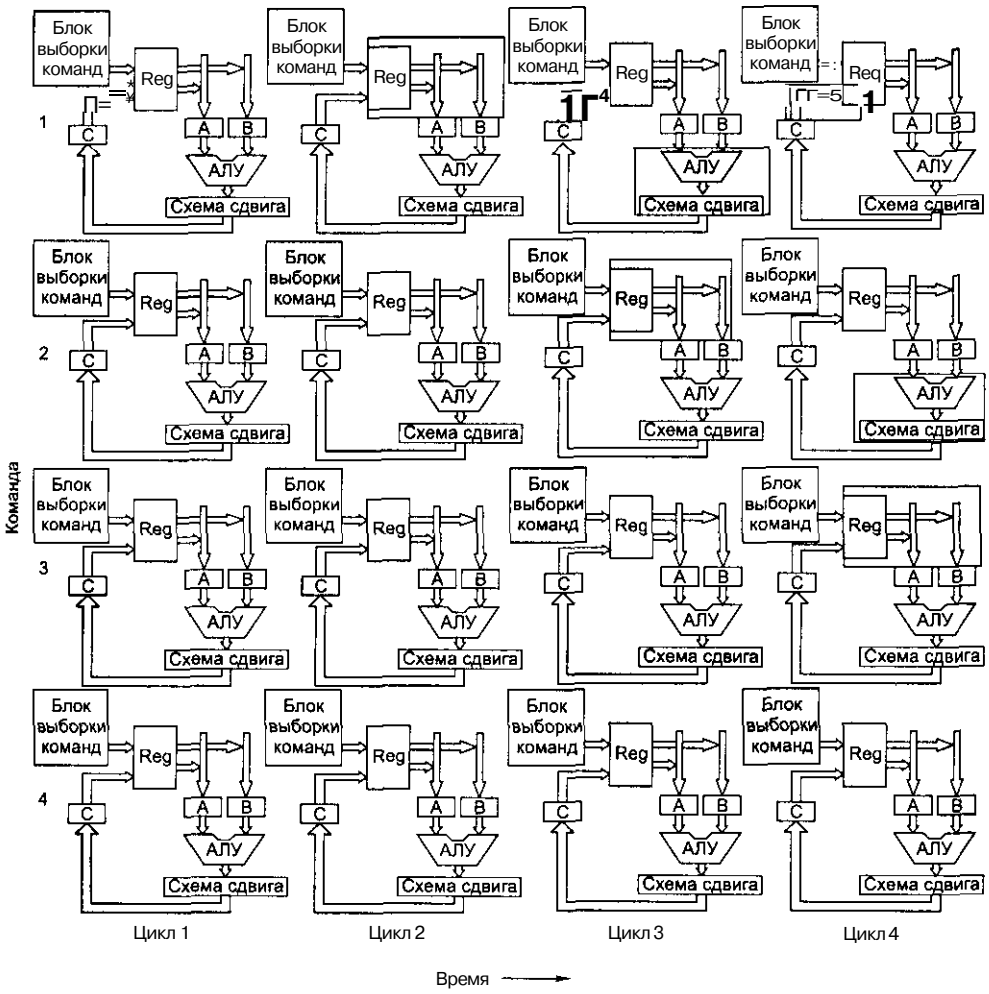


Рис. 4.22. Графическое изображение работы конвейера

Если бы мы показали цикл 5 и следующие, модель была бы точно такой же, как в цикле 4: все четыре части тракта данных работали бы независимо друг от друга. Данный конвейер содержит 4 стадии: для вызова команд, для доступа к операндам, для работы АЛУ и для записи результата обратно в регистры. Он похож на конвейер, изображенный на рис. 2.3, *а,* только у него отсутствует стадия декодирования (расшифровки). Здесь важно подчеркнуть, что хотя выполнение одной команды занимает 4 цикла, в каждом цикле начинается новая команда и завершается предыдущая.

Можно рассматривать схему на рис. 4.22 не вертикально (по колонкам), а горизонтально (по строчкам). При выполнении команды 1 в цикле 1 функционирует блок выборки команд. В цикле 2 значения регистров помещаются на шины A и B. В цикле три происходит работа АЛУ и схемы сдвига. Наконец, в цикле 4 полученные результаты сохраняются в регистрах. Отметим, что имеется 4 доступные части

аппаратного обеспечения, и во время каждого цикла определенная команда использует только одну из них, оставляя свободными другие части для других команд.

Проведем аналогию с конвейером на заводе по производству машин. Чтобы изложить основную суть работы такого конвейера, представим, что ровно каждую минуту звучит гонг, и в этот момент все машины передвигаются по линии на один пункт. В каждом пункте рабочие выполняют определенную операцию с машиной, которая находится перед ними, например ставят колеса или тормоза. При каждом ударе гонга (это 1 цикл) одна новая машина поступает на конвейер и одна собранная машина сходит с конвейера. Завод выпускает одну машину в минуту независимо от того, сколько времени занимает сборка одной машины. В этом и состоит суть работы конвейера. Такой подход в равной степени применим и к процессорам, и к производству машин.

Конвейер с 7 стадиями: Mic-4

Мы не упомянули о том факте, что каждая микрокоманда выбирает следующую за ней микрокоманду. Большинство из них просто выбирают следующую команду в текущей последовательности, но последняя из них, например swap6, часто совершает межуровневый переход, который останавливает работу конвейера, поскольку после этого перехода вызывать команды заранее уже становится невозможно. Поэтому нам нужно придумать лучшую технологию.

Следующая (и последняя) микроархитектура — Mic-4. Ее основные части проиллюстрированы на рис. 4.23, но значительное количество деталей не показано, чтобы сделать схему более понятной. Как и Mic-3, эта микроархитектура содержит блок выборки команд, который заранее вызывает слова из памяти и сохраняет различные значения MBR.

Блок выборки команд передает входящий поток байтов в новый компонент — блок декодирования. Этот блок содержит внутреннее ПЗУ, которое индексируется кодом операции IJVM. Каждый элемент (ряд) блока состоит из двух частей: длины команды IJVM и индекса в другом ПЗУ— ПЗУ микроопераций. Длина команды IJVM нужна для того, чтобы блок декодирования мог разделить входящий поток байтов и установить, какие байты являются кодами операций, а какие операндами. Если длина текущей команды составляет 1 байт (например, длина команды POP), то блок декодирования определяет, что следующий байт — это код операции. Если длина текущей команды составляет 2 байта, блок декодирования определяет, что следующий байт — это операнд, сразу за которым следует другой код операции. Когда появляется префиксная команда WIDE, следующий байт преобразуется в специальный расширенный код операции, например, WIDE+ILOAD превращается в WIDE_ILOAD.

Блок декодирования передает индекс в ПЗУ микроопераций, который он находит в своей таблице, следующему компоненту, **блоку формирования очереди.** Этот блок содержит логические схемы и две внутренние таблицы: одна — для ПЗУ и одна — для ОЗУ. В ПЗУ находится микропрограмма, причем каждая команда IJVM содержит набор последовательных элементов, которые называются **микрооперации-**

ями. Эти элементы должны быть расположены в строгом порядке, и, например, переход из wide_iload2 в iload2, который допустим в микроархитектуре Mic-2, не разрешается. Каждая последовательность микроопераций должна выполняться полностью, в некоторых случаях последовательности дублируются.

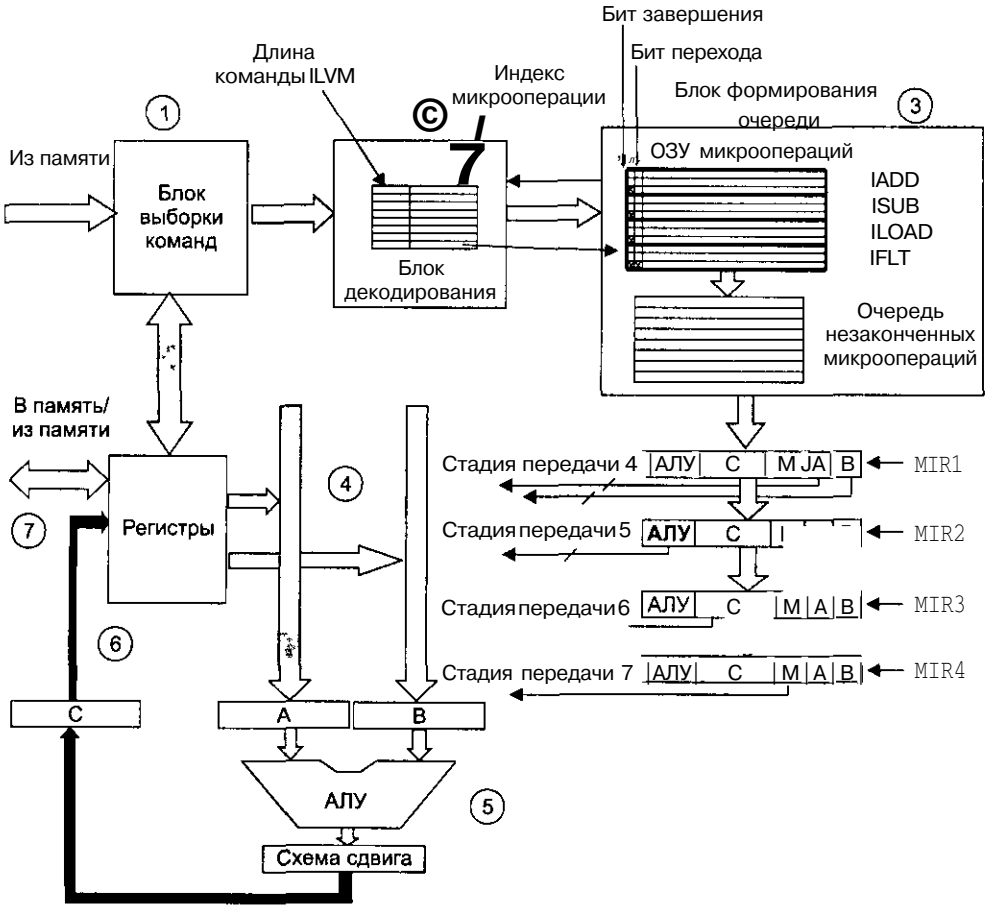


Рис. 4.23. Основные компоненты микроархитектуры Mic-4

Структура микрооперации сходна со структурой микрокоманды (см. рис. 4.4), только в данном случае поля NEXT_ADDRESS и JAM отсутствуют и требуется новое поле для определения входа на шину A. Имеется также два новых бита: бит завершения (Final bit) и бит перехода (Goto bit). Бит завершения устанавливается на последней микрооперации каждой последовательности (чтобы обозначить эту операцию). Бит перехода нужен для указания на микрооперации, которые являются условными микропереходами. По формату они отличаются от обычных микроопераций. Они состоят из битов JAM и индекса в ПЗУ микроопераций. Микрокоманды, которые раньше осуществляли какие-либо действия с трактом данных, а также выполняли условные микропереходы (например, iflt4), теперь нужно разбивать на две микрооперации.

Блок формирования очереди работает следующим образом. Он получает от блока декодирования индекс микрооперации ПЗУ. Затем он отыскивает микрооперацию и копирует ее во внутреннюю очередь. Затем он копирует следующую микрооперацию в ту же очередь, а также следующую за этой микрооперацией. Так продолжается до тех пор, пока не появится микрооперация с битом завершения. Тогда блок копирует эту последнюю микрооперацию и останавливается. Если блоку не встретилась микрооперация с битом перехода и у него осталось достаточно свободного пространства, он посылает сигнал подтверждения приема блоку декодирования. Когда блок декодирования воспринимает сигнал подтверждения, он посылает блоку формирования очереди следующую команду IJVM.

Таким образом, последовательность команд IJVM в памяти в конечном итоге превращается в последовательность микроопераций в очереди. Эти микрооперации передаются в регистры MIR, которые посылают сигналы тракту данных. Но есть еще один фактор, который нам нужно рассмотреть: поля каждой микрооперации не действуют одновременно. Поля A и B активны во время первого цикла, поле АЛУ активно во время второго цикла, поле C активно во время третьего цикла, а все операции с памятью происходят в четвертом цикле.

Чтобы все эти операции выполнялись правильно, мы ввели 4 независимых регистра MIR в схему на рис. 4.23. В начале каждого цикла (на рис. 4.2 это время *Aw)* значение MIR3 копируется в регистр MIR4, значение MIR2 копируется в регистр MIR3, значение MIR1 копируется в регистр MIR2, а в MIR1 загружается новая микрооперация из очереди. Затем каждый регистр MIR выдает сигналы управления, но используются только некоторые из них. Поля A и B из регистра MIR1 применяются для выбора регистров, которые запускают защелки A и B, а поле АЛУ в регистре MIR1 не используется и не связано ни с чем на тракте данных.

В следующем цикле микрооперация передается в регистр MIR2, а выбранные регистры в данный момент находятся в защелках A и B. Поле АЛУ теперь используется для запуска АЛУ. В следующем цикле поле C запишет результаты обратно в регистры. После этого микрооперация передается в регистр MIR4 и инициирует любую необходимую операцию памяти, используя загруженное значение регистра MAR (или MDR для записи).

Нужно обсудить еще один аспект микроархитектуры Mic-4: микропереходы. Некоторым командам IJVM нужен условный переход, который осуществляется с помощью бита N. Когда происходит такой переход, конвейер не может продолжать работу. Именно поэтому нам пришлось добавить в микрооперацию бит перехода. Когда в блок формирования очереди поступает микрооперация с таким битом, блок воздерживается от передачи сигнала о получении данных блоку декодирования. В результате машина будет простаивать до тех пор, пока этот переход не разрешится.

Предположительно, некоторые команды IJVM, не зависящие от этого перехода, уже переданы в блок декодирования, но не в блок формирования очереди, поскольку он еще не выдал сигнал о получении. Чтобы разобраться в этой путанице и вернуться к нормальной работе, требуется специальное аппаратное обеспечение и особые механизмы, но мы не будем рассматривать их в этой книге. Э. Дейкстра, написавший знаменитую работу «GOTO Statement Considered Harmful» («Выражение GOTO губительное), был прав.

Мы начали с микроархитектуры Mic-1 и, пройдя довольно долгий путь, закончили микроархитектурой Mic-4. Микроархитектура Mic-i представляла собой очень простой вариант аппаратного обеспечения, поскольку практически все управление осуществлялось программным обеспечением. Микроархитектура Mic-4 является конвейеризированной структурой с семью стадиями и более сложным аппаратным обеспечением. Данный конвейер изображен на рис. 4.24. Цифры в кружочках соответствуют компонентам рис. 4.23. Микроархитектура Mic-4 автоматически вызывает заранее поток байтов из памяти, декодирует его в команды IJVM, превращает их в последовательность операций с помощью ПЗУ и применяет их по назначению. Первые три стадии конвейера при желании можно связать с задающим генератором тракта данных, но работа будет происходить не в каждом цикле. Например, блок выборки команд совершенно точно не может передавать новый код операции блоку декодирования в каждом цикле, поскольку выполнение команды IJVM занимает несколько циклов и очередь быстро переполнится.

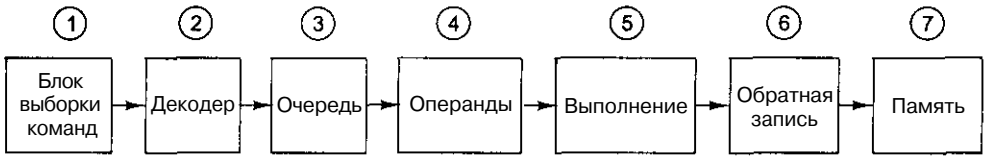


Рис. 4.24. Конвейер Mic-4

В каждом цикле значения регистров MIR перемещаются, а микрооперация, находящаяся в начале очереди, копируется в регистр MIR1. Затем сигналы управления из всех четырех регистров MIR передаются по тракту данных, вызывая определенные действия. Каждый регистр MIR контролирует отдельную часть тракта данных и, следовательно, различные микрошаги.

В данной разработке содержится конвейеризированный процессор. Благодаря этому отдельные шаги становятся очень короткими, а тактовая частота — высокой. Многие процессоры проектируются именно таким образом, особенно те, которым приходится выполнять старый набор команд (CISC). Например, процессор Pentium II в некоторых аспектах сходен с микроархитектурой Mic-4, как мы увидим позднее в этой главе.

Увеличение производительности

Все производители компьютеров хотят, чтобы их системы работали как можно быстрее. В этом разделе мы рассмотрим ряд передовых технологий для повышения производительности системы (в первую очередь процессора и памяти), которые исследуются в настоящее время. Поскольку в компьютерной промышленности наблюдается огромное количество конкурентов, между появлением новой идеи о том, как повысить скорость работы компьютера, и воплощением этой идеи обычно проходит очень небольшой период времени. Следовательно, большинство идей, которые мы сейчас будем обсуждать, уже применяются в производстве.

Усовершенствования, которые мы будем обсуждать, распадаются на две категории: усовершенствование реализации и усовершенствование архитектуры.

Усовершенствования реализации — это такие способы построения нового процессора и памяти, после применения которых система работает быстрее, но архитектура при этом не меняется. Изменение реализации без изменения архитектуры означает, что старые программы будут работать на новой машине, а это очень важно для успешной продажи. Чтобы усовершенствовать реализацию, можно, например, использовать более быстрый задающий генератор, но это не единственный способ. Отметим, что улучшение производительности от компьютера 80386 к 80486, Pentium, Pentium Pro, а затем Pentium II происходило без изменения архитектуры.

Однако некоторые типы усовершенствований можно осуществить только путем изменения архитектуры. Иногда, например, нужно добавить новые команды или регистры, причем таким образом, чтобы старые программы могли работать на новых моделях. В этом случае для достижения полной производительности программное обеспечение должно быть изменено или, по крайней мере, заново скомпилировано на новом компиляторе.

Однако один раз в несколько десятилетий разработчики понимают, что старая архитектура уже никуда не годится и что единственный способ развивать технологии дальше — начать все заново. Таким революционным скачком было появление RISC в 80-х годах, и следующий прорыв уже приближается. Мы рассмотрим наш пример (Intel IA-64) в главе 5.

Далее в этом разделе мы расскажем о четырех различных технологиях увеличения производительности процессора. Начнем мы с трех установившихся способов усовершенствования реализации, а затем перейдем к методу, для которого требуется поддержка архитектуры. Это кэш-память, прогнозирование ветвления, исполнение с изменением последовательности, подмена регистров и спекулятивное исполнение.

Кэш-память

Одним из самых важных вопросов при разработке компьютеров было и остается построение такой системы памяти, которая могла бы передавать операнды процессору с той же скоростью, с которой он их обрабатывает. Быстрый рост скорости работы процессора, к сожалению, не сопровождается столь же высоким ростом скорости работы памяти. Относительно процессора память работает все медленнее и медленнее с каждым десятилетием. С учетом огромной важности основной памяти эта ситуация сильно ограничивает развитие систем с высокой производительностью и направляет исследование таким путем, чтобы обойти проблему очень низкой по сравнению с процессором скорости работы памяти. И, откровенно говоря, эта ситуация ухудшается с каждым годом.

Современные процессоры предъявляют определенные требования к системе памяти и относительно времени ожидания (задержки в доставке операнда), и относительно пропускной способности (количества данных, передаваемых в единицу времени). К сожалению, эти два аспекта системы памяти сильно расходятся. Обычно с увеличением пропускной способности увеличивается время ожидания. Например, технологии конвейеризации, которые используются в микроархитектуре Mic-3, можно применить к системе памяти, при этом запросы памяти будут

обрабатываться более рационально, с перекрытием. Но, к сожалению, как и в микроархитектуре Mic-3, это приводит к увеличению времени ожидания отдельных операций памяти. С увеличением скорости задающего генератора становится все сложнее обеспечить такую систему памяти, которая может передавать операнды за один или два цикла.

Один из способов решения этой проблемы — добавление кэш-памяти. Как мы говорили в разделе «Кэш-память» главы 2, кэш-память содержит наиболее часто используемые слова, что повышает скорость доступа к ним. Если достаточно большой процент нужных слов находится в кэш-памяти, время ожидания может сильно сократиться.

Одной из самых эффективных технологий одновременного увеличения пропускной способности и уменьшения времени ожидания является применение нескольких блоков кэш-памяти. Основная технология — введение отдельной кэш-памяти для команд и отдельной для данных **(разделенной** кэш-памяти). Такая кэш-память имеет несколько преимуществ. Во-первых, операции могут начинаться независимо в каждой кэш-памяти, что удваивает пропускную способность системы памяти. Именно по этой причине в микроархитектуре Mic-1 нам понадобились два отдельных порта памяти: особый порт для каждой кэш-памяти. Отметим, что каждая кэш-память имеет независимый доступ к основной памяти.

В настоящее время многие системы памяти гораздо сложнее этих. Между разделенной кэш-памятью **и** основной памятью часто помещается **кэш-память второго уровня.** Вообще говоря, может быть три и более уровней кэш-памяти, поскольку требуются более продвинутые системы. На рис. 4.25 изображена система с тремя уровнями кэш-памяти. Прямо на микросхеме центрального процессора находится небольшая кэш-память для команд и небольшая кэш-память для данных, обычно от 16 до 64 Кбайт. Есть еще кэш-память второго уровня, которая расположена не на самой микросхеме процессора, а рядом с ним в том же блоке. Кэш-память второго уровня соединяется с процессором через высокоскоростной тракт данных. Эта кэш-память обычно не является разделенной и содержит смесь данных и команд. Ее размер — от 512 Кбайт до 1 Мбайт. Кэш-память третьего уровня находится на той же плате, что **и** процессор, **и** обычно состоит из статического ОЗУ в несколько мегабайтов, которое функционирует гораздо быстрее, чем динамическое ОЗУ основной памяти. Обычно все содержимое кэш-памяти первого уровня находится в кэш-памяти второго уровня, а все содержимое кэш-памяти второго уровня находится в кэш-памяти третьего уровня.

Существует два типа локализации адресов. Работа кэш-памяти зависит от этих типов локализации. **Пространственная локализация** основана на вероятности, что в скором времени появится потребность обратиться к ячейкам памяти, которые расположены рядом с недавно вызванными ячейками. Исходя из этого наблюдения в кэш-память переносится больше данных, чем требуется в данный момент. **Временная локализация** имеет место, когда недавно запрашиваемые ячейки запрашиваются снова. Это может происходить, например, с ячейками памяти, находящимися рядом с вершиной стека или с командами внутри цикла. Принцип временной локализации используется при выборе того, какие элементы выкинуть из

кэш-памяти в случае промаха кэш-памяти. Обычно отбрасываются те элементы, к которым давно не было обращений.

Рис. 4.25. Система с тремя уровнями кэш-памяти

Во всех типах кэш-памяти используется следующая модель. Основная память разделяется на блоки фиксированного размера, которые называются **строками кэш-памяти.** Строка кэш-памяти состоит из нескольких последовательных байтов (обычно от 4 до 64). Строки нумеруются, начиная с 0, то есть если размер строки составляет 32 байта, то строка 0 — это байты с 0 по 31, строка 1 — байты с 32 по 63 и т. д. В любой момент несколько строк находится в кэш-памяти. Когда происходит обращение к памяти, контроллер кэш-памяти проверяет, есть ли нужное слово в данный момент в кэш-памяти. Если есть, то можно сэкономить время, требуемое на доступ к основной памяти. Если данного слова в кэш-памяти нет, то какая-либо строка из нее удаляется, а вместо нее помещается нужная строка из основной памяти или из кэш-памяти более низкого уровня. Существует множество вариаций данной схемы, но в их основе всегда лежит идея держать в кэш-памяти как можно больше часто используемых строк, чтобы число успешных обращений к кэш-памяти было максимальным.

Кэш-память прямого отображения

Самый простой тип кэш-памяти — это **кэш-память прямого отображения.** Пример одноуровневой кэш-памяти прямого отображения показан на рис. 4.26, *а.* Данная кэш-память содержит 2048 элементов. Каждый элемент (ряд) может вмещать ровно одну строку из основной памяти. Если размер строки кэш-памяти 32 байта

(для этого примера), кэш-память может вмещать 64 Кбайт. Каждый элемент кэш-памяти состоит из трех частей:

1. Бит достоверности указывает, есть ли достоверные данные в элементе или нет. Когда система загружается, все элементы маркируются как недостоверные.
2. Поле «Тег» состоит из уникального 16-битного значения, указывающего соответствующую строку памяти, из которой поступили данные.
3. Поле «Данные» содержит копию данных памяти. Это поле вмещает одну строку кэш-памяти в 32 байта.

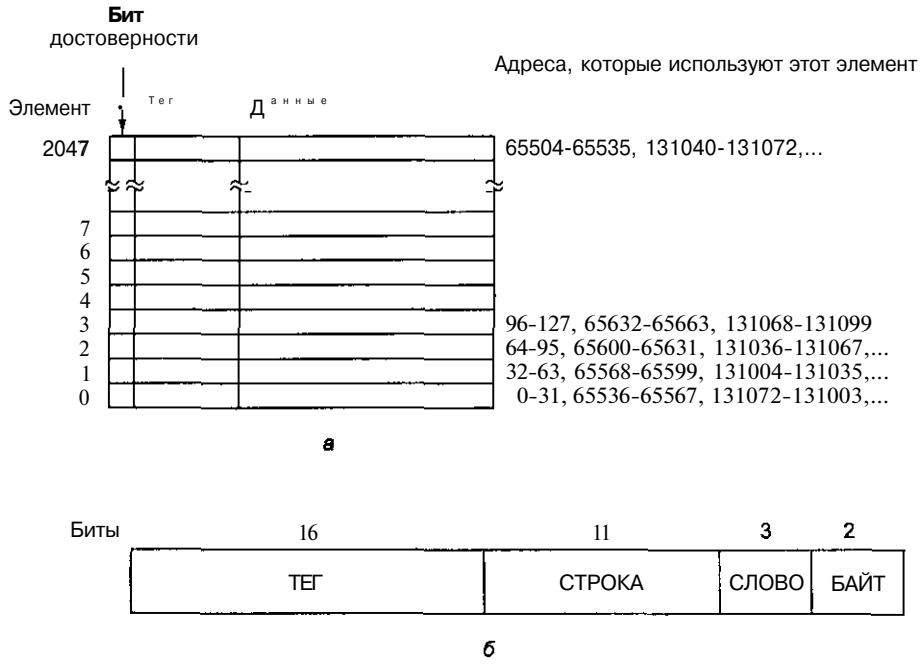


Рис. 4.26. Кэш-память прямого отображения (а); 32-битный виртуальный адрес (б)

В кэш-памяти прямого отображения данное слово может храниться только в одном месте. Если дан адрес слова, то в кэш-памяти его можно искать только в одном месте. Если его нет на этом определенном месте, значит, его вообще нет в кэш-памяти. Для хранения и удаления данных из кэш-памяти адрес разбивается на 4 компонента, как показано на рис. 4.26, *б:*

1. Поле «ТЕГ» соответствует битам, сохраненным в поле «Тег» элемента кэш-памяти.
2. Поле «СТРОКА» указывает, какой элемент кэш-памяти содержит соответствующие данные, если они есть в кэш-памяти.
3. Поле «СЛОВО» указывает, на какое слово в строке производится ссылка.
4. Поле «БАЙТ» обычно не используется, но если требуется только один байт, поле сообщает, какой именно байт в слове нужен. Для кэш-памяти, поддерживающей только 32-битные слова, это поле всегда будет содержать 0.

Когда центральный процессор выдает адрес памяти, аппаратное обеспечение выделяет из этого адреса 11 битов поля «СТРОКА» и использует их для поиска в кэш-памяти одного из 2048 элементов. Если этот элемент действителен, то производится сравнение поля «Тег» основной памяти и поля «Тег» кэш-памяти. Если юля равны, это значит, что в кэш-памяти есть слово, которое запрашивается. Такая :итуация называется **удачным обращением в кэш-память. В** случае удачного обрадения слово берется прямо из кэш-памяти, и тогда не нужно обращаться к основной памяти. Из элемента кэш-памяти берется только нужное слово. Остальная часть элемента не используется. Если элемент кэш-памяти недействителен (недостоверен) или поля «Тег» не совпадают, то нужного слова нет в памяти. Такая ситуация называется **промахом кэш-памяти.** В этом случае 32-байтная строка вызывается -l3 основной памяти и сохраняется в кэш-памяти, заменяя тот элемент, который там был. Однако если существующий элемент кэш-памяти изменяется, его нужно тписать обратно в основную память до того, как он будет отброшен.

Несмотря на сложность решения, доступ к нужному слову может быть чрезвы-1айно быстрым. Поскольку известен адрес, известно и точное нахождение слова, *'.ели оно имеется в кэш-памяти.* Это значит, что можно считывать слово из кэштамяти и доставлять его процессору и одновременно с этим устанавливать, правильное ли это слово (путем сравнения полей «Тег»), Поэтому процессор в действительности получает слово из кэш-памяти одновременно или даже до того, как станет известно, требуемое это слово или нет.

При такой схеме последовательные строки основной памяти помещаются в пос-1едовательные элементы кэш-памяти. Фактически в кэш-памяти может хранить-ся до 64 Кбайт смежных данных. Однако две строки, адреса которых различаются ювно на 64 К (65, 536 байт) или на любое целое кратное этому числу, не могут >дновременно храниться в кэш-памяти (поскольку они имеют одно и то же значение в поле «СТРОКА»). Например, если программа обращается к данным с адресом X, а затем выполняет команду, которой требуются данные с адресом X+ 65, 536 (или с любым другим адресом в той же строке), вторая команда требует пере-1агрузки элемента кэш-памяти. Если это происходит достаточно часто, то могут возникнуть проблемы. В действительности, если кэш-память плохо работает, то тучше бы вообще не было кэш-памяти, поскольку при каждой операции с памятью считывается целая строка, а не одно слово.

Кэш-память прямого отображения — это самый распространенный тип кэштамяти, и она достаточно эффективна, поскольку коллизии, подобные описанной выше, случаются крайне редко[1] или вообще не случаются. Например, очень хоропий компилятор может учитывать подобные коллизии при размещении команд и щнных в памяти. Отметим, что указанный выше случай не произойдет в системе, •де команды и данные находятся раздельно, поскольку конфликтующие запросы >удут обслуживаться разными блоками кэш-памяти. Таким образом, мы видим второе преимущество наличия двух блоков кэш-памяти вместо одного: большая 'ибкость при разрешении конфликтных ситуаций.

На самом деле подобные коллизии не столь уж и редки из-за TOI О, ЧТО при страничном способе организации виртуальной памяти и организации параллельного выполнения нескольких задач страницы как бы «перемешиваются». Разбиение программы на страницы осуществляется случайным образом, поэтому и «локальность кода» может быть нарушена. — *Примеч. научн. ред.*

Ассоциативная кэш-память с множественным доступом

Как было сказано выше, различные строки основной памяти конкурируют за право занять одну и ту же область в кэш-памяти. Если программе, которая применяет кэш-память, изображенную на рис. 4.26, а, часто требуются слова с адресами 0 и 65 536, то будут иметь место постоянные конфликты и каждое обращение потенциально повлечет за собой вытеснение какой-то определенной строки кэш-памяти. Чтобы разрешить эту проблему, нужно сделать так, чтобы в каждом элементе кэш-памяти помещалось по две и более строк. Кэш-память с n возможными элементами для каждого адреса называется **n-входовой ассоциативной кэш-памятью.** Четырехвходовая ассоциативная кэш-память изображена на рис. 4.27.

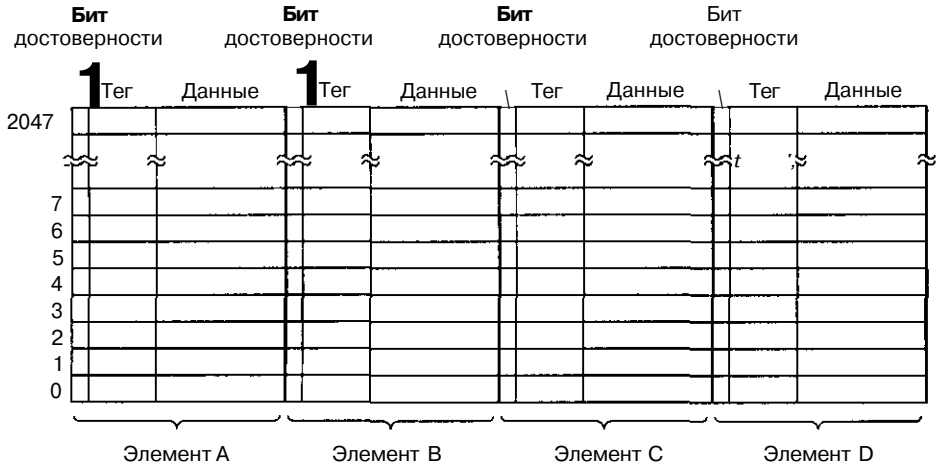


Рис. 4.27. Четырехвходовая ассоциативная кэш-память

Ассоциативная кэш-память с множественным доступом по сути гораздо сложнее, чем кэш-память прямого отображения, поскольку хотя элемент кэш-памяти **и** можно вычислить из адреса основной памяти, требуется проверить n элементов кэш-памяти, чтобы узнать, есть ли там нужная нам строка. Тем не менее практика показывает, что двувходовая или четырехвходовая ассоциативная кэш-память дает хороший результат, поэтому внедрение этих дополнительных схем вполне оправданно.

Использование ассоциативной кэш-памяти с множественным доступом ставит разработчика перед выбором. Если нужно поместить новый элемент в кэш-память, какой именно из старых элементов нужно убрать? Для многих целей хорошо подходит алгоритм **LRU (Least Recenly Used — алгоритм удаления наиболее давно использовавшихся элементов).** Имеется определенный порядок каждого набора ячеек, которые могут быть доступны из данной ячейки памяти. Всякий раз, когда осуществляется доступ к любой строке, в соответствии с алгоритмом список обновляется и маркируется элемент, к которому произведено последнее обращение. Когда требуется заменить какой-нибудь элемент, убирается тот, который находится в конце списка, то есть тот, который использовался давно по сравнению со всеми другими.

Возможна также 2048-входовая ассоциативная кэш-память, которая содержит один набор из 2048 элементов. Здесь все адреса памяти располагаются в этом наборе, поэтому при поиске требуется сравнивать нужный адрес со всеми 2048 тегами **в** кэш-памяти. Отметим, что для этого каждый элемент кэш-памяти должен содержать специальную логическую схему. Поскольку поле «СТРОКА» в данный момент имеет длину 0, поле «ТЕГ» — это весь адрес за исключением полей «СЛОВО» и «БАЙТ». Более того, когда строка кэш-памяти замещается, все 2048 ячеек являются возможными кандидатами на смену. Для сохранения упорядоченного списка потребовался бы громоздкий учет использования системных ресурсов, поэтому применение алгоритма LRU становится недопустимым. (Помните, что этот список следует обновлять при каждой операции с памятью.) Интересно, что кэш-память с высокой ассоциативностью часто не сильно превосходит по производительности кэш-память с низкой ассоциативностью, а в некоторых случаях работает даже хуже. Поэтому ассоциативность выше четырех встречается редко.

Наконец, особой проблемой для кэш-памяти является запись. Когда процессор записывает слово, а это слово находится в кэш-памяти, он, очевидно, должен или обновить слово, или отбросить данный элемент кэш-памяти. Практически во всех разработках используется обновление кэш-памяти. А что же можно сказать об обновлении копии в основной памяти? Эту операцию можно отложить на потом до того момента, когда строка кэш-памяти будет готова к замене алгоритмом LRU. Выбор труден, и ни одно из решений не является предпочтительным. Немедленное обновление элемента основной памяти называется **сквозной записью.** Этот подход обычно гораздо проще реализуется, и к тому же, он более надежен, поскольку современная память всегда может восстановить предыдущее состояние, если произошла ошибка. К сожалению, при этом требуется передавать больший поток информации к памяти, поэтому в более сложных проектах стремятся использовать альтернативный подход — **обратную** запись.

С процессом записи связана еще одна проблема: а что происходит, если нужно записать что-либо в ячейку, которая в текущий момент не находится в кэш-памяти? Должны ли данные переноситься в кэш-память или просто записываться в основную память? И снова ни один из ответов не является во всех отношениях лучшим. В большинстве разработок, в которых применяется обратная запись, данные переносятся в кэш-память. Эта технология называется **заполнением по записи** (write allocation). С другой стороны, в тех разработках, где применяется сквозная запись, обычно элемент в кэш-память при записи не помещается, поскольку эта возможность усложняет разработку. Заполнение по записи полезно только в том случае, если имеют место повторные записи в одно и то же слово или в разные слова в пределах одной строки кэш-памяти.

Прогнозирование ветвления

Современные компьютеры сильно конвейеризированы. Конвейер, изображенный на рис. 4.25, имеет семь стадий; более сложно организованные компьютеры содержат конвейеры с десятью и более стадиями. Конвейеризация лучше работает с линейным кодом, поэтому блок выборки команд может просто считывать последовательные слова из памяти и отправлять их в блок декодирования заранее, еще до того, как они понадобятся.

Единственная проблема состоит в том, что эта модель совершенно не реалистична. Программы вовсе не являются последовательностями линейного кода. В них полно команд перехода. Рассмотрим простые утверждения листинга 4.4. Переменная i сравнивается с 0 (вероятно, это самый распространенный тест на практике). В зависимости от результата другой переменной, к, присваивается одно из двух возможных значений.

Листинг 4.4. Фрагмент программы

```
if (i—0)
        к-1,
else
        к=2:
```

Возможный перевод на язык ассемблера показан в листинге 4.5. Язык ассемблера мы будем рассматривать позже в этой книге, и детали сейчас не важны, но при определенных машине и компиляторе программа, более или менее похожая на программу листинга 4.5, вполне возможна. Первая команда сравнивает переменную i с 0. Вторая совершает переход к Else, если i не равно 0. Третья команда присваивает значение 1 переменной к. Четвертая команда совершает переход к следующему высказыванию программы. Компилятор поместил там метку Next. Пятая команда присваивает значение 2 переменной к.

Листинг 4.5. Перевод программы листинга 4.4 на язык ассемблер

```
        CMP 1. 0   . сравнение i с 0
        BNE Else   , переход к Else, если они не равны
Then    MOV k. 1   , присваивание значения 1 переменной к
        BR Next    , безусловный переход к Next
Else    MOV к 2    , присваивание значения 2 переменной к
Next
```

Мы видим, что две из пяти команд являются переходами. Более того, одна из них, BNE, — это условный переход (переход, который осуществляется тогда и только тогда, когда выполняется определенное условие, в данном случае это равенство цвух операндов предыдущей команды CMP). Самый длинный линейный код состоит здесь из двух команд. Вследствие этого вызывать команды с высокой скоростью для передачи в конвейер очень трудно.

На первый взгляд может показаться, что безусловные переходы, например команда BR Next в листинге 4.5, не влекут за собой никаких проблем. Вообще говоря, в данном случае нет никакой двусмысленности в том, куда дальше идти. Почему же блок выборки команд не может просто продолжать считывать команды из целевого адреса (то есть из того места, куда будет затем осуществлен переход)?

Сложность объясняется самой природой конвейеризации. На рис. 4.23, например, мы видим, что декодирование происходит на второй стадии. Следовательно, блоку выборки команд приходится решать, откуда вызывать следующую команду еще до того, как он узнает, команду какого типа он только что вызвал. Только в следующем цикле он сможет узнать, что получил команду безусловного перехода, и до этого момента он уже начал вызывать команду, следующую за безусловным переходом. Вследствие этого существенная часть конвейеризированных машин (например, UltraSPARC II) обладает таким свойством, что сначала выполняется команда, *следующая после* безусловного перехода, хотя по логике вещей этого не

должно быть. Позиция после перехода называется отсрочкой ветвления. Pentium II (а также машина, используемая в листинге 4.5) не обладает таким качеством, но обойти эту проблему путем усложнения внутреннего устройства чрезвычайно тяжело. Оптимизирующий компилятор постарается найти какую-нибудь полезную команду, чтобы поместить ее в отсрочку ветвления, но часто ничего подходящего нет, поэтому компилятор вынужден вставлять туда команду NOP. Это сохраняет правильность программы, но зато программа становится больше по объему и работает медленнее.

С условными переходами дело обстоит еще хуже. Во-первых, они тоже содержат отсрочки ветвления, а во-вторых, блок выборки команд узнает, откуда нужно считывать команду, гораздо позже. Первые конвейеризированные машины просто простаивали до тех пор, пока не становилось известно, нужно ли совершать переход или нет. Простаивание по три или четыре цикла при каждом условном переходе, особенно если 20% команд являются условными переходами, сильно снижает производительность.

Поэтому большинство машин прогнозируют, будет производиться условный переход, который встретился на пути, или нет. Для этого, например, можно предполагать, что все условные переходы назад будут осуществляться, а все условные переходы вперед не будут. Что касается первой части предположения, причина такого выбора состоит в том, что переходы назад часто помещаются в конце цикла. Большинство циклов выполняется много раз, поэтому переход к началу цикла будет встречаться очень часто.

Со второй частью данного предположения дело обстоит сложнее. Некоторые переходы вперед осуществляются в случае обнаружения ошибки в программном обеспечении (например, файл не может быть открыт). Ошибки случаются редко, поэтому в большинстве случаев подобные переходы не происходят. Естественно, существует множество переходов вперед, не связанных с ошибками, поэтому процент успеха здесь не так высок, как в переходах назад. Однако это правило по крайней мере лучше, чем ничего.

Если переход правильно предсказан, то ничего особенного делать не нужно. Просто продолжается выполнение программы. Проблема возникает в том случае, когда переход предсказан неправильно. Вычислить, куда нужно перейти, и перейти именно туда несложно. Самое сложное — отменить команды, которые уже выполнены, но которые не нужно было выполнять.

Существует два способа отмены команд. Первый способ — продолжать выполнять команды, вызванные после спрогнозированного условного перехода, до тех пор, пока одна из этих команд не попытается изменить состояние машины (например, сохранить значение в регистре). Тогда вместо того, чтобы перезаписывать этот регистр, нужно поместить вычисленное значение во временный (скрытый) регистр, а затем, когда уже станет известно, что прогноз был правильным, просто скопировать это значение в обычный регистр. Второй способ — записать значение любого регистра, который, вероятно, скоро будет переписан (например, в скрытый временный регистр), поэтому машина может вернуться в предыдущее состояние в случае неправильно предсказанного перехода. Реализация обоих способов очень сложна и требует громоздкого учета использования системных ресурсов. А если встречается второй условный переход до того, как стало известно, был ли первый условный переход предсказан правильно, все может совершенно запутаться.

Динамическое прогнозирование ветвления

Ясно, что точные прогнозы очень ценны, поскольку это позволяет процессору работать с полной скоростью. В настоящее время проводится множество исследований, целью которых является усовершенствование алгоритмов прогнозирования ветвления (например, [32,70,108,125,138,163]). Один из подходов — хранить специальную таблицу (в особом аппаратном обеспечении), в которую центральный процессор записывает условные переходы, когда они встречаются, и там их можно искать, когда они снова появятся. Простейшая версия такой схемы показана на рис. 4.28, *а*. В данном случае эта таблица содержит одну ячейку для каждой команды условного перехода. В ячейке находится адрес команды перехода, а также бит, который указывает, был ли сделан переход, когда эта команда встретилась последний раз. Прогноз состоит в том, что программа пойдет тем же путем, каким она пошла в прошлый раз после этой команды перехода. Если прогноз неверен, бит в таблице меняется.

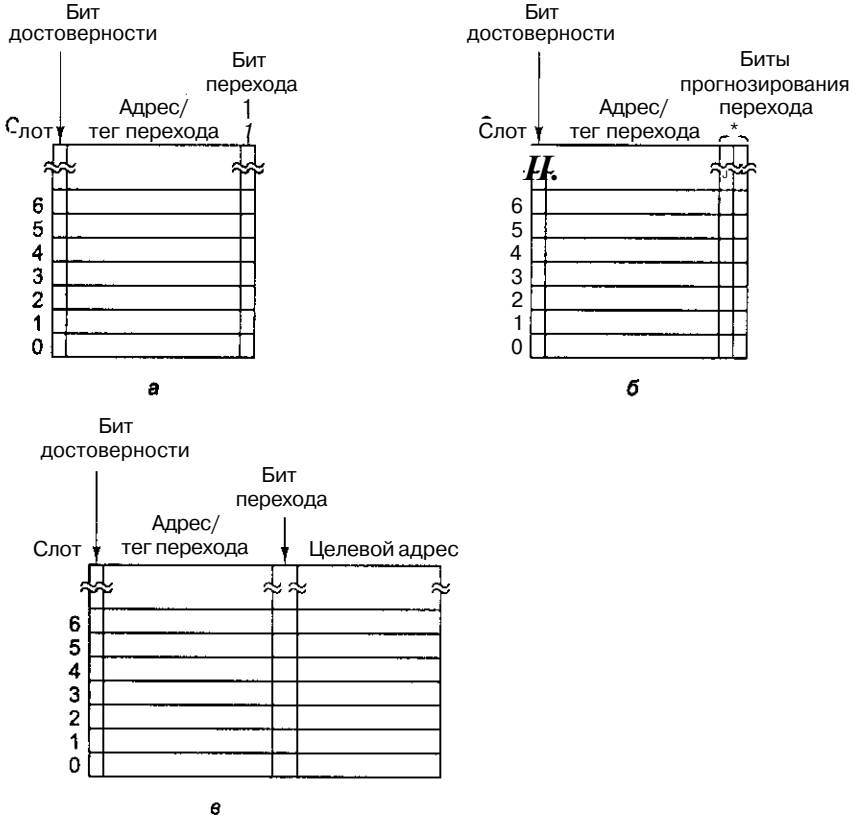


Рис. 4.28. Таблица динамики ветвлений с 1-битным указателем перехода (а), таблица динамики ветвлений с 2-битным указателем перехода (б), соответствие между адресом команды перехода и целевым адресом (в)

Существует несколько способов организации данной таблицы. В действительности точно такие же способы используются при организации кэш-памяти,

Рассмотрим машину с 32-битными командами, которые расположены таким образом, что два младших бита каждого адреса памяти — 00. Таблица содержит 2^n ячеек (строк). Из команды перехода можно извлечь n+2 младших бита и осуществить сдвиг вправо на два бита. Это n-битное число можно использовать в качестве индекса в таблице, где проверяется, совпадает ли адрес, сохраненный там, с адресом перехода. Как и в случае с кэш-памятью, здесь нет необходимости сохранять n+2 младших бита, поэтому их можно опустить (то есть сохраняются только старшие адресные биты — тег). Если адреса совпали, бит прогнозирования используется для предсказания перехода. Если тег неправильный или элемент недействителен, значит, имеет место несовпадение. В этом случае можно применять правило перехода вперед/назад.

Если таблица динамики переходов содержит, скажем, 4096 элементов, то адреса 0, 16384, 32768,... будут конфликтовать; аналогичная проблема встречается и при работе с кэш-памятью. Здесь возможно такое же решение: двухальтернативный, четырехальтернативный, n-альтернативный ассоциативный элемент. Как и у кэш-памяти, предельный случай — один n-альтернативный ассоциативный элемент.

При достаточно большом размере таблицы и достаточной ассоциативности эта схема хорошо работает в большинстве ситуаций. Тем не менее систематически встречается одна проблема. Когда происходит выход из цикла, переход в конце будет предсказан неправильно, и, что еще хуже, этот неправильный прогноз изменит бит в таблице, который теперь будет указывать, что переход совершать не надо. В следующий раз, когда опять будет выполняться цикл, переход в конце первого прохождения цикла будет спрогнозирован неправильно. Если цикл находится внутри другого цикла или внутри часто вызываемой процедуры, эта ошибка будет повторяться слишком часто.

Для устранения такой ситуации мы немного изменим метод, чтобы прогноз менялся только после двух последовательных неправильных предсказаний. Такой подход требует наличия двух предсказывающих битов в таблице: один указывает, предполагается ли совершить переход или нет, а второй указывает, что было сделано в прошлый раз. Таблица показана на рис. 4.28, *б*.

Этот алгоритм можно представить в виде конечного автомата с четырьмя состояниями (рис. 4.29). После ряда последовательных успешных предсказаний «перехода нет» конечный автомат будет находиться в состоянии 00 и в следующий раз также прогнозировать, что «перехода нет». Если этот прогноз неправильный, автомат переходит в состояние 01, но в следующий раз все равно предсказывает отсутствие перехода. Только в том случае, если это последнее предсказание ошибочно, конечный автомат перейдет в состояние 11 и будет все время прогнозировать наличие перехода. Фактически, левый бит — это прогноз, а правый бит — это то, что было сделано в прошлый раз (то есть был ли совершен переход). В данной разработке используется только 2 специальных бита, но возможно применение и 4, и 8 битов.

Это не первый конечный автомат, который мы рассматриваем. На рис. 4.19 тоже изображен конечный автомат. На самом деле все наши микропрограммы можно считать конечными автоматами, поскольку каждая строка представляет особое состояние, в котором может находиться автомат, с четко определенными переходами к конечному набору других состояний. Конечные автоматы очень широко используются во всех аспектах разработки аппаратного обеспечения.

Рис. 4.29. Двубитный конечный автомат для прогнозирования переходов

До сих пор мы предполагали, что цель каждого условного перехода известна. Обычно или в явном виде давался адрес, к которому нужно перейти (он содержался прямо в самой команде), или было известно смещение относительно текущей команды (то есть число со знаком, которое нужно было прибавить к счетчику команд). Часто это предположение имеет силу, но некоторые команды условного перехода вычисляют целевой адрес, выполняя определенные арифметические действия над значениями регистров, а затем уже переходят туда. Даже если взять конечный автомат, изображенный на рис. 4.29, который точно прогнозирует переходы, такой прогноз будет не нужен, поскольку целевой адрес неизвестен. Один из возможных выходов из подобной ситуации — сохранить в таблице адрес, к которому был осуществлен переход в прошлый раз, как показано на рис. 4.28, *в*. Тогда, если в таблице указано, что в прошлый раз, когда встретилась команда перехода по адресу 516, переход был совершен в адрес 4000, и если сейчас предсказывается совершение перехода, то целевым адресом снова будет 4000.

Еще один подход к прогнозированию ветвления — следить, были ли совершены последние k условных переходов, независимо от того, какие это были команды [108]. Это k-битное число, которое хранится в **сдвиговом регистре динамики переходов,** затем сравнивается параллельно со всеми элементами таблицы с k-битным ключом, и в случае совпадения применяется то предсказание, которое найдено в этом элементе. Удивительно, но эта технология работает достаточно хорошо.

Статическое прогнозирование ветвления

Все технологии прогнозирования ветвления, которые обсуждались до сих пор, являются динамическими, то есть выполняются во время работы программы. Они также приспосабливаются к текущему поведению программы, и это их положительное качество. Отрицательной стороной этих технологий является то, что они требуют специализированного и дорогостоящего аппаратного обеспечения, а также наличия очень сложных микросхем.

Можно пойти другим путем и призвать на помощь компилятор. Когда компилятор получает такое выражение, как

```
for (1=0  1 < 1000000,  1++) {   }
```

это знает, что переход в конце цикла будет происходить практически всегда. Если бы только был способ сообщить это аппаратному обеспечению, можно было бы избавиться от огромного количества работы.

Хотя это связано с изменением архитектуры (а не только с вопросом реализации), в некоторых машинах, например UltraSPARC II, имеется еще один набор команд условного перехода помимо обычных (которые нужны для обратной совместимости). Новые команды содержат бит, по которому компилятор определяет, совершать переход или не совершать. Когда встречается такой бит, блок выборки команд просто делает то, что ему сказано. Более того, нет необходимости тратить драгоценное пространство в таблице предыстории переходов для этих команд, что сокращает количество конфликтных ситуаций.

Наконец, наша последняя технология прогнозирования ветвления основана на профилировании [37]. Это тоже статическая технология, только в данном случае программа не заставляет компилятор вычислять, какие переходы нужно совершать, а какие нет. В данном случае программа действительно выполняется, а ветвления фиксируются. Эта информация поступает в компилятор, который затем использует специальные команды условного перехода для того, чтобы сообщить аппаратному обеспечению, что нужно делать.

Исполнение с изменением последовательности и подмена регистров

Большинство современных процессоров являются и конвейеризированными и суперскалярными, как показано на рис. 2.5. Это значит, что там есть блок выборки команд, который заранее вызывает команды из памяти и передает их в блок декодирования. Блок декодирования, в свою очередь, передает декодированные команды в соответствующие функциональные блоки для выполнения. В некоторых случаях этот блок может разбивать отдельные команды на микрооперации, перед тем как отправить их в функциональные блоки.

Ясно, что самым простым является компьютер, в котором все команды выполняются в том порядке, в котором они вызываются из памяти (предполагается, что прогнозирование переходов всегда оказывается верным). Однако такое последовательное выполнение не всегда дает оптимальную производительность из-за взаимной зависимости команд. Если команде требуется значение, которое вычисляется предыдущей командой, вторая команда не может начать выполняться, пока первая не выдаст нужную величину. В такой ситуации реальной взаимозависимости второй команде приходится ждать. Существуют и другие виды взаимозависимостей, но о них мы поговорим позже.

Чтобы обойти эти проблемы и достичь лучшей производительности, некоторые процессоры пропускают взаимозависимые команды и переходят к следующим (независимым) командам. Думаю, не нужно говорить, что алгоритм распределения команд должен давать такой же результат, как если бы все команды выполнялись в том порядке, в котором они написаны. А теперь продемонстрируем на конкретном примере, как происходит переупорядочение команд.

Чтобы изложить основную суть проблемы, начнем с машины, которая запускает команды в том порядке, в котором они расположены в программе, и требует, чтобы выполнение команд завершалось также в порядке, соответствующем программному. Важность второго требования прояснится позднее.

Наша машина содержит 8 регистров, видимых для программиста, от R0 до R7. Все арифметические команды используют три регистра: два — для операндов и один — для результата, как и в микроархитектуре Mic-4. Мы предполагаем, что если команда декодируется в цикле n, выполнение начинается в цикле n+1. В случае с простой командой, например командой сложения или вычитания, запись обратно в выходной регистр происходит в конце цикла n+2. В случае с более сложной командой, например командой умножения, запись в регистр происходит в конце цикла n+3 Чтобы сделать наш пример реалистичным, мы позволим блоку декодирования выпускать до двух команд за цикл. Некоторые суперскалярные процессоры могут выпускать 4 или даже 6 команд за цикл.

Последовательность выполнения команд показана в табл. 4.12. В первом столбце приводится номер цикла, во втором — номер команды, а в третьем — сама команда. В четвертом столбце указано, выдача каких команд произошла (максимум две команды за цикл). Цифры в пятом столбце сообщают, какие команды завершены Помните, что в нашем примере мы требуем, чтобы команды и запускались, и завершались в строгом порядке, поэтому выдача команды k+1 может произойти только после выдачи команды k, а результат команды k+1 не может быть записан в выходной регистр до того, как завершится выполнение команды k. Оставшиеся 16 столбцов мы обсудим ниже.

Таблица 4.12. Суперскалярный процессор с последовательной выдачей и последовательным завершением команд

Цикл	#	Команда	Выдача	Завершение	Считываемые регистры 0	1	2	3	4	5	6	7	Записываемые регистры 0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	-		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										**1**	
	5	R7=R1'R2	5			2	1		1										1	1
6	6	R1=R0-R2	-			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1						1							
	7	R3=R3*R1	-		1		1						1							
10					1		1						1							
11				6																
12			7			1		1								1				
	8	R1=R4+R4	-			1		1								1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2				1							
17									2				1							
18				8																

После декодирования команды блок декодирования должен определить, запускать ли команду сразу или нет. Для этого блок декодирования должен знать состояния всех регистров. Если, например, текущей команде требуется регистр, значение которого еще не подсчитано, текущая команда не может быть выпущена, и центральный процессор должен простаивать.

Следить за состоянием регистров будет специальное устройство — счетчик обращений (scoreboard), который впервые появился в CDC 6600. Счетчик обращений содержит небольшой счетчик для каждого регистра, который показывает, сколько раз этот регистр используется командами, выполняющимися в данный момент, в качестве источника. Если одновременно может выполняться максимум 15 команд, тогда будет достаточно 4-битного счетчика. Когда запускается команда, элементы счетчика обращений, соответствующие регистрам операндов, увеличиваются на 1. Когда выполнение команды завершено, соответствующие элементы счетчика уменьшаются на 1.

Счетчик обращений также содержит счетчики для регистров, которые используются в качестве пунктов назначения. Поскольку допускается только одна запись за раз, эти счетчики могут быть размером в один бит. Правые 16 столбцов табл. 4.12 демонстрируют показания счетчика обращений.

В реальных машинах счетчик обращений также следит за использованием функционального блока, чтобы избежать выдачи команды, для которой нет доступного функционального блока. Для простоты мы предполагаем, что подходящий функциональный блок всегда имеется в наличии, поэтому функциональные блоки в таблице не показаны.

В первой строке табл. 4.12 показана команда 1, которая перемножает значения регистров R0 и K1,ипомещаетрезультатврегистр113. Поскольку ни один из этих регистров еще не используется, команда запускается, а счетчик обращений показывает, что регистры R0 и R1 считываются, а регистр R3 записывается. Ни одна из последующих команд не может записывать результат в эти регистры и не может считывать регистр R3 до тех пор, пока не завершится выполнение команды 1. Поскольку это команда умножения, она закончится в конце цикла 4. Значения счетчика обращений, приведенные в каждой строке, отражают состояние регистров после запуска команды, записанной в этой же строке. Пустые клетки соответствуют значению 0.

Поскольку рассматриваемый пример — это суперскалярная машина, которая может запускать две команды за цикл, вторая команда выдается также во время цикла 1. Она складывает значения регистров R0 и R2, а результат сохраняет в регистре R4. Чтобы определить, можно ли запускать эту команду, применяются следующие правила:

1. Если какой-нибудь операнд записывается, запускать команду нельзя (RAW-взаимозависимость).
2. Если считывается регистр результатов, запускать команду нельзя (WAR-взаимозависимость).
3. Если записывается регистр результатов, запускать команду нельзя (WAW-взаимозависимость).

Мы уже рассматривали RAW-взаимозависимости, имеющие место, когда команде в качестве источника нужно использовать результат предыдущей команды,

которая еще не завершилась. Два других типа взаимозависимостей менее серьезные. По существу, они связаны с конфликтами ресурсов. В **WAR-взаимозависимости** (Write After Read — запись после чтения) одна команда пытается перезаписать регистр, который предыдущая команда еще не закончила считывать. **WAW-взаимозависимость (Write After Write — запись после записи)** сходна с WAR-взаимозависимостью. Этого можно избежать, если вторая команда будет помещать результат где-либо в другом месте еще (возможно, временно). Если ни одна из трех упомянутых ситуаций не возникает и нужный функциональный блок доступен, то команду можно выпустить. В этом случае команда 2 использует регистр R0, который в данный момент считывается незаконченной командой, но подобное перекрытие допустимо, поэтому команда 2 может запускаться. Сходным образом команда 3 запускается во время цикла 2.

А теперь перейдем к команде 4, которая должна использовать регистр R4. К сожалению, из таблицы мы видим, что в регистр R4 в данный момент производится запись (см. строку 3 в таблице). Здесь имеет место RAW-взаимозависимость, поэтому блок декодирования простаивает до тех пор, пока регистр R4 не станет доступен. Во время простаивания блок декодирования прекращает получать команды из блока выборки команд. Когда внутренние буферы блока выборки команд заполнятся, он прекращает вызывать команды из памяти.

Следует упомянуть, что следующая команда, команда 5, не конфликтует ни с одной из заверигенных команд. Ее можно было бы декодировать и выпустить, если бы в нашей разработке не требовалось, чтобы команды выдавались по порядку.

Посмотрим, что происходит в цикле 3. Команда два, а это команда сложения (два цикла), завершается в конце цикла 3. Но ее результат не может быть сохранен в регистре R4 (который тогда освободится для команды 4). Почему? Потому что данная разработка требует записи результатов в регистры в соответствии с порядком программы. Но зачем? Что плохого произойдет, если сохранить результат в регистре R4 сейчас и сделать это значение доступным?

Ответ на этот вопрос очень важен. Предположим, что команды могут завершаться в произвольном порядке. Тогда в случае прерывания будет очень сложно сохранить состояние машины так, чтобы его можно было потом восстановить. В частности, нельзя будет сказать, что все команды до какого-то адреса были выполнены, а все команды после этого адреса не были выполнены. Это называется **точным прерыванием** и является желательной характеристикой центрального процессора [99]. Сохранение результатов в произвольном порядке делает прерывания неточными, и именно поэтому в некоторых машинах требуется соблюдение жесткого порядка в завершении команд.

Вернемся к нашему примеру. В конце четвертого цикла результаты всех трех команд могут быть сохранены, поэтому в цикле 5 может быть выпущена команда 4, а также недавно декодированная команда 5. Всякий раз, когда завершается какая-нибудь команда, блок декодирования должен проверять, нет ли простаивающей команды, которую теперь уже можно выпустить.

В цикле 6 команда 6 простаивает, потому что ей нужно записать результат в регистр R1, а регистр R1 занят. Выполнение команды начинается только в цикле 9. Чтобы завершить всю последовательность из 8 команд, требуется 15 циклов из-за многочисленных ситуаций взаимозависимости, хотя аппаратное обеспечение

способно выдавать по две команды за цикл. В колонках «Выдача» и «Завершение» табл. 4.12 видно, что все команды выдаются из блока декодирования по порядку и завершаются эти команды тоже по порядку.

Рассмотрим альтернативный подход: исполнение с изменением последовательности. В такой разработке выполнение команд может начинаться в произвольном порядке и завершаться также в произвольном порядке. В табл. 4.13 показана та же последовательность из восьми команд, только теперь разрешен другой порядок выдачи команд и сохранения результатов в регистрах.

Первое различие встречается в цикле 3. Несмотря на то, что команда 4 простаивает, мы можем декодировать и запустить команду 5, поскольку она не создает конфликтной ситуации ни с одной из выполняющихся программ. Однако пропуск команд порождает новую проблему. Предположим, что команда 5 использует операнд, который вычисляется пропущенной командой 4. При текущем состоянии счетчика обращений мы этого не заметим. В результате нам придется расширить счетчик обращений, чтобы следить за записями, которые совершают пропущенные команды. Это можно сделать, добавив еще одно битовое отображение, по одному биту на регистр, для контроля за записями, которые делают простаивающие команды (эти счетчики в таблице не показаны). Правило для запуска команд следует расширить, с тем чтобы предотвратить запуск команды, операнд которой должен быть записан той предшествующей командой, что шла до нее, но была пропущена.

Таблица 4.13. Работа суперскалярного процессора с изменением последовательности запуска и завершения команд

Цикл	#	Команда	Выдача	Завершение	Считываемые регистры 0	1	2	3	4	5	6	7	Записываемые регистры 0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5^R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	-		3	2	1									1	1	1		
3	5	R7^R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0-R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	-		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Теперь посмотрим на команды 6, 7 и 8 в табл. 4.12. Здесь мы видим, что команда 6 помещает вычисленное значение в регистр R1, и это значение использует-

ся командой 7. Мы также видим, что это значение больше не используется, потому что команда 8 переписывает значение регистра R1. Нет никакой надобности использовать регистр R1 для хранения результата команды 6. Еще хуже то, что далеко не лучшим является выбор R1 в качестве промежуточного регистра, хотя с точки зрения программиста, привыкшего к идее последовательного выполнения команд без перекрытий, этот выбор является самым разумным.

В таблице 4.12 мы ввели новый метод для решения этой проблемы: **подмена регистров.** Блок декодирования меняет регистр R1 в команде 6 (цикл 3) и в команде 7 (цикл 4) на скрытый регистр S1, который невидим для программиста. Теперь команда 6 может запускаться одновременно с командой 5. Современные процессоры содержат десятки скрытых регистров, которые используются для процедуры подмены. Такая технология часто устраняет WAR- и WAW-взаимозависимости.

В команде 8 мы снова применяем подмену регистров. На этот раз регистр R1 переименовывается в S2, поэтому операция сложения может начаться до того, как регистр R1 освободится, а освободится он только в конце цикла 6. Если окажется, что результат в этот момент должен быть в регистре R1, содержимое регистра S2 всегда можно скопировать туда. Еще лучше то, что все будущие команды, которым нужен этот результат, могут в качестве источника использовать регистры, переименованные в тот регистр, где действительно хранится нужное значение. В любом случае выполнение команды 8 начнется раньше,

В настоящих (не гипотетических) компьютерах подмена регистров происходит с многократным вложением. Существует множество скрытых регистров и таблица, в которой показывается соответствие видимых для программиста регистров и скрытых регистров. Например, чтобы найти местоположение регистра R0, нужно обратиться к элементу 0 этой таблицы. На самом деле реального регистра R0 нет, а есть только связь между именем R0 и одним из скрытых регистров. Эта связь часто меняется во время выполнения программы, чтобы избежать взаимозависимостей.

Обратите внимание на четвертый и пятый столбцы табл. 4.12. Вы видите, что команды запускаются не по порядку и завершаются также не по порядку. Вывод весьма прост: изменяя последовательность выполнения команд и подменяя регистры, мы можем ускорить процесс вычисления почти в два раза.

Спекулятивное выполнение

В предыдущем разделе мы ввели понятие переупорядочения команд. Эта процедура нужна для улучшения производительности. В действительности имелось в виду переупорядочение команд в пределах одного базового элемента программы. Рассмотрим этот аспект подробнее.

Компьютерные программы можно разбить на **базовые элементы,** каждый из которых представляет собой линейную последовательность команд с точкой входа в начале и точкой выхода в конце. Базовый элемент не содержит никаких управляющих структур (например, условных операторов i f или операторов цикла **whi** I e), поэтому при трансляции на машинный язык нет никаких ветвлений. Базовые элементы связываются операторами управления.

Программа в такой форме может быть представлена в виде ориентированного графа, как показано на рис. 4.30. Здесь мы вычисляем сумму кубов четных и нечетных целых чисел до какого-либо предела и помещаем результаты в *evensum* и *oddsum* соответственно (листинг 4.6). В пределах каждого базового элемента технологии, упомянутые в предыдущем разделе, работают отлично.

Листинг 4.6. Фрагмент программы

```
evesum=0,
oddsum-0;
i=0;
while (I<limit) {
       k-i*i*i.
       1f(((1/2)*2)==1)
           evensum=evensunH-k;
       else
           oddsum=oddsum+k;
       i-i+I:
}
```

Проблема состоит в том, что большинство базовых элементов очень короткие и в них недостаточно параллелизма. Следовательно, нужно сделать так, чтобы переупорядочение последовательности команд можно было применять не только в пределах конкретного базового элемента. Выгоднее всего будет передвинуть потенциально медленную операцию в графе повыше, чтобы ее выполнение началось раньше. Это может быть команда LOAD, операция с плавающей точкой или даже начало длинной цепи зависимостей. Перемещение кода вверх по ребру графа называется подъемом.

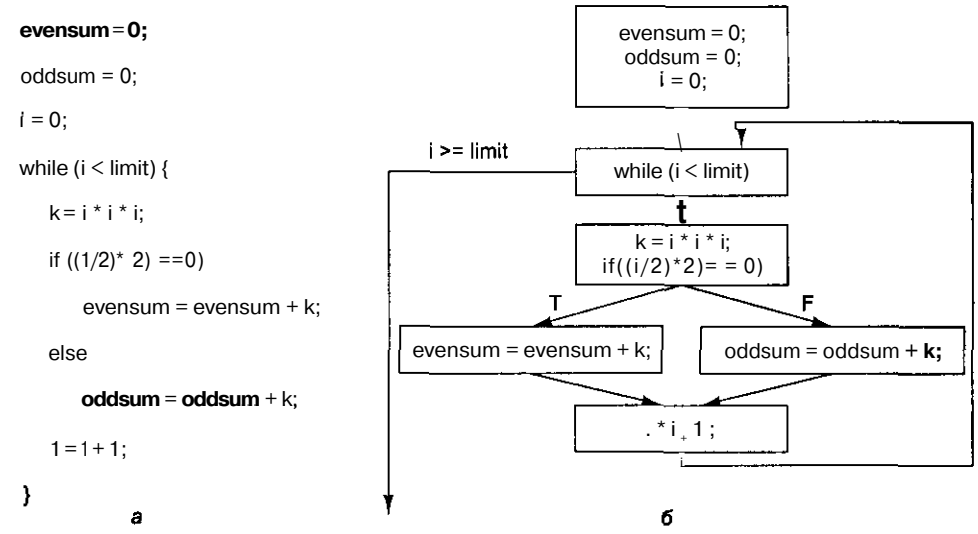


Рис. 4.30. Граф базового элемента для фрагмента программы, приведенного в листинге 4.6

Посмотрите на рис. 4.30. Представим, что все переменные были помещены в регистры, кроме *evensum* и *oddsum* (из-за недостатка регистров). Тогда имело бы смысл переместить команды LOAD в начало цикла до вычисления переменной k,

чтобы выполнение этих команд началось раньше, а полученные результаты были бы доступны в тот момент, когда они понадобятся. Естественно, при каждой итерации требуется только одно значение, поэтому остальные команды LOAD будут отбрасываться, но если кэш-память и основная память конвейеризированы, то подобная процедура имеет смысл. Выполнение команды до того, как стало известно, понадобится ли вообще эта команда, называется **спекулятивным выполнением.** Чтобы использовать эту технологию, требуется поддержка компилятора, аппаратного обеспечения, а также некоторое усовершенствование архитектуры. В большинстве случаев переупорядочение команд за пределами одного базового элемента находится вне компетенции аппаратного обеспечения, поэтому компилятор должен перемещать команды явным образом.

В связи со спекулятивным выполнением команд возникают некоторые интересные проблемы. Например, очень важно, чтобы ни одна из спекулятивных команд не имела окончательного результата, который нельзя отменить, поскольку позднее может оказаться, что эти команды не нужно было выполнять Обратимся к листингу 4.6 и рис. 4 30. Очень удобно производить сложение, как только появляется значение **к** (даже до условною оператора if), но нежелательно сохранять результаты в памяти. Чтобы предотвратить перезапись регистров до того, как стало известно, полезны ли полученные результаты, нужно переименовать (подменить) все выходные регистры, которые используются спекулятивной командой. Как вы можете себе представить, счетчик обращений для отслеживания всего этого очень сложен, но при наличии соответствующего аппаратного обеспечения его вполне можно сделать.

Однако при наличии спекулятивных команд возникает еще одна проблема, которую нельзя решить путем подмены регистров А что происходит, если спекулятивная команда вызывает исключение (exception)? В качестве примера можно привести команду LOAD, которая вызывает промах кэш-памяти в компьютере с достаточно большим размером строки кэш-памяти (скажем, 256 байт) и памятью, которая работает гораздо медленнее, чем центральный процессор и кэш. Если нам требуется команда **LOAD** и работа машины останавливается на много циклов, на то время, пока загружается строка кэш-памяти, то это не так страшно, поскольку данное слово действительно нужно Но если машина простаивает, чтобы вызвать слово, которое, как окажется позднее, совершенно ни к чему, это совершенно не рационально. Если подобных «оптимизаций» слишком много, то центральный процессор будет работать медленнее, чем если бы этих «оптимизаций» вообще не было. (Если машина содержит виртуальную память, которая обсуждается в главе 6, то спекулятивное выполнение команды LOAD может даже вызвать обращение к отсутствующей странице Подобные ошибки могут сильно повлиять на производительность, поэтому важно их избегать)

В ряде современных компьютеров данная проблема решается следующим образом. В них содержится специальная команда SPECULATIVE-LOAD, которая производит попытку вызвать слово из кэш-памяти, а если слова там нет, просто прекращает вызов. Если значение находится там и если в данный момент оно действительно требуется, его можно использовать, но если оно в данный момент не требуется, аппаратное обеспечение должно сразу получить это значение. А если окажется, что данное значение нам не нужно, то никаких потерь не будет.

Более сложную ситуацию можно проиллюстрировать следующим выражением:

if (x>0) **z-y/x;**

где x, y и z — переменные с плавающей точкой. Предположим, что все эти переменные поступают в регистры заранее и что команда деления с плавающей точкой (эта команда выполняется медленно) перемещается вверх и выполняется еще до условного оператора if. К сожалению, если x равен 0, то программа завершается в результате попытки деления на 0. Таким образом, спекулятивная команда приводит к сбою в изначально правильной программе. Еще хуже то, что программист изменяет программу, чтобы предотвратить подобную ситуацию, но сбой все равно происходит.

Одно из возможных решений — специальные версии команд, которые могут вызвать исключения (exceptions). Кроме того, к каждому регистру добавляется специальный бит **(poison bit).** Если спекулятивная команда дает сбой, она не вызывает trap (ловушку), а устанавливает бит присутствия в регистр результатов. Если этот регистр позднее используется обычной командой, происходит trap (как и должно быть). Однако если этот результат не используется, бит присутствия сбрасывается и не причиняет программе никакого вреда.

Примеры микроархитектурного уровня

В этом разделе мы рассмотрим три современных процессора в свете понятий, изученных в этой главе. Наше изложение будет кратким, поскольку компьютеры чрезвычайно сложны, содержат миллионы вентилей и у нас нет возможности давать подробное описание. Примеры будут те же, которые мы использовали до сих пор; Pentium II, UltraSPARC II и picojava П.

Микроархитектура процессора Pentium II

Pentium II — один из процессоров семейства Intel. Он поддерживает 32-битные операнды и арифметику, 64-битные операции с плавающей точкой, а также 8- и 16-битные операнды и операции, которые унаследованы от предыдущих процессоров данного семейства. Процессор может адресовать до 64 Гбайт памяти и считывать слова из памяти по 64 бита за раз. Обычная система Pentium II изображена на рис. 3.47.

Как мы уже говорили раньше и как показано на рис. 3.40, картридж с однорядным расположением контактов (SEC) системы Pentium II состоит из двух интегральных схем: центрального процессора (на котором находится разделенная кэш-память первого уровня) и объединенной кэш-памяти второго уровня. На рис. 4.31 показаны основные компоненты центрального процессора: блок вызова/декодирования, блок отправки/выполнения и блок возврата, которые вместе действуют как конвейер высокого уровня. Эти три блока обмениваются данными через пул команд — место для хранения информации о частично выполненных командах. Информация в пуле команд находится в таблице, которая называется **ROB (ReOrder Buffer — буфер переупорядочивания команд).** Если излагать кратко,

блок вызова/декодирования вызывает команды и разбивает их на микрооперации для хранения в ROB. Блок отправки выполнения получает микрооперации из буфера ROB и выполняет их. Блок возврата завершает выполнение каждой операции и обновляет регистры. Команды поступают в буфер ROB по порядку, могут выполняться в произвольном порядке, но завершаться опять должны по порядку.

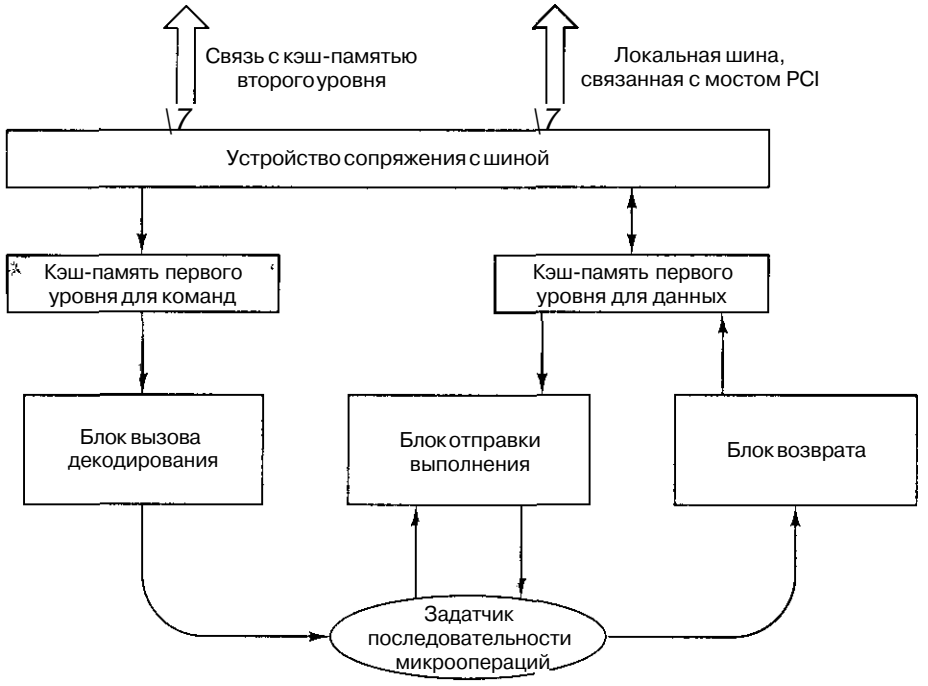


Рис. 4.31. Микроархитектура Pentium И

Блок сопряжения с шиной отвечает за обмен информацией с системой памяти (и с кэш-памятью второго уровня, и с основной памятью). Кэш-память второго уровня не связана с локальной шиной, поэтому блок сопряжения с шиной отвечает за вызов данных из основной памяти через локальную шину, а также за загрузку всех блоков кэш-памяти. Система Pentium II использует протокол синхронизации кэш-памяти MESI, который мы будем рассматривать, когда дойдем до мультипроцессоров в разделе «Архитектуры UMA SMP с шинной организацией» главы 8.

Блок вызова/декодирования

Блок вызова/декодирования отличается высокой степенью конвейеризации (содержит семь стадий). На рис. 4.32 эти семь стадий обозначены IFU0,..., ROB. Блок отправки/выполнения и блок возврата имеют еще пять стадий, то есть всего стадий 12. Команды поступают на конвейер на стадии IFUO (IFU — аббревиатура от Instruction Fetch Unit — блок выборки команд), куда из кэша команд загружаются целые 32-байтные строки. Всякий раз, когда внутренний буфер пуст, туда копируется следующая строка кэш-памяти. Регистр NEXT IP (NEXT Instruction Pointer — следующий указатель команды) управляет процессом вызова команд.

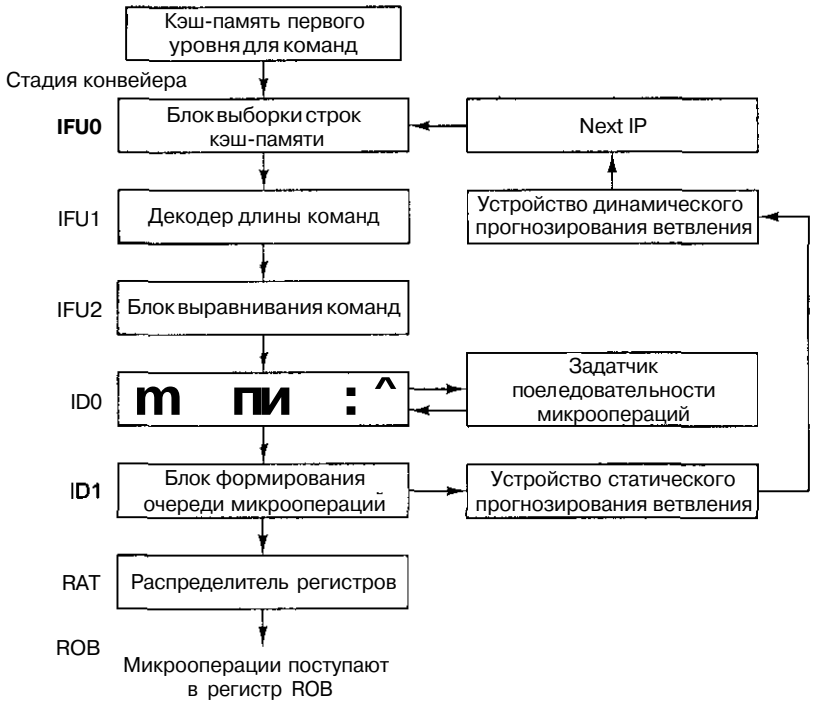


Рис.4.32. Внутренняя структура блока вызова/декодирования (в упрощенном виде)

Поскольку в наборе команд Intel, который часто называют IA-32 (32-разрядная архитектура для процессоров Intel), содержатся команды разной длины и различного формата, на следующей стадии, IFU1, происходит анализ потока байтов, чтобы определить начало каждой команды. В случае необходимости на стадии IFU1 может рассматриваться до 30 команд архитектуры IA-32 вперед. К сожалению, вследствие этого обычно встречаются 4 или 5 условных переходов, не все из которых правильно прогнозируются, поэтому в обработке такого большого количества команд заранее нет особого смысла. На стадии IFU2 команды выравниваются, поэтому в следующей стадии они без труда декодируются.

Декодирование начинается на стадии Ш0 (Instruction Decoding — декодирование команды). Декодирование в системе Pentium II состоит из превращения каждой команды IA-32 в одну или несколько микроопераций, как и в микроархитектуре Mic-4. Простые команды, например перемещение из одного регистра в другой, переделываются в одну микрооперацию. Выполнение более сложных команд может занимать до четырех микроопераций. Несколько чрезвычайно сложных команд требуют еще больше микроопераций и используют ПЗУ последовательности микроопераций для упорядочения этих микроопераций.

На стадии ID0 имеется три внутренних декодера. Два из них предназначены для простых команд, а третий обрабатывает остальные команды. На выходе получается последовательность микроопераций. Каждая микрооперация содержит код операции, два входных и один выходной регистр.

Очередь микрокоманд выстраивается на стадии ID1. Этот блок аналогичен блоку формирования очереди, изображенному на рис. 4.23. На этой стадии также про-

исходит прогнозирование ветвления (сначала статическое, на всякий случай). Прогноз зависит от нескольких факторов, но для переходов, связанных с текущей командой, считается, что переходы назад будут производиться, а переходы вперед — нет Затем идет динамическое прогнозирование с использованием специального алгоритма, как показано на рис. 4 29, только в данном случае для прогнозирования используется не два, а четыре бита. Должно быть ясно, что если речь идет о конвейере с 12 стадиями, очень велика вероятность неправильного предсказания, и поэтому нужно так много битов. Если перехода в таблице динамики нет, используется статическое прогнозирование.

Чтобы избежать взаимозависимостей WAR и WAW, система Pentium II поддерживает переименования (подмены), как мы видели в табл. 4.13. Реальные регистры в командах IA-32 могут быть заменены в микрооперациях любым из 40 внутренних временных регистров, находящихся в буфере ROB. Подмена происходит на стадии RAT.

И наконец, микрооперации копируются в буфер ROB со скоростью три микрооперации за цикл. Сюда же собираются операнды, если они имеются в наличии. Если операнды микрооперации и регистр результатов доступны, а операционный блок свободен, микрооперацию можно выпустить. В противном случае она находится в буфере ROB, пока не появятся все необходимые ресурсы.

Блок отправки/выполнения

Перейдем к блоку отправки/выполнения, который изображен па рис. 4.33. Этот блок устанавливает очередность и выполняет микрооперации, разрешает взаимозависимости и конфликты ресурсов. Хотя за один цикл можно декодировать всего три команды (на стадии ID0), за один цикл можно выпустить для выполнения целых пять микроопераций, по одной на каждый порт. Такую скорость нельзя поддерживать, поскольку она превышает способности работы блока возврата. Микрооперации могут запускаться не по порядку, но блок возврата должен завершать их выполнение по порядку. Чтобы следить за микрооперациями, регистрами и функциональными блоками, требуется сложный счетчик обращений. Когда операция готова для выполнения, она может начаться, даже если другие операции, которые поступили в буфер ROB раньше нее, еще не готовы, Если несколько микроопераций пригодны для выполнения одним и тем же функциональным блоком, с помощью сложного алгоритма выбирается важнейшая из них, и именно она и запускается следующей Например, выполнение перехода гораздо важнее, чем выполнение арифметического действия, поскольку первый из них влияет на ход программы.

Блок отправки/выполнения состоит из резервации и функциональных блоков, которые связаны с пятью портами Резервация представляет собой очередь из 20 элементов для микроопераций, которые имеют собственные операнды. Они ожидают своей очереди в резервации, пока не освободится нужный функциональный блок.

Между резервацией и функциональными блоками имеется пять портов. Некоторые функциональные блоки разделяют один порт, как показано на рисунке. Блок загрузки и блоки запоминающих устройств выдают информацию для операций загрузки и сохранения соответственно Для запоминающих устройств есть два порта. Поскольку через один порт за один цикл может выдаваться только одна микрооперация, то если двум микрооперациям нужно пройти через один и тот же порт, одной из них придется подождать.

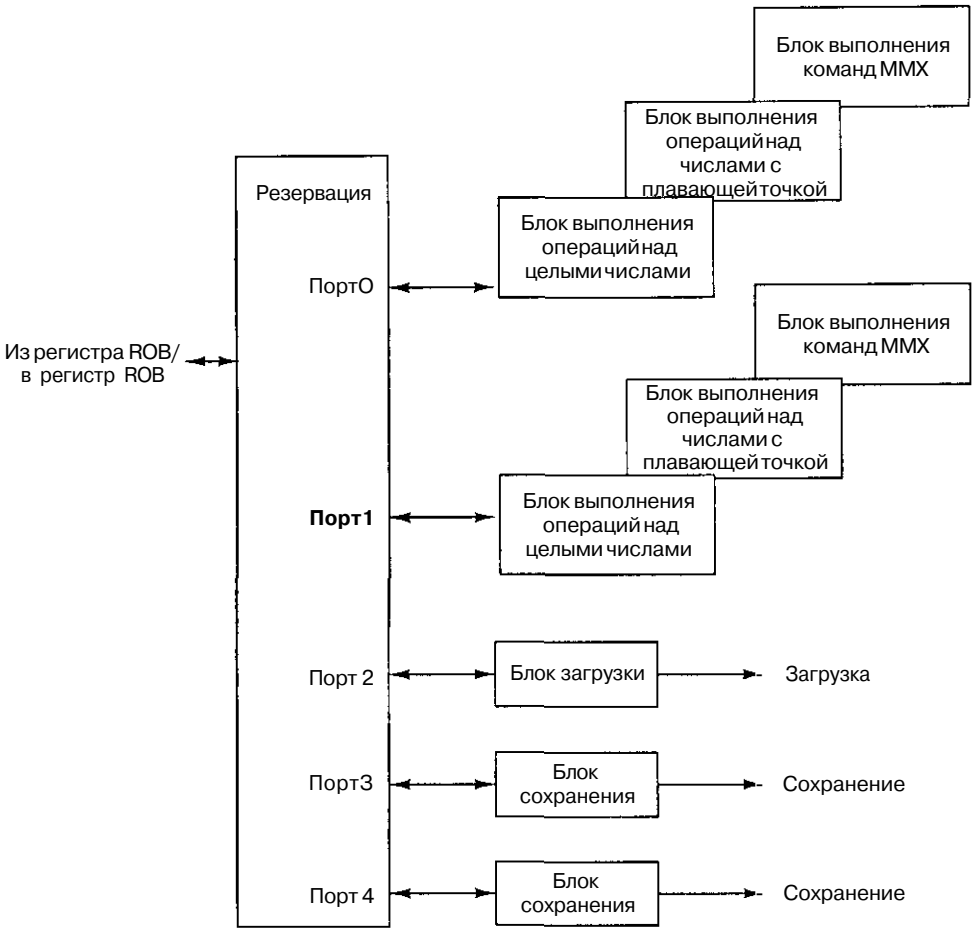


Рис. 4.33. Блокотправки/выполнения

Блоквозврата

Когда микрооперация выполнена, она переходит обратно в резервацию, а затем в буфер ROB, и там ожидает возврата. Блок возврата отвечает за отправку результатов в нужные места — в соответствующий регистр или в другие устройства блока отправки/выполнения, которым требуется данное значение. Блок отправки/выполнения содержит «официальные» регистры, то есть те, в которых хранятся значения завершенных команд. Блок возврата содержит ряд «промежуточных» регистров, значения которых были вычислены командой, которая еще не завершилась, поскольку выполнение предыдущих команд не закончилось.

Система Pentium II поддерживает процедуру спекулятивного выполнения, поэтому некоторые команды будут выполняться напрасно, и их результаты никуда не нужно будет сбрасывать. Именно поэтому и нужна способность возвращаться в предыдущее состояние. Если стало известно, что какая-то микрооперация пришла из команды, которую не нужно было выполнять, результаты этой микрооперации

отбрасываются. Все это контролирует блок возврата. Только результаты «официально» выполненных команд могут возвращаться в регистры, причем это должно происходить в том же порядке, что и в программе, даже если команды выполнялись в произвольном порядке.

Микроархитектура процессора UltraSPARC II

Серия UltraSPARC, произведенная компанией Sun, — это реализация версии 9 архитектуры SPARC. Все модели сходны друг с другом и различаются главным образом производительностью и ценой. Тем не менее, чтобы избежать путаницы, в этом разделе мы будем говорить о системе UltraSPARC II и описывать те характеристики, по которым этот процессор отличается от других процессоров того же семейства.

UltraSPARC II — это 64-разрядная машина с 64-разрядными регистрами и 64-разрядным трактом данных, но в целях совместимости с машинами версии 8 (которые являются 32-разрядными) она может обращаться с 32-разрядными операндами, а программное обеспечение, написанное для 32-разрядных версий SPARC, изменять не нужно. Хотя внутренняя архитектура машины использует 64 разряда, ширина шины памяти составляет 128 битов, аналогично процессору Pentium II с 32-разрядной архитектурой и 64-разрядной шиной памяти. Ядро системы UltraSPARC II показано на рис. 3.44.

Вся серия SPARC с самого начала представляла собой систему RISC. У большинства команд есть два входных и один выходной регистр, поэтому они хорошо подходят для конвейерного выполнения в одном цикле. Разбивать старые команды CISC на микрооперации RISC, как в системе Pentium II, не нужно.

UltraSPARC II — это суперскалярная машина, которая может выдавать 4 команды за цикл. Команды запускаются по порядку, но завершаться могут и в произвольном порядке. Тем не менее прерывания являются точными (то есть всегда точно известно, в каком месте программы была машина, когда произошло прерывание). Существует аппаратная поддержка для спекулятивных загрузок в виде команды PREFETCH, которая не вызывает ошибок при промахе кэша. Она даже не блокирует последовательные обращения к памяти. Следовательно, компилятор может вставлять одну или несколько команд PREFETCH задолго до того, как они понадобятся, в надежде, что они пригодятся позже, не испытывая никаких неприятностей в случае, если нужного слово не окажется в кэш-памяти,

Общий обзор системы UltraSPARC II

Диаграмма UltraSPARC II представлена на рис. 4.34. Все указанные компоненты расположены на микросхеме центрального процессора. Исключение составляет кэш-память второго уровня, которая является внешней по отношению к процессору. Кэш команд — это 16 Кбайт двувходовой ассоциативной кэш-памяти, со строками по 32 байта и с возможностью возврата половины строки. Половина строки кэш-памяти (16 байтов) содержит ровно четыре команды, и все эти четыре команды могут выдаваться за один цикл. Кэш-память данных — это 16 Кбайт кэш-памяти прямого отображения со сквозной записью и без заполнения по записи. Здесь тоже используются 32-байтные строки, разделенные на 2 части по 16 байтов.

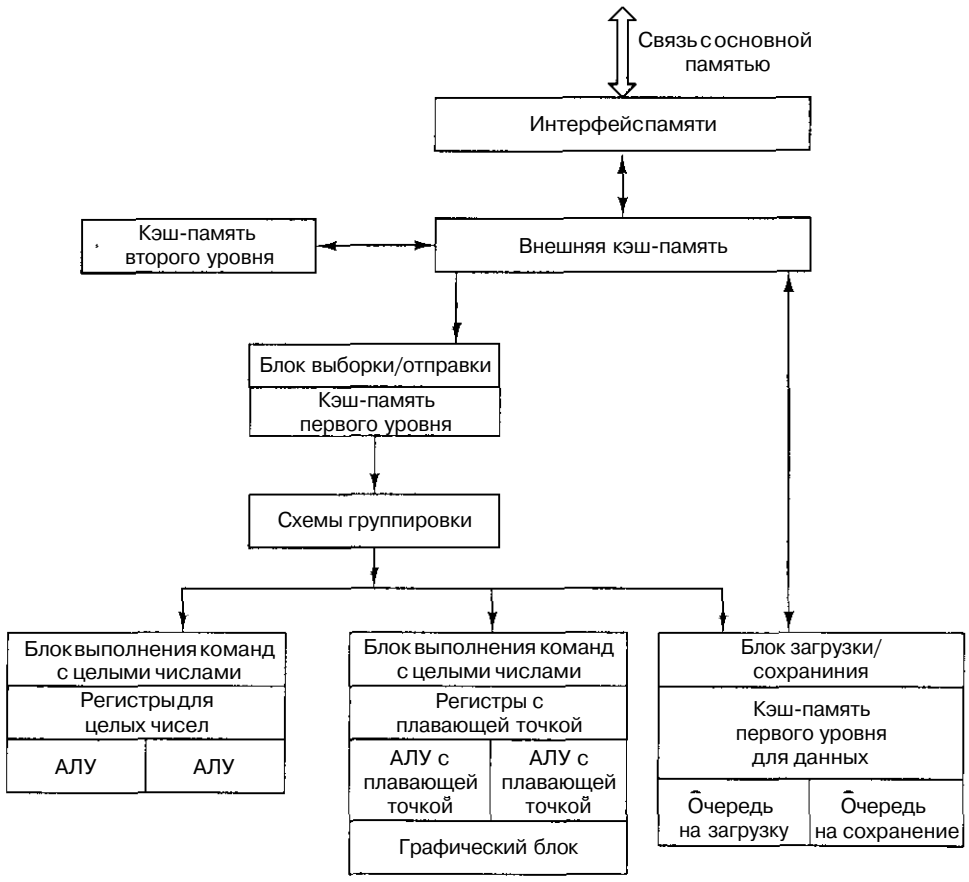


Рис. 4.34. Микроархитектура UltraSPARC II

В случае промаха кэш-памяти первого уровня нужная строка ищется в кэш-памяти второго уровня. Если поиск завершился успехом, строка копируется в кэш-память первого уровня. В случае неудачи внешняя кэш-память (кэш-память второго уровня) посылает устройству сопряжения с памятью команду вызова строки из основной памяти.

Рассмотрим функциональные блоки системы UltraSPARC II. Блок выборки отправки в целом похож на блок вызова/декодирования в системе Pentium II (см. рис. 4.31). Однако у этого блока работа проще, поскольку входные команды уже представлены в трех регистрах в виде микроопераций, поэтому их не нужно разбивать. Блок может вызывать команды и из кэш-памяти первого уровня, и из кэш-памяти второго уровня без потери времени. За один цикл вызываются четыре команды.

Чтобы сократить неприятные последствия неправильно предсказанных переходов, каждая группа из четырех команд в кэш-памяти команд содержит адрес, который указывает, какую именно половинчатую строку нужно взять следующей. Кроме того, предсказывающее устройство находится внутри блока выборки отправки.

При этом используется 2-битный алгоритм прогнозирования, сходный с тем, который показан на рис. 4.29. Более того, UltraSPARC II содержит ряд команд перехода, в которых компилятор может сообщать аппаратному обеспечению, каким именно способом предсказывать переход. Вызванные заранее команды помещаются в очередь из 12 элементов, а затем передаются в схему группировки.

Схема группировки — это блок, который выбирает по четыре команды за один раз из очереди для запуска. Задача состоит в том, чтобы найти 4 команды, которые можно выпустить одновременно. Блок целых чисел содержит два раздельных АЛУ, что позволяет выполнять две команды параллельно. Блок вычислений с плавающей точкой также содержит два АЛУ. Следовательно, в одной группе может находиться по две команды каждого типа, но не четыре команды одного типа. Чтобы сделать группирование оптимальным, команды должны запускаться не по порядку, что разрешается Завершаются они также в произвольном порядке.

Блок целых чисел и блок вычислений с плавающей точкой содержат собственные регистры, поэтому команды берут операнды прямо внутри блока и там же оставляют результаты. Регистр целых чисел и регистр с плавающей точкой разделены, поэтому значения никогда не переходят из одного блока в другой. В блоке вычислений с плавающей точкой также находится графический блок, который выполняет специальные команды для двух- и трехмерных изображений, аудио и видео, аналогичные командам MMX в системе Pentium II.

Блок загрузки/сохранения управляет командами LOAD и STORE. Если они имеются в кэш-памяти данных первого уровня, то выполняются без задержки. В противном случае нужная строка берется из кэш-памяти второго уровня или основной памяти (если речь идет о команде LOAD) или нужное слово записывается туда (если речь идет о команде STORE). Если бы кэш-память данных была write-allocate, тогда в случае промаха кэш-памяти при записи (команда STORE) нужная строка переносилась бы из кэш-памяти второго уровня или из основной памяти. На самом деле, если нужно сохранить отдельное слово, просто осуществляется сквозная запись в кэш-память второго уровня, а в случае промаха — в основную память. Чтобы избежать блокирования из-за отсутствия нужного слова в кэш-памяти данных, блок загрузки/сохранения хранит очереди незавершенных команд LOAD и STORE, поэтому можно продолжать обрабатывать новые команды, пока завершается выполнение старых.

Конвейеризация системы UltraSPARC II

Система UltraSPARC II содержит конвейер с 9 стадиями. Некоторые из этих стадий различны для команд с целыми числами и команд с плавающей точкой (рис. 4.35). На первой стадии вызываются команды из кэш-памяти команд (если это возможно). При благоприятных обстоятельствах (отсутствии промахов кэш-памяти, неправильного прогнозирования ветвлений, сложных команд, наличии правильной смеси команд и т. п.) машина может продолжать вызывать и запускать по 4 команды за цикл. На стадии декодирования перед копированием команд в очередь к каждой команде прибавляются дополнительные биты. Эти биты ускоряют последующую обработку (например, сразу отправляя команду в соответствующий функциональный блок).

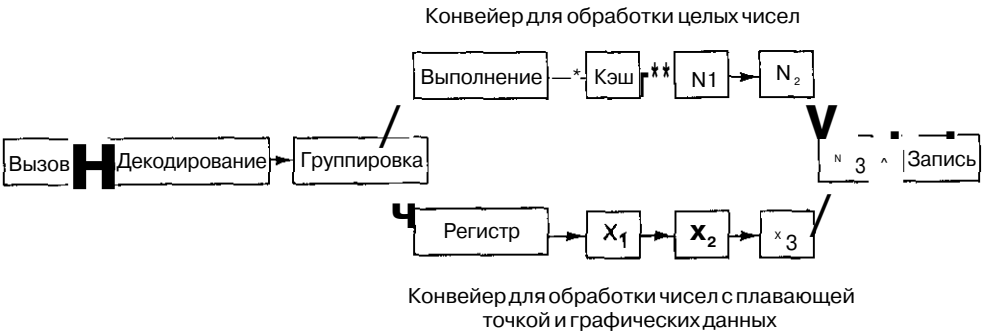


Рис. 4.35. Конвейер системы UltraSPARC II

Стадия группировки соответствует схеме группировки, которую мы рассматривали раньше. На этой стадии декодированные команды объединяются в группы. В каждой группе находится по 4 команды. Все команды одной группы должны быть подобраны таким образом, чтобы их можно было выполнять одновременно.

С этого момента стадии конвейера для операций с целыми числами и для операций с плавающей точкой разделяются. На стадии 4 в блоке целых чисел большинство команд выполняется прямо за один цикл. Однако команды STORE и LOAD требуют дополнительной обработки на стадии кэширования. На стадиях Ni и N_2 не производится никаких действий для команд, но эти стадии нужны для синхронизации работы двух конвейеров. Если каждая команда с целыми числами будет завершаться на несколько секунд позже, это не такая уж большая потеря, зато конвейер работает равномерно.

Блок с плавающей точкой содержит отдельные 4 стадии. Первая нужна для доступа к регистрам с плавающей точкой. Следующие три нужны для выполнения команды. Все команды с плавающей точкой выполняются за три цикла, за исключением деления (на эту операцию требуется 12 циклов) и квадратного корня (здесь нужно 22 цикла), поэтому длинная последовательность других команд не снижает скорости работы конвейера.

Стадия N_3, общая для обоих блоков, нужна для разрешения исключительных ситуаций, например деления на нуль. Наконец, на последней стадии результаты записываются обратно в регистры. Эта стадия напоминает блок возврата системы Pentium II в том, что если команда прошла через эту стадию, она завершена.

Микроархитектура процессора picoJava II

В системе picoJava II двоичные программы JVM могут работать практически без интерпретации. Большинство команд JVM выполняются непосредственно аппаратным обеспечением за один цикл. Около 30 команд JVM являются микропрограммными. Только очень небольшое число команд не может выполняться аппаратным обеспечением picoJava II и вызывает traps (ловушки). Эти команды связаны с особенностями JVM, которые мы не обсуждали, например создание и управление сложными программными объектами.

Общий обзор системы picoJava II

Диаграмма микроархитектуры picoJava II представлена на рис. 4.36. Микросхема процессора содержит разделенную кэш-память первого уровня. Кэш-память команд факультативна. Ее объем может составлять 1 Кбайт, 2 Кбайт, 4 Кбайт, 8 Кбайт или 16 Кбайт. Это кэш-память прямого отображения. Размер строки составляет 16 байтов. Кэш-память данных тоже факультативна, и ее объем может составлять 1 Кбайт, 2 Кбайт, 4 Кбайт, 8 Кбайт или 16 Кбайт. Это двувходовая ассоциативная кэш-память. Размер строки также составляет 16 байтов. Она использует обратную запись и заполнение по записи. Каждая кэш-память соединяется с шиной памяти по 32-битному каналу. Система microjava 701 имеет оба блока кэш-памяти в обязательном порядке, объем каждого из них составляет 16 Кбайт. Факультативный блок с плавающей точкой также является частью разработки picoJava II.

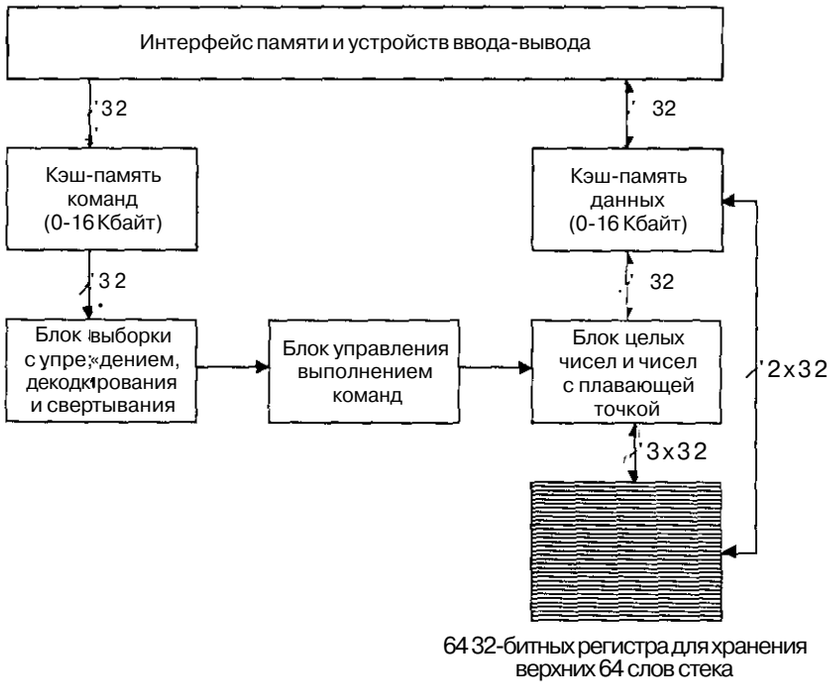


Рис. 4.36. Диаграмма системы picoJava II с кэш-памятью первого уровня и блоком с плавающей точкой. Это конфигурация системы microJava 701

Кэш-память команд передает в блок вызова, декодирования и свертывания по 8 байтов за раз. Этот блок, в свою очередь, связан с контроллером выполнения и с основным трактом данных (блоком операций с целыми числами и с плавающей точкой). Ширина тракта данных составляет 32 бита для целочисленных операций. Этот тракт данных может также управляться с плавающей точкой с одинарной и двойной точностью (IEEE 754).

Наиболее интересная часть рис. 4.36 — это регистровый файл, состоящий из 64 32-битных регистров. В этих регистрах могут содержаться верхние 64 слова стека JVM, что сильно повышает скорость доступа к словам в стеке. И стек операндов,

и стек локальных переменных под ним могут находиться в регистровом файле, Доступ к регистровому файлу «свободный» (то есть происходит без задержек, тогда как доступ к кэш-памяти данных требует дополнительного цикла). Ширина канала между регистровым файлом и блоком операций с целыми числами и с плавающей точкой составляет 96 битов. За один цикл канал выдерживает 2 32-битных считывания из стека и одну 32-битную запись в стек.

Если, например, стек операндов состоит из двух слов, то в регистровом файле может находиться до 62 слов локальных переменных. Естественно, при помещении еще одного слова в стек возникает проблема. Происходит так называемый **дриббинг** — это когда одно или несколько слов, находящихся глубоко в стеке, записываются обратно в память. Точно так же, если несколько слов выталкиваются из стека операндов, в регистровом файле освобождается место, и поэтому некоторые слова, находящиеся глубоко в стеке, могут перезагружаться в регистровый файл. Специальные регистры на микросхеме определяют, насколько полным должен быть регистр, чтобы слова из нижней части стека записывались в память, и насколько пустым он может быть для того, чтобы перезагрузить регистровый файл из памяти. Чтобы легко произвести дриббинг без копирования, регистровый файл действует как кольцевой буфер с указателями на самое нижнее и на самое верхнее слова. Дриббинг происходит автоматически всякий раз, когда регистровый файл переполняется или пустеет.

Конвейер системы picoJava И

Конвейер системы picoJava II состоит из шести стадий. Он показан на рис. 4.37. На первой стадии из кэш-памяти команд в буфер команд вызываются команды по 8 байтов за раз. Емкость буфера команд составляет 16 байтов. На следующей стадии команды декодируются и определенным образом объединяются. На выходе из блока декодирования получается последовательность микроопераций, каждая из которых содержит код операции и три номера регистров (двух входных и одного выходного регистров). В этом отношении машина picoJava II сходна с Pentium II: обе машины получают поток команд CISC, который превращается в последовательность микроопераций RISC. Однако, в отличие от Pentium II, машина picoJava II не является суперскалярной и микрооперации выполняются и завершаются в том порядке, в котором они запускаются. В случае промаха кэш-памяти, если операнд приходится вызывать из основной памяти, процессор должен простаивать.

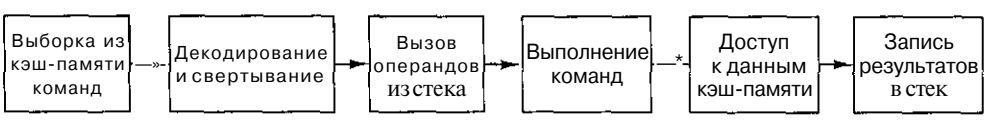


Рис. 4.37. Шесть стадий конвейера в машине picoJava II

На третьей стадии вызываются операнды из стека (фактически из регистрового файла), чтобы они были в наличии для четвертой стадии. Четвертая стадия — это блок выполнения команд. На пятой стадии в случае необходимости производится обращение к кэш-памяти данных (например, чтобы сохранить там результаты). Наконец, на шестой стадии результаты записываются обратно в стек.

Свертывание команд

Как мы упоминали выше, блок декодирования способен свертывать команды вместе. Чтобы объяснить, как происходит этот процесс, рассмотрим следующее выражение:

n=k+m.

Трансляция на (I)JVM может быть следующей:

```
ILOAD 7
ILOAD 1
IADD
    ISTORE 3,
```

Предполагается, что k, m и n — локальные переменные 7,1 и 3 соответственно. Процесс выполнения этих четырех команд изображен на рис. 4,38, *а*.

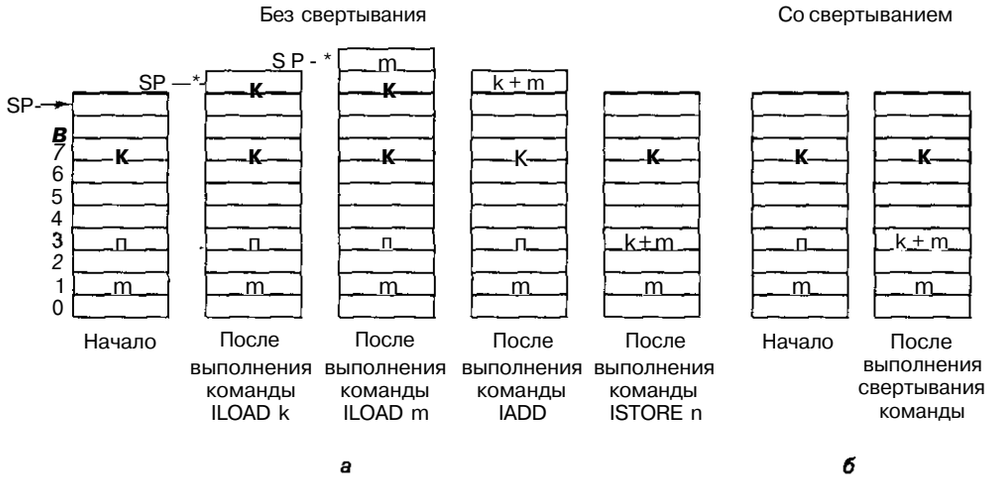


Рис. 4.38. Выполнение последовательности из четырех команд для вычисления выражения n=k+m (а); та же последовательность, свернутая до одной команды (б)

Если предположить, что все три переменные находятся достаточно высоко в стеке, настолько высоко, что все они содержатся в регистровом файле, то для выполнения этой последовательности команд вообще не требуются обращения к памяти. Первая команда ILOAD 7 копирует слово, находящееся в седьмой локальной переменной, в вершину стека и увеличивает указатель стека на 1. Сходным образом команда ILOAD 1 производит копирование из регистра в регистр. Команда IADD складывает два регистра, а команда ISTORE копирует значение регистра в регистровый файл. Избавление от любых обращений к памяти — главный способ улучшения производительности.

Однако машина picojava II делает не только это. Данная последовательность из четырех команд просто складывает два регистра и сохраняет полученное значение в третьем регистре. Блок декодирования определяет это условие и запускает одну микрооперацию: трехрегистровую команду ADD. Таким образом, вместо четырех команд JVM, для которых требуется 9 обращений к памяти, мы получаем одну

микрооперацию для сложения, как показано на рис. 4.38,*б*. И хотя на входе в конвейер были команды CISC с многочисленными обращениями к памяти, в результате выполнена была всего одна простая микрооперация. Таким образом, picojava II может выполнять программы на языке Java, скомпилированные для JVM, так же быстро, как будто они были скомпилированы на машинный язык компьютера RISC.

Как мы только что увидели, возможность сворачивать несколько команд JVM в одну микрооперацию является ключом к высокой производительности. Следовательно, стоит кратко изложить, как блок декодирования осуществляет свертывание. Для этого команды распределяются по шести группам (табл. 4.14). Первая группа содержит команды, которые не сворачиваются. Во второй находятся команды загрузки локальных переменных. BIJVM имеется одна такая команда, ILOAD, а JVM содержит и другие команды. Третья группа состоит из команд запоминания, например ISTORE. Четвертая и пятая группы предназначены для команд переходов с одним и двумя операндами соответственно. Последняя группа состоит из команд, которые выталкивают два операнда из стека, выполняют с ними какие-нибудь вычисления и помещают результат обратно в стек.

Таблица 4.14. Распределение команд JVM по группам для свертывания

Группа	Описание	Пример
NF	Несвертываемые команды	GOTO
LV	Помещают слово в стек	ILOAD
MEM	Выталкивают слово из стека	ISTORE
BG1	Операции с использованием одного стекового операнда	IFEQ
BG2	Операции с использованием двух стековых операндов	IF^CMPEQ
OP	Вычисления над двумя операндами с одним результатом	IADD

Блок декодирования передает 74-битные микрооперации операционному блоку (через блок вызова операндов). Большинство этих микроопераций содержат код операции и три регистра и могут быть выполнены за один цикл. Когда блок декодирования выталкивает команды JVM из буфера команд, он превращает их в последовательность микроопераций. Кроме того, блок декодирования определяет, какие последовательности команд JVM можно свернуть в одну микрооперацию. В такой последовательности может быть до четырех команд. Если обнаружена подходящая последовательность, выдается соответствующая микрооперация, а изначальные команды отбрасываются.

В табл. 4.15 приведены некоторые типичные последовательности команд JVM, которые можно свернуть. Когда в блоке декодирования оказывается одна из таких последовательностей, он замещает обычное разбиение команд на одну микрооперацию, которая выполняет работу всей этой последовательности за один цикл. Например, он превращает последовательность из четырех команд в одну трехрегистровую микрооперацию ADD, как мы видели на рис. 4.38. Процесс свертывания происходит только тогда, когда требуемые локальные переменные находятся достаточно близко от вершины стека, то есть содержатся в регистровом файле.

Таблица 4.15. Некоторые последовательности команд JVM, которые можно сворачивать

Последовательность команд				Пример
LV	LV	OP	MEM	ILOAD, ILOAD, IADD, ISTORE
LV	LV	OP		ILOAD, ILOAD, IADD
LV	LV	BG2		ILOAD, ILOAD, IF_CMPEQ
LV	BG1			ILOAD, IFEQ
LV	BG2			ILOAD, IF_CMPEQ
LV	MEM			ILOAD, ISTORE
OP	MEM			IADD, ISTORE

Свертывание команд происходит довольно часто, поэтому существенная часть программы JVM может быть выполнена настолько быстро, как будто она была скомпилирована прямо для конвейеризированного процессора RISC. Измерения показывают, что picojava II может выполнять программы на языке Java в пять раз быстрее, чем если те же программы скомпилировать на машинный язык для Pentium, который работает с такой же тактовой частотой, и в 15 раз быстрее, чем при интерпретируемом выполнении той же программы на машине Pentium.

В машине picojava II используется чрезвычайно примитивный алгоритм прогнозирования ветвлений: она всегда предсказывает, что перехода не будет. За этим стоит идея сохранить микросхему простой и дешевой, а не тратить существенное пространство микросхемы на схемы прогнозирования. Однако благодаря длине конвейера (6 стадий вместо 12, как в системе Pentium II) проигрыш при непраэдшдкил предсказании перехода составляет всего три цикла.

Сравнение Pentium, UltraSPARC и picojava

Данные три примера во многом отличаются друг от друга, однако у них есть удивительная общность, которая может сказать кое-что о том, как лучше разрабатывать компьютер. Машина Pentium II содержит старый набор команд CISC, который инженеры компании Intel были бы рады выкинуть в бухту Сан-Франциско, но тогда они нарушили бы законы о загрязнении воды. UltraSPARC II — система RISC. Picojava II — машина со стековой организацией и командами различной длины, которые совершают огромное число обращений к памяти.

Несмотря на эти различия, все три машины имеют сходные функциональные блоки. Все функциональные блоки принимают микрооперации, которые содержат код операции, два входных регистра и один выходной регистр. Все они могут выполнять микрооперацию за один цикл. Все они конвейеризированы и применяют прогнозирование ветвления. Все они содержат разделенную кэш-память для команд и для данных, объем каждой составляет 16 Кбайт.

Такое внутреннее сходство не случайно, и причиной его является вовсе не постоянные переходы с одной работы на другую инженеров Силиконовой долины. Когда мы рассматривали микроархитектуры Mic-3 и Mic-4, мы увидели, что достаточно просто построить конвейеризированный тракт данных, который исполь-

зует два регистра в качестве источников, пропускает значения этих регистров через АЛУ и сохраняет результат в регистре. На рисунке 4.22 представлено графическое изображение такого конвейера. Для современной техники это наиболее эффективная разработка.

Главное различие между Pentium II, UltraSPARC II и picojava II — переход от набора команд к функциональному блоку. Компьютеру Pentium II приходится разбивать команды CISC, чтобы переделать их в трехрегистровый формат, который нужен для функционального блока. Именно этот процесс показан на рис. 4.32 — разбиение больших команд на маленькие микрооперации. У машины picojava II обратная проблема — как скомбинировать несколько команд вместе, чтобы получить простую микрооперацию. Такой процесс называется свертыванием. Машине UltraSPARC II вообще не нужно ничего делать, поскольку ее первоначальные команды уже представляют собой маленькие удобные микрооперации. Вот почему большинство новых архитектур команд — архитектуры типа RISC, если, конечно, у них нет какого-нибудь скрытого мотива (например, вызов программ на языке Java через Интернет и их выполнение на произвольной машине).

Полезно будет сравнить нашу последнюю разработку, микроархитектуру Mic-4, с этими тремя реальными машинами. Mic-4 больше всего похожа на Pentium II. Обе системы интерпретируют команды, которые не являются командами типа RISC. Для этого обе системы разбивают команды на микрооперации с кодом операции, двумя входными регистрами и одним выходным регистром. В обоих случаях помещаются в очередь для дальнейшего выполнения. Микроархитектура Mic-4 запускает микрооперации строго по порядку, выполняет их строго по порядку и завершает выполнение тоже строго по порядку, а Pentium II запускает по порядку, выполняет в произвольном порядке, а завершает опять по порядку. Кроме того, внутренняя структура конвейера Pentium II, особенно та его часть, которая показана на рис. 4.32, во многом сходна с Mic-4.

А теперь сравним Mic-4 с picojava П. Хотя кажется, что эти две архитектуры должны быть похожи по логике вещей — они интерпретируют один и тот же набор команд, но на самом деле это не совсем так. Причина этого различия состоит в том, что блоки декодирования имеют диаметрально противоположные стратегии. Mic-4 берет каждую входящую команду IJVM и сразу разбивает ее на микрооперации, а picojava II пытается соединить (свернуть) несколько команд IJVM в одну микрооперацию. Простая операция присваивания i=j+k занимает 14 циклов на Mic-4 и 1 цикл на picojava II. В данном случае свертывание улучшает производительность в 14 раз. Очевидно, что второй метод ведет к более быстрому выполнению команд, но сложность процесса свертывания тоже существенна.

Mic-4 и UltraSPARC II вообще нельзя сравнивать, поскольку команды системы UltraSPARC II — это команды RISC (то есть трехрегистровые микрооперации). Их не нужно ни разбивать, ни объединять. Их можно выполнять в том виде, в каком они есть, каждую за один цикл тракта данных.

Все четыре машины конвейеризированы. Pentium II имеет 12 стадий, UltraSPARC II — 9 стадий, picojava II — 6 стадий, Mic-4 — 7 стадий. У Pentium II больше стадий, поскольку этой машине приходится разбивать сложные команды. UltraSPARC II содержит больше стадий, чем ему нужно, поскольку конвейер для целочисленных вычислений был искусственно удлинен на 2 стадии, чтобы опера-

ции с целыми числами занимали столько же времени, сколько занимают операции с плавающей точкой. Отсюда следует вывод: при современном состоянии техники оптимальным является конвейер с шестью или семью стадиями, который обрабатывает трехрегистровые микрооперации. Микроархитектура Mic-4 дает хорошее представление о том, как работает такой конвейер (по крайней мере, с командами, которые не являются командами типа RISC).

Краткое содержание главы

Основным компонентом любого компьютера является тракт данных. Он содержит несколько регистров, две или три шины, один или несколько функциональных блоков, например АЛУ, и схему сдвига. Основной цикл состоит из вызова нескольких операндов из регистров и их передачи по шинам к АЛУ и другому функциональному блоку. После выполнения операции результаты сохраняются опять в регистрах.

Тракт данных может управляться задатчиком последовательности, который вызывает микрокоманды из управляющей памяти. Каждая микрокоманда содержит биты, управляющие трактом данных в течение одног о цикла. Эти биты определяют, какие операнды нужно выбирать, какую операцию нужно выполнять и что нужно делать с результатами. Кроме того, каждая микрокоманда определяет своего последователя (обычно в ней содержится адрес следующей микрокоманды). Некоторые микрокоманды изменяют этот базовый адрес с помощью операции ИЛИ

IJVM — это машина со стековой организацией и с 1-байтными кодами операций, которые помещают слова в стек, выталкивают слова из стека и выполняют различные операции над словами из стека (например, складывают их) В главе приводится микропрограмма для микроархитектуры Mic-1. Если добавить блок выборки команд для загрузки команд из потока байтов, то можно устранить большое количество обращений к счетчику команд, и тогда скорость работы машины сильно повысится,

Существует множество способов разработки микроархитектурного уровня Есть много различных вариантов[1] двухшинная архитектура — трехшинная архитектура, кодированные поля микрокоманды — декодированные поля микрокоманды, наличие или отсутствие вызова с упреждением и многие другие Mic-1 — это простая машина с программным управлением, последовательным выполнением команд и полным отсутствием параллелизма. Mic-4, напротив, является высокопараллельной микроархитектурой с конвейером с семью стадиями.

Производительность компьютера можно повысить несколькими способами. Главный способ — использование кэш-памяти. Кэш-память прямого отображения и ассоциативная кэш-память с множественным доступом широко используются для того, чтобы ускорить обращения к памяти. Кроме того, применяется прогнозирование ветвления (как статическое, так и динамическое), исполнение с изменением последовательности и спекулятивное выполнение команд.

Наши три примера, Pentium II, UltraSPARC II и picojava II, во многом отличаются друг от друга, но при этом удивительно похожи в плане выполнения команд.

Pentium II берет команды CISC и разбивает их на микрооперации, которые обрабатываются суперскалярной архитектурой с прогнозированием ветвления, изменением последовательности команд и спекулятивным выполнением. UltraSPARC II — это современный 64-разрядный процессор с командами типа RISC. Здесь тоже используется прогнозирование ветвления, исполнение с изменением последовательности и спекулятивные команды. Picojavall представляет собой более простой процессор, предназначенный для дешевых устройств, поэтому у него нет таких особенностей, как динамическое прогнозирование ветвления. Однако при применении свертывания команд эта машина способна выполнять команды JVM достаточно быстро, как будто это регистровые команды RISC. Все три машины содержат сходные функциональные блоки, которые обрабатывают трехрегистровые микрооперации, когда они проходят через конвейер.

Вопросы и задания

1. В табл. 4.1 показан один из способов получения результата *Л* на выходе из АЛУ. Приведите другой способ.
2. В микроархитектуре Mic-1 требуется 1 не на установку регистра MIR, 1 не — на передачу значения регистра на шину B, 3 не — на запуск АЛУ и схемы сдвига и 1 не — на передачу результатов обратно в регистры. Длительность синхронизирующего импульса составляет 2 не. Может ли такая машина работать с частотой 100 МГц? А 150 МГц?
3. На рис. 4.5 регистр шины B закодирован в 4-битном поле, а шина C представлена в виде битового отображения. Почему?
4. На рис. 4.5 есть блок «Старший бит». Нарисуйте его схему.
5. Когда в микрокоманде установлено поле JMPC, регистр MBR соединяется операцией ИЛИ с полем NEXT_ADDRESS, чтобы получить адрес следующей микрокоманды. Существуют ли такие обстоятельства, при которых имеет смысл использовать JMPC, если NEXT_ADDRESS — OxlFF?
6. Предположим, что в примере, приведенном в листинге 4.1, выражение i-0: добавляется после условного оператора. Каким будет новый код ассемблера? Предполагается, что компилятор является оптимизирующим.
7. Напишите две трансляции IJVM для следующего высказывания на языке Java: i=j+k+4;
8. Напишите на языке Java выражение, которое произвело следующую программу IJVM:

```
ILOAD j
ILOAD k
ISUB
BIPUSH 6
ISUB
DUP
IADD
ISTORE i
```

9. В этой главе мы упомянули, что во время трансляции выражения
   if Ш goto LI; else goto L2
   в двоичную форму *L2* должно находиться среди младших 256 слов управляющей памяти. А возможно ли иметь *L1,* скажем, в ячейке с адресом 0x40, а *L2* — в ячейке с адресом 0x140? Объясните, почему.
10. В микропрограмме для Mic-1 в микрокоманде if_cmpeq3 значение регистра MDR копируется в регистр H, а в следующей строке от него отнимается значение регистра TOS. Казалось бы, это удобнее записать в одном высказывании:
    if_cmpeq3 Z-MDR-TOS. rd
    Почему этого не делают?
11. Сколько времени потребуется машине Mic-1, которая работает с частотой 200 МГц, на выполнение следующего высказывания на языке Java: i=j+k; Ответ дайте в наносекундах.
12. Тот же вопрос, что и предыдущий, только для машины Mic-2 с частотой 200 МГц. Опираясь на это вычисление, ответьте, сколько времени займет выполнение программы на машине Mic-2, если эта программа выполняется на машине Mic-1 за 100 не?
13. На машине JVM существуют специальные 1-байтные коды операций для загрузки в стек локальных переменных от 0 до 3, которые используются вместо обычной команды ILOAD. Какие изменения нужно внести в машину IJVM, чтобы наилучшим образом использовать эти команды?
14. Команда ISHR (целочисленный арифметический сдвиг вправо) есть в машине JVM, но ее нет в машине IJVM. Команда берет два верхних слова стека и заменяет их одним словом (результатом). Второе сверху слово стека — это операнд, который нужно сдвинуть. Он сдвигается вправо на значение от 0 до 31 включительно, в зависимости от значения пяти самых младших битов верхнего слова в стеке (остальные 27 битов игнорируются). Знаковый бит дублируется вправо на столько же битов, на сколько осуществляется сдвиг. Код операции для команды ISHR 122 (0x7A).
    1. Какая арифметическая операция эквивалентна сдвигу вправо на 2?
    2. Расширьте систему микрокоманд, чтобы включить эту команду в IJVM.
15. Команда ISHR (целочисленный сдвиг влево) имеется в JVM, но отсутствует в IJVM. Команда берет два верхних слова стека и замещает их одним значением (результатом). Второе сверху слово в стеке — операнд, который нужно сдвинуть. Он сдвигается влево на значение от 0 до 31 включительно, в зависимости от значения пяти младших бит верхнего слова в стеке (остальные 2 бита верхнего слова игнорируются). Нули сдвигаются влево на столько же битов, на сколько осуществляется сдвиг. Код операции ISHL 120 (0x78).
    1. Какая арифметическая операция эквивалентна сдвигу влево на 2?
    2. Расширьте систему микрокоманд, чтобы включить эту команду в систему IJVM.
16. Команде INVOKEVIRTUAL в машине JVM нужно знать, сколько у нее параметров. Зачем?

17. Напишите микропрограмму для Mic-1, чтобы реализовать команду JVM POP TWO. Эта команда убирает два верхних слова из стека.
18. Реализуйте команду JVM DLOAD для Mic-2. Эта команда содержит 1-байтный индекс и помещает локальную переменную, находящуюся в этом месте, в стек. Затем она помещает следующее старшее слово в стек.
19. Нарисуйте конечный автомат для учета очков при игре в теннис. Правила игры в теннис следующие. Чтобы выиграть, вам нужно получить как минимум 4 очка и у вас должно быть как минимум на 2 очка больше, чем у вашего соперника. Начните с состояния (0, 0), то есть с того, что ни у кого из вас еще нет очков. Затем добавьте состояние (1,0). Это значит, что игрок Л получил очко. Дугу из состояния (0,0) к состоянию (1,0) обозначьте буквой *A*. Затем добавьте состояние (0,1), чтобы показать, что игрок Л получил очко, а дугу к состоянию (0, 1) обозначьте буквой *B,* Продолжайте добавлять состояния и дуги до тех пор, пока не нарисуете все возможные состояния.
20. Вернитесь к предыдущему вопросу. Существуют ли такие состояния, которые могут выйти из строя, но при этом никак не повлияют на результат любой игры? Если да, то какие из них эквивалентны?
21. Нарисуйте конечный автомат для прогнозирования ветвления, более надежный, чем тот, который изображен на рис. 4.29. Он должен изменять предсказание только после трех последовательных неудачных предсказаний.
22. Сдвиговый регистр, изображенный на рис. 4.18, имеет максимальную емкость 6 байтов. Можно ли сконструировать более дешевый блок выборки команд с 5-байтным сдвиговым регистром? А с 4-байтным?
23. Предыдущий вопрос связан с более дешевыми блоками выборки команд. Теперь рассмотрим более дорогие. Встанет ли когда-нибудь вопрос о том, чтобы сконструировать сдвиговый регистр гораздо большей емкости, скажем, 12 байтов? Если да, то почему? Если нет, то почему?
24. В микропрограмме для микроархитектуры Mic-2 микрокоманда if_icmpeq6 совершает переход к T, если Z установлено на 1. Однако микрокоманда T та же, что и gotol. А возможно ли перейти к gotol сразу, и станет ли машина работать быстрее после этого?
25. В микроархитектуре Mic-4 блок декодирования отображает код операции IJVM в индекс ПЗУ, где хранятся соответствующие микрооперации. Кажется, что было бы проще опустить стадию декодирования и сразу передать код операции IJVM в очередь. Тогда можно использовать код операции IJVM в качестве индекса в ПЗУ, точно так же, как в микроархитектуре Mic-1. Что не так в этом плане?
26. Компьютер содержит двухуровневую кэш-память. Предположим, что 80% обращений к памяти — удачные обращения в кэш-память первого уровня, 15% — в кэш-память второго уровня, а 5% — промахи кэша. Время доступа составляет 5 не, 15 не и 60 не соответственно, причем время доступа в кэш-память второго уровня и в основную память отсчитывается с того момента, как стало известно, что они нужны (например, доступ к кэш-памяти второго уровня не может начаться, пока не произойдет промах кэш-памяти первого уровня). Каково среднее время доступа?

27. В конце раздела «Кэш-память» мы сказали, что заполнение по записи выгодно только в том случае, если имеют место повторные записи в одну и ту же строку кэш-памяти. А если после записи следуют многочисленные считывания из одной и той же строки, не будет ли заполнение по записи также большим преимуществом?
28. В черновом варианте этой книги на рис. 4.27 вместо 4-входовой ассоциативной кэш-памяти была изображена 3-входовая ассоциативная кэш-память. Один из рецензентов заявил, что читателей это может сильно смутить, поскольку три — это не степень двойки, а компьютеры все делают в двоичной системе. Поскольку потребитель всегда прав, рисунок изменили на 4-входовую ассоциативную кэш-память. Был ли рецензент прав? Аргументируйте.
29. Компьютер с конвейером из пяти стадий при обработке условных переходов простаивает следующие три цикла. Насколько эти простаивания снизят производительность, если 20% команд являются условными переходами? Другие причины простаиваний не учитывайте.
30. Предположим, что компьютер вызывает до 20 команд заранее. В среднем 4 из этих команд являются условными переходами, причем вероятность правильного прогнозирования каждого из этих условных переходов равно 90%. Какова вероятность, что предварительный вызов команд на правильном пути?
31. Предположим, что нам пришлось изменить структуру машины, показанную в табл. 4.12, чтобы использовать 16 регистров вместо 8. Тогда мы изменим команду 6, чтобы использовать регистр R8 в качестве ее выходного регистра. Что в этом случае будет происходить в циклах, начиная с цикла 6?
32. Обычно взаимозависимости затрудняют работу конвейеризированных процессоров. Можно ли что-нибудь сделать с WAW-взаимозависимостью, чтобы улучшить положение вещей? Какие существуют средства оптимизации?
33 Перепишите интерпретатор Mic-1 таким образом, чтобы регистр LV указывал на первую локальную переменную, а не на связующий указатель.
34. Напишите моделирующую программу для одновходовой кэш-памяти прямого отображения. Сделайте число элементов и длину строки параметрами программы. Поэкспериментируйте с этой программой и изложите полученные данные.

Глава 5
Уровень архитектуры команд

В этой главе подробно обсуждается уровень архитектуры команд. Он расположен между микроархитектурным уровнем и уровнем операционной системы, как показано на рис 1.2. Исторически этот уровень развился прежде всех остальных уровней и изначально был единственным. В наши дни этот уровень очень часто называют «архитектурой» машины, а иногда (что неправильно) «языком ассемблера»

Уровень архитектуры команд имеет особое значение: он является связующим звеном между программным и аппаратным обеспечением. Конечно, можно было бы сделать так, чтобы аппаратное обеспечение сразу непосредственно выполняло программы, написанные на C, C++, FORTRAN 90 или других языках высокого уровня, но это не очень хорошая идея. Преимущество компиляции перед интерпретацией было бы тогда потеряно. Кроме того, из чисто практических соображений компьютеры должны уметь выполнять программы, написанные на разных языках, а не только на одном.

В сущности, все разработчики считают, что нужно транслировать программы, написанные на различных языках высокого уровня, в общую промежуточную форму — на уровень архитектуры команд — и соответственно конструировать аппаратное обеспечение, которое может непосредственно выполнять программы этого уровня (уровня архитектуры команд). Уровень архитектуры команд связывает компиляторы и аппаратное обеспечение. Это язык, который понятен и компиляторам, и аппаратному обеспечению. На рис. 5.1 показана взаимосвязь компиляторов, уровня архитектуры команд и аппаратного обеспечения.

В идеале при создании новой машины разработчики архитектуры команд должны консультироваться и с составителями компиляторов, и с теми, кто конструирует аппаратное обеспечение, чтобы выяснить, какими особенностями должен обладать уровень команд. Если составители компилятора требуют наличия какой-то особенности, которую инженеры не могут реализовать, то такая идея не пройдет. Точно так же, если разработчики аппаратного обеспечения хотят ввести в компьютер какую-либо новую особенность, но составители программного обеспечения не знают, как построить программу, чтобы использовать эту особенность, то такой проект никогда не будет воплощен. После долгих обсуждений и моделирования появится уровень команд, оптимизированный для нужных языков программирования, который и будет реализован

Но все это в теории. А теперь перейдем к суровой реальности Когда появляется новая машина, первый вопрос, который задают все потенциальные покупатели «Совместима ли машина с предыдущими версиями?». Второй вопрос: «Могу ли я

запустить на ней мою старую операционную систему?» И третий вопрос: «Будут ли работать мои прикладные программы на этой машине и не потребуется ли их изменять?» Если какой-нибудь из этих вопросов получает ответ «нет», разработчики должны будут объяснить, почему. Покупатели редко рвутся выбросить все старое программное обеспечение и начать все заново.

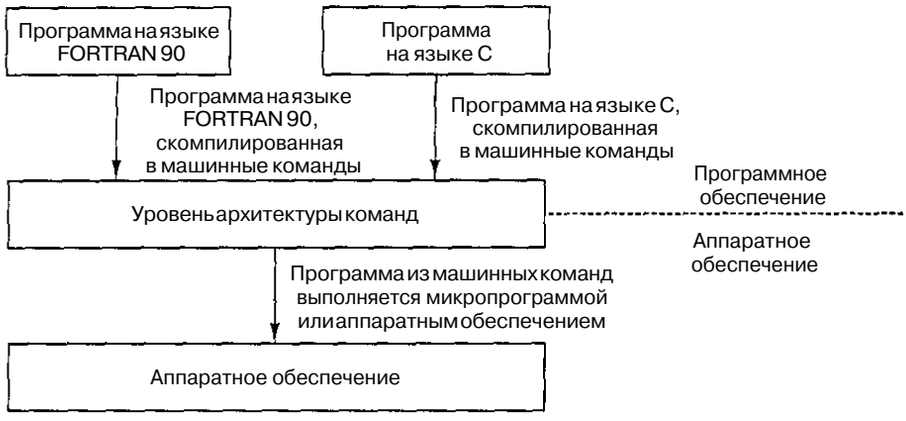


Рис. 5.1. Уровень команд — это промежуточное звено между компиляторами и аппаратным обеспечением

Этот факт заставляет компьютерных разработчиков сохранять один и тот же уровень команд в разных моделях или, по крайней мере, делать его **обратно совместимым.** Под обратной совместимостью мы понимаем способность новой машины выполнять старые программы без изменений. Тем не менее новая машина может содержать новые команды и другие особенности, которые могут использоваться новым программным обеспечением. Разработчики должны делать уровень команд совместимым с предыдущими моделями, но они вправе творить все что угодно с аппаратным обеспечением, поскольку едва ли кого-нибудь из покупателей волнует, что собой представляет реальное аппаратное обеспечение и какие действия оно выполняет. Они могут переходить от микропрограммной разработки к непосредственному выполнению, добавлять конвейеры, суперскалярные устройства и т п, при условии что сохранится обратная совместимость с предыдущим уровнем команд. Основная цель — убедиться, что старые программы работают на новой машине. Тогда возникает проблема[1] построение лучших машин, но с обратной совместимостью.

Все это вовсе не значит, что разработка уровня команд не имеет никакого значения. Хорошо разработанный уровень архитектуры команд имеет огромные преимущества перед плохим, особенно в отношении вычислительных возможностей и стоимости Производительность эквивалентных машин с различными уровнями команд может различаться на 25%. Мы просто хотим сказать, что рынок несколько затрудняет (хотя и не делает невозможным) устранение старой архитектуры команд и введение новой. Тем не менее иногда появляются новые уровни команд универсального назначения, а на специализированных рынках (например, на рынке встроенных систем или на рынке мультимедийных процессоров) они возникают гораздо чаще. Следовательно, важно понимать принципы разработки этого уровня.

Какую архитектуру команд можно считать хорошей? Существует два основных фактора. Во-первых, хорошая архитектура должна определять набор команд, которые можно эффективно реализовать в современной и будущей технике, что приводит к рентабельным разработкам на несколько поколений. Плохой проект реализовать сложнее. При плохо разработанной архитектуре команд может потребоваться большее количество вентилей для процессора и больший объем памяти для выполнения программ. Кроме того, машина может работать медленнее, поскольку такая архитектура команд ухудшает возможности перекрывания операций, поэтому для достижения более высокой производительности здесь потребуется более сложный проект. Разработка, в которой используются особенности конкретной техники, может повлечь за собой производство целого поколения компьютеров, и эти компьютеры сможет опередить только более продвинутая архитектура команд.

Во-вторых, хорошая архитектура команд должна обеспечивать ясную цель для оттранслированной программы. Регулярность и полнота вариантов — важные черты, которые не всегда свойственны архитектуре команд. Эти качества важны для компилятора, которому трудно сделать лучший выбор из нескольких возможных, особенно когда некоторые очевидные на первый взгляд варианты не разрешены архитектурой команд. Если говорить кратко, поскольку уровень команд является промежуточным звеном между аппаратным и программным обеспечением, он должен быть удобен и для разработчиков аппаратного обеспечения, и для составителей программного обеспечения.

Общий обзор уровня архитектуры команд

Давайте начнем изучение уровня команд с вопроса о том, что он собой представляет. Этот вопрос на первый взгляд может показаться простым, но на самом деле здесь есть очень много сложностей. В следующем разделе мы обсудим некоторые из этих проблем. Затем мы рассмотрим модели памяти, регистров и команд.

Свойства уровня команд

В принципе уровень команд — это то, каким представляется компьютер программисту машинного языка. Поскольку сейчас ни один нормальный человек не пишет программ на машинном языке, мы переделали это определение. Программа уровня архитектуры команд — это то, что выдает компилятор (в данный момент мы игнорируем вызовы операционной системы и символический язык ассемблера). Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т. д. Вся эта информация в совокупности и определяет уровень архитектуры команд.

В соответствии с этим определением такие вопросы, как программируется ли микроархитектура или нет, конвейеризирован компьютер или нет, является он суперскалярным или нет и т. д., не относятся к уровню архитектуры команд, поскольку составитель компилятора не видит всего этого. Однако это замечание не

совсем справедливо, поскольку некоторые из этих свойств влияют на производительность, а производительность является видимой для программиста. Рассмотрим, например, суиерскалярную машину, которая может выдавать back-to-back команды в одном цикле, при условии что одна команда целочисленная, а одна — с плавающей точкой. Если компилятор чередует целочисленные команды и команды с плавающей точкой, то производительность заметно улучшится. Таким образом, детали суперскалярной операции видны на уровне команд, и границы между различными уровнями размыты.

Для одних архитектур уровень команд определяется формальным документом, который обычно выпускается промышленным консорциумом, для других — нет. Например, V9 SPARC (Version 9 SPARC) и JVM имеют официальные определения [156, 85]. Цель такого официального документа — дать возможность различным производителям выпускать машины данного конкретного вида, чтобы эти машины могли выполнять одни и те же программы и получать при этом одни и те же результаты,

В случае с системой SPARC подобные документы нужны для того, чтобы различные предприятия могли выпускать идентичные микросхемы SPARC, отличающиеся друг от друга только производительностью и ценой. Чтобы эта идея работала, поставщики микросхем должны знать, что делает микросхема SPARC (на уровне команд). Следовательно, в документе говорится о том, какая модель памяти, какие регистры присутствуют, какие действия выполняют команды и т. д., а не о том, что представляет собой микроархитектура.

В таких документах содержатся нормативные разделы, в которых излагаются требования, и информативные разделы, которые предназначены для того, чтобы помочь читателю, но не являются частью формального определения. В нормативных разделах описаны требования и запреты. Например, такое высказывание, как:

выполнениезарезервированногокодаоперациидолжновызыватьсистемноепрерывание

означает, что если программа выполняет код операции, который не определен, то он должен вызывать системное прерывание, а не просто игнорироваться. Может быть и альтернативный подход:

результатвыполнениязарезервированногокодаоперациионределяетсяреализацией.

Это значит, что составитель компилятора не может просчитать какие-то конкретные действия, предоставляя конструкторам свободу выбора. К описанию архитектуры часто прилагаются тестовые комплекты для проверки, действительно ли данная реализация соответствует техническим требованиям.

Совершенно ясно, почему V9 SPARC имеет документ, в котором определяется уровень команд: это нужно для того, чтобы все микросхемы V9 SPARC могли выполнять одни и те же программы. По той же причине существует специальный документ для JVM: чтобы интерпретаторы (или такие микросхемы, как picojava II) могли выполнять любую допустимую программу JVM. Для уровня команд процессора Pentium II такого документа нет, поскольку компания Intel не хочет, чтобы другие производители смогли запускать микросхемы Pentium II. Компания Intel даже обращалась в суд, чтобы запретить производство своих микросхем другими предприятиями.

Другое важное качество уровня команд состоит в том, что в большинстве машин есть, по крайней мере, два режима. **Привилегированный режим** предназначен для запуска операционной системы. Он позволяет выполнять все команды. **Пользовательский** режим предназначен для запуска программных приложений. Этот режим не позволяет выполнять некоторые чувствительные команды (например, те, которые непосредственно манипулируют кэш-памятью). В этой главе мы в первую очередь сосредоточимся на командах и свойствах пользовательского режима.

Модели памяти

Во всех компьютерах память разделена на ячейки, которые имеют последовательные адреса. В настоящее время наиболее распространенный размер ячейки — 8 битов, но раньше использовались ячейки от 1 до 60 битов (см. табл. 2.1). Ячейка из 8 битов называется байтом. Причина применения именно 8-битных байтов такова: символы ASCII кода занимают 7 битов, поэтому один символ ASCII плюс бит четности как раз подходит под размер байта. Если в будущем будет доминировать UNICODE, то ячейки памяти, возможно, будут 16-битными. Вообще говоря, число 2^4 лучше, чем 2^3, поскольку 4 — степень двойки, а 3 — нет.

Байты обычно группируются в 4-байтные (32-битные) или 8-байтные (64-битные) слова с командами для манипулирования целыми словами. Многие архитектуры требуют, чтобы слова были выровнены в своих естественных границах. Так, например, 4-байтное слово может начинаться с адреса 0,4,8 и т. д., но не с адреса 1 или 2. Точно так же слово из 8 байтов может начинаться с адреса 0,8 или 16, но не с адреса 4 или 6. Расположение 8-байтных слов показано на рис. 5.2.

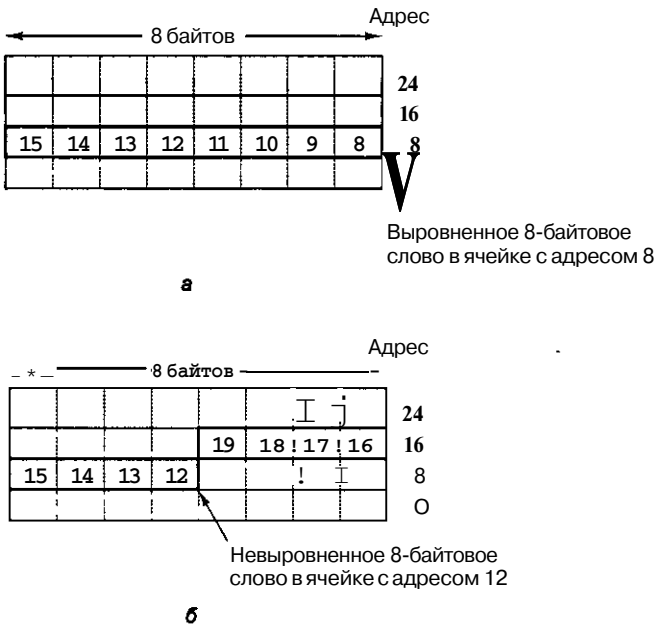


Рис. 5.2. Расположение слова из 8 байтов в памяти: выровненное (а); невыровненное (б). Некоторые машины требуют, чтобы слова в памяти были выровнены

Выравнивание адресов требуется довольно часто, поскольку при этом память работает более эффективно. Например, Pentium II, который вызывает из памяти по 8 байтов за раз, использует 36-битные физические адреса, но содержит только 33 адресных бита, как показано на рис. 3.41. Следовательно, Pentium II даже не сможет обратиться к невыровненной памяти, поскольку младшие три бита не определены явным образом. Эти биты всегда равны 0, и все адреса памяти кратны 8 байтам.

Тем не менее требование выравнивания адресов иногда вызывает некоторые проблемы. В процессоре Pentium II программы могут обращаться к словам, начиная с любого адреса, — это качество восходит к модели 8088 с шиной данных шириной в 1 байт, в которой не было такого требования, чтобы ячейки располагались в 8-байтных границах. Если программа в процессоре Pentium II считывает 4-байтное слово из адреса 7, аппаратное обеспечение должно сделать одно обращение к памяти, чтобы вызвать байты с 0-го по 7-й, и второе обращение к памяти, чтобы вызвать байты с 8-го по 15-й. Затем центральный процессор должен извлечь требуемые 4 байта из 16 байтов, считанных из памяти, и скомпоновать их в нужном порядке, чтобы сформировать 4-байтное слово.

Возможность считывать слова с произвольными адресами требует усложнения микросхемы, которая после этого становится больше по размеру и дороже. Разработчики были бы рады избавиться от такой микросхемы и просто потребовать, чтобы все программы обращались к словам памяти, а не к байтам. Однако на вопрос инженеров: «Кому нужно исполнение старых программ для машины 8088, которые неправильно обращаются к памяти?» последует ответ продавцов: «Нашим покупателям».

Большинство машин имеют единое линейное адресное пространство, которое простирается от адреса 0 до какого-то максимума, обычно 2^{32} байтов или 2^{64} байтов. В некоторых машинах содержатся отдельные адресные пространства для команд и для данных, так что при вызове команды с адресом 8 и вызове данных с адресом 8 происходит обращение к разным адресным пространствам. Такая система гораздо сложнее, чем единое адресное пространство, но зато она имеет два преимущества. Во-первых, появляется возможность иметь 2^{32} байтов для программы и дополнительные 2^{32} байтов для данных, используя только 32-битные адреса. Во-вторых, поскольку запись всегда автоматически происходит только в пространство данных, случайная перезапись программы становится невозможной, и следовательно, устраняется один из источников программных сбоев.

Отметим, что отдельные адресные пространства для команд и для данных — это не то же самое, что разделенная кэш-память первого уровня. В первом случае все адресное пространство целиком дублируется, и считывание из любого адреса вызывает разные результаты в зависимости от того, что именно считывается: слово или команда. При разделенной кэш-памяти существует только одно адресное пространство, просто в разных блоках кэш-памяти хранятся разные части этого пространства.

Еще один аспект модели памяти — семантика памяти. Естественно ожидать, что команда LOAD, которая встречается после команды STORE и которая обращается к тому же адресу, возвратит только что сохраненное значение. Тем не менее, как мы видели в главе 4, во многих машинах микрокоманды переупорядочиваются.

Таким образом, существует реальная опасность, что память не будет действовать так, как ожидается. Ситуация усложняется в случае с мультипроцессором, когда каждый процессор посылает разделенной памяти поток запросов на чтение и запись, которые тоже могут быть переупорядочены.

Системные разработчики могут применять один из нескольких подходов к этой проблеме. С одной стороны, все запросы памяти могут быть упорядочены в последовательность таким образом, чтобы каждый из них завершался до того, как начнется следующий. Такая стратегия сильно вредит производительности, но зато дает простейшую семантику памяти (все операции выполняются в строгом программном порядке).

С другой стороны, не дается вообще никаких гарантий. Чтобы сделать обращения к памяти упорядоченными, программа должна выполнить команду SYNC, которая блокирует запуск всех новых операций памяти до тех пор, пока предыдущие операции не будут завершены. Эта идея сильно затрудняет работу тех, кто пишет компиляторы, поскольку для этого им нужно очень хорошо знать, как работает соответствующая микроархитектура, но зато разработчикам аппаратного обеспечения предоставлена полная свобода в оптимизации использования памяти.

Возможны также промежуточные модели памяти, в которых аппаратное обеспечение автоматически блокирует запуск определенных операций с памятью (например, тех, которые связаны с RAW- или WAR-взаимозависимостью), при этом запуск всех других операций не блокируется. Хотя разработка этих особенностей на уровне команд довольно утомительна (по крайней мере, для составителей компиляторов и программистов на языке ассемблера), сейчас существует тенденция использовать такой подход. Эта тенденция вызвана такими реализациями, как переупорядочение микрокоманд, конвейеры, многоуровневая кэш-память и т. д. Другие неестественные примеры такого рода мы рассмотрим в этой главе чуть позже.

Регистры

Во всех компьютерах имеется несколько регистров, которые видны на уровне команд. Они нужны там для того, чтобы контролировать выполнение программы, хранить временные результаты, а также для некоторых других целей. Обычно регистры, которые видны на микроархитектурном уровне, например TOS и MAR (см. рис. 4.1), не видны на уровне команд. Тем не менее некоторые из них, например счетчик команд и указатель стека, присутствуют на обоих уровнях. Регистры, которые видны на уровне команд, всегда видны на микроархитектурном уровне, поскольку именно там они реализуются.

Регистры уровня команд можно разделить на две категории: специальные регистры и регистры общего назначения. Специальные регистры включают счетчик команд и указатель стека, а также другие регистры с особой функцией. Регистры общего назначения содержат ключевые локальные переменные и промежуточные результаты вычислений. Их основная функция состоит в том, чтобы обеспечить быстрый доступ к часто используемым данным (обычно избегая обращений к памяти). Машины RISC с высокоскоростными процессорами и медленной (относительно медленной) памятью обычно содержат как минимум 32 регистра общего назначения, а в новых процессорах количество этих регистров постоянно растет.

В некоторых машинах регистры общего назначения полностью симметричны и взаимозаменяемы. Если все регистры эквивалентны, для хранения временного результата компилятор может использовать и регистр R1, и регистр R25. Выбор регистра не имеет никакого значения.

В других машинах некоторые регистры общего назначения могут быть специализированы. Например, в процессоре Pentium II существует регистр EDX, который может использоваться в качестве регистра общего назначения, но который также получает половину произведения и содержит половину делимого при делении.

Даже если регистры общего назначения полностью взаимозаменяемы, операционная система или компиляторы часто принимают соглашения о том, каким образом используются эти регистры. Например, некоторые регистры могут содержать параметры вызываемых процедур, а другие могут использоваться в качестве временных регистров. Если компилятор помещает важную локальную переменную в регистр R1, а затем вызывает библиотечную процедуру, которая воспринимает регистр R1 как временный регистр, доступный для нее, то когда библиотечная процедура возвращает значение, регистр R1 может содержать ненужные данные. А если существуют какие-либо системные соглашения по поводу того, как нужно использовать регистры, составители компиляторов и программисты на языке ассемблера должны следовать им.

Кроме регистров, доступных на уровне команд, всегда существует довольно большое количество специальных регистров, доступных только в привилегированном режиме. Эти регистры контролируют различные блоки кэш-памяти, основную память, устройства ввода-вывода и другие элементы аппаратного обеспечения машины. Данные регистры используются только операционной системой, поэтому компиляторам и пользователям не обязательно знать об их существовании.

Есть один регистр управления, который представляет собой привилегированно-пользовательский гибрид. Это **флаговый регистр,** или **PSW (Program State Word — слово состояния программы.** Этот регистр содержит различные биты, которые нужны центральному процессору. Самые важные биты — это **коды условия.** Они устанавливаются в каждом цикле АЛУ и отражают состояние результата предыдущей операции. Биты кода условия включают:

- N — устанавливается, если результат был отрицательным (Negative);
- Z — устанавливается, если результат был равен 0 (Zero);
- V — устанавливается, если результат вызвал переполнение (oVerflow);
- C — устанавливается, если результат вызвал выход переноса самого левого бита (Carry out);
- A — устанавливается, если произошел выход переноса бита 3 (Auxiliary carry — служебный перенос);
- P — устанавливается, если результат четный (Parity).

Коды условия очень важны, поскольку они используются при сравнениях и условных переходах. Например, команда CMP обычно вычитает один операнд из другого и устанавливает коды условия на основе полученной разности. Если

операнды равны, то разность будет равна 0 и во флаговом регистре будет установлен бит Z. Последующая команда BEQ (Branch Equal — переход в случае равенства) проверяет бит Z и совершает переход, если он установлен.

Флаговый регистр содержит не только коды условия. Его содержимое меняется от машины к машине. Дополнительные поля указывают режим машины (например, пользовательский или привилегированный), трассовый бит (который используется для отладки), уровень приоритета процессора, а также статус разрешения прерываний. Флаговый регистр обычно можно считать в пользовательском режиме, но некоторые поля могут записываться только в привилегированном режиме (например, бит, который указывает режим).

Команды

Главная особенность уровня, который мы сейчас рассматриваем, — это набор машинных команд. Они управляют действиями машины. В этом наборе всегда присутствуют команды LOAD и STORE (в той или иной форме) для перемещения данных между памятью и регистрами и команда MOVE для копирования данных из одного регистра в другой. Всегда присутствуют арифметические и логические команды и команды для сравнения элементов данных и переходов в зависимости от результатов. Некоторые типичные команды мы уже рассматривали (см. табл. 4.2.). А в этой главе мы рассмотрим многие другие команды.

Общий обзор уровня команд машины Pentium II

В этой главе мы обсудим три совершенно разные архитектуры команд: IA-32 компании Intel (она реализована в Pentium II), Version 9 SPARC (она реализована в процессорах SPARC) и JVM (она реализована в picojavall). Мы не преследуем цель дать исчерпывающее описание каждой из этих архитектур. Мы просто хотим продемонстрировать важные аспекты архитектуры команд и показать, как эти аспекты меняются от одной архитектуры к другой. Начнем с машины Pentium II.

Процессор Pentium II развивался на протяжении многих лет. Его история восходит к самым первым микропроцессорам, как мы говорили в главе 1. Основная архитектура команд обеспечивает выполнение программ, написанных для процессоров 8086 и 8088 (которые имеют одну и ту же архитектуру команд), а в машине даже содержатся элементы 8080 — 8-разрядный процессор, который был популярен в 70-е годы. На процессор 8080, в свою очередь, сильно повлияли требования совместимости с процессором 8008, который был основан на процессоре 4004 (4-битной микросхеме, применявшейся еще в каменном веке).

С точки зрения программного обеспечения, компьютеры 8086 и 8088 были 16-разрядными машинами (хотя компьютер 8088 содержал 8-битную шину данных). Их последователь 80286 также был 16-разрядным. Его главным преимуществом был больший объем адресного пространства, хотя небольшое число программ использовали его, поскольку оно состояло из 16 384 64 K сегментов, а не представляло собой линейную 2^{24}-байтную память.

Процессор 80386 был первой 32-разрядной машиной, выпущенной компанией Intel. Все последующие процессоры (80486, Pentium, Pentium Pro, Pentium II, Celeron и Хеоп) имеют точно такую же 32-разрядную архитектуру, которая называется **IA-32,** поэтому мы сосредоточим наше внимание именно на этой архитектуре. Единственным существенным изменением архитектуры со времен процессора 80386 было введение команд MMX в более поздние версии системы Pentium и их включение **в** Pentium II и последующие процессоры.

Pentium II имеет 3 операционных режима, в двух из которых он работает как 8086. В **реальном режиме** все особенности, которые были добавлены к процессору со времен системы 8088, отключаются, и Pentium II работает как простой компьютер 8088. Если программа совершает ошибку, то происходит полный отказ системы. Если бы компания Intel занималась разработкой человеческих существ, то внутрь каждого человека был бы помещен бит, который превращает людей обратно в режим шимпанзе (примитивный мозг, отсутствие речи, питание в основном бананами и т.д.).

На следующей ступени находится **виртуальный режим 8086,** который делает возможным выполнение старых программ, написанных для 8088, с защитой. Чтобы запустить старую программу 8088, операционная система создает специальную изолированную среду, которая работает как процессор 8088, за исключением того, что если программа дает сбой, операционной системе передается соответствующая информация и полного отказа системы не происходит. Когда пользователь WINDOWS начинает работу с MS-DOS, ""рограмма, которая действует там, запускается в виртуальном режиме 8086, чтобы программа WINDOWS не могла вмешиваться в программы MS-DOS.

Последний режим — это защищенный режим, в котором Pentium II работает как Pentium II, а не как 8088. В этом режиме доступны 4 уровня привилегий, которые управляются битами во флаговом регистре. Уровень 0 соответствует привилегированному режиму на других компьютерах и имеет полный доступ к машине. Этот уровень используется операционной системой. Уровень 3 предназначен для пользовательских программ. Он блокирует доступ к определенным командам и регистрам управления, чтобы ошибки какой-нибудь пользовательской программы не привели к поломке всей машины. Уровни 1 и 2 используются редко. Pentium II имеет огромное адресное пространство. Память разделена на 16 384 сегмента, каждый из которых идет от адреса 0 до адреса $2^{3J}-1$. Однако большинство операционных систем (включая UNIX и все версии WINDOWS) поддерживают только один сегмент, поэтому большинство прикладных программ видят линейное адресное пространство в 2^{32} байтов, а иногда часть этого пространства занимает сама операционная система. Каждый байт в адресном пространстве имеет свой адрес. Слова состоят из 32 битов. Байты нумеруются справа налево (то есть самый первый адрес соответствует самому младшему байту).

Регистры процессора Pentium II показаны на рис. 5.3. Первые четыре регистра EAX, EBX, ECX и EDX 32-битные. Это регистры общего назначения, хотя у каждого из них есть определенные особенности. EAX — основной арифметический регистр; EBX предназначен для хранения указателей (адресов памяти); ECX связан с организацией циклов; EDX нужен для умножения и деления — этот регистр

вместе с EAX содержит 64-битные произведения и делимые. Каждый из этих регистров имеет 16-разрядный регистр в младших 16 битах и 8-разрядный регистр в младших 8 битах. Данные регистры позволяют легко манипулировать 16-битными и 8-битными значениями соответственно. В компьютерах 8088 и 80286 есть только 8-битные и 16-битные регистры. 32-битные регистры появились в системе 80386 вместе с приставкой E (Extended — расширенный).

Следующие три регистра также являются регистрами общего назначения, но с большей степенью специализации. Регистры ESI и EDI предназначены для хранения указателей, особенно для команд манипулирования цепочками, где ESI указывает на входную цепочку, а EDI — на выходную цепочку. Регистр EBP тоже предназначен для хранения указателей. Обычно он используется для указания на основу текущего фрейма локальных переменных, как и регистр LV в машине IJVM. Такой регистр обычно называют **указателем фрейма.** Наконец, регистр ESP — это указатель стека.

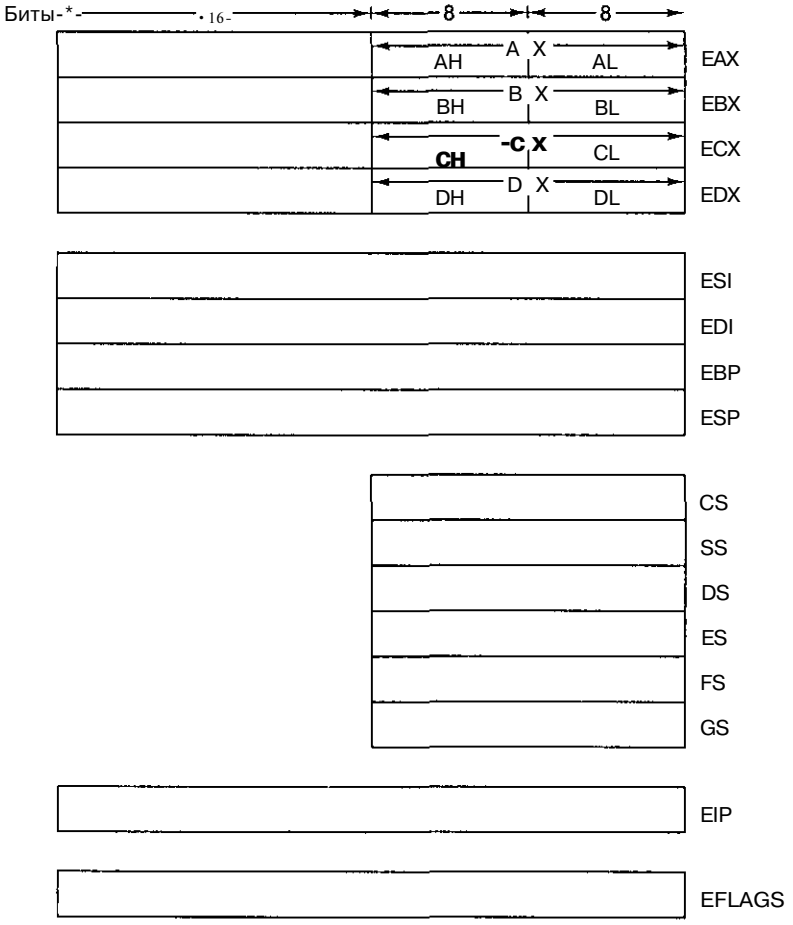


Рис. 5.3. Основные регистры процессора Pentium II

Следующая группа регистров от CS до GS — сегментные регистры. Это электронные трилобиты — атавизмы, оставшиеся от процессора 8088, который обращался к 2^{20} байтам памяти, используя 16-битные адреса. Достаточно сказать, что когда Pentium II установлен на использование единого линейного 32-битного адресного пространства, их можно смело проигнорировать. Регистр EIP — это счетчик программ (Extended Instruction Pointer — расширенный указатель команд). Регистр EFLAGS — это флаговый регистр.

Общий обзор уровня команд системы UltraSPARC II

Архитектура SPARC была впервые введена в 1987 году компанией Sun Microsystems. Эта архитектура была одной из первых архитектур промышленного назначения типа RISC. Она была основана на исследовании, проведенном в Беркли в 80-е годы [110,113]. Изначально система SPARC была 32-разрядной архитектурой, но UltraSPARC II — это 64-разрядная машина, основанная на Version 9, и именно ее мы будем описывать в этой главе. В целях согласованности с остальными частями книги мы будем называть данную систему UltraSPARC II, хотя на уровне команд все машины UltraSPARC идентичны.

Структура памяти машины UltraSPARC II очень проста: память представляет собой линейный массив из 2^{64} байтов. К сожалению, память настолько велика (18 446 744 073 709 551 616 байтов), что в настоящее время ее невозможно реализовать. Современные реализации имеют ограничение на размер адресного пространства, к которому они могут обращаться *(2^{44}* байтов у UltraSPARC II), но в будущем это число увеличится. Байты нумеруются слева направо, но нумерацию можно изменить и сделать ее справа налево, установив бит во флаговом регистре.

Важно, что архитектура команд имеет больше байтов, чем требуе!ся для реализации, поскольку в будущем, скорее в-его, понадобится увеличить размер памяти, к которой может обращаться процессор. Одна из самых серьезных проблем при разработке архитектур состоит в том, что архитектура команд ограничивает размер адресуемой памяти. В информатике существует один вопрос, который совершенно невозможно разрешить: никогда не хватает того количества битов, которое имеется в данный момент. Когда-нибудь ваши внуки спросят у вас, как же могли работать компьютеры, которые содержат всего-навсего 32-битные адреса и только 4 Гбайт памяти.

Архитектура команд SPARC достаточно проста, хотя организация регистров была немного усложнена, чтобы сделать вызовы процедур более эффективными. Практика показывает, что организация регистров требует больших усилий и в общем эти усилия не стоят того, но правило совместимости не позволяет избавиться от этого.

В системе UltraSPARC II имеется две группы регистров: 32 64-битных регистра общего назначения и 32 регистра с плавающей точкой. Регистры общего назначения называются R0-R31, но в определенных контекстах используются другие названия. Варианты названий регистров и их функции приведены в табл. 5.1.

Таблица 5.1. Регистры общего назначения в системе UltraSPARC II

Регистр	Другой вариант названия	Функция
R0	G0	Связан с 0. То, что сохранено в этом регистре, просто игнорируется
R1-R7	G1-G7	Содержит глобальные переменные
R8-R13	O0-O5	Содержит параметры вызываемой процедуры
R14	SP	Указатель стека
R15	07	Временный регистр
R16-R23	L0-L7	Содержит локальные переменные для текущей процедуры
R24-R29	I0-I5	Содержит входные параметры
R30	FP	Указатель на основу текущего стекового фрейма
R31	17	Содержит адрес возврата для текущей процедуры

Все регистры общего назначения 64-битные. Все они, кроме R0, значение которого всегда равно 0, могут считываться и записываться при помощи различных команд загрузки и сохранения. Функции, приведенные в табл. 5.1, отчасти определены по соглашению, но отчасти основаны на том, как аппаратное обеспечение обрабатывает их. Вообще не стоит отклоняться от этих функций, если вы не являетесь крупным специалистом, блестяще разбирающимся в компьютерах SPARC. Программист должен быть уверен, что программа правильно обращается к регистрам и выполняет над ними допустимые арифметические действия. Например, очень легко загрузить числа с плавающей точкой в регистры общего назначения, а затем произвести над ними целочисленное сложение, операцию, которая приведет к полнейшей чепухе, но которую центральный процессор обязательно выполнит, если этого потребует программа.

Глобальные переменные используются для хранения констант, переменных и указателей, которые нужны во всех процедурах, хотя они могут загружаться и перезагружаться при входе в процедуру и при выходе из процедуры, если нужно. Регистры 1x и Ox используются для передачи параметров процедурам, чтобы избежать обращений к памяти. Ниже мы расскажем, как это происходит.

Специальные регистры используются для особых целей. Регистры FP и SP ограничивают текущий фрейм. Первый указывает на основу текущего фрейма и используется для обращения к локальным переменным, точно так же как LV на рис. 4.9. Второй указывает на текущую вершину стека и изменяется, когда слова помещаются в стек или выталкиваются оттуда. Значение регистра FP изменяется только при вызове и завершении процедуры. Третий специальный регистр — R31. Он содержит адрес возврата для текущей процедуры.

В действительности процессор UltraSPARC II имеет более 32 регистров общего назначения, но только 32 из них видны для программы в любой момент времени. Эта особенность, называемая **регистровыми окнами,** предназначена для повышения эффективности вызова процедур. Система регистровых окон проиллюстрирована рис. 5.4. Основная идея — имитировать стек, используя при этом регистры. То есть существует несколько наборов регистров, точно так же как и несколько

фреймов в стеке. Ровно 32 регистра общего назначения видны в любой момент. Регистр CWP (Current Window Pointer — указатель текущего окна) следит за тем, какой набор регистров используется в данный момент.

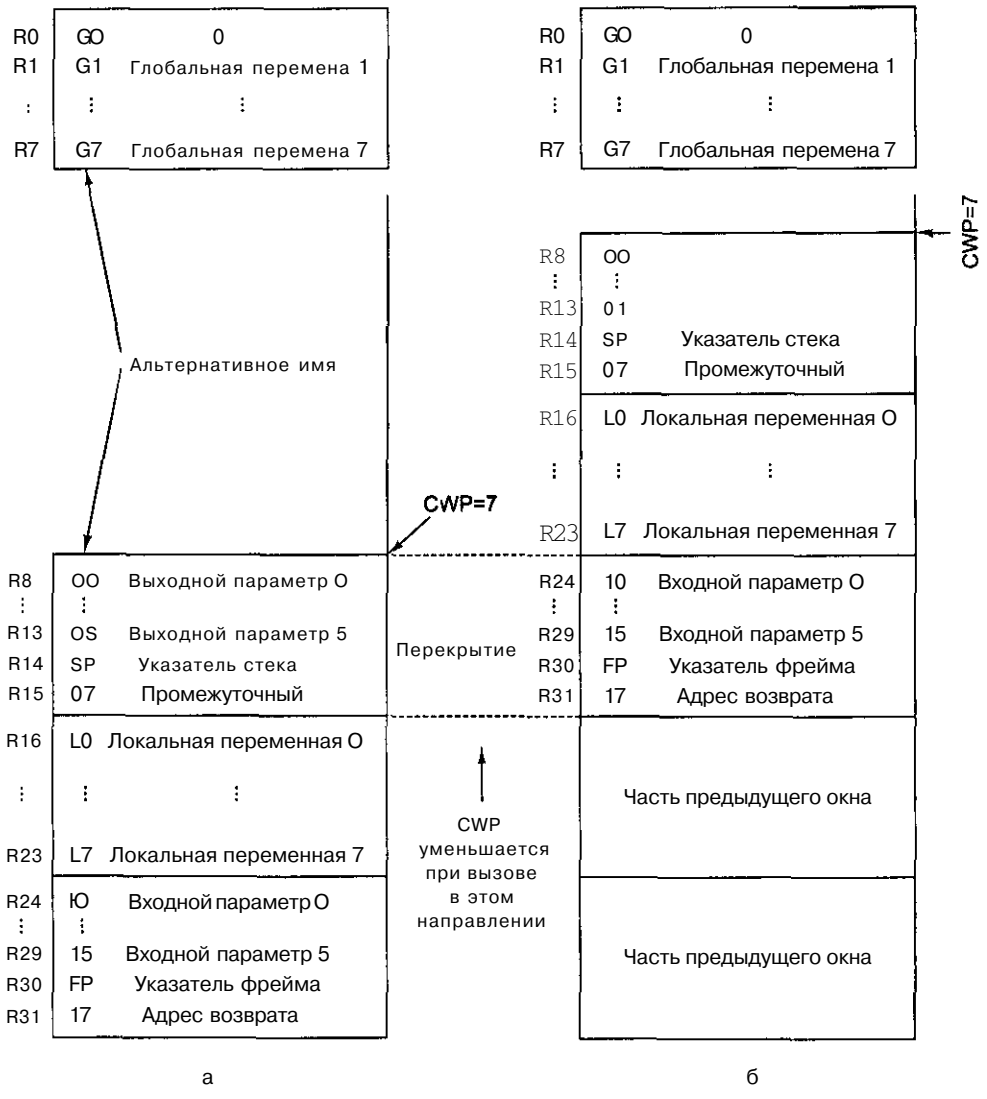


Рис. 5.4. Регистровые окна системы UltraSPARC II

Команда вызова процедуры скрывает старый набор регистров и путем изменения CWP предоставляет новый набор, который может использовать вызванная процедура. Однако некоторые регистры переносятся из вызывающей процедуры к вызванной процедуре, что обеспечивает эффективный способ передачи параметров между процедурами. Для этого некоторые регистры переименовываются: после вызова процедуры прежние выходные регистры с R8 по R15 все еще видны, но

теперь это входные регистры с R24 по R31. Восемь глобальных регистров не меняются. Это всегда один и тот же набор регистров.

В отличие от памяти, которая квазибесконечна (по крайней мере, в отношении стеков), если происходит многократное вложение процедур, машина выходит из регистровых окон. В этот момент самый старый набор регистров сбрасывается в память, чтобы освободить новый набор. Точно так же после многократных выходов из процедур может понадобиться вызвать набор регистров из памяти. В целом такая усложненность является большой помехой и, вообще говоря, не очень полезна. Такая система выгодна только в том случае, если нет многократных вложений процедур.

В системе UltraSPARC II есть также 32 регистра с плавающей точкой, которые могут содержать либо 32-битные (одинарная точность), либо 64-битные (двойная точность) значения. Возможно также использовать пары этих регистров, чтобы поддерживать 128-битные значения.

Архитектура UltraSPARC II — архитектура загрузки/хранения. Это значит, что единственные операции, которые непосредственно обращаются к памяти — это команды LOAD (загрузка) и STORE (сохранение), служащие для перемещения данных между регистрами и памятью. Все операнды для команд арифметических и логических действий должны браться из регистров или предоставляться самой командой (но не должны браться из памяти), и все результаты должны сохраняться в регистрах (но не в памяти).

Общий обзор виртуальной машины Java

Уровень команд машины JVM необычен, но достаточно прост. Мы уже отчасти рассмотрели его во время изучения машины JVM. Модель памяти JVM точно такая же, как у IJVM, о которой мы говорили в главе 4 (см. рис. 4.9), но с одной дополнительной областью, о которой мы сейчас расскажем. Порядок байтов обратный.

Память содержит 4 основные области: фрейм локальных переменных, стек операндов, область процедур и набор констант. Напомним, что в реализациях Mic-x машины IJVM на эти области указывают регистры LV, SP, PC и CPP. Доступы к памяти должны осуществляться только по смещению от одного из этих регистров; указатели и абсолютные адреса памяти не используются. Хотя JVM не требует наличия этих регистров, в большинстве реализаций такие регистры (или подобные им) имеются.

Отсутствие указателей для доступа к локальным переменным и константам не случайно. Это нужно для достижения одной из главных целей языка Java: возможности загружать двоичную программу из Интернета и выполнять ее, не опасаясь шпионских программ или какого-либо сбоя в машине, на которой это программа выполняется. Если ограничить использование указателей, можно добиться высокой безопасности.

Напомним, что ни одна из областей памяти, определенных в машине IJVM, не может быть очень большой. Объем области процедур может быть всего лишь 64 Кбайт. Пространство, занимаемое локальными переменными, не должно превышать 64 Кбайт. Набор констант также ограничивается 64 Кбайт. JVM характеризуется теми же ограничениями по той же причине: смещения для индексирования этих областей ограничиваются 16 битами.

Область локальных переменных меняется с каждой процедурой, поэтому каждая вызываемая процедура имеет собственные 64 Кбайт для своих локальных переменных. Точно так же определенный набор констант распространяется только на определенный класс, поэтому каждый из них имеет собственные 64 Кбайт. Здесь нет места для хранения больших массивов и динамических структур данных, например списков и деревьев. Именно поэтому в JVM включается дополнительная область памяти, так называемая «куча», которая предназначена для хранения динамических объектов, а также очень крупных объектов. Когда компилятор Java воспринимает следующее выражение:

```
int a[]-new -int[4096]
```

он посылает сигнал распределителю памяти, который определяет место в памяти для «кучи» и возвращает ей указатель. Таким образом, указатели используются в JVM, но программисты не могут непосредственно манипулировать ими.

Если в «куче» создаются все новые и новые структуры данных, она в конце концов переполнится. Когда специальная система определяет, что «куча» почти заполнилась, она вызывает **программу чистки памяти (сборщик мусора),** которая ищет ненужные объекты в «куче». Для этого применяется сложный алгоритм. Ненужные объекты отбрасываются, чтобы освободить место в «куче». Схема действия автоматической программы чистки памяти полностью отличается от использования стека локальных переменных. Эти два метода находятся в отношении дополнительности; каждый из них имеет свое место.

JVM не содержит регистров общего назначения, которые могут загружаться или сохраняться под управлением программы. Это машина со стековой организацией. Хотя для наших дней это необычно, такая машина дает простую архитектуру команд, которую легко компилировать.

Однако у машины со стековой организацией есть один недостаток: здесь требуется большое количество обращений к памяти. Но, как мы видели в разделе «Микроархитектура процессора picojava II» главы 4, при умелой разработке можно устранить большую часть из них путем свертывания команд JVM. Более того, поскольку данная архитектура очень проста, ее можно реализовать в небольшом кусочке кремния, оставив большую часть пространства микросхемы свободной для кэш-памяти первого уровня, которая в дальнейшем сократит число обращений к основной памяти. (Размер кэш-памяти picojava II составляет максимум 16 Кбайт + 16 Кбайт, поскольку основной целью было создание дешевой микросхемы.)

Типыданных

Всем компьютерам нужны данные. Для многих компьютерных систем основной задачей является обработка финансовых, промышленных, научных, технических и других данных. Внутри компьютера данные должны быть представлены в какой-либо особой форме. На уровне архитектуры команд используются различные типы данных. Они будут описаны ниже.

Ключевым вопросом является вопрос о том, существует ли аппаратная поддержка для конкретного типа данных. Под аппаратной поддержкой подразумевается, что одна или несколько команд ожидают данные в определенном формате **и** пользо-

ватель не может брать другой формат. Например, бухгалтеры привыкли писать знак «минус» справа у отрицательных чисел, а специалисты по вычислительной технике — слева. Предположим, что, пытаясь произвести впечатление на своего начальника, глава компьютерного центра в бухгалтерской фирме изменил все числа во всех компьютерах, чтобы знаковый бит был самым правым битом (а не самым левым). Несомненно, это произведет большое впечатление на начальника, поскольку все программное обеспечение больше не будет функционировать правильно. Аппаратное обеспечение требует определенного формата для целых чисел и не будет работать должным образом, если целые числа поступают в другом формате.

Теперь рассмотрим другую бухгалтерскую фирму, которая только что заключила договор на проверку федерального долга. 32-битная арифметика здесь не подойдет, поскольку числа превышают 2^{32} (около 4 миллиардов). Одно из возможных решений — использовать два 32-битных целых числа для представления каждого числа, то есть все 64 бита. Если машина не поддерживает такие **числа с удвоенной точностью,** то все арифметические операции над ними должны выполняться программным обеспечением, но эти две части могут располагаться в произвольном порядке, поскольку для аппаратного обеспечения это не важно. Это пример типа данных без аппаратной поддержки и, следовательно, без аппаратной реализации. В следующих разделах мы рассмотрим типы данных, которые поддерживаются аппаратным обеспечением и для которых требуются специальные форматы.

Числовые типы данных

Типы данных можно разделить на две категории: числовые и нечисловые. Среди числовых типов данных главными являются целые числа. Они бывают различной длины: обычно 8,16,32 и 64 бита. Целые числа применяются для подсчета различных предметов (например, они показывают, сколько на складе имеется отверток), для идентификации различных объектов (например, банковских счетов), а также для других целей. В большинстве современных компьютеров целые числа хранятся в двоичной записи, хотя в прошлом использовались и другие системы. Двоичные числа обсуждаются в приложении А.

Некоторые компьютеры поддерживают целые числа и со знаком, и без знака. В целом числе без знака нет знакового бита, и все биты содержат данные. Этот тип данных имеет преимущество: у него есть дополнительный бит, поэтому 32-битное слово может содержать целое число без знака от 0 до $2^{n}-1$ включительно. Двоичное целое число со знаком, напротив, может содержать числа только до $2^{31}-1$, но зато включает и отрицательные числа.

Для выражения нецелых чисел (например, 3,5) используются числа с плавающей точкой. Они обсуждаются в приложении Б. Их длина составляет 32,64, а иногда и 128 битов. В большинстве компьютеров есть команды для выполнения операций с числами с плавающей точкой. Во многих компьютерах имеются отдельные регистры для целочисленных операндов и для операндов с плавающей точкой.

Некоторые языки программирования, в частности COBOL, допускают в качестве типа данных десятичные числа. Машины, предназначенные для программ на языке COBOL, часто поддерживают десятичные числа в аппаратном обеспечении,

обычно кодируя десятичный разряд в 4 бита и затем объединяя два десятичных разряда в байт (двоично-десятичный формат). Однако результаты арифметических действий над такими десятичными числами будут некорректны, поэтому требуются специальные команды для коррекции десятичной арифметики. Эти команды должны знать выход переноса бита 3. Вот почему код условия часто содержит бит служебного переноса. Между прочим, проблема 2000 года была вызвана программистами на языке COBOL, которые решили, что дешевле будет представлять год в виде двух десятичных разрядов, а не в виде 16-битного двоичного числа.

Нечисловые типы данных

Хотя самые первые компьютеры работали в основном с числами, современные компьютеры часто используются для нечисловых приложений, например, для обработки текстов или управления базой данных. Для этих приложений нужны другие, нечисловые, типы данных. Они часто поддерживаются командами уровня архитектуры команд. Здесь очень важны символы, хотя не каждый компьютер обеспечивает аппаратную поддержку для них. Наиболее распространенными символьными кодами являются ASCII и UNICODE. Они поддерживают 7-битные и 16-битные символы соответственно. Эти коды обсуждались в главе 2.

На уровне команд часто имеются особые команды, предназначенные для операций с цепочками символов. Эти цепочки иногда разграничиваются специальным символом в конце. Вместо этого для определения конца цепочки может использоваться поле длины цепочки. Команды могут выполнять копирование, поиск, редактирование цепочек и другие действия.

Кроме того, важны значения булевой алгебры. Этих значений два: истина и ложь. Теоретически булево значение может представлять один бит: 0 — ложь, а 1 — истина (или наоборот). На практике же используется байт или слово, поскольку отдельные биты в байте не имеют собственных адресов, и следовательно, к ним трудно обращаться. В обычных системах применяется следующее соглашение: 0 означает ложь, а все остальное означает истину.

Единственная ситуация, в которой булево значение представлено 1 битом, — это когда имеется целый массив бит и 32-битное слово может содержать 32 булевых значения. Такая структура данных называется битовым отображением. Она встречается в различных контекстах. Например, битовое отображение может использоваться для того, чтобы следить за свободными блоками на диске. Если диск содержит n блоков, тогда битовое отображение содержит n битов.

Последний тип данных — это указатели, которые представляют собой машинные адреса. Мы уже неоднократно рассматривали указатели. В машинах Mic-д: регистры SP, PC, LV и CPP — это примеры указателей. Доступ к переменной на фиксированном расстоянии от указателя (а именно так работает команда I LOAD) широко применяется на всех машинах.

Типы данных процессора Pentium II

Pentium II поддерживает двоичные целые числа со знаком, целые числа без знака, числа двоично-десятичной системы счисления и числа с плавающей точкой по стандарту IEEE 754 (табл. 5.2). Эта машина является 8-, 16-разрядной и оперирует с

целыми числами такой длины. У нее имеются многочисленные арифметические команды, булевы операции и операции сравнения. Операнды необязательно должны быть выровнены в памяти, но если адреса слов кратны 4 байтам, то наблюдается более высокая производительность.

Таблица 5.2. Числовые типы данных процессора Pentium II. Поддерживаемые типы отмечены крестом (x)

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	x	x	x		
Целые числа без знака	x	x	x		
Двоично-десятичные целые числа	x				
Числа с плавающей точкой			x	x	

Pentium II также может манипулировать 8-разрядными символами ASCII: существуют специальные команды для копирования и поиска цепочек символов. Эти команды используются и для цепочек, длина которых известна заранее, и для цепочек, в конце которых стоит специальный маркер. Они часто используются в библиотеках операций над строками.

Типы данных машины UltraSPARC II

UltraSPARC II поддерживает широкий ряд форматов данных (табл. 5.3). Эта машина может поддерживать 8-, 16-, 32- и 64-битные целочисленные операнды со знаком и без знака. Целые числа со знаком используют дополнительный код. Кроме того, имеются операнды с плавающей точкой по 32,64 и 128 битов, которые соответствуют стандарту ШЕЕ 754 (для 32-битных и 64-битных чисел). Двоично-десятичные числа не поддерживаются. Все операнды должны быть выровнены в памяти.

Таблица 5.3. Числовые типы данных компьютера UltraSPARC II

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	x	x	x	x	
Целые числа без знака	x	x	x	x	
Двоично-десятичные целые числа					
Числа с плавающей точкой			x	x	x

UltraSPARC II представляет собой регистровую структуру, и почти все команды оперируют с 64-разрядными регистрами. Символьные и строковые типы данных специальными командами аппаратного обеспечения не поддерживаются.

Типы данных виртуальной машины Java

Java — это язык со строгим контролем типов. Это значит, что каждый операнд имеет особый тип и размер, который известен в период компиляции. Это отражено в типах, поддерживаемых JVM. JVM поддерживает числовые типы, приведенные в табл. 5Л. Целые числа со знаком используют дополнительный код.

Целые числа без знака в языке Java не присутствуют и не поддерживаются JVM, как и двоично-десятичные числа.

Таблица 5.4. Числовые типы данных для JVM

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	×	×	×	×	
Целые числа без знака					
Двоично-десятичные целые числа					
Числа с плавающей точкой			×	×	

JVM поддерживает символы, но не традиционные 8-битные символы ASCII, а 16-битные символы UNICODE. Указатели поддерживаются главным образом для внутреннего использования компилятора и системы обслуживания. Пользовательские программы не могут непосредственно обращаться к указателям. Указатели используются в основном для ссылок на объекты.

Форматы команд

Команда состоит из кода операции и некоторой дополнительной информации, например, откуда поступают операнды и куда должны отправляться результаты. Процесс определения, где находятся операнды (то есть их адреса), называется адресацией.

На рисунке 5.5 показано несколько возможных форматов для команд второго уровня. Команды всегда содержат код операции, который сообщает, какие действия выполняет команда. В команде может присутствовать ноль, один, два или три адреса.

Рис. 5.5. Четыре формата команд: безадресная команда (а); одноадресная команда (б); двухадресная команда (в); трехадресная команда (г)

В одних машинах все команды имеют одинаковую длину; в других команды могут быть разной длины. Команды могут быть короче слова, длиннее слова или быть равными слову по длине. Если все команды одной длины, то это упрощает декодирование, но часто требует большего пространства, поскольку все команды должны быть такой же длины, как самая длинная. На рис. 5.6 показано несколько возможных соотношений между длиной команды и длиной слова.

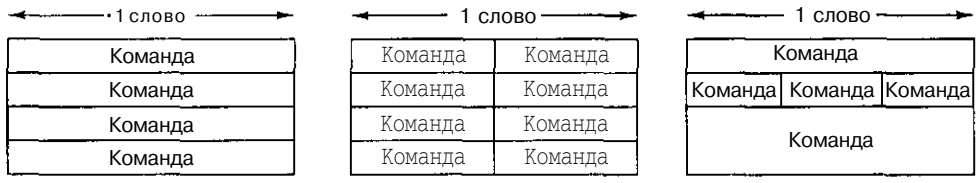


Рис. 5.6. Некоторые возможные отношения между длиной команды и длиной слова

Критерии разработки для форматов команд

Если разработчикам нужно выбрать форматы команд для их машины, они должны рассмотреть ряд факторов. Нельзя недооценивать сложность этого решения. Если компьютер получился удачным с коммерческой точки зрения, то набор команд может продолжать существовать 20 лет и более. Имеет огромное значение способность добавлять новые команды и использовать другие возможности, которые появляются на протяжении определенного периода времени, но только в том случае, если архитектура и компания, создавшая эту архитектуру, существуют достаточно долго.

Эффективность конкретной архитектуры команд зависит от технологии, которая применялась при разработке компьютера. За длительный период времени эта технология будет подвергнута значительным изменениям, и некоторые характеристики архитектуры команд окажутся неудачными (если оглянуться на прошлое). Например, если доступ к памяти осуществляется быстро, то подойдет стековая архитектура (как в JVM), но если достун к памяти медленный, тогда желательно иметь много регистров (как в UltraSPARC II). Тем читателям, которые считают, что этот выбор сделать просто, мы предлагаем взять лист бумаги и записать следующие предположения: 1) какова будет типичная скорость тактового генератора через 20 лет и 2) каково будет типичное время доступа к ОЗУ через 20 лет. Аккуратно сложите этот лист бумаги и спрячьте его в надежном месте, а через 20 лет разверните и прочитайте, что на нем написано. Те из вас, кто принял этот вызов, могут не использовать лист бумаги, а просто отправить свои предсказания в Интернет.

Даже дальновидные разработчики не всегда могут сделать правильный выбор. И даже если бы они могли его сделать, они бы просуществовали недолго, поскольку если такая развитая архитектура команд будет стоить дороже, чем архитектуры у конкурентов, то компания долго не продержится.

Если речь идет об одинаковых машинах, то лучше иметь короткие команды, чем длинные. Программа, состоящая из n 16-битных команд, занимает в два раза меньше пространства памяти, чем программа из n 32-битных команд. Поскольку цены на память постоянно падают, этот фактор не имел бы значения в будущем, если бы программное обеспечение не разрасталось гораздо быстрее, чем происходит снижение цен на память.

Более того, минимизация размера команд может усложнить их декодирование и перекрытие. Следовательно, достижение минимального размера команды должно уравниваться со временем, затрачиваемым на декодирование и выполнение команд.

Есть еще одна очень важная причина минимизации длины команд, и она становится все важнее с увеличением скорости работы процессоров: пропускная способность памяти (число битов в секунду, которое память может предоставлять). Значительный рост скорости работы процессора за последнее десятилетие не соответствует увеличению пропускной способности памяти. Ограничения здесь связаны с неспособностью системы памяти передавать команды и операнды с той же скоростью, с какой процессор может обрабатывать их. Пропускная способность памяти зависит от технологии разработки. Трудности, связанные с пропускной способностью, имеют отношение не только к основной памяти, но и ко всем видам кэш-памяти.

Если пропускная способность кэш-памяти команд составляет t бит/с, а средняя длина команды r битов, то кэш-память способна передавать самое большее t/r команд в секунду. Отметим, что это *верхний предел* скорости, с которой процессор может выполнять команды, хотя в настоящее время предпринимаются попытки преодолеть этот барьер. Ясно, что скорость, с которой могут выполняться команды (то есть скорость работы процессора), может ограничиваться длиной команд. Чем короче команды, тем быстрее работает процессор. Поскольку современные процессоры способны выполнять несколько команд за один цикл, то вызов нескольких команд за цикл обязателен. Этот аспект кэш-памяти команд делает размер команд важным критерием, который нужно учитывать при разработке.

Второй критерий разработки — достаточный объем пространства в формате команды для выражения всех требуемых операндов. Машина с 2" операциями и со всеми командами менее п битов невозможна. В этом случае в коде операции не было бы достаточно места для того, чтобы указать, какая нужна команда. И история снова и снова доказывает, что обязательно нужно оставлять большое количество свободных кодов операций для будущих дополнений к набору команд.

Третий критерий связан с числом битов в адресном поле. Рассмотрим проект машины с 8-битными символами и основной памятью, которая должна содержать 2^{32} символов. Разработчики вольны были приписать последовательные адреса блокам по 8, 16, 24 или 32 бита.

Представим, что бы случилось, если бы команда разработчиков разбилась на две воюющие группы, одна из которых утверждает, что основной единицей памяти должен быть 8-битный байт, а другая требует, чтобы основной единицей памяти было 32-битное слово. Первая группа предложила бы память из 2^{32} байтов с номерами О, 1,2,3, ...,4 294 967 295. Вторая группа предложила бы память из 2^{30} слов с номерами 0,1,2,3, ...,1073 741823.

Первая группа скажет, что для того чтобы сравнить два символа при организации по 32-битным словам, программе приходится не только вызывать из памяти слова, содержащие эти символы, но и выделять соответствующий символ из каждого слова для сравнения. А это потребует наличия дополнительных команд и, следовательно, дополнительного пространства. 8-битная организация, напротив, обеспечивает адрес для каждого символа, что значительно упрощает процедуру сравнения.

Сторонники 32-битной организации скажут, что их проект требует всего лишь 2^{30} отдельных адресов, что дает длину адреса всего 30 битов, тогда как при 8-битной организации требуется целых 32 бита для обращения к той же самой памяти.

Если адрес короткий, то и команда будет более короткой. Она будет занимать меньше пространства в памяти, и к тому же для ее вызова потребуется меньше времени. В качестве альтернативы они могут сохранить 32-битный адрес для обращения к памяти в 16 Гбайт вместо каких-то там 4 Гбайт.

Этот пример демонстрирует, что для получения оптимальной дискретности памяти требуются более длинные адреса и, следовательно, более длинные команды. Одна крайность — это организация памяти, при которой адресуется каждый бит (например, Burroughs B1700). Другая крайность — это память, состоящая из очень длинных слов (например, серия CDC Cyber содержала 60-битные слова).

Современные компьютерные системы пришли к компромиссу, который, в каком-то смысле, объединил в себе худшие качества обоих вариантов. Они требуют, чтобы адреса были у отдельных байтов, но при обращении к памяти всегда считывается одно, два, а иногда даже четыре слова сразу. В результате считывания одного байта из памяти на машине UltraSPARC II вызывается сразу минимум 16 байтов (см. рис. 3.44), а иногда и вся строка кэш-памяти в 64 байта.

Расширение кода операций

В предыдущем разделе мы увидели, что короткие адреса противостоят удачной дискретности памяти. В этом разделе мы рассмотрим компромиссы, связанные с кодами операций и адресами. Рассмотрим команду размером (n+k) битов с кодом операции в к битов и одним адресом в п битов. Такая команда допускает $2^к$ различных операций и 2" адресуемых ячеек памяти. В качестве альтернативы те же (n+k) битов можно разбить на код операции в (к-1) битов и адрес в (п+1) битов. При этом будет либо в два раза меньше команд, но в два раза больше памяти, либо то же количество памяти, но дискретность вдвое выше. Код операции на (к+1) битов и адрес в (п-1) битов дает большее количество операций, но ценой этого преимущества является либо меньшее количество ячеек памяти, либо не очень удачная дискретность при том же объеме памяти. Наряду с простыми компромиссами между битами кода операции и битами адреса, которые были только что описаны, возможны и более сложные. Схема, которая будет обсуждаться в следующих разделах, называется расширением кода операций.

Понятие расширения кода операций можно пояснить на примере. Рассмотрим машину, в которой длина команд составляет 16 битов, а длина адресов — 4 бита, как показано на рис. 5.7. Эта ситуация вполне разумна для машины, содержащей 16 регистров (а следовательно, 4-битный адрес регистра), над которыми совершаются все арифметические операции. Один из возможных вариантов — наличие в каждой команде 4-битного кода операции и трех адресов, что дает 16 трехадресных команд.

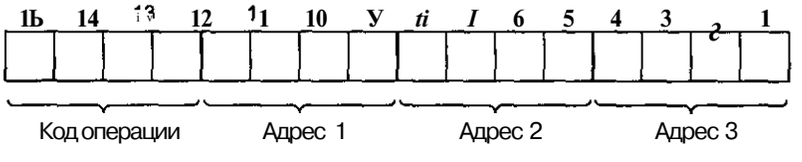


Рис. 5.7. Команда с 4-битным кодом операции и тремя 4-битными адресными полями

Если разработчикам нужно 15 трехадресных команд, 14 двухадресных команд, 31 одноадресная команда и 16 безадресных команд, они могут использовать коды операций от 0 до 14 в качестве трехадресных команд, а код операции 15 уже интерпретировать по-другому (рис. 5.8).

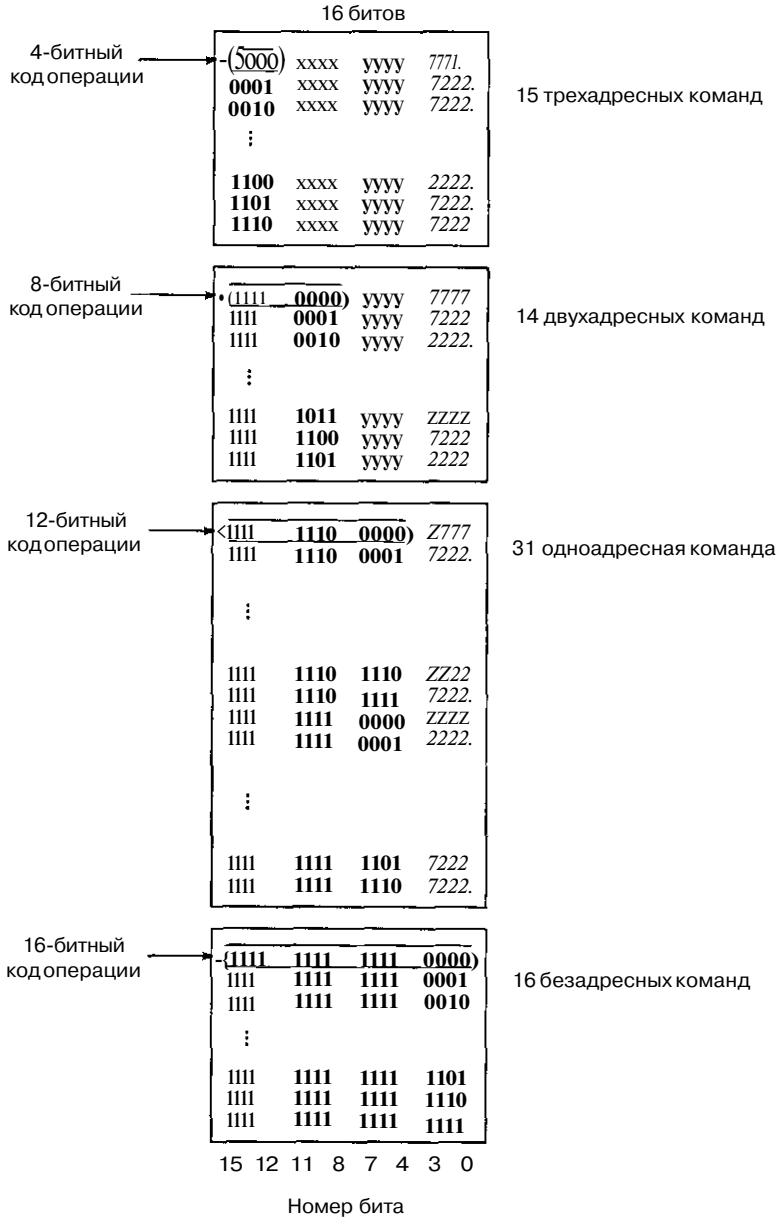


Рис. 5.8. Расширение кода операции допускает 15 трехадресных команд, 14 двухадресных команд, 31 одноадресную команду и 16 безадресных команд. Поля xxxx, *yyyy* и *zzz2* — это 4-битные адресные поля

Это значит, что код операции 15 содержится в битах с 8 по 15, а не с 12 по15. Биты с 0 по 3 и с 4 по 7, как и раньше, формируют два адреса. Все 14 двухадресных команд содержат число 1111 в старших четырех битах и числа от 0000 до 1101 в битах с 8 по 11. Команды с числом 1111 в старших четырех битах и числом 1110 или 1111 в битах с 8 по 11 будут рассматриваться особо. Они будут трактоваться таким образом, как будто их коды операций находятся в битах с 4 по 15. В результате получаем 32 новых кода операций. А поскольку требуется всего 31 код, то код 111111111111 означает, что действительный код операции находится в битах с О по 15, что дает 16 безадресных команд.

Как видим, код операции становится все длиннее и длиннее: трехадресные команды имеют 4-битный код операции, двухадресные команды — 8-битный код операции, одноадресные команды — 12-битный код операции, а безадресные команды — 16-битный код операции.

Идея расширения кода операций наглядно демонстрирует компромисс между пространством для кодов операций и пространством для другой информации. Однако на практике все не так просто и понятно, как в нашем примере. Есть только два способа изменения размера кодов операций. С одной стороны, можно иметь все команды одинаковой длины, приписывая самые короткие коды операций тем командам, которым нужно больше всего битов для спецификации чего-либо другого. С другой стороны, можно свести к минимуму *средний* размер команды, если выбрать самые короткие коды операций для часто используемых команд и самые длинные — для редко используемых команд.

Если довести эту идею до конца, можно свести к минимуму среднюю длину команды, закодировав каждую команду, чтобы максимально уменьшить число требуемых битов. К сожалению, это приведет к наличию команд разных размеров, которые не будут выровнены в границах байтов. Пока существуют архитектуры команд с таким свойством (например Intel 432), выравнивание будет иметь большое значение для быстрого декодирования команд. Тем не менее эта идея часто применяется на уровне байтов. Ниже мы рассмотрим архитектуру команд JVM, чтобы показать, как можно менять форматы команд, чтобы максимально уменьшить размер программы.

Форматы команд процессора Pentium II

Форматы команд процессора Pentium II очень сложны и нерегулярны. Они содержат до шести полей разной длины, пять из которых факультативны. Общая модель показана на рис. 5.9. Эта ситуация сложилась из-за того, что архитектура развивалась на протяжении нескольких поколений и ранее в нее были включены не очень удачно выбранные характеристики. Из-за требования обратной совместимости позднее их нельзя было изменить. Например, если один из операндов команды находится в памяти, то другой может и не находиться в памяти. Следовательно, существуют команды сложения двух регистров, команды прибавления регистра к слову из памяти и команды прибавления слова из памяти к регистру, но не существует команд для прибавления одного слова памяти к другому слову памяти.

В первых архитектурах Intel все коды операций были размером 1 байт, хотя для изменения некоторых команд часто использовался так называемый префиксный

байт. Префиксный байт — это дополнительный код операции, который ставится перед командой, чтобы изменить ее действие. Примером префиксного байта может служить команда WIDE в машинах IJVM и JVM. К сожалению, в какой-то момент компания Intel вышла за пределы кодов операций, и один код операции, OxFF, определялся как **код смены алфавита** и использовался для разрешения второго байта команды.

Отдельные биты в кодах операций процессора Pentium II дают довольно мало информации о команде. Единственной структурой такого рода в поле кода операции является младший бит в некоторых командах, который указывает, что именно вызывается — слово или байт, а также соседний бит, который указывает, является ли адрес памяти (если он вообще есть) источником или пунктом назначения. Таким образом, в большинстве случаев код операции должен быть полностью декодирован, чтобы установить, к какому классу относится операция, которую нужно выполнить, и, следовательно, какова длина этой команды. Это очень сильно снижает производительность, поскольку необходимо производить декодирование еще до того, как будет определено, где начинается следующая команда.

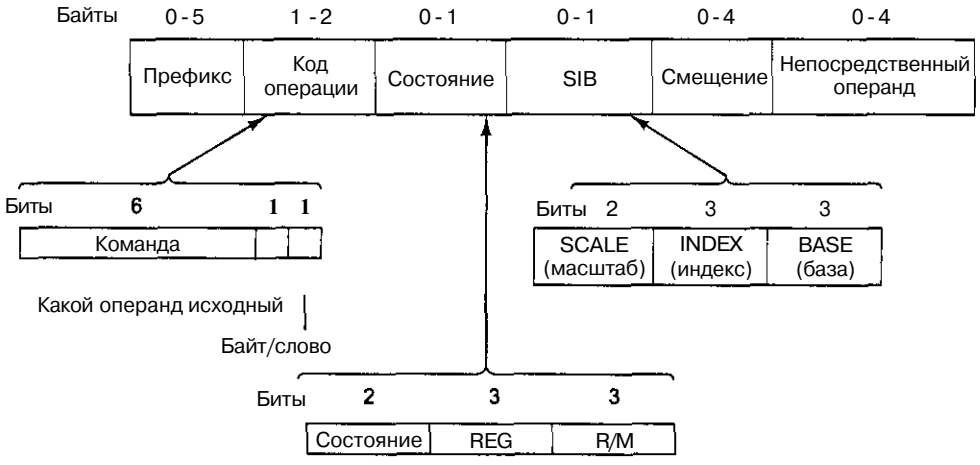


Рис. 5.9. Форматы команд процессора Pentium II

В большинстве команд вслед за байтом кода операции, который указывает местонахождение операнда в памяти, следует второй байт, который сообщает всю информацию об операнде. Эти 8 битов распределены в 2-битном поле MOD и двух 3-битных регистровых полях REG и R/M. Иногда первые три бита этого байта используются в качестве расширения для кода операции, давая в сумме 11 битов для кода операции. Тем не менее 2-битное поле означает, что существует только 4 способа обращения к операндам, и один из операндов всегда должен быть регистром. Логически должен быть определяем любой из регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но правила кодирования команд запрещают некоторые комбинации, поскольку эти комбинации используются для особых случаев. В некоторых типах команд требуется дополнительный байт, называемый **SIB (Scale, Index,** Base — масштаб, индекс, база), который дает дополнительную спецификацию. Эта схема не идеальна, но она является компромиссом между требованием

обратной совместимости и желанием добавлять новые особенности, которые не были предусмотрены изначально.

Добавим еще, что некоторые команды имеют 1, 2 или 4 дополнительных байта для определения адреса команды (смещение), а иногда еще 1,2 или 4 байта, содержащих константу (непосредственный операнд).

Форматы команд процессора UltraSPARC II

Архитектура команд процессора UltraSPARC II состоит из 32-битных команд, выровненных в памяти. Команды очень просты. Каждая из них определяет только одно действие. Типичная команда указывает два регистра, из которых поступают входные операнды, и один выходной регистр. Вместо одного из регистров команда может использовать константу со знаком. При выполнении команды LOAD два регистра (или один регистр и 13-битная константа) складываются вместе для определения адреса памяти, который нужно считать. Данные оттуда записываются в другой указанный регистр.

Изначально машина SPARC имела ограниченное число форматов команд. Они показаны на рис. 5.10. Со временем добавлялись новые форматы. Когда писалась эта книга, число форматов уже было равно 31. Большинство новых вариантов были получены путем отнимания нескольких битов из какого-нибудь поля. Например, изначально для команд перехода использовался формат 3 с 22-битным смещением. Когда были добавлены прогнозируемые переходы, 3 из 22 битов убирались: один из них стал использоваться для прогнозирования (совершать или не совершать переход), а два оставшихся определяли, какой набор битов условного кода нужно использовать. В результате получилось 19-битное смещение. Приведем другой пример. Существует много команд для переделывания одного типа данных в другой (целые числа в числа с плавающей точкой и т. д.). Для большинства этих команд используется вариант формата lb, в котором поле непосредственной константы разбито на 5-битное поле, указывающее входной регистр, и 8-битное поле, которое обеспечивает дополнительные биты кода операции. Однако в большинстве команд все еще используются форматы, показанные на рисунке.

Первые два бита каждой команды помогают определить формат команды и сообщают аппаратному обеспечению, где найти оставшуюся часть кода операции, если она есть. В формате 1a оба источника операндов представляют собой регистры; в формате lb один источник — регистр, а второй — константа в промежутке от -4096 до +4095. Бит 13 определяет один из этих двух форматов. (Биты нумеруются с 0.) В обоих случаях местом сохранения результатов всегда является регистр. Достаточный объем пространства обеспечен для 64 команд, некоторые из которых сохранены на будущее.

Поскольку все команды 32-битные, включить в команду 32-битную константу невозможно. Команда SETHI устанавливает 22 бита, оставляя пространство для другой команды, чтобы установить оставшиеся 10 битов. Это единственная команда, которая использует данный формат.

Для непрогнозируемых условных переходов используется формат 3, в котором поле УСЛОВИЕ определяет, какое условие нужно проверить. Бит A нужен для

того, чтобы избегать пустых операций при определенных условиях. Прогнозируемые переходы используют тот же формат, но только с 19-битным смещением, как было сказано выше.

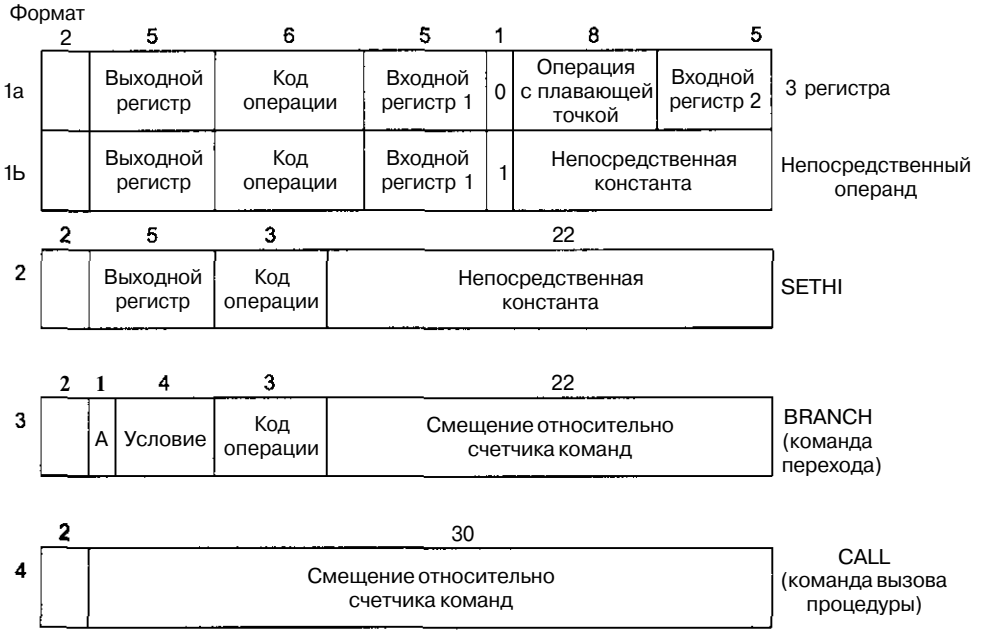


Рис. 5.10. Изначальные форматы команд процессора SPARC

Последний формат используется для команды вызова процедуры (CALL). Эта команда особая, поскольку только в ней для определения адреса требуется 30 битов. В данной архитектуре существует один 2-битный код операции. Требуемый адрес — это целевой адрес, разделенный на четыре.

Форматы команд JVM

Большинство форматов команд машины JVM чрезвычайно просты. Все форматы показаны на рис. 5.11. Их простота объясняется тем, что машина JVM сравнительно новая. Но подождите 10 лет. Все команды начинаются с кода операции в 1 байт. В некоторых командах за кодом операции следует второй байт. Это может быть индекс (как в команде ILOAD), константа (как в команде BIPUSH) или указатель типа данных (как в команде NEWARRAY, которая создает одномерный массив указанного типа в «куче»). Третий формат по сути такой же, как и второй, только вместо 8-битной константы там присутствует 16-битная константа (как, например, у команд WIDE ILOAD или GOTO). Формат 4 используется только для команды IINC. Формат 5 используется только для команды MULTINEWARRAY, которая создает многомерный массив «в куче». Формат 6 нужен только для команды INVOKEINTERFACE, которая вызывает процедуру при определенных обстоятельствах. Формат 7 предназначен только для команды WIDE IINC, чтобы обеспечить 16-битный индекс и 16-битную

константу, которая прибавляется к выбранной переменной. Формат 8 применяется только для команд WIDE GOTO и WIDE JSR, чтобы осуществлять переходы на большие расстояния в памяти и вызовы определенных процедур. Последний формат используется только двумя командами, и обе эти команды нужны для реализации оператора языка Java switch. Таким образом, все команды JVM, за исключением восьми особых команд, используют простые и короткие форматы 1, 2 и 3.

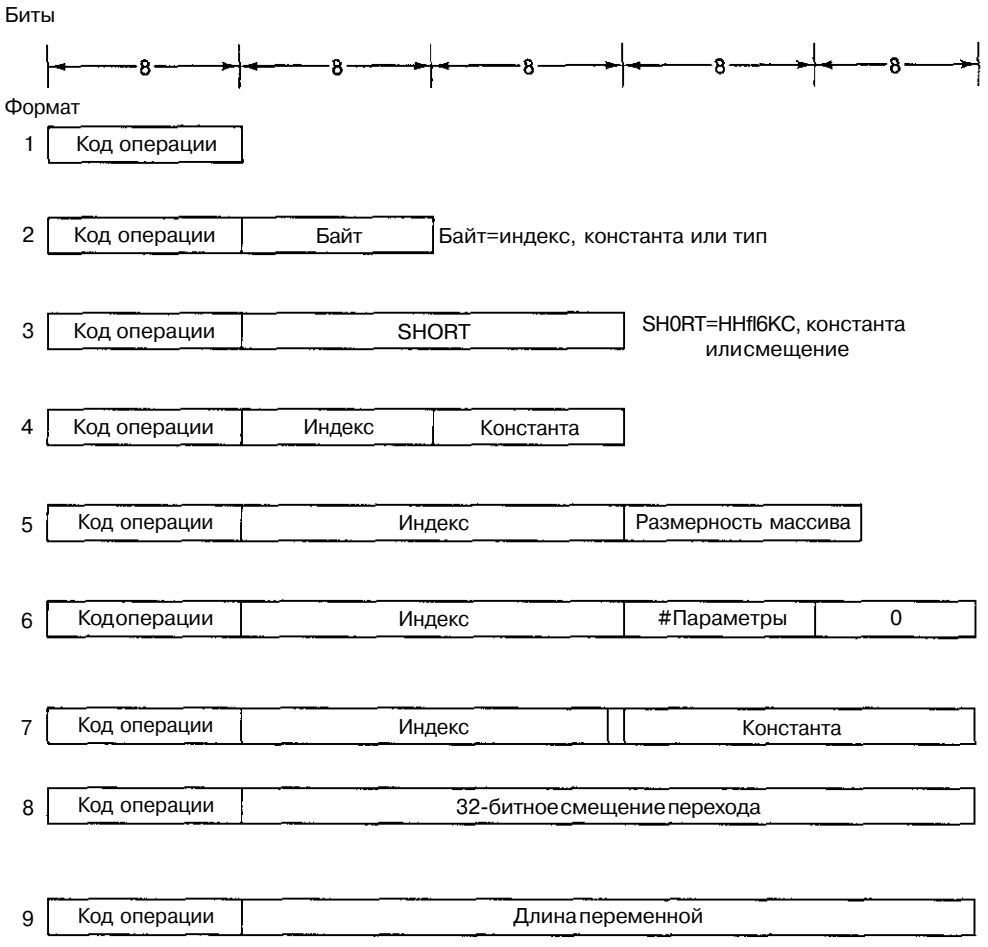


Рис. 5.11. Форматы команд JVM

В действительности дела обстоят гораздо лучше, чем может показаться. Команды виртуальной машины Java кодируются таким образом, чтобы большинство наиболее распространенных команд кодировались в одном байте. Большинство из 256 возможных команд, кодируемых в одном байте, представляют собой особые случаи общей формы команд IJVM.

Давайте, например, рассмотрим, как в машине Java происходит загрузка локальной переменной. Существует три различных способа определения локальной переменной. Самый короткий вариант покрывает самые распространенные случаи,

а самый длинный — все возможные случаи. JVM содержит команду ILOAD, использующую 8-битный индекс для определения локальной переменной, которую нужно поместить в стек. Мы также показали, как префикс WIDE позволяет использовать тот же код операции для определения любого из первых 65 536 элементов во фрейме локальных переменных. Для команды WIDE ILOAD требуется 4 байта: 1 — для WIDE, 1 — для ILOAD и 2 — для 16-битного индекса. Такое разделение объясняется тем, что большинство команд ILOAD используют одну из первых 256 локальных переменных. Префикс WIDE нужен для универсальности, применимости к любым ситуациям, и используется он редко.

Но разработчики машины JVM пошли еще дальше. Так как параметры процедуры передаются в первые несколько слов фрейма локальных переменных, команда ILOAD чаще всего использует элементы фрейма локальных переменных с невысокими индексами. Разработчики JVM решили, что стоит назначить отдельные 1-байтные коды операций для каждой из возможных комбинаций. Команда ILOAD_O помещает в стек локальную переменную 0. Эта команда полностью эквивалентна 2-байтной команде ILOAD 0, за исключением того, что она занимает один байт вместо двух. Точно также команды ILOAD_1, ILOADJ? и IL0AD_3 (коды операций OxlB, OxlC и OxlD соответственно) помещают в стек локальные переменные 1, 2 и 3. Отметим, что локальную переменную 1, например, можно загрузить одним из трех способов: ILOAD_1, ILOAD 1 и WIDE ILOAD 1.

Многие команды имеют варианты, подобные этим. Существуют специальные тоъшвдл, полностью эквивалентные BIPUSH, для значений 0,1, 2, 3,4, 5, а также -1. Есть, кроме того, особые команды для записи переменных из стека в первые 4 слова пространства локальных переменных.

Отметим, что эти варианты повлекли за собой некоторые убытки. Только 256 различных команд могут определяться в одном байте. Поскольку 4 из этих 256 команд решили отвести на загрузку первых четырех локальных переменных, число команд уменьшилось на 4. Эти команды вместе с основной командой ILOAD в сумме составляют 5 команд. Префикс WIDE тоже использует одно из 256 возможных значений (а это даже не команда, а просто префикс), но он применяется к различным кодам операций.

Для спецификации загрузки операндов из набора констант разработчики использовали немного другой метод, поскольку они ожидали различия в распределении адресов. Они предоставили две версии команды: LDC и LDC_W. Вторая форма команды (она была включена в IJVM) может вызывать любое из 65 536 слов в наборе констант. В первой форме требуется только однобайтный индекс, но такая команда может вызывать только одно их первых 256 слов. Вызов любого из первых 256 слов можно осуществить с помощью 2-байтной команды, а вызов любого слова — с помощью 3-байтной команды. На эти 2 варианта требуется 2 кода из 256 кодов операций. Набор команд был бы более простым и регулярным, если бы разработчики выбрали ту же технологию, которую они использовали для команды ILOAD, то есть использовали бы префикс WIDE, а не команду LDC_W. Однако в этом случае для вызова констант из верхних 256 слов потребовалось бы 4 байта, а не 3.

Технология объединения кодов операций и индексов в один байт, а также размещения 256 доступных байтов в соответствии с частотой их использования была впервые предложена автором данной книги в 1978 году, но нашла применение только спустя два десятилетия [145].

Адресация

Разработка кодов операций является важной частью архитектуры команд. Однако значительное число битов программы используется для того, чтобы определить, откуда нужно брать операнды, а не для того, чтобы узнать, какие операции нужно выполнить. Рассмотрим команду ADD, которая требует спецификации трех операндов: двух источников и одного пункта назначения. (Термин «операнд» обычно используется применительно ко всем трем элементам, хотя пункт назначения — это место, где сохраняется результат.) Так или иначе команда ADD должна сообщать, где найти операнды и куда поместить результат. Если адреса памяти 32-битные, то спецификация этой команды требует помимо кода операции еще три 32-битных адреса. Адреса занимают гораздо больше бит, чем коды операции.

Два специальных метода предназначены для уменьшения размера спецификации. Во-первых, если операнд должен использоваться несколько раз, его можно переместить в регистр. В использовании регистра для переменной есть двойная польза: скорость доступа увеличивается, а для определения операнда требуется меньшее количество битов. Если имеется 32 регистра, любой из них можно определить, используя всего лишь 5 битов. Если при выполнении команды ADD применять только регистровые операнды, для определения всех трех операндов понадобится только 15 битов, а если бы эти операнды находились в памяти, понадобилось бы целых 96 битов.

Однако использование регистров может вызвать другую проблему. Если операнд, находящийся в памяти, должен сначала загружаться в регистр, то потребуется большее число битов для определения адреса памяти. Во-первых, для переноса операнда в регистр нужна команда LOAD. Для этого требуется не только код операции, но и полный адрес памяти, а также нужно определить целевой регистр. Поэтому если операнд используется только один раз, помещать его в регистр не стоит.

К счастью, многочисленные измерения показали, что одни и те же операнды используются многократно. Поэтому большинство новых архитектур содержат большое количество регистров, а большинство компиляторов доходят до огромных размеров, чтобы хранить локальные переменные в этих регистрах, устраняя таким образом многочисленные обращения к памяти. Это сокращает и размер, и время выполнения программы.

Второй метод подразумевает определение одного или нескольких операндов неявным образом. Для этого существует несколько технологий. Один из способов — использовать одну спецификацию для входного и выходного операндов. В то время как обычная трехадресная команда ADD использует форму

```
0ESTINATI0N=S0URSEI+S0URSE2.
```

двухадресную команду можно сократить до формы

```
REGISER2-REGISTER2+S0URSE1.
```

Недостаток этой команды состоит в том, что содержимое REGISTER2 не сохранится. Если первоначальное значение понадобится позднее, его нужно сначала скопировать в другой регистр. Компромисс здесь заключается в том, что двухадресные команды короче, но они не так часто используются. У разных разработчи-

ков разные предпочтения. В Pentium II, например, используются двухадресные команды, а в UltraSPARC II — трехадресные.

Мы сократили число операндов команды ADD с трех до двух. Продолжим сокращение дальше. Первые компьютеры имели только один регистр, который назывался аккумулятором. Команда ADD, например, всегда прибавляла слово из памяти к аккумулятору, поэтому нужно было определять только один операнд (операнд памяти). Эта технология хорошо работала для простых вычислений, *но* когда были нужны промежуточные результаты, аккумулятор приходилось записывать обратно в память, а позднее вызывать снова. Следовательно, эта технология нам не подходит.

Итак, мы перешли от трехадресной команды ADD к двухадресной, а затем к одноадресной. Что же остается? Ноль адресов? Да. В главе 4 мы увидели, как машина IJVM использует стек. Команда IADD не имеет адресов. Входные и выходные операнды не показываются явным образом. Ниже мы рассмотрим стековую адресацию более подробно.

Способы адресации

До сих пор мы не рассказывали о том, как интерпретируются биты адресного поля для нахождения операнда. Один из возможных вариантов состоит в том, что они содержат адрес операнда. Помимо огромного поля, необходимого для определения полного адреса памяти, данный метод имеет еще одно ограничение: этот адрес должен определяться во время компиляции. Существуют и другие возможности, которые обеспечивают более короткие спецификации, а также могут определять адреса динамически. В следующих разделах мы рассмотрим некоторые из этих форм, которые называются **способами адресации.**

Непосредственная адресация

Самый простой способ определения операнда — содержать в адресной части сам операнд, а не адрес операнда или какую-либо другую информацию, описывающую, где находится операнд. Такой операнд называется **непосредственным операндом,** поскольку он автоматически вызывается из памяти одновременно с командой; следовательно, он сразу непосредственно становится доступным. Один из вариантов команды с непосредственным адресом для загрузки в регистр R1 константы 4 показан на рис. 5.12.

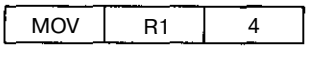


Рис. 5.12. Команда с непосредственным адресом для загрузки константы 4 в регистр 1

При непосредственной адресации не требуется дополнительного обращения к памяти для вызова операнда. Однако у такого способа адресации есть и некоторые недостатки. Во-первых, таким способом можно работать только с константами. Во-вторых, число значений ограничено размером поля. Тем не менее эта технология используется во многих архитектурах для определения целочисленных констант.

Прямая адресация

Следующий способ определения операнда — просто дать его полный адрес. Такой способ называется **прямой адресацией.** Как и непосредственная адресация, прямая адресация имеет некоторые ограничения: команда всегда будет иметь доступ только к одному и тому же адресу памяти. То есть значение может меняться, а адрес — нет. Таким образом, прямая адресация может использоваться только для доступа к глобальным переменным, адреса которых известны во время компиляции. Многие программы содержат глобальные переменные, поэтому этот способ широко используется. Каким образом компьютер узнает, какие адреса непосредственные, а какие прямые, мы обсудим позже.

Регистровая адресация

Регистровая адресация по сути сходна с прямой адресацией, только в данном случае вместо ячейки памяти определяется регистр. Поскольку регистры очень важны (из-за быстрого доступа и коротких адресов), этот способ адресации является самым распространенным на большинстве компьютеров. Многие компиляторы доходят до огромных размеров, чтобы определить, к каким переменным доступ будет осуществляться чаще всего (например, индекс цикла), и помещают эти переменные в регистры.

Такой способ адресации называют **регистровой адресацией. В** архитектурах с загрузкой с запоминанием, например UltraSPARC II, практически все команды используют исключительно этот способ адресации. Он не используется только в том случае, когда операнд перемещается из памяти в регистр (команда LOAD) или из регистра в память (команда STORE). Даже в этих командах один из операндов является регистром — туда отправляется слово из памяти **или** оттуда перемещается слово в память.

Косвенная регистровая адресация

При таком способе адресации определяемый операнд берется из памяти или отправляется в память, но адрес не зафиксирован жестко в команде, как при прямой адресации. Вместо этого адрес содержится в регистре. Если адрес используется таким образом, он называется **указателем.** Преимущество косвенной адресации состоит в том, что можно обращаться к памяти, не имея в команде полного адреса. Кроме того, при разных выполнениях данной команды можно использовать разные слова памяти.

Чтобы понять, почему может быть полезно использование разных слов при каждом выполнении команды, представим себе цикл, который проходит по 1024-элементному одномерному массиву целых чисел для вычисления суммы элементов в регистре R1. Вне этого цикла какой-то другой регистр, например R2, может указывать первый элемент массива, а еще один регистр, например R3, может указывать первый адрес после массива. Массив содержит 1024 целых числа по 4 байта каждое. Если массив начинается с A, то первый адрес после массива будет A+4096. Типичная программа ассемблера, выполняющая это вычисление для двухадресной машины, показана в листинге 5.1.

Листинг 5.1. Программа на ассемблере для вычисления суммы элементов массива

```
        MOVRI.#0        накопление суммы в R1. изначально 0
        MOV R2,#A       ;R2=aflpec массива A
        MOV R3,#A+4096  ;R3=aflpec первого слова после A
LOOP:   ADD R1.(R2)      получение операнда через регистр R2
        ADD R2,#4        увеличение R2 на одно слово(4байта)
        CMP R2.R3        ;проверка на завершение
        BLT LOOP         :если R2<R3. продолжать цикл
```

В этой маленькой программе мы использовали несколько способов адресации. Первые три команды используют регистровую адресацию для первого операнда (пункт назначения) и непосредственную адресацию для второго операнда (константа, обозначенная символом #). Вторая команда помещает в R2 не содержимое A, а *адрес* A. Именно это и сообщает ассемблеру знак #. Сходным образом третья команда помещает в R3 первое слово после массива.

Интересно отметить, что само тело цикла не содержит каких-либо адресов памяти. В четвертой команде используется регистровая и косвенная адресация. В пятой команде применяется регистровая и непосредственная адресация, а в шестой — два раза регистровая. Команда BLT могла бы использовать адрес памяти, однако более привлекательным является определение адреса с помощью 8-битного смещения, связанного с самой командой BLT. Таким образом, полностью избегая адресов памяти, мы получили короткий и быстрый цикл. Кстати, эта программа предназначена для Pentium II, только мы переименовали команды и регистры и для упрощения понимания изменили запись.

Теоретически есть еще один способ выполнения этого вычисления без использования косвенной регистровой адресации. Этот цикл мог бы содержать команду для прибавления A к регистру R1, например

```
ADDR1.A
```

Тогда при каждом шаге команда должна увеличиваться на 4. Таким образом, после одного шага команда будет выглядеть следующим образом;

```
ADD R1.A+4
```

и так далее до завершения цикла.

Программа, которая сама изменяется подобным образом, называется **самоизменяющейся программой.** Эта идея была предложена Джоном фон Нейманом и применялась в старых компьютерах, где не было косвенной регистровой адресации. В настоящее время самоизменяющиеся программы считаются неудобными и очень трудными для понимания. Кроме того, их выполнение нельзя разделить между несколькими процессорами. Они даже не могут правильно выполняться на машинах с разделенной кэш-памятью первого уровня, если в кэш-памяти команд нет специальной схемы для обратной записи (поскольку разработчики предполагали, что программы сами себя не изменяют).

Индексная адресация

Часто нужно уметь обращаться к словам памяти по известному смещению. Подобные примеры мы видели в машине IJVM, где локальные переменные определяются по смещению от регистра LV. Обращение к памяти по регистру и константе смещения называется **индексной адресацией.**

В машине IJVM при доступе к локальной переменной используется указатель ячейки памяти (LV) в регистре плюс небольшое смещение в самой команде, как показано на рис. 4.14, *а*. Есть и другой способ: указатель ячейки памяти в команде и небольшое смещение в регистре. Чтобы показать, как это работает, рассмотрим следующий пример. У нас есть два одномерных массива A и B по 1024 слова в каждом. Нам нужно вычислить A, И Bi для всех пар, а затем соединить все эти 1024 логических произведения операцией ИЛИ, чтобы узнать, есть ли в этом наборе хотя бы одна пара, не равная нулю. Один из вариантов — поместить адрес массива A в один регистр, а адрес массива B — в другой регистр, а затем последовательно перебирать элементы массивов, аналогично тому, как мы делали в предыдущей программе (см. листинг 5.1). Такая программа, конечно же, будет работать, но ее можно усовершенствовать, как показано в листинге 5.2.

Листинг 5.2. Программа на языке ассемблера для вычисления операции ИЛИ от (Ai И Bi) для массива из 1024 элементов

```
        MOV RI,#0       ;собирает результаты выполнения ИЛИ в R1.
        MOV R2.#0       :R2= л от текущего произведения A[i] И B[i]
        MOV R3.#4096    ;R3=первое ненужное значение индекса

LOOP:   MOV R4.A(R2)    ;R4-A[i]
        AND R4,B(R2)    ;R4=A[I] И B[i]
        OR R1.R4
        ADD R2.#4       И-1+4
        CMP R2.R3       ;нужно ли продолжать?
        BLT LOOP        ;если R2<R3, мы не закончили и нужно продолжать
```

Здесь нам требуется 4 регистра:

1. R1 — содержит результаты суммирования логических произведений.
2. R2 — индекс i, который используется для перебора элементов массива.
3. R3 — константа 4096. Это самое маленькое значение i, которое не используется.
4. R4 — временный регистр для хранения каждого произведения.

После инициализации регистров мы входим в цикл из шести команд. Команда напротив LOOP вызывает элемент Ai в регистр R4. При вычислении источника здесь используется индексная адресация. Регистр (R2) и константа (адрес элемента A) складываются, и полученный результат используется для обращения к памяти. Сумма этих двух величин поступает в память, но не сохраняется ни в одном из видимых пользователем регистров. Запись

```
MOV R4.ACR2)
```

означает, что для определения пункта назначения используется регистровая адресация, где R4 — это регистр, а для определения источника используется индексная адресация, где A — это смещение, а R2 — это регистр. Если A имеет значение, скажем, 124300, то соответствующая машинная команда будет выглядеть так, как показано на рис. 5.13.

Рис. 5.13. Возможное представление команды MOV R4, A{R2)

Во время первого прохождения цикла регистр R2 принимает значение 0 (поскольку регистр инициализируется таким образом), поэтому нужное нам слово A0 находится в ячейке с адресом 124300. Это слово загружается в регистр R4. При следующем прохождении цикла R2 принимает значение 4, поэтому нужное нам слово A1 находится в ячейке с адресом 124304 и т. д.

Как мы говорили раньше, здесь смещение — это указатель ячейки памяти, а значение регистра — это небольшое целое число, которое во время вычисления меняется. Такая форма требует, чтобы поле смещения в команде было достаточно большим для хранения адреса, поэтому такой способ не очень эффективен. Тем не менее этот способ часто оказывается самым лучшим.

Относительная индексная адресация

В некоторых машинах применяется способ адресации, при котором адрес вычисляется путем суммирования значений двух регистров и смещения (смещение факультативно). Такой подход называется **относительной индексной адресацией.** Один из регистров — это база, а другой — это индекс. Такая адресация очень удобна при следующей ситуации. Вне цикла мы могли бы поместить адрес элемента A в регистр R5, а адрес элемента B в регистр R6. Тогда мы могли бы заменить две первые команды цикла LOOP на

```
LOOP:   MOV R4,(R2+R5)
AND R4,(R2+R6)
```

Было бы идеально, если бы существовал способ адресации по сумме двух регистров без смещения. С другой стороны, даже команда с 8-битным смещением была бы большим достижением, поскольку мы оба смещения могли бы установить на 0. Однако если смещения всегда составляют 32 бита, тогда мы ничего не выиграем, используя такую адресацию. На практике машины с такой адресацией обычно имеют форму с 8-битным и 16-битным смещением.

Стековая адресация

Мы уже говорили, что очень желательно сделать машинные команды как можно короче. Конечный предел в сокращении длины адреса — это команды без адресов. Как мы видели в главе 4, безадресные команды, например IADD, возможны при наличии стека. В этом разделе мы рассмотрим стековую адресацию более подробно.

Обратная польская запись

В математике существует древняя традиция помещать оператор между операндами (x+y), а не после операндов (xy+). Форма с оператором между операндами называется **инфиксной записью.** Форма с оператором после операндов называется **постфиксной** или **обратной польской записью** в честь польского логика Я. Лукасевича (1958), который изучал свойства этой записи.

Обратная польская запись имеет ряд преимуществ над инфиксной записью для выражения алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она удобна для вычисления формул в машинах со

стеками. В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что axb+c значит (axb)+c, а не ax(Ь+c), поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево над логической операцией И? Кто знает? Обратная польская запись устраняет такие недоразумения.

Существует несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. Ниже изложена переделка идеи Э. Дейкстры. Предположим, что формула состоит из следующих символов: переменных, двухоперандных операторов +, -, *, /, а также левой и правой скобок. Чтобы отметить конец формулы, мы будем вставлять символ -L после последнего символа одной формулы и перед первым символом следующей формулы.

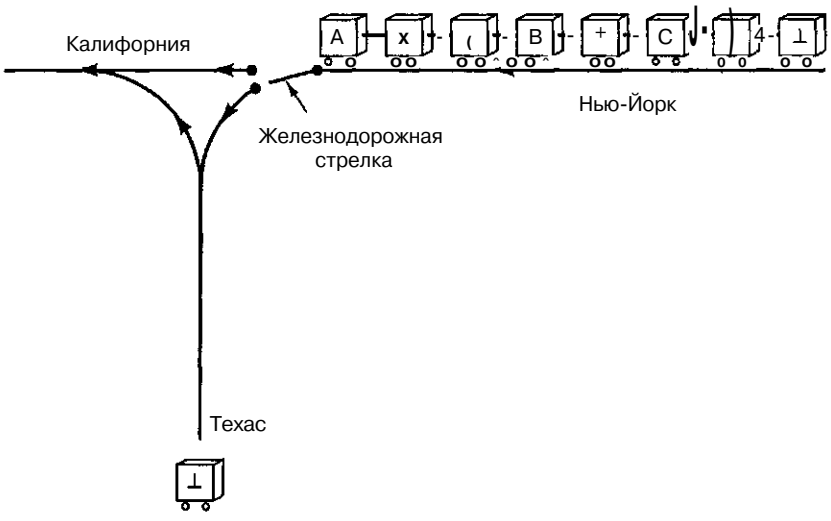


Рис. 5.14. Каждый вагон представляет собой один символ в формуле, которую нужно переделать из инфиксной формы в обратную польскую запись

На рис. 5.14 нарисована железная дорога из Нью-Йорка в Калифорнию с развилкой, ведущей в Техас. Каждый символ формулы представлен одним вагоном. Поезд движется на запад (налево). Перед развилкой каждый вагон должен останавливаться и узнавать, должен ли он двигаться прямо в Калифорнию, или ему нужно по пути заехать в Техас. Вагоны, содержащие переменные, всегда направляются Калифорнию и никогда не едут в Техас. Вагоны, содержащие все прочие символы, должны перед вхождением на развилку узнавать о содержимом ближайшего вагона, отправившегося в Техас.

В таблице на рис. 5.15 показана зависимость ситуации от того, какой вагон отправился последним в Техас и какой вагон находится у развилки. Первый 1 всегда отправляется в Техас. Числа соответствуют следующим ситуациям:

1. Вагон на развилке направляется в Техас.
2. Последний вагон, направившийся в Техас, разворачивается и направляется в Калифорнию.
3. Вагон, находящийся на развилке, и последний вагон, отправившийся в Техас, угоняются и исчезают (то есть оба удаляются).

4. Остановка. Символы, находящиеся в Калифорнии, представляют собой формулу в обратной польской записи, если читать слева направо.
5. Остановка. Произошла ошибка. Изначальная формула была некорректно сбалансирована.

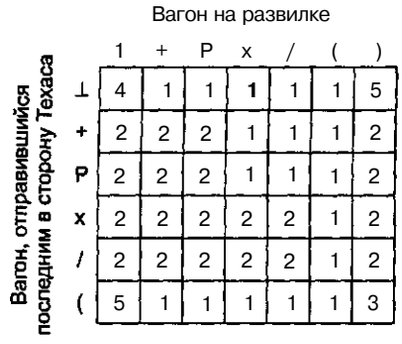


Рис. 5.15. Алгоритм преобразования инфиксной записи в обратную польскую запись

После каждого действия производится новое сравнение вагона, находящегося у развилки (это может быть тот же вагон, что и в предыдущем сравнении, а может быть следующий вагон), и вагона, который на данный момент последним ушел на Техас. Этот процесс продолжается до тех пор, пока не будет достигнут шаг 4. Отметим, что линия на Техас используется как стек, где отправка вагона в Техас — это помещение элемента в стек, а разворот вагона, отправленного в Техас, в сторону Калифорнии — это выталкивание элемента из стека.

Таблица 5.5. Некоторые примеры инфиксных выражений и их эквиваленты в обратной польской записи

Инфиксная запись	Обратная польская запись
A+BxC	ABCx+
AxB+C	ABxC+
AxB+CxD	A Bx C Dx+
(A+B)/(C-D)	A B+C D-/
AxB/C	ABxC/
((A+B) xC+D)/(E+F+G)	A B+CxD+ E F + G +/

Порядок переменных в инфиксной и обратной польской записи одинаков. Однако порядок операторов не всегда один и тот же. В обратной польской записи операторы появляются в том порядке, в котором они будут выполняться. В табл. 5.5 даны примеры инфиксных формул и их эквивалентов в обратной польской записи.

Вычисление формул в обратной польской записи

Обратная польская запись — идеальная запись для вычисления формул на компьютере со стеком. Формула состоит из n символов, каждый из которых является или операндом, или оператором. Алгоритм для вычисления формулы в обратной

польской записи с использованием стека прост. Нужно просто прочитать обратную польскую запись слева направо. Если встречается операнд, его нужно поместить в стек. Если встречается оператор, нужно выполнить соответствующую команду.

В таблице 5.6 показано вычисление выражения

(8+2x5)/(1+3x2-4)

в машине JVM. Соответствующая формула в обратной польской записи выглядит следующим образом:

825x+132x+4-/

В таблице мы ввели команды умножения и деления IMUL и IDIV. Число на вершине стека — это правый операнд (а не левый). Это очень важно для операций деления и вычитания, поскольку порядок операндов в данном случае имеет значение (в отличие от операций сложения и умножения). Другими словами, команда I0IV определяется следующим образом: сначала в стек помещается числитель, потом знаменатель, и тогда выполнение операции дает правильный результат. Отметим, что преобразовать обратную польскую запись в код (I)JVM очень легко: нужно просто просканировать формулу в обратной польской записи и выдавать одну команду с каждым символом. Если символ является константой или переменной, нужно выдавать команду помещения этой константы или переменной в стек. Если символ является оператором, нужно выдавать команду для выполнения данной операции.

Способы адресации для команд перехода

До сих пор мы рассматривали только те команды, которые оперируют с данными. Командам перехода (а также командам вызова процедур) также нужны особые способы адресации для определения целевого адреса. Способы, о которых мы говорили в предыдущих разделах, работают и для большинства команд перехода. Один из возможных вариантов — прямая адресация, когда целевой адрес просто полностью включается в команду.

Другие способы адресации тоже имеют смысл. Косвенная регистровая адресация позволяет программе вычислять целевой адрес, помещать его в регистр, а затем переходить туда. Такой способ дает максимальную гибкость, поскольку целевой адрес вычисляется во время выполнения программы. Но он также предоставляет огромные возможности для появления ошибок, которые практически невозможно найти.

Индексная адресация, при которой известно смещение от регистра, также является вполне разумным способом. Этот способ обладает теми же свойствами, что и косвенная регистровая адресация.

Еще один вариант — относительная адресация по счетчику команд. В данном случае для получения целевого адреса смещение (со знаком), находящееся в самой команде, прибавляется к программному счетчику. По сути, это индексная адресация, где в качестве регистра используется PC.

Таблица 5.6. Использование стека для вычисления формулы в обратной польской записи

Шаг	Оставшаяся цепочка	Команда	Стек
1	825x+132x+4-/	**BIPUSH** 8	**8**
2	25x+132x+4-/	**BIPUSH 2**	**8,2**
3	5x+132x+4-/	**BIPUSH 5**	**8,2,5**
4	x+132x+4-/	**IMUL**	**8,10**
5	+132X+4-/	**IADD**	**18**
6	132x+4-/	**BIPUSH 1**	**18,1**
7	32x+4-/	**BIPUSH 3**	**18,1,3**
8	2x+4-/	**BIPUSH 2**	**18,1,3,2**
9	x+**4-/**	**IMUL**	**18,1,6**
10	+**4-/**	**IADD**	**18,7**
11	**4-**/	**BIPUSH 4**	**18,7,4**
12	-/	**ISUB**	**18,3**
13	/	**IDIV**	**6**

Ортогональность кодов операций и способов адресации

С точки зрения программного обеспечения, команды и способы адресации должны иметь регулярную структуру, число форматов команд должно быть минимальным. При такой структуре компилятору гораздо проще порождать нужный код. Все коды операций должны допускать все способы адресации, где это имеет смысл. Более того, все регистры должны быть доступны для всех типов регистров (включая указатель фрейма (FP), указатель стека (SP) и программный счетчик (PC)).

Рассмотрим 32-битные форматы команд для трехадресной машины (рис. 5.16). Здесь поддерживаются до 256 кодов операций. В формате 1 каждая команда имеет два входных регистра и один выходной регистр. Этот формат используется для всех арифметических и логических команд.

Неиспользованное 6-битное поле в конце формата может использоваться для дальнейшей дифференциации команд. Например, можно иметь один код для всех операций с плавающей точкой, а различаться эти операции будут по дополнительному полю. Кроме того, если установлен бит 23, тогда используется формат 2, а второй операнд уже не является регистром, а является 13-битной непосредственной константой со знаком. Команды LOAD и STORE тоже могут использовать этот формат для обращения к памяти при индексном способе адресации.

Необходимо также иметь небольшое число дополнительных команд (например, команды условных переходов), но они легко подходят под формат 3. Например, можно приписать один код операции каждому (условному) переходу, вызову процедуры и т. д., тогда останется 24 бита для смещения по счетчику команд. Если предположить, что это смещение считается в словах, период будет составлять ± 32 Мбайт. Несколько кодов операций можно зарезервировать для команд LOAD

и STORE, которым нужны длинные смещения из формата 3. Они не будут общими (например, только регистр R0 будет загружаться и сохраняться), и использоваться будут довольно редко.

Рис. 5.16. Разработка форматов команд для трехадресной машины

Теперь рассмотрим разработку для двухадресной машины, в которой в качестве любого операнда может использоваться слово из памяти (рис. 5.17). Такая машина может прибавлять слово из памяти к регистру, прибавлять регистр к слову из памяти, складывать два регистра или складывать два слова из памяти. В настоящее время осуществлять доступ к памяти довольно дорого, поэтому данный проект не очень популярен, но если с развитием технологий доступ к памяти в будущем станет дешевле, такой подход будет считаться простым и эффективным. Машины PDP-11 и VAX были очень популярны и доминировали на рынке мини-компьютеров в течение двух десятилетий. В этих машинах использовались форматы, сходные с тем, который изображен на рис. 5.17.

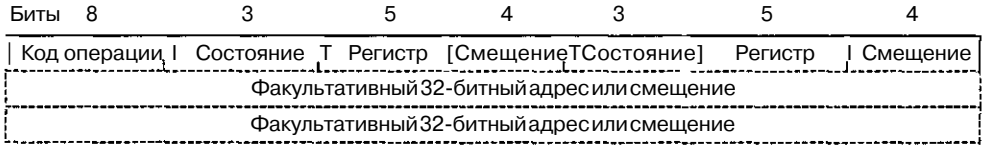


Рис. 5.17. Разработка форматов команд для двухадресной машины

Здесь мы снова имеем 8-битный код операции, но теперь у нас есть 12 битов для определения источника и 12 битов для определения пункта назначения. Для каждого операнда 3 бита дают метод адресации, 5 битов дают регистр и 4 бита дают смещение. Имея 3 бита для установления метода адресации, мы можем поддерживать непосредственную, прямую, регистровую, косвенную регистровую индексную и стековую адресации, и при этом еще остается место для двух дополнительных методов, которые, возможно, появятся в будущем. Это простая разработка, которую легко компилировать; она достаточно гибкая, особенно если счетчик программ, указатель стека и указатель локальных переменных находятся среди регистров общего назначения, к которым можно получить доступ.

Единственная проблема, которая здесь есть, — это то, что при прямой адресации нам нужно большее количество битов для адреса. В машинах PDP-11 и VAX к

команде было добавлено дополнительное слово для адреса каждого прямо адресуемого операнда. Мы тоже могли бы использовать один из двух доступных способов адресации для индексной адресации с 32-битным смещением, которое следует за командой. Тогда в худшем случае при прибавлении слова из памяти к слову из памяти, когда обращение к обоим операндам производится с помощью прямой адресации, или при использовании длинной индексной формы команда была бы 96 битов в длину и занимала бы 3 цикла шины (один — для команды и два — для данных). С другой стороны, большинству разработок типа RISC потребовалось бы по крайней мере 96 битов, а может и больше, для прибавления произвольного слова из памяти к другому произвольному слову из памяти, и тогда нужно было бы по крайней мере 4 цикла шины.

Помимо форматов, изображенных на рис. 5.17, возможны и другие варианты. В данной разработке можно выполнять операцию

с помощью одной 32-битной команды, при условии что i HJ находятся среди первых 16 локальных переменных. Для переменных после 16 нам приходится переходить к 32-битным смещениям. Можно сделать другой формат с одним 8-битным смещением вместо двух 4-битных и правилом, что это смещение может использоваться либо источником, либо пунктом назначения, но не тем и другим одновременно. Варианты компромиссов не ограничены, и разработчики должны учитывать многие факторы, чтобы получить хороший результат.

Способы адресации процессора Pentium II

Способы адресации процессора Pentium II чрезвычайно нерегулярны и зависят от того, в каком формате находится конкретная команда — 16-битном или 32-битном. Мы не будем рассматривать 16-битные команды. Вполне достаточно 32-битных. Поддерживаемые способы адресации включают непосредственную, прямую, регистровую, косвенную регистровую индексную и специальную адресацию для обращения к элементам массива. Проблема заключается в том, что не все способы применимы ко всем командам и не все регистры могут использоваться при всех способах адресации. Это сильно усложняет работу составителя компилятора.

Байт MODE на рис. 5.9 управляет способами адресации. Один из операндов определяется по комбинации полей MOD и R/M. Второй операнд всегда является регистром и определяется по значению поля REG. В таблице 5.7 приведен список 32 комбинаций значений 2-битного поля **MOD** и 3-битного поля R/M. Например, если оба поля равны 0, операнд считывается из ячейки памяти с адресом, который содержится в регистре EAX.

Колонки 01 и 10 включают способы адресации, при которых значение регистра прибавляется к 8-битному или 32-битному смещению, которое следует за командой. Если выбрано 8-битное смещение, оно перед сложением получает 32-битное зьэковое расширение. Например, команда ADD с полем R/M=011, полем MOD=01 и смещением, равным 6, вычисляет сумму регистра EBX и 6, и в качестве одного из операндов считывает слово из полученного адреса памяти. Значение регистра EBX не изменяется.

Таблица 5.7. 32-битные способы адресации процессора Pentium II. M[x] — это слово в памяти с адресом x

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX+СМЕЩЕНИЕ 8]	M[EAX+СМЕЩЕНИЕ 32]	EAX или AL
001	M[ECX]	M[ECX+СМЕЩЕНИЕ 8]	M[ECX+СМЕЩЕНИЕ 32]	ECX или CL
010	M[EDX]	MtEDX+СМЕЩЕНИЕ 8]	M[EOX+СМЕЩЕНИЕ 32]	EDX или DL
011	M[EBX]	M[EBX+СМЕЩЕНИЕ8]	M[EBX+СМЕЩЕНИЕ32]	EBX или BL
100	SIB	SIB и СМЕЩЕНИЕ 8	SIB и СМЕЩЕНИЕ 32	ESP или AH
101	Прямая адресация	M[EBP+СМЕЩЕНИЕ8]	M[EBP+СМЕЩЕНИЕ32]	EBP или CH
110	M[ESI]	M[ESI+CMEU4EHME8]	M[ЕЭ1+СМЕЩЕНИЕ32]	ESI или DH
111	M[EDI]	M[EDI+CMEU4EHHE 8]	MtEDI+СМЕЩЕНИЕ32]	EDI или BH

При MOD=11 предоставляется выбор из двух регистров. Для команд со словами берется первый вариант, для команд с байтами — второй. Отметим, что здесь не все регулярно. Например, нельзя осуществить косвенную адресацию через EBP или прибавить смещение к ESP.

Иногда вслед за байтом MODE следует дополнительный байт **SIB (Scale, Index, Base — масштаб, индекс, база)** (см. рис. 5.9). Байт SIB определяет масштабный коэффициент и два регистра. Когда присутствует байт SIB, адрес операнда вычисляется путем умножения индексного регистра на 1, 2, 4 или 8 (в зависимости от SCALE), прибавлением его к базовому регистру и, наконец, возможным прибавлением 8- или 32-битного смещения, в зависимости от значения поля MOD. Практически все регистры могут использоваться и в качестве индекса, и в качестве базы.

Форматы SIB могут пригодиться для обращения к элементам массива. Рассмотрим следующее выражение на языке Java:

```
for (i=0; i<n; i++) a[i]=0;
```

где a — это массив 4-байтных целых чисел, относящийся к текущей процедуре. Обычно регистр EBP используется для указания на базу стекового фрейма, который содержит локальные переменные и массивы, как показано на рис. 5.18. Компилятор должен хранить i в регистре EAX. Для доступа к элементу a[i] он будет использовать формат SIB, в котором адрес операнда равен сумме 4xEAX, EBP и 8. Эта команда может сохраняться в a[i] за одну команду.

А стоит ли применять такой способ адресации? На этот вопрос трудно ответить. Без сомнения, эта команда при надлежащем использовании сохраняет несколько циклов. Насколько часто она используется, зависит от компилятора и от приложения. Проблема здесь в том, что эта команда занимает определенное количество пространства микросхемы, которое можно было бы использовать для других целей, если бы этой команды не было. Например, можно было бы сделать больше кэш-память первого уровня.

Мы представили несколько возможных компромиссов, с которыми постоянно сталкиваются разработчики. Обычно перед тем как воплотить какую-либо идею в кремнии, производятся обширные моделирующие прогоны, но для этого нужно

иметь представление о том, какова рабочая нагрузка. Можно быть уверенным, что разработчики машины 8088 не включили web-браузер в набор тестов. Решения, принятые 20 лет назад, могут оказаться абсолютно неудачными с точки зрения современных приложений. Однако если какая-либо особенность была включена в машину, избавиться от нее уже невозможно по причине требования совместимости.

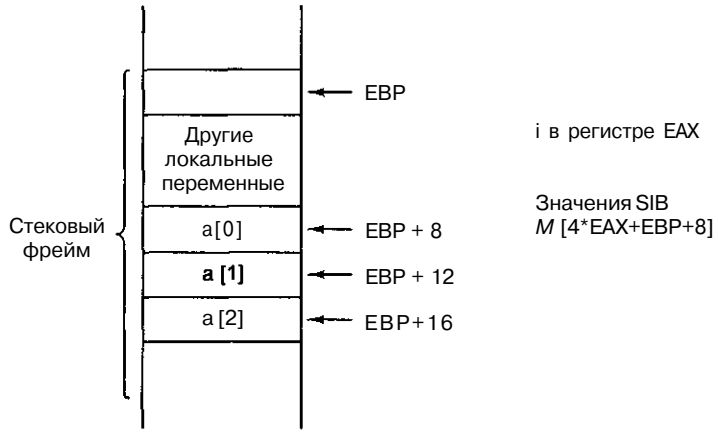


Рис. 5.18. Обращение к элементу массива a[i]

Способы адресации процессора UltraSPARC II

В архитектуре команд процессора UltraSPARC все команды используют непосредственную или регистровую адресацию, за исключением тех команд, которые обращаются к памяти. При регистровом способе адресации 5 битов просто сообщают, какой регистр нужно использовать. При непосредственной адресации данные обеспечивает 13-битная константа со знаком. Для арифметических, логических и подобных команд никакие другие способы адресации не используются.

К памяти обращаются команды трех типов: команды загрузки (LOAD), команды сохранения (STORE) и одна команда синхронизации мультипроцессора. Для команд LOAD и STORE есть два способа обращения к памяти. Первый способ' вычисляется сумма двух регистров, а затем через полученное значение производится косвенная адресация. Второй способ представляет собой обычное индексирование с 13-битным смещением со знаком.

Способы адресации машины JVM

У машины JVM нет общих способов адресации в том смысле, что каждая команда содержит несколько битов, которые сообщают, как нужно вычислить адрес (как в Pentium II, например). Вместо этого здесь с каждой командой связан один особый способ адресации. Поскольку в JVM нет видимых регистров, регистровая и косвенная регистровая адресация здесь невозможна. Несколько команд, например BIPUSH, используют непосредственную адресацию. Единственный оставшийся

доступный способ — индексная адресация. Она используется командами ILOAD, ISTORE, LDCW, а также несколькими командами, которые определяют переменную, связанную с каким-нибудь неявным регистром, обычно LV или CPP. Команды перехода тоже используют индексную адресацию, при этом PC рассматривается как регистр.

Сравнение способов адресации

Мы только что рассмотрели несколько способов адресации. Способы адресации машин Pentium II, UltraSPARC II и JVM изложены в табл. 5.8. Как мы уже говорили, не каждый способ может использоваться любой командой.

Таблица 5.8. Сравнение способов адресации

Способадресации	Pentium II	UltraSPARC II	**JVM**
Непосредственная	x	x	×
Прямая	x		
Регистровая	x	x	
Косвенная регистровая	x		
Индексная	x	x	×
Относительная индексная		x	
Стековая			×

На практике для эффективной архитектуры команд вовсе не требуется большого количества различных способов адресации. Поскольку практически весь код, написанный на этом уровне, будет порождаться компиляторами, способов адресации должно быть мало, и они должны быть четкими и ясными. Машина должна предлагать либо все возможные варианты, либо только один вариант.

В остальных промежуточных случаях может оказаться так, что компилятор не способен сделать выбор.

Поэтому самые простые архитектуры используют очень небольшое число способов адресации, причем на каждый из этих способов накладываются жесткие ограничения. Обычно практически для любых применений достаточно непосредственной, прямой, регистровой и индексной адресации. Каждый регистр (включая указатель локальных переменных, указатель стека и счетчик программ) должен быть пригоден к употреблению всякий раз, когда этот регистр требуется. Более сложные способы адресации могут сократить число команд, но при этом придется ввести последовательности операций, которые трудно будет выполнять параллельно с другими последовательными операциями.

Мы рассмотрели возможные компромиссы между кодами операций и адресами и между различными способами адресации. Когда вы сталкиваетесь с новым компьютером, вы должны изучить все команды и способы адресации не только для того, чтобы знать, какие из них имеются в наличии, но и для того, чтобы понять, почему был сделан именно такой выбор и каковы были бы последствия при другом выборе.

Типы команд

Команды можно грубо поделить на несколько групп, которые повторяются от машины к машине, хотя и могут различаться в деталях. Кроме того, в каждом компьютере всегда имеется несколько необычных команд, которые добавлены в целях совместимости с предыдущими моделями или из-за того, что у разработчика возникла блестящая идея, или потому, что правительство заплатило производителю, чтобы тот включил эту команду в набор команд. Ниже мы попытаемся описать все наиболее распространенные категории. Отметим, что мы не претендуем на исчерпывающее изложение.

Команды перемещения данных

Копирование данных из одного места в другое — одна из самых распространенных операций. Под копированием мы понимаем создание нового объекта с точно таким же набором битов, как у исходного объекта. Такое понимание слова «перемещение» несколько отличается от его обычного значения. Если мы говорим, что какой-то человек переместился из Нью-Йорка в Калифорнию, это не значит, что в Калифорнии была создана идентичная копия этого человека, а оригинал остался в Нью-Йорке. Когда мы говорим, что содержимое ячейки памяти 2000 переместилось в какой-либо регистр, мы всегда подразумеваем, что там была создана идентичная копия и что оригинал все еще находится в ячейке 2000. Команды перемещения данных лучше было бы назвать командами дублирования данных, но термин «перемещение данных» уже устоялсч.

Есть две причины, по которым данные могут копироваться из одного места в другое. Одна из них фундаментальна: присваивание переменным значений. Операция присваивания

```
A=B
```

выполняется путем копирования значения, которое находится в ячейке памяти с адресом B, в ячейку A, поскольку программист приказал это сделать. Вторая причина копирования данных — предоставить возможность быстрого обращения к ним. Как мы уже видели, многие команды могут обращаться к переменным только в том случае, если они имеются в регистре. Поскольку существует два возможных источника элемента данных (память и регистр) и существует два возможных пункта назначения для элемента данных (память и регистр), следовательно, существует 4 различных способа копирования. В одних компьютерах содержится 4 команды для 4 случаев, в других — одна команда для всех 4 случаев. Некоторые компьютеры используют команду LOAD для перемещения из памяти в регистр, команду STORE — для перемещения из регистра в память, команду MOVE — для перемещения из одного регистра в другой регистр, но не имеют никакой команды для копирования из одной части памяти в другую.

Команды перемещения данных должны как-то указывать, какое именно количество данных нужно переместить. Существуют команды для перемещения разного количества данных — от одного бита до всей памяти. В машинах с фиксированной длиной слова обычно перемещается ровно одно слово. Любые перемещения другого количества данных (больше слова или меньше слова) должны выполнять-

ся программным обеспечением с использованием сдвигов и слияний. Некоторые архитектуры команд дают возможность копировать отрезки данных размером меньше слова (они обычно измеряются в байтах), а также сразу несколько слов. Копирование нескольких слов рискованно, особенно если максимальное количество слов достаточно большое, поскольку такая операция может занять много времени и, возможно, ее придется прерывать в середине. Некоторые машины с изменяемой длиной слов содержат команды, которые определяют только адреса источника и места назначения, но не количество данных. Перемещение продолжается до тех пор, пока не появится специальное поле конца данных.

Бинарные операции

Бинарные операции — это такие операции, которые берут два операнда и получают из них результат. Все архитектуры команд содержат команды для сложения и вычитания целых чисел. Команды умножения и деления целых чисел также имеются практически во всех случаях. Думаю, нет необходимости объяснять, почему компьютеры оснащены арифметическими командами.

Следующая группа бинарных операций содержит булевы команды. Существует 16 булевых функций от двух переменных, но есть очень немного машин, в которых имеются команды для всех 16. Обычно присутствуют И, ИЛИ и НЕ; иногда кроме них еще есть ИСКЛЮЧАЮЩЕЕ ИЛИ, НЕ-ИЛИ и НЕ-И.

Важным применением команды И является выделение битов из слов. Рассмотрим машину со словами длиной 32 бита, в которой на одно слово приходится четыре 8-битных символа. Предположим, что нужно отделить второй символ от остальных трех, чтобы его напечатать. Это значит, что нужно создать слово, которое содержит этот символ в правых 8 битах с нулями в левых 24 битах (так называемое **выравнивание по правому биту).**

Чтобы извлечь нужный нам символ, слово, содержащее этот символ, соединяется операцией И с константой, которая называется **маской.** В результате этой операции все ненужные биты меняются на нули:

10110111 10111100 11011011 10001011 А
00000000 11111111 00000000 00000000 В (маска)
00000000 10111100 00000000 00000000 А И В

Затем результат сдвигается на 16 битов вправо, чтобы нужный символ находился в правом конце слова.

Важным применением команды ИЛИ является помещение битов в слово. Эта операция обратна операции извлечения. Чтобы изменить правые 8 битов 32-битного слова, не повредив при этом остальные 24 бита, сначала нежелательные 8 битов надо заменить на нули, а затем новый символ соединить операцией ИЛИ с полученным результатом, как показано ниже:

10110111 10111100 11011011 10001011 А
11111111 11111111 11111111 00000000 В (маска)
10110111 10111100 ПОПОИ 00000000 АИ В
00000000 00000000 00000000 01010111 С
10110111 10111100 ПОПОИ 01010111 (А И В) ИЛИ С

Операция И убирает единицы, и в полученном результате никогда не бывает больше единиц, чем в любом из двух операндов. Операция ИЛИ вставляет единицы, и поэтому в полученном результате всегда по крайней мере столько же единиц, сколько в операнде с большим количеством единиц. Команда ИСКЛЮЧАЮЩЕЕ ИЛИ, в отличие от них, симметрична в отношении единиц и нулей. Такая симметрия иногда может быть полезной, например при порождении случайных чисел.

Большинство компьютеров сегодня поддерживают команды с плавающей точкой, которые в основном соответствуют арифметическим операциям с целыми числами. Большинство машин содержит по крайней мере 2 варианта таких чисел: более короткие для скорости и более длинные на тот случай, если требуется высокая точность вычислений. Существует множество возможных форматов для чисел с плавающей точкой, но сейчас практически везде применяется единый стандарт IEEE 754. Числа с плавающей точкой и этот стандарт обсуждаются в приложении Б.

Унарные операции

Унарные операции используют один операнд и производят один результат. Поскольку в данном случае нужно определять на один адрес меньше, чем в бинарных операциях, команды иногда бывают короче, хотя часто требуется определять другую информацию.

Команды для сдвига и циклического сдвига очень полезны. Они часто даются в нескольких вариантах. Сдвиги — это операции, при которых биты сдвигаются налево или направо, при этом биты, которые сдвигаются за пределы слова, утрачиваются. Циклические сдвиги — это сдвиги, при которых биты, вытесненные с одного конца, появляются на другом конце. Разница между обычным сдвигом и циклическим сдвигом показана ниже:

00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 сдвиг вправо на 2 бита
11000000 00000000 00000000 00011100 циклический сдвиг вправо на 2 бита

Обычные и циклические сдвиги влево и вправо очень важны. Если n-битное слово циклически сдвигается влево на k битов, результат будет такой же, как при циклическом сдвиге вправо на n-k битов.

Сдвиги вправо часто выполняются с расширением по знаку. Это значит, что позиции, освободившиеся на левом конце слова, заполняются изначальным знаковым битом (0 или 1), как будто знаковый бит перетащили направо. Кроме того, это значит, что отрицательное число останется отрицательным. Ниже показаны сдвиги на 2 бита вправо:

1111111 11111111 11111111 11110000A
0011111 11111111 11111111 11111100 A сдвинуто без знакового расширения
1111111 11111111 11111111 11111100 A сдвинуто со знаковым расширением

Операция сдвига используется при умножении и делении на 2. Если положительное целое число сдвигается влево на k битов, результатом будет изначальное число, умноженное на 2^k. Если положительное целое число сдвигается вправо на k битов, результатом будет изначальное число, деленное на 2^k.

Сдвиги могут использоваться для повышения скорости выполнения некоторых арифметических операций. Рассмотрим выражение 18xп, где п — положительное целое число. 18xn=16xn+2xn. 16xn можно получить путем сдвига копии п на 4 бита влево. 2xп можно получить, сдвинув п на 1 бит влево. Сумма этих двух чисел равна 18xп. Таким образом, это произведение можно вычислить путем одного перемещения, двух сдвигов и одного сложения, что обычно гораздо быстрее, чем сама операция умножения. Конечно, компилятор может применять такую схему, только если один из множителей является константой.

Сдвиг отрицательных чисел даже со знаковым расширением дает совершенно другие результаты. Рассмотрим, например, число -1 в обратном двоичном коде. При сдвиге влево на 1 бит получается число -3. При сдвиге влево еще на 1 бит получается число -7:

11111111 11111111 11111111 11111110 число-1 в обратном двоичном коде
11111111 11111111 11111111 11111100 число-1 сдвигается влево на 1 бит (-3)
11111111 11111111 11111111 11111000 число-1 сдвигается влево на 2 бита (-7)

Сдвиг влево отрицательных чисел в обратном двоичном коде не умножает число на 2. Однако сдвиг вправо производит деление корректно.

А теперь рассмотрим число -1 в дополнительном двоичном коде. При сдвиге вправо на 6 бит с расширением по знаку получается число -1, что неверно, поскольку целая часть от -1/64 равна 0:

11111111 11111111 11111111 11111111 число -1 в дополнительном двоичном коде
11111111 11111111 11111111 11111111 число-1, сдвинутое влево на 6 битов, равно -1

Как мы видим, сдвиг вправо вызывает ошибки. Однако при сдвиге влево число умножается на 2.

Операции циклического сдвига нужны для манипулирования последовательностями битов в словах. Если нужно проверить все биты в слове, при циклическом сдвиге слова последовательно по 1 биту каждый бит помещается в знаковый бит, где его можно легко проверить, а когда все биты проверены, можно восстановить изначальное значение слова. Операции циклического сдвига гораздо удобнее операций обычного сдвига, поскольку при этом не теряется информация: произвольная операция циклического сдвига может быть отменена другой операцией циклического сдвига.

В некоторых бинарных операциях очень часто используются совершенно определенные операнды, поэтому в архитектуры команд часто включаются унарные операции для их быстрого выполнения. Например, перемещение нуля в память или регистр чрезвычайно часто выполняется при начале вычислений. Перемещение нуля — это особый случай команды перемещения данных. Поэтому для повышения производительности часто вводится операция CLR с единственным адресом той ячейки, которую нужно очистить (то есть установить на 0).

Прибавление 1 к слову тоже часто используется при различных подсчетах. Унарная форма команды ADD — это операция INC, которая прибавляет 1. Другой пример — операция NEG. Отрицание X — это на самом деле бинарная операция вычитания 0-X, но поскольку операция отрицания очень часто применяется, в архитектуру

команд вводится команда NEG. Важно понимать разницу между арифметической операцией NEG и логической операцией NOT. Операция NEG производит **аддитивную инверсию** числа (такое число, сумма которого с изначальным числом дает 0). Операция NOT просто инвертирует все биты в слове. Эти операции очень похожи, а для системы, в которой используется представление в обратном двоичном коде, они идентичны. (В арифметике дополнительных кодов для выполнения команды NEG сначала инвертируются все биты, а затем к полученному результату прибавляется 1.)

Унарные и бинарные операции часто объединяются в группы по функциям, которые они выполняют, а вовсе не по числу операндов. В первую группу входят арифметические операции, в том числе операция отрицания. Во вторую группу входят логические операции и операции сдвига, поскольку эти две категории очень часто используются вместе для извлечения данных.

Сравнения и условные переходы

Практически все программы должны проверять свои данные и на основе результатов изменять последовательность команд, которые нужно выполнить. Рассмотрим функцию квадратного корня Ox. Если число x отрицательное, процедура сообщает об ошибке; если число положительное, процедура вычисляет квадратный корень. Функция *sqrt* должна проверять x, а затем совершать переход в зависимости от того, положительно число x или отрицательно.

Это можно сделать с помощью специальных команд условного перехода, которые проверяют какое-либо условие и совершают переход в определенный адрес памяти, если условие выполнено. Иногда определенный бит в команде указывает, нужно ли осуществлять переход в случае выполнения условия или в случае невыполнения условия соответственно. Часто целевой адрес является не абсолютным, а относительным (он связан с текущей командой).

Самое распространенное условие, которое нужно проверить, — равен ли определенный бит нулю или нет. Если команда проверяет знаковый бит числа и совершает переход к метке (LABEL) при условии, что бит равен 1, то если число было отрицательным, будут выполняться те утверждения, которые начинаются с метки LABEL, а если число было положительным или было равно 0, то будут выполняться те утверждения, которые следуют за условным переходом.

Во многих машинах содержатся биты кода условия, которые указывают на особые условия. Например, там может быть бит переполнения, который принимает значение 1 всякий раз, когда арифметическая операция выдает неправильный результат. Проверяя этот бит, мы проверяем выполнение предыдущей арифметической операции, и если произошла ошибка, то запускается программа обработки ошибок.

В некоторых процессорах есть специальный разряд (бит) переноса, который принимает значение 1, если происходит перенос из самого левого бита (например, при сложении двух отрицательных чисел). Бит переноса нельзя путать с битом переполнения. Проверка бита переноса необходима для вычислений с повышенной точностью (то есть когда целое число представлено двумя или более словами).

Проверка на ноль очень важна при выполнении циклов и в некоторых других случаях. Если бы все команды условного перехода проверяли только 1 бит, то тогда для проверки определенного слова на 0 нужно было бы отдельно проверять каждый бит, чтобы убедиться, что ни один бит не равен 1. Чтобы избежать подобной ситуации, во многие машины включается команда, которая должна проверять слово и осуществлять переход, если оно равно 0. Конечно же, это решение просто перекладывает ответственность на микроархитектуру. На практике аппаратное обеспечение обычно содержит регистр, все биты которого соединяются операцией ИЛИ, чтобы выдать на выходе один бит, по которому можно определить, содержит ли регистр биты, равные 1. Бит Z на рис. 4.1 обычно вычисляется следующим образом: сначала все выходные биты АЛУ соединяются операцией ИЛИ, а затем полученный результат инвертируется.

Операция сравнения слов или символов очень важна, например, при сортировке. Чтобы произвести сравнение, требуется три адреса: два нужны для элементов данных, а в третий адрес будет совершаться переход в случае выполнения условия. В тех компьютерах, где форматы команд позволяют содержать три адреса в команде, проблем не возникает. Но если такие форматы не предусмотрены, нужно что-то сделать, чтобы обойти эту проблему.

Одно из возможных решений — ввести команду, которая выполняет сравнение и записывает результат в один или несколько битов условия. Следующая команда может проверить биты условия и совершить переход, если два сравниваемых значения были равны, или неравны, или первое из них было больше второго и т. д. Такой подход применяется в Pentium II и UltraSPARC II.

В сравнении двух чисел есть некоторые тонкости. Сравнение — это не такая простая операция, как вычитание. Если очень большое положительно число сравнивается с очень большим отрицательным числом, операция вычитания приведет к переполнению, поскольку результат вычитания не может быть представлен. Тем не менее команда сравнения должна определить, удовлетворено ли условие, и возвратить правильный ответ. При сравнении не должно быть переполнений.

Кроме того, при сравнении чисел нужно решить, считаются ли числа числами со знаком или числами без знака. Трехбитные бинарные числа можно упорядочить двумя способами. От самого маленького к самому большому:

Без знака	Со знаком
000	100 (самое маленькое)
001	101
010	ПО
011	111
100	000
101	001
НО	010
111	011 (самое большое)

В колонке слева приведены положительные числа от 0 до 7 по возрастанию. В колонке справа показаны целые числа со знаком от -4 до +3 в дополнительном двоичном коде. Ответ на вопрос: «Какое число больше: 011 или 100?» зависит от того, считаются ли числа числами со знаком. В большинстве архитектур есть команды для обращения с обоими типами упорядочения.

Команды вызова процедур

Процедура — это группа команд, которая выполняет определенную задачу и которую можно вызвать из нескольких мест программы. Вместо термина процедура часто используется термин **подпрограмма,** особенно когда речь идет о программах на языке ассемблера. Когда процедура закончила задачу, она должна вернуться к соответствующему оператору. Следовательно, адрес возврата должен как-то передаваться процедуре или сохраняться где-либо таким образом, чтобы можно было определить местонахождение после завершения задачи.

Адрес возврата может помещаться в одном из трех мест: в памяти, в регистре или в стеке. Самое худшее решение — поместить этот адрес в одну фиксированную ячейку памяти. Тогда если процедура будет вызывать другую процедуру, второй вызов приведет к потере первого адреса возврата.

Более удачное решение — сохранить адрес возврата в первом слове процедуры. Тогда первой выполняемой командой будет второе слово процедуры. После завершения процедуры происходит переход к первому слову, а если аппаратное обеспечение в первом слове наряду с адресом возврата дает код операции, то происходит непосредственный переход к этой операции. Процедура может вызывать другие процедуры, поскольку в каждой процедуре имеется пространство для одного адреса возврата. Но если процедура вызывает сама себя, эта схема не работает, поскольку первый адрес возврата будет уничтожен вторым вызовом. Способность процедуры вызывать саму себя, называемая **рекурсией,** очень важна и для теоретиков, и для практиков. Более того, если процедура А вызывает процедуру В, процедура В вызывает процедуру С, а процедура С вызывает процедуру А (непосредственная или цепочечная рекурсия), эта схема сохранения адреса возврата также не работает.

Еще более удачное решение — помещать адрес возврата в регистр. Если процедура рекурсивна, ей придется помещать адрес возврата в другое место каждый раз, когда она вызывается.

Самое лучшее решение — поместить адрес возврата в стек. Когда процедура завершена, она выталкивает адрес возврата из стека. При такой форме вызова процедур рекурсия не порождает никаких проблем; адрес возврата будет автоматически сохраняться таким образом, чтобы избежать уничтожения предыдущего адреса возврата. Мы рассматривали такой способ сохранения адреса возврата в машине ПУМ(см.рис.4.10).

Управление циклом

Часто возникает необходимость выполнять некоторую группу команд фиксированное количество раз, поэтому некоторые машины содержат команды для облегчения этого процесса. Все эти схемы содержат счетчик, который увеличивается или уменьшается на какую-либо константу каждый раз при выполнении цикла. Кроме того, этот счетчик каждый раз проверяется. При выполнении определенного условия цикл завершается.

Определенная процедура запускает счетчик вне цикла и затем сразу начинает выполнение цикла. Последняя команда цикла обновляет счетчик, и если условие завершения цикла еще не выполнено, то происходит возврат к первой команде цикла. Если условие выполнено, цикл завершается и начинается выполнение ко-

манды, идущей сразу после цикла. Цикл такого типа с проверкой в начале представлен в листинге 5.3. (Мы не могли здесь использовать язык Java, поскольку в нем нет оператора goto.)

Цикл такого типа всегда будет выполняться хотя бы один раз, даже если n≤0. Рассмотрим программу, которая поддерживает данные о персонале компании. В определенном месте программа начинает считывать информацию о конкретном работнике. Она считывает число n — количество детей у работника, и выполняет цикл n раз, по одному разу на каждого ребенка. Она считывает его имя, пол и дату рождения, так что компания может послать ему или ей подарок. Если у работника нет детей, n будет равно 0, но цикл все равно будет выполнен один раз, что даст ошибочные результаты.

В листинге 5.4 представлен другой способ выполнения проверки, который дает правильные результаты даже при n≤0. Отметим, что если одна команда выполняет и увеличение счетчика, и проверку условия, разработчики вынуждены выбирать один из двух методов.

Листинг 5.3. Цикл с проверкой в конце

```
        i=1:
L1:     первый оператор:
        .
        .
        .
        последний оператор:
        1-1+1:
        if (i<n) goto LI:
```

Листинг 5.4. Цикл с проверкой в начале

```
        i=1:
LI:     if(i>n) goto L2:
        первый  оператор:
        .
        .
        .
        последний оператор:
        i=i+1:
        goto LI:
L2:
```

Рассмотрим программу, которую нужно произвести для следующего выражения:

```
for (i=0; i<n: 1++) {операторы}
```

Если у компилятора нет никакой информации о числе n, он должен применять подход, приведенный в листинге 5.4, чтобы корректно обработать случай n≤0. Однако если компилятор может определить, что n>0 (например, узнав, как определено n), он может использовать более удобный код, изложенный в листинге 5.3. Когда-то в стандарте языка FORTRAN требовалось, чтобы все циклы выполнялись один раз. Это позволяло всегда порождать более эффективный код (листинг 5.3). В 1977 году этот дефект был исправлен, поскольку даже приверженцы языка FORTRAN начали осознавать, что плохо иметь оператор цикла с такой странной семантикой, пусть он и позволяет экономить одну команду перехода на каждый цикл.

Команды ввода-вывода

Ни одна другая группа команд не различается настолько сильно в разных машинах, как команды ввода-вывода. В современных персональных компьютерах используются три различные схемы ввода-вывода:

1. Программируемый ввод-вывод с активным ожиданием.
2. Ввод-вывод с управлением по прерываниям.
3. Ввод-вывод с прямым доступом к памяти.

Мы рассмотрим каждую из этих схем по очереди.

Самым простым методом ввода-вывода является программируемый ввод-вывод, который часто используется в дешевых микропроцессорах, например во встроенных системах или в таких системах, которые должны быстро реагировать на внешние изменения (это системы, работающие в режиме реального времени). Эти процессоры обычно имеют одну входную и одну выходную команды. Каждая из этих команд выбирает одно из устройств ввода-вывода. Между фиксированным регистром в процессоре и выбранным устройством ввода-вывода передается один символ. Процессор должен выполнять определенную последовательность команд при каждом считывании и записи символа.

В качестве примера данного метода рассмотрим терминал с четырьмя 1-байтными регистрами, как показано на рис. 5.19. Два регистра используются для ввода: регистр состояния устройства и регистр данных. Два регистра используются для вывода: тоже регистр состояния устройства и регистр данных. Каждый из них имеет уникальный адрес. Если используется ввод-вывод с распределением памяти, все 4 регистра являются частью адресного пространства, и будут считываться и записываться с помощью обычных команд. В противном случае для чтения и записи регистров используются специальные команды ввода-вывода, например IN и OUT. В обоих случаях ввод-вывод осуществляется путем передачи данных и информации о состоянии устройства между центральным процессором и этими регистрами.

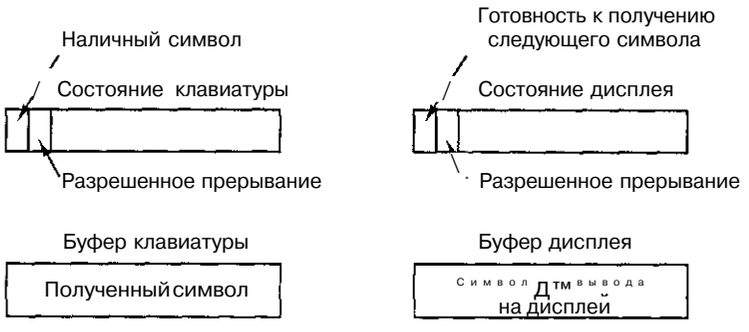


Рис. 5.19. Регистры устройств для простого терминала

Регистр состояния клавиатуры содержит 2 бита, которые используются, и 6 битов, которые не используются. Аппаратное обеспечение устанавливает самый левый бит на 1 всякий раз, когда появляется символ. Если программное обеспечение ранее установило на 1 бит 6, то производится прерывание. В противном случае прерывания не происходит. При программируемом вводе-выводе для получения входных данных центральный процессор обычно находится в цикле, периодически считывая регистр состояния клавиатуры, пока бит 7 не примет значение 1. Когда это случается, программное обеспечение считывает буферный регистр клавиатуры, чтобы получить символ. Считывание регистра данных вызывает установку бита CHARACTER AVAILABLE (наличия символа) на 0.

Вывод осуществляется сходным образом. Чтобы написать символ на экране, программное обеспечение сначала считывает регистр состояния дисплея, чтобы узнать, установлен ли бит READY (бит готовности) на 1. Если он не установлен,

программное обеспечение проходит цикл снова и снова до тех пор, пока данный бит не примет значение 1. Это значит, что устройство готово принять символ. Как только терминал приходит в состояние готовности, программное обеспечение записывает символ в буферный регистр дисплея, который переносит символ на экран и дает сигнал устройству установить бит готовности в регистре состояния дисплея на 0. Когда символ уже отображен, а терминал готов к обработке следующего символа, бит READY снова устанавливается на 1 контроллером.

В качестве примера программируемого ввода-вывода рассмотрим процедуру, написанную на Java (листинг 5.5). Эта процедура вызывается с двумя параметрами: массивом символов, который нужно вывести, и количеством символов, которые присутствуют в массиве (до 1 K). Тело процедуры представляет собой цикл, который выводит по одному символу. Сначала центральный процессор должен подождать, пока устройство будет готово, и только после этого он выводит символ, и эта последовательность действий повторяется для каждого символа. Процедуры *in* и *out* — это обычные процедуры языка ассемблера для чтения и записи регистров устройств, которые определяются по первому параметру, из или в переменную, которая определяется по второму параметру. Деление на 128 убирает младшие 7 битов, при этом бит READY остается в бите 0.

Листинг 5.5. Пример программируемого ввода-вывода

```
public static void output_buffer(int buf[]. int count) {
       //Вывод блока данных на устройство
       int status, i. ready;

       for (i=0; i<count; i++) {
          do {
          status=in(display_status_reg);  // получение информации
                                             о состоянии устройства
          ready=(status«7)&0x01:          // выделение бита
                                             готовности
    } while (ready==1);
    out(display_buffer_reg, buf[i]);
    }
    }
```

Основной недостаток программируемого ввода-вывода заключается в том, что центральный процессор проводит большую часть времени в цикле, ожидая готовности устройства. Такой процесс называется **активным ожиданием.** Если центральному процессору больше ничего не нужно делать (например, в стиральной машине), в этом нет ничего страшного (хотя даже простому контроллеру часто нужно контролировать несколько параллельных процессов). Но если процессору нужно выполнять еще какие-либо действия, например запускать другие программы, то активное ожидание здесь не подходит, и нужно искать другие методы ввода-вывода.

Чтобы избавиться от активного ожидания, нужно, чтобы центральный процессор запускал устройство ввода-вывода и указывал этому устройству, что необходимо осуществить запрос на прерывание, когда оно завершит свою работу. Посмотрите на рис. 5.19. Установив бит разрешения прерываний в регистре устройства, программное обеспечение сообщает, что аппаратное обеспечение будет передавать сигнал о завершении работы устройства ввода-вывода. Подробнее мы рассмотрим прерывания ниже в этой главе, когда перейдем к обсуждению передачи управления.

Во многих компьютерах сигнал прерывания порождается путем логического умножения (И) бита разрешения прерываний и бита готовности устройства. Если программное обеспечение сначала разрешает прерывание (перед запуском устройства ввода-вывода), прерывание произойдет сразу же, поскольку бит готовности будет равен 1. Таким образом, может понадобиться сначала запустить устройство, а затем сразу после этого ввести прерывание. Запись байта в регистр состояния устройства не изменяет бита готовности, который может только считываться.

Ввод-вывод с управлением по прерываниям — это большой шаг вперед по сравнению с программируемым вводом-выводом, но все же он далеко не совершенен. Дело в том, что прерывание требуется для каждого передаваемого символа. Следовательно, нужно каким-то образом избавиться от большинства прерываний.

Решение лежит в возвращении к программируемому вводу-выводу. Но только эту работу должен выполнять кто-то другой. Посмотрите на рис. 5.20. Мы добавили новую микросхему — контроллер **прямого доступа к памяти (ПДП)** с прямым доступом к шине.

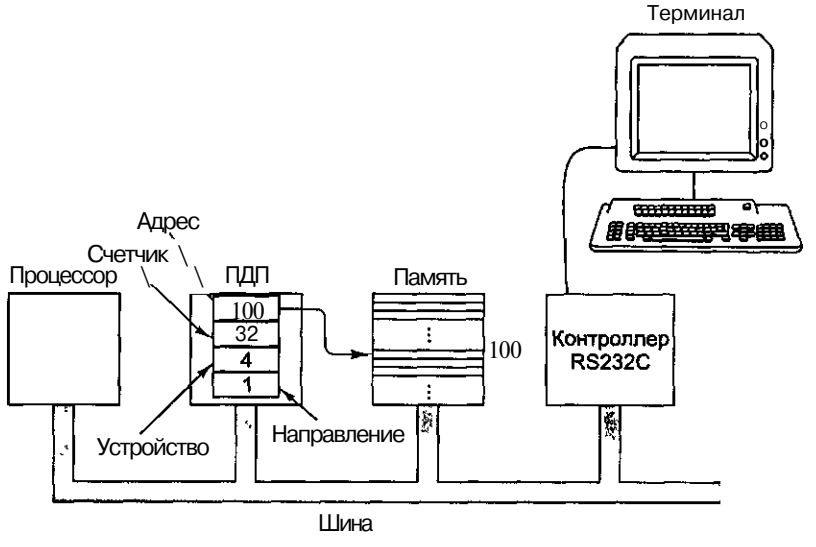


Рис. 5.20. Система с контроллером прямого доступа к памяти

Микросхема ПДП имеет по крайней мере 4 регистра. Все они могут загружаться программным обеспечением, работающим на центральном процессоре. Первый регистр содержит адрес памяти, который нужно считать или записать. Второй регистр содержит число, которое показывает количество передаваемых байтов или слов. Третий регистр содержит номер устройства или адрес устройства ввода-вывода, определяя, таким образом, какое именно *устройство нам требуется*. Четвертый регистр сообщает, должны ли данные считываться с устройства или записываться на него.

Чтобы записать блок из 32 байтов из адреса памяти 100 на терминал (например, устройство 4), центральный процессор записывает числа 32,100 и 4 в первые три регистра ПДП и код записи (например, 1) в четвертый регистр, как показано

на рис. 5.20. Контроллер ПДП, инициализированный таким способом, делает запрос на доступ к шине, чтобы считать байт 100 из памяти, точно так же как если бы центральный процессор считывал этот байт. Получив нужный байт, контроллер ПДП посылает устройству 4 запрос на ввод-вывод, чтобы записать на него байт. После завершения этих двух операций контроллер ПДП увеличивает значение регистра адреса на 1 и уменьшает значение регистра счетчика на 1. Если значение счетчика больше 0, то следующий байт считывается из памяти и записывается на устройство ввода-вывода.

Когда значение счетчика доходит до 0, контроллер ПДП останавливает передачу данных и устанавливает линию прерывания на микросхеме процессора. При наличии ПДП центральному процессору нужно только инициализировать несколько регистров. После этого центральный процессор может выполнять какую-либо другую работу до тех пор, пока передача данных не завершится. При завершении передачи данных центральный процессор получает сигнал прерывания от контроллера ПДП. Некоторые контроллеры ПДП содержат два, три и более наборов регистров, так что они могут управлять несколькими процессами передачи одновременно.

Отметим, что если какое-нибудь высокоскоростное устройство, например диск, будет запускаться контроллером ПДП, то потребуется очень много циклов шины и для обращений к памяти, и для обращений к устройству. Во время этих циклов центральному процессору придется ждать (ПДП всегда имеет приоритет над центральным процессором на доступ к шине, поскольку устройства ввода-вывода обычно не допускают задержек). Процесс отбирания контроллером ПДП циклов шины у центрального процессора называется **захватом цикла.** Но выигрыш в том, что не нужно обрабатывать одно прерывание при каждом передаваемом байте (слове), сильно перевешивает потери, происходящие из-за захвата циклов.

Команды процессора Pentium II

В этом и следующих двух разделах мы рассмотрим наборы команд трех машин: Pentium II, UltraSPARC II и picojava II. Каждая из них содержит базовые команды, которые обычно порождаются компиляторами, а также набор команд, которые редко используются или используются только операционной системой. Мы будем рассматривать обычные команды. Начнем с Pentium II.

Команды Pentium II представляют собой смесь команд 32-битного формата и команд, которые восходят к процессору 8088. На рисунке 5.21 приведены наиболее распространенные команды с целыми числами, которые широко используются в настоящее время. Этот список далеко не полный, поскольку в него не вошли команды с плавающей точкой, команды управления, а также некоторые редкие команды с целыми числами (например, использование 8-битного байта для выполнения поиска по таблице). Тем не менее этот список дает представление о том, какие действия может выполнять Pentium П.

Многие команды Pentium II обращаются к одному или к двум операндам, которые находятся или в регистрах, или в памяти. Например, бинарная команда ADD

складывает два операнда, а унарная команда INC увеличивает значение одного операнда на 1. Некоторые команды имеют несколько похожих вариантов. Например, команды сдвига могут сдвигать слово либо вправо, либо влево, а также могут рассматривать знаковый бит особо или нет. Большинство команд имеют несколько различных кодировок в зависимости от природы операндов.

На рисунке 5.21 поля SRC — это источники информации (SOURCE). Они не изменяются. Поля DST — это пункты назначения (DESTINATION). Они обычно изменяются командой. Существуют правила, определяющие, что может быть источником, а что пунктом назначения, но здесь мы не будем о них говорить. Многие имеют три варианта: для 8-, 16- и 32-битных операндов соответственно. Они различаются по коду операции И/ИЛИ по одному биту в команде. На рис. 5.21 приведены в основном 32-битные команды.

Для удобства мы разделили команды на несколько групп. Первая группа содержит команды, которые перемещают данные между частями машины: регистрами, памятью и стеком. Вторая группа содержит арифметические операции со знаком и без знака. Для умножения и деления 64-битное произведение или делимое хранится в двух регистрах: EAX (младшие биты) и EDX (старшие биты).

Третья группа включает двоично-десятичную арифметику. Здесь каждый байт рассматривается как два 4-битных полубайта. Каждый полубайт содержит 1 десятичный разряд (от 0 до 9). Комбинации битов от 1010 до 1111 не используются. Таким образом, 16-битное целое число может содержать десятичное число от 0 до 9999. Хотя такая форма хранения неэффективна, она устраняет необходимость переделывать десятичные входные данные в двоичные, а затем обратно в десятичные для вывода. Эти команды используются для выполнения арифметических действий над двоично-десятичными числами. Они широко используются в программах на языке COBOL.

Логические команды и команды сдвига манипулируют битами в слове или байте. Существует несколько комбинаций.

Следующие две группы связаны с проверкой и сравнением и осуществлением перехода в зависимости от полученного результата. Результаты проверки и сравнения хранятся в различных битах регистра EFLAGS. Значок Jxx стоит вместо набора команд, которые совершают условный переход в зависимости от результатов предыдущего сравнения (то есть в зависимости от битов в регистре EFLAGS).

В Pentium II есть несколько команд для загрузки, сохранения, перемещения, сравнения и сканирования цепочек символов или слов. Перед этими командами может стоять специальный префиксный байт REP (repetition — повторение), который заставляет команду повторяться до тех пор, пока не будет выполнено определенное условие (например, пока регистр ECX, значение которого уменьшается на 1 после каждого повторения, не будет равен 0). Таким образом, различные действия (перемещение, сравнение и т. д.) могут производиться над произвольными блоками данных.

Последняя группа содержит команды, которые не вошли ни в одну из предыдущих групп. Это команды перекодирования, команды управления, команды ввода-вывода и команды остановки процессора.

Команды перемещения

а	MOV DST, SRC	Перемещает SRC в DST
	PUSH SRC	Помещает SRC в стек
	POP DST	Выталкивает слово из стека и помещает его в DST
	XCHGDS1.DS2	Меняет местами DS1 и DS2
	LEA DST, SRC	Загружает действительный адрес SRC в DST
	CMOV DST, SRC	Условное перемещение

Арифметические команды

б	ADD DST, SRC	Складывает SRC и DST
	SUB DST, SRC	Вычитает SRC из DST
	MUL SRC	Умножает EAX на SRC (без знака)
	IMUL SRC	Умножает EAX на SRC (со знаком)
	DIV SRC	Делит EDX:EAX на SRC (без знака)
	IDV SRC	Делит EDX:EAX на SRC (со знаком)
	ADC DST, SRC	Складывает SRC с DST и прибавляет бит переноса
	SBB DST, SRC	Вычитает DST и переносит из SRC
	INC DST	Прибавляет 1 к DST
	DEC DST	Вычитает 1 из DST
	NEG DST	Отрицает DST (вычитает DST из 0)

Двоично-десятичные команды

в	DAA	Десятичная коррекция
	DAS	Десятичная коррекция для вычитания
	AAA	Коррекция кода ASCII для сложения
	AAS	Коррекция кода ASCII для вычитания
	AAM	Коррекция кода ASCII для умножения
	AAD	Коррекция кода ASCII для деления

Логические команды

г	**AND** DST, SRC	Логическая операция И над SRC и DST
	OR DST, SRC	Логическая операция ИЛИ над SRC и DST
	XOR DST, SRC	Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ над SRC и DST
	NOT DST	Замещение DST дополнением до 1

Команды сдвига/циклического сдвига

д	SAL/SAR DST, #	Сдвиг DST влево/вправо на # битов
	SHL/SHR DST, #	Логический сдвиг DST влево/вправо на # битов
	ROL/ROR DST, #	Циклический сдвиг DST влево/вправо на # битов
	ROL/ROR DST, #	Циклический сдвиг DST по переносу на # битов

Команды тестирования/сравнения

е	TSTSRC1.SRC2	Операнды логической операции И, установка флагов
	CMPSRC1.SRC2	Установка флагов на основе вычитания SRC1-SRC2

Рис. 5.21. Команды с целыми числами в Pentium И (начало)

Команды передачи управления

JMPADDR	Переход к адресу
Jxx ADDR	Условные переходы на основе флагов
CALL ADDR	Вызов процедуры по адресу
RET	Выход из процедуры
IRET	Выход из прерывания
LOOPxx	Продолжает цикл до удовлетворения определенного условия
INT ADDR	Инициирует программное прерывание
INTO	Совершает прерывание, если установлен бит переполнения

Команды для операций над цепочками

LODS	Загружает цепочку
STOS	Сохраняет цепочку
MOVS	Перемещает цепочку
CMPS	Сравнивает две цепочки
SCAS	Сканирование цепочки

Коды условия

STC	Устанавливает бит переноса в регистре EFLAGS
CLC	Сбрасывает бит переноса в регистре EFLAGS
CMC	Образует дополнение бита переноса в регистре EFLAGS
STD	Устанавливает бит направления в регистре EFLAGS
CLD	Сбрасывает бит направления в регистре EFLAGS
STI	Устанавливает бит прерывания в регистре EFLAGS
CLI	Сбрасывает бит прерывания в регистре EFLAGS
PUSHFD	Помещает регистр EFLAGS в стек
POPFD	Выталкивает содержимое регистра EFLAGS из стека
LAHF	Загружает AH из регистра EFLAGS
SAHF	Сохраняет AH в регистре EFLAGS

Прочие команды

SWAP DST	Изменяет порядок байтов DST
CWQ	Расширяет EAX до EDX:EAX для деления
SWDE	Расширяет 16-битное число в AX до EAX
ENTER SIZE, LV	Создает стековый фрейм с байтами размера
LEAVE	Удаляет стековый фрейм, созданный командой ENTER
NOP	Пустая операция
HLT	Останов
IN AL, PORT	Переносит байт из порта в АЛУ
OUT PORT, AL	Переносит байт из АЛУ в порт
WAIT	Ожидает прерывания

SRC = источник (source); # = на сколько битов происходит сдвиг;
DST = пункт назначения (destination); LV = # локальных переменных

Рис. 5.21. Команды с целыми числами в Pentium II (окончание)

Pentium II имеет ряд префиксов. Один из них (REP) мы уже упомянули. Префикс — это специальный байт, который может ставиться практически перед любой командой (подобно WIDE в IJVM). Префикс REP заставляет команду, идущую за ним, повторяться до тех пор, пока регистр ECX не примет значение 0, как было сказано выше. REPZ и REPNZ заставляют команду выполняться снова и снова, пока код выполнения условия Z не примет значение 1 или 0 соответственно. Префикс LOCK резервирует шину для всей команды, чтобы можно было осуществлять многопроцессорную синхронизацию. Другие префиксы используются для того, чтобы команда работала в 16-битном или 32-битном формате. При этом не только меняется длина операндов, но и полностью переопределяются способы адресации.

Команды UltraSPARC II

Все целочисленные команды пользовательского режима UltraSPARC II приведены на рис. 5.22. Здесь не даются команды с плавающей точкой, команды управления (например, команды управления кэш-памятью, команды перезагрузки системы), команды, включающие адресные пространства, отличные от пользовательских, или устаревшие команды. Набор команд удивительно мал: UltraSPARC II — это процессор типа RISC.

Структура команд LOAD и STORE очень проста. Эти команды имеют варианты для 1, 2,4 и 8 байтов. Если в 64-разрядный регистр загружается число размером меньше 64 битов, это число может быть либо расширено по знаку, либо дополнено нулями. Существуют команды для обоих вариантов.

Следующая группа команд предназначена для арифметических операций. Команды с буквами CC в названии устанавливают биты кода условия. На машинах CISC большинство команд устанавливают коды условия, но в машине типа RISC это нежелательно, поскольку ограничивает способность компилятора перемещать команды, стараясь заполнить отсрочки. Если изначальный порядок команд A... B... C, где A устанавливает коды условия, а B проверяет их, то компилятор не может вставить C между A и B, если C устанавливает условные коды. По этой причине многие команды имеют два варианта, при этом компилятор обычно использует ту команду, которая не устанавливает коды условия, если не планируется проверить их позже. Команды умножения, деления со знаком и деления без знака тоже поддерживаются.

Кроме этого, поддерживается специальный формат 30-битных чисел с автоматическим опознаванием типа данных за счет поля тега. Он используется для таких языков, как Smalltalk и Prolog, в которых тип переменных может меняться во время выполнения программы. При наличии таких чисел компилятор может породить команду ADD, а во время выполнения программы машина определяет, нужна ли в данном случае целочисленная команда ADD или команда ADD с плавающей точкой.

Группа команд сдвига включает одну команду сдвига влево и две команды сдвига вправо. Каждая из них имеет два варианта: 32-битный и 64-битный. Команды сдвига в основном используются для манипуляции с битами. Большинство машин CISC имеют довольно много различных команд обычного и циклического сдвига, и практически все они совершенно бесполезны.

Команды загрузки

а	LDSB ADDR, DST	Загружает байт со знаком (8 битов)
	LDUB ADDR, DST	Загружает байт без знака (8 битов)
	LDSH ADDR, DST	Загружает полуслово со знаком (8 битов)
	LDUH ADDR, DST	Загружает полуслово без знака (16 битов)
	LDSW ADDR, DST	Загружает слово со знаком (32 бита)
	LDUW ADDR, DST	Загружает слово без знака (32 бита)
	LDX ADDR, DST	Загружает расширенные слова (64 бита)

Команды сохранения

б	STB SRC, ADDR	Сохраняет байт (8 битов)
	STH SRC, ADDR	Сохраняет полуслово (16 битов)
	STW SRC, ADDR	Сохраняет слово (32 битов)
	STX SRC, ADDR	Загружает расширенное слово (64 бита)

Арифметические команды

в	ADDR1.S2, DST	Сложение
	ADDCC	Сложение с установкой кода условия
	ADDC	Сложение с переносом
	ADDCCC	Сложение с переносом и установкой кода условия
	SUBR1.S2, DST	Вычитание
	SUBCC	Вычитание с установкой кода условия
	SUBC	Вычитание с переносом
	SUBCCC	Вычитание с установкой кода переноса
	MULXR1.S2, DST	Умножение
	SDIVXR1.S2, DST	Деление со знаком
	UDIVXR1.S2, DST	Деление без знака
	TADCCR1.S2, DST	Сложение с использованием поля тега

Команды сдвига/циклического сдвига

г	SLLR1.S2, DST	Логический сдвиг влево (32 бита)
	SLLXR1.S2, DST	Логический сдвиг влево (64 бита)
	SRLR1.S2, DST	Логический сдвиг вправо (32 бита)
	SRLXR1.S2, DST	Логический сдвиг вправо (64 бита)
	SRAR1.S2, DST	Арифметический сдвиг вправо (32 бита)
	SRAXR1.S2, DST	Арифметический сдвиг вправо (64 бита)

Логические команды

д	ANDR1.S2, DST"	Логическое И
	ANDCC "	Логическое И с установкой кода условия
	ANDN "	Логическое НЕ-И
	ANDNCC"	Логическое НЕ-И с установкой кода условия
	ORR1.S2, DST"	Логическое ИЛИ
	ORCC"	Логическое ИЛИ с установкой кода условия
	ORN"	Логическое НЕ-ИЛИ
	ORNCC"	Логическое НЕ-ИЛИ с установкой кода условия
	XORR1.S2, DST"	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
	XORCC"	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ с установкой кода условия
	XNOR"	Логическое ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ
	XNORCC"	Логическое ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ с установкой кода условия

Рис. 5.22. Основные целочисленные команды UltraSPARC II (начало)

Передача управления

	BPcc ADDR	Переход с прогнозированием
	BPr SRC, ADDR	Переход в регистр
	CALL ADDR	Вызов процедуры
	RETURN ADDR	Выход из процедуры
	JMPL ADDR, DST	Переход со связыванием
в	SAVER1.S2, DST	Расширение регистровых окон
	RESTORE "	Восстановление регистровых окон
	Tcc CC, TRAP#	Системное прерывание при определенном условии
	PREFETCH FNC	Выборка данных из памяти с упреждением
	LDSTUB ADDR, R	Атомарная операция загрузки/сохранения
	MEMBAR MASK	Барьер памяти

Прочие команды

	SETHI CON, DST	Установка битов с 10 по 13
	MOVcc CC, S2, DST	Перемещение при определенном условии
	MOVr, R1.S2, DST	Перемещение в зависимости от значения регистра
	NOP	Пустая операция
ж	POPCS1.DST	Подсчет генеральной совокупности
	RDCCR V, DST	Чтение регистра кода условия
	WRCCR, R1.S2, V	Запись регистра кода условия
	RDPC V, DST	Чтение счетчика команд

SRC = входной регистр (source register);
DST = выходной регистр (destination redister);
R1 = входной регистр;
S2 = источник: регистр; или непосредственно получаемые данные;
ADDR = адрес памяти;

TRAP# = номер системного прерывания;
FCN = код функции;
MASK = тип операции;
CON = константа;
V = указатель регистра;

CC = набор кодов условия;
R = выходной регистр;

ее = условие;
г = LZ, LEZ, 2, NZ, GZ, GEZ

Рис. 5.22. Основные целочисленные команды UltraSPARC II (окончание)

Логические команды аналогичны арифметическим. Эта группа включает команды AND (И), (Ж(ИЛИ), EXCLUSIVE OR (ИСКЛЮЧАЮЩЕЕ ИЛИ), ANDN (НЕ-И), ORN (НЕ-ИЛИ) и XNOR (ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ). Значение последних трех команд спорно, но они могут выполняться за один цикл и не требуют практически никакого дополнительного аппаратного обеспечения, поэтому они часто включаются в набор команд. Даже разработчики машин RISC порой поддаются искушению.

Следующая группа содержит команды передачи управления. BPcc представляет собой набор команд, которые совершают переходы при различных условиях и определяют прогноз компилятора по поводу перехода. Команда BPr проверяет регистр и совершает переход, если условие подтвердилось.

Предусмотрено два способа вызова процедур. Для команды CALL используется формат 4 (см. рис. 5.10) с 30-битным смещением. Этого значения достаточно для того, чтобы добраться до любой команды в пределах 2 Гбайт от вызывающего оператора в любом направлении. Команда CALL копирует адрес возврата в регистр R15, который после вызова превращается в регистр R31.

Второй способ вызова процедуры — команда JMPL, для которой используется формат 1a или 1b, позволяющая помещать адрес возврата в любой регистр. Такая форма может быть полезной в том случае, если целевой адрес вычисляется во время выполнения.

Команды SAVE и RESTORE манипулируют регистровым окном и указателем стека. Обе команды совершают прерывание, если следующее (предыдущее) окно недоступно.

В последней группе содержатся команды, которые не попали ни в одну из групп. Команда SETHI необходима, поскольку невозможно поместить 32-битный непосредственный операнд в регистр. Для этого команда SETHI устанавливает биты с 10 по 31, а затем следующая команда передает оставшиеся биты, используя непосредственный формат.

Команда POPC подсчитывает число битов со значением 1 в слове. Последние три команды предназначены для чтения и записи специальных регистров.

Ряд широко распространенных команд CISC, которые отсутствуют в этом списке, можно легко получить, используя либо регистр GO, либо операнд-константу (формат 1b). Некоторые из них даны в табл. 5.9. Эти команды узнаются ассемблером UltraSPARC II и часто порождаются компиляторами. Многие из них используют тот факт, что регистр GO связан с 0 и что запись в этот регистр не произведет никакого результата.

Команды компьютера picoJava II

Настало время рассмотреть уровень команд машины picoJava II. Здесь реализован полный набор команд JVM (226 команд), а также 115 дополнительных команд, предназначенных для C, C++ и операционной системы. Мы сосредоточимся главным образом на командах JVM, поскольку компилятор Java производит только эти команды. Архитектура команд JVM не содержит регистров, доступных пользователю, а также не имеет некоторых других особенностей, обычных для большинства центральных процессоров. (В процессоре picoJava II есть 64 встроенных регистра для вершины стека, но пользователи их не видят.) Большинство команд JVM помещают слова в стек, оперируют словами, находящимися в стеке, и выталкивают слова из стека. Большинство команд JVM выполняются непосредственно аппаратным обеспечением picoJava II, но некоторые из них микропрограммируются, а некоторые даже передаются программе обработки для выполнения.

Таким образом, для того чтобы заставить машину работать, требуется небольшая система уровня команд, но эта система гораздо меньше по размеру, чем полный интерпретатор JVM, и вызывается она только в редких случаях. Она содержит код для интерпретации нескольких сложных команд, загрузчик класса, верификатор байт-кода, администратор потока и программу чистки памяти («сборщик мусора»).

Таблица 5.9. Некоторые моделируемые команды UltraSPARC II

Команда	Как получить команду
MOV SRC, DST	Выполнить команду OR над SRC и G0 и сохранить результат в DST
CMP SRC1, SRC2	Вычесть SRC2 из SRC1 (команда SUBCC) и сохранить результат в G0
TST SRC	Выполнить команду ORCC над SRC и G0 и сохранить результат в G0
NOT DST	Выполнить команду XNOR над DST и G0
NEG DST	Вычесть SRC2 из SRC1 (команда SUBCC) и сохранить результат в G0
INC DST	Прибавить 1 к DST (непосредственный операнд) — команда ADD
DEC DST	Отнять 1 от DST (непосредственный операнд) — команда SUB
CLR DST	Выполнить команду OR над G0 и G0 и сохранить результат в DST
NOP	SETHI G0 на 0
RET	JMPL%I7+8, %G0

JVM содержит относительно небольшой набор простых команд. Набор всех команд JVM (за исключением некоторых расширенных, коротких и быстрых вариантов команд) приведен на рис. 5.23.

Команды JVM типизированы. Одну и ту же операцию с разными типами данных выполняют разные команды. Например, команда ILOAD помещает в стек целое 32-битное число, а команда ALOAD помещает в стек 32-битный указатель. Такое строгое разделение необязательно для правильного выполнения программы, поскольку в обоих случаях 32 бита, которые находятся в определенной ячейке памяти, передаются в стек независимо от типа этого 32-битного слова. Такое жесткое разграничение типов требуется для того, чтобы можно было проверить во время выполнения программы, не нарушены ли какие-нибудь ограничения (например, не пытается ли программа превратить целое число в указатель, чтобы обратиться к памяти).

Перейдем к командам JVM. Первая команда в списке — typeLOAD IND8.

На самом деле это не одна команда, а шаблон для порождения команд. Команды JVM регулярны, поэтому вместо того чтобы приводить все команды по одной, в некоторых случаях мы будем давать правило для порождения команд. В данном случае слово type заменяет одну из четырех букв: I, L, F и D, которые соответствуют типам integer (целые числа), long, float (32-битные числа с плавающей точкой) и double (64-битные числа с плавающей точкой) соответственно. Следовательно, здесь подразумевается 4 команды (ILOAD, LLOAD, FLOAD и DLOAD), каждая из которых содержит 8-битный индекс IND8 для нахождения локальной переменной и помещает в стек значение соответствующей длины и типа. Мы рассматривали только одну из этих команд — ILOAD, но остальные действуют точно так же и отличаются только по числу слов, помещаемых в стек, и по типу значения.

Кроме этих четырех команд существует еще четыре команды загрузки typeALOAD. Эти команды помещают в стек элементы массива. Во всех четырех случаях сначала в стек должен загружаться указатель на массив и индекс элемента массива. Затем эти команды выталкивают индекс и указатель, производят вычисление, чтобы найти элемент массива, и помещают этот элемент в стек. При вычислении нужно знать размер элементов, который определяется по типу. Эти команды получают индекс массива из стека, поэтому сама команда не содержит операнда.

Команды загрузки

а	typeLOAD_IND8	Помещает локальную переменную в стек
	typeALOAD	Помещает элемент массива в стек
	BALOAD	Помещает байт из массива в стек
	SALOAD	Помещает short integer из массива в стек
	CALOAD	Помещает символ из массива в стек
	AALOAD	Помещает указатель из массива в стек

Команды сохранения

б	typeSTORE_IND8	Выталкивает из стека значение и сохраняет его в локальной переменной
	typeASTORE	Выталкивает из стека значение и сохраняет его в массиве
	BASTORE	Выталкивает из стека байт и сохраняет его в массиве
	SASTORE	Выталкивает из стека short и сохраняет его в массиве
	CASTORE	Выталкивает из стека символ и сохраняет его в массиве
	AASTORE	Выталкивает из стека указатель и сохраняет его в массиве

Команды помещения в стек

в	BIPUSH CON8	Помещает небольшую константу в стек
	SIPUSHCON16	Помещает 16-битную константу в стек
	LDC_IND8	Помещает в стек константу из набора констант
	typeCONST_*	Помещает в стек непосредственную константу
	ACONS_NULL	Помещает в стек нулевой указатель

Арифметические команды

г	typeADD	Сложение
	typeSUB	Вычитание
	typeMUL	Умножение
	typeDIV	Деление
	typeREM	Остаток
	typeNEG	Отрицание

Логические команды/команды сдвига

д	HAND	Логическое И
	NOR	Логическое ИЛИ
	iIXOR	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
	iISHL	Сдвиг влево
	IISHR	Сдвиг вправо
	iIUSHR	Сдвиг вправо без знака

Команды преобразования

е	x2y	Преобразует x в y
	i2c	Преобразует целое число в символ
	i2b	Преобразует целое число в байт

Команды управления стеком

ж	DUPxx	Шесть команд дублирования
	POP	Выталкивает целое число из стека и отбрасывает его
	POP2	Выталкивает два целых числа из стека и отбрасывает их
	SWAP	Меняет местами два верхних целых числа в стеке

Рис. 5.23. Набор командам (начало)

Команды сравнения

IF_ ICMPrel 0FFSET16	Условный переход
IF_ACMPEQ 0FFSET16	Переход в случае равенства двух значений
IF_ACMPNE0FFSET16	Переход в случае неравенства двух значений
IFrel 0FFSET16	Проверяет одно значение и совершает переход
IFNULLOFFSET16	Совершает переход, если значение равно 0
IFNONNULLOFFSET16	Совершает переход, если значение не равно 0
LCMP	Сравнивает два числа long
FCMPL	Сравнивает два числа с плавающей точкой на <
FCMPG	Сравнивает два числа с плавающей точкой на >
DCMPL	Сравнивает два числа типа double на <
DCMPG	Сравнивает два числа типа double на >

Команды передачи управления

INVOKEVIRTUAL IND16	Вызов процедуры
INVOKESTATIC IND16	Вызов процедуры
INVOKEINTRFACE	Вызов процедуры
INVOKESPECIAL IND16	Вызов процедуры
JSROFFSET16	Вызов процедуры
tipeRETURN	Возвращает значение
ARETURN	Возвращает указатель
RETURN	Возвращает пустой тип
RET IND8	Выход из процедуры
G0T0 0FFSET16	Безусловный переход

Операции с массивами

ANEWARRAYIND16	Создает массив переменных
NEWARRAY ATYPE	Создает массив из массивов
MULTINEWARRAY 1N16, D	Создает многомерный массив
ARRAYLENGTH	Выдает длину массива

Прочие команды

IINCIND8, CON16	Увеличивает локальную переменную на 1
WIDE	Префикс
NOP	Пустая операция
GETFIELDIND16	Считывает поле из объекта
PUTFIELD IND16	Записывает слово в объект
GETSTATIC IND16	Получает статическое поле из класса
NEWIND16	Создает новый объект
INSTANCEOF 0FFSET16	Определяет тип объекта
CHECKCASTIND16	Проверяет тип объекта
ATHROW	Обработка исключения
LOOKUPSWITCH...	Разбросанные многоуровневые переходы
TABLESWITCH...	Компактные многоуровневые переходы
MONITORENTER	Входит в управляющую прграмму
MONITOREXIT	Выходит из управляющей прграммы

IND 8/16 = индекс локальной переменной; CON 8/16, D, ATYPE = константа;
type, x, y = I, L, F, D;
OFFSET 16 для команд перехода

Рис. 5.23. Набор команд JVM (окончание)

Последние четыре команды этой группы также предназначены для работы с элементами массива, но только других типов. Они поддерживают byte (байт — 8 битов), short (16 битов), char (символ — 16 битов) и pointer (указатель — 32 бита). Таким образом, всего существует 12 команд LOAD.

Команды typeSTORE обратны командам typeLOAD. Каждая команда выталкивает элемент из стека и сохраняет его в локальной переменной. Одну из этих команд (ISTORE) мы уже рассматривали, когда изучали машину IJVM. Здесь также имеются команды для сохранения элементов массива. В вершине стека находится значение нужного типа. Под ним находится индекс, а еще ниже — указатель на массив. Все три элемента удаляются из стека этой командой.

Команды PUSH помещают значение в стек. Команду BI PUSH мы уже рассматривали. Команда SIPUSH выполняет ту же операцию, но только с 16-битным числом. Команда LDC помещает в стек значение из набора констант. Следующая команда представляет целую группу команд всех четырех основных типов (integer, long, float и double). Каждая команда содержит только код операции, и каждый код операции помещает определенное значение в стек. Например, команда ICONST_0 (код операции 0x03) помещает в стек 32-битное слово 0. То же самое действие можно произвести с помощью команды BIPUSH, но это займет два байта. Благодаря оптимизации самых распространенных команд программы JVM получаются небольшими по размеру. Поддерживаются следующие значения: Integers (целые числа) -1, 0, 1, 2, 3, 4, 5; longs 0 и 1; floats (числа с плавающей точкой) 0,0, 1,0 и 2,0 и doubles 0,0 и 1,0. Команда CONST_NULL помещает в стек нулевой указатель. Сочетание кодов операций и самых распространенных адресов в одной 1-байтной команде сильно сокращает размер команды, что экономит память и время на передачу бинарных программ на языке Java по Интернету.

Арифметические операции абсолютно регулярны. Для каждого из основных четырех типов имеется 4 команды. Три логические операции и три операции сдвига применяются только для целых чисел и чисел типа long. Наличие команды AND для чисел с плавающей точкой противоречило бы строгим правилам типизирования JVM. Строка x2y в таблице представляет 4x4 команд преобразования. Значения каждого типа могут переделываться в любой другой тип (но только не в тот же самый), поэтому здесь имеется 12 команд. Самой типичной из них является команда I2F, которая превращает целое число в число с плавающей точкой.

Группа команд управления стеком содержит команды, которые дублируют верхнее значение или два значения стека и помещают их в различные части стека (не обязательно в вершину). Остальные команды выталкивают значения из стека и меняют местами два верхних значения.

Команды группы сравнения выталкивают одно или два значения из стека и проверяют их. Если из стека выталкивается два значения, то одно из них вычитается из другого, а результат проверяется. Если выталкивается одно значение, то оно и проверяется. Суффикс rel замещает реляционные операторы: LT, LE, EQ, NE, GE и GT. Команды со смещением совершают переход, если определенное условие подтверждено. Остальные команды помещают результат обратно в стек.

Следующая группа предназначена для вызова процедур и возвращения значений. При изучении машины IJVM мы рассматривали очень простые версии команд INVOKEVIRTUAL и IRETURN. Полные версии содержат гораздо больше параметров,

и существует множество команд, которые покрывают самые различные случаи. В этой книге мы не будем описывать эти команды. Подробнее см. [85].

Еще одну группу образуют 4 команды для создания одномерных и многомерных массивов и проверки их длины. В машине JVM массивы хранятся в «куче» и периодически очищаются (процесс «сборки мусора»), когда они уже больше не нужны.

Последняя группа включает в себя оставшиеся команды. Каждая из этих команд имеет специальное назначение, связанное с какой-нибудь особенностью языка Java. Описание этих команд не входит в задачи этой книги.

А теперь нужно сказать пару слов о самом уровне команд picojava II. Это машина с обратным порядком байтов (хотя существует несколько команд, которые можно переделать в формат с прямым порядком байтов). Слова состоят из 32 битов, хотя существуют команды для работы с единицами по 8,16 и *64* бита. Стек в памяти располагается от верхних адресов к нижним в отличие от IJVM (спецификация JVM допускает оба варианта).

Машина picojava II была разработана для программ на Java, C и C++. Но чтобы запустить программы на C и C++, нужен компилятор, который превращает C и C++ в команды picojava И. После того как программа на C или C++ была скомпилирована на JVM, все способы оптимизации аппаратного обеспечения, которые мы описывали в главе 4, становятся применимы для C и C++.

Чтобы программы на C и C++ могли работать на машине picojava II, к уровню архитектуры команд было добавлено 115 дополнительных команд. Большинство из них составляют два или более байтов в длину и начинаются с одного из двух зарезервированных кодов JVM (OxFE и OxFF), которые показывают, что дальше следует расширенная команда. Ниже мы дадим краткий обзор особенностей picojava II, не характерных для JVM.

В машине picojava II содержится 25 32-битных регистров. Четыре из них по функциям эквивалентны регистрам PC, LV, SP и CPP машины IJVM. Регистр OPLIM помещает определенное предельное значение в SP. Ести значение SP выходит за пределы OPLIM, то происходит прерывание. Эта особенность позволяет представлять стек в виде связного списка участков стека, а не как один непрерывный блок памяти. Регистр FRAME отмечает конец фрейма локальных переменных и указывает на слово, которое содержит счетчик команд вызывающей процедуры.

Среди других регистров можно назвать слово состояния программы — это регистр, который следит, насколько заполнен 64-регистровый стековый кэш, и четыре регистра, которые используются для управления потоком. Кроме того, существует 4 регистра для ловушек и прерываний и 4 регистра для вызова процедур и возвращения значений в командах на языках C и C++. Поскольку picojava II не имеет виртуальной памяти, для ограничения определенной части памяти, к которой может иметь доступ текущая программа на C или C++, используются два специальных регистра. Расширенные команды можно разделить на 5 категорий. К первой категории относятся команды для чтения и записи верхних регистров. Ко второй категории относятся команды для работы с указателями. Они позволяют считывать из памяти и записывать в память произвольные слова. Большинство из этих команд выталкивают машинный адрес из стека, а затем помещают в стек содержи-

мое *байта, слова и т.д., находящегося в ячейке с этим адресом. Такие команды нарушают* типовую безопасность языка Java, но они нужны для C и C++.

В третью категорию входят команды для программ на C и C++, например вызов процедур и выход из процедур без применения команд JVM. К четвертой группе относятся команды, которые управляют аппаратным обеспечением, например кэш-памятью. В пятую категорию включены команды разного рода, например проверки при включении. Программы, использующие эти дополнительные команды, не переносимы на другие машины JVM.

Сравнение наборов команд

Рассмотренные наборы команд очень сильно отличаются друг от друга. Pentium II — это классическая двухадресная 32-битная машина CISC. Она пережила долгую историю, у нее особые и нерегулярные способы адресации, и она содержит множество команд, которые обращаются к памяти. UltraSPARC II — это современная трехадресная 64-битная машина RISC с архитектурой загрузки/сохранения, всего двумя способами адресации и компактным и эффективным набором команд. JVM — это машина со стековой организацией, практически без способов адресации, с регулярными командами и очень плотным кодированием команд.

В основу разработки компьютера Pentium II легли три основных фактора:

1. Обратная совместимость.
2. Обратная совместимость.
3. Обратная совместимость.

При нынешнем положении вещей никто не стал бы разрабатывать такую нерегулярную машину с таким маленьким количеством абсолютно разных регистров. По этой причине очень сложно писать компиляторы. Из-за недостатка регистров компиляторам постоянно приходится сохранять переменные в памяти, а затем вновь загружать их, что очень невыгодно даже при наличии трех уровней кэш-памяти. Только благодаря таланту инженеров компании Intel процессор Pentium II работает достаточно быстро, несмотря на все недостатки уровня команд. Но, как мы увидели в главе 4, реализация этого процессора чрезвычайно сложна и требует транзисторов в два раза больше, чем picojava II, и почти в полтора раза больше, чем UltraSPARC II.

Современная разработка уровня команд представлена в процессоре UltraSPARC II. Он содержит полную 64-битную архитектуру команд (с шиной на 128 битов). Процессор содержит много регистров и имеет набор команд, в котором преобладают трехрегистровые операции, а также имеется небольшая группа команд LOAD и STORE. Все команды одного размера, хотя число форматов вышло из-под контроля. Большинство новых разработок очень похожи на UltraSPARC II, но содержат меньше форматов команд.

JVM — машина совершенно другого рода. Здесь уровень команд изначально разрабатывался так, чтобы небольшие программы можно было передавать по Интернету и интерпретировать на программном обеспечении другого компьютера. Это была разработка для одного языка. Все это привело к использованию стека и коротким командам разной длины с очень высокой плотностью (в среднем всего

1,8 байта на команду). Создание аппаратного обеспечения, которое выполняет одну команду JVM за раз и при выполнении одной команды обращается к памяти два или три раза, кажется нонсенсом. Но благодаря помещению на микросхему стека из 64 слов и переделыванию целых последовательностей команд в современные трехадресные команды RISC машина picojava II умудряется неплохо работать с очень неэффективной архитектурой команд.

Ядро современного компьютера представляет собой сильно конвейеризированное трехрегистровое устройство загрузки/сохранения типа RISC. UltraSPARC II просто открыто сообщает об этой структуре пользователю. Pentium II скрывает эту систему RISC, перенимая старую архитектуру команд и разбивая команды CISC на микрооперации RISC. Машина picojava II также таит в себе ядро RISC, комбинируя несколько команд для получения одной команды RISC.

Поток управления

Поток управления — это последовательность, в которой команды выполняются динамически, то есть во время работы программы. При отсутствии переходов и вызовов процедур команды вызываются из последовательных ячеек памяти. Вызов процедуры влечет за собой изменение потока управления, выполняемая в данный момент процедура останавливается, и начинается выполнение вызванной процедуры. Сопрограммы связаны с процедурами и вызывают сходные изменения в потоке управления. Они нужны для моделирования параллельных процессов. Ловушки (traps) и прерывания тоже меняют поток управления при возникновении определенных ситуаций. Все это мы обсудим в следующих разделах.

Последовательный поток управления и переходы

Большинство команд не меняют поток управления. После выполнения одной команды вызывается и выполняется та команда, которая идет вслед за ней в памяти. После выполнения каждой команды счетчик команд увеличивается на число, соответствующее длине команды. Счетчик команд представляет собой линейную функцию от времени, которая увеличивается на среднюю длину команды за средний промежуток времени. Иными словами, процессор выполняет команды в том же порядке, в котором они появляются в листинге программы, как показано на рис. 5.24, *а*.

Если программа содержит переходы, то это простое соотношение между порядком расположения команд в памяти и порядком их выполнения больше не соответствует действительности. При наличии переходов счетчик команд больше не является монотонно возрастающей функцией от времени, как показано на рис. 5.24, *б*. В результате последовательность выполнения команд из самой программы уже не видна. Если программисты не знают, в какой последовательности процессор будет выполнять команды, это может привести к ошибкам. Такое наблюдение побудило Дейкстру [31] написать статью под названием «Оператор GOTO нужно считать вредным», в котором он предлагал избегать в программах оператора goto. Эта статья дала толчок революции в программировании, одним из нововведений которой было устранение операторов goto более структурированными формами потока управления, например циклами while. Конечно, эти про-

граммы компилируются в программы второго уровня, которые могут содержать многочисленные переходы, поскольку реализация операторов if, while и структур языков высокого уровня требует совершения переходов.

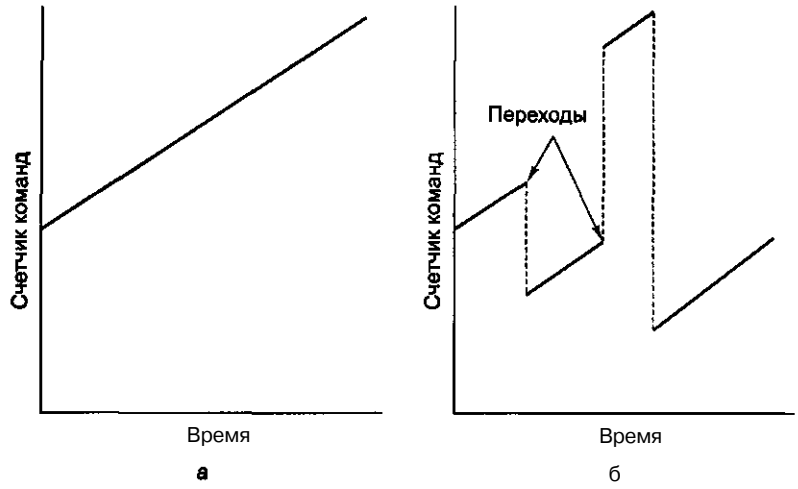


Рис. 5.24. Счетчик команд как функция от времени (приближенно): без переходов (а); с переходами (б)

Процедуры

Самым важным способом структурирования программ является процедура. С одной стороны, вызов процедуры, как и команда перехода, изменяет поток управления, но в отличие от команды перехода после выполнения задачи управление возвращается к команде, которая вызвала процедуру.

С другой стороны, тело процедуры можно рассматривать как определение новой команды на более высоком уровне. С этой точки зрения вызов процедуры можно считать отдельной командой, даже если процедура очень сложная. Чтобы понять часть программы, содержащую вызов процедуры, нужно знать, что она делает и как она это делает.

Особый интерес представляет **рекурсивная процедура.** Это такая процедура, которая вызывает сама себя либо непосредственно, либо через цепочку других процедур. Изучение рекурсивных процедур дает значительное понимание того, как реализуются вызовы процедур и что в действительности представляют собой локальные переменные. А теперь рассмотрим пример рекурсивной процедуры.

«Ханойская башня» — это древняя задача, которая имеет простое решение с использованием рекурсии. В одном монастыре в Ханое есть три золотых колышка. Вокруг первого из них располагались 64 концентрических золотых диска, каждый из них с отверстием посередине для колышка. Диаметр дисков уменьшается снизу вверх. Второй и третий колышки абсолютно пусты. Монахи переносят все диски на колышек 3 по одному диску, но диск большего размера не может находиться сверху на диске меньшего размера. Говорят, что когда они закончат, наступит конец света. Если вы хотите потренироваться, вы можете использовать пласти-

ковые диски, и не 64, а поменьше, но когда вы решите эту задачу, ничего страшного не произойдет. Чтобы произошел конец света, требуется 64 диска, и все они должны быть из золота. На рисунке 5.25 показана начальная конфигурация, где число дисков (п) равно 5.

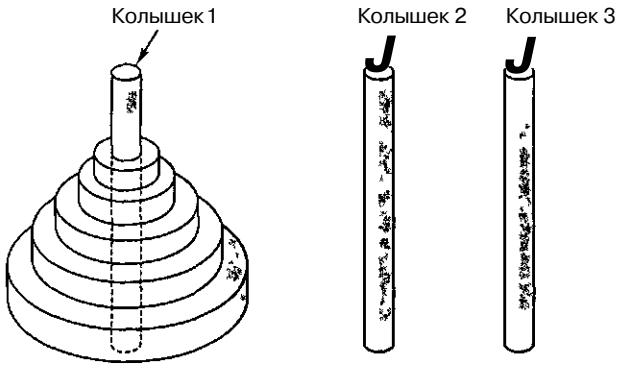


Рис. 5.25. Исходное положение в задаче «Ханойская башня» для пяти дисков

Чтобы переместить п дисков с колышка 1 на колышек 3, нужно сначала перенести п-1 дисков с колышка 1 на колышек 2, затем перенести один диск с колышка1 на колышек 3, а потом перенести п-1 диск с колышка 2 на колышек 3. Решение этой задачи проиллюстрировано на рис. 5.26.

Для решения задачи нам нужна процедура, которая перемещает п дисков с колышка i на колышек]. Когда эта процедура вызывается,

towers (n I j)

решение выводится на экран. Сначала процедура проверяет, равно ли п единице. Если да, то решение тривиально: нужно просто переместить один диск с i на j. Если п не равно 1, решение состоит из трех частей, как было сказано выше, и каждая из этих частей представляет собой рекурсивную процедуру.

Полное решение показано в листинге 5.6. Вызов процедуры

towers (3. 1 3)

порождает еще три вызова

```
towers (2  1  2)
towers (I  1.3)
towers (2, 2  3)
```

Первый и третий вызовы производят по три вызова каждый, и всего получится семь.

Листинг 5.6. Процедура для решения задачи «Ханойская башня»

```
public void towers (int n. int I. int j)
int k.
if (n==I)
System out рппШС'Переместить диск из" + i + "на" + j). else
k=6-i-j.
towers(n-I. I, k)
towers (1. i. j).
towers (n-1. k. j).
```

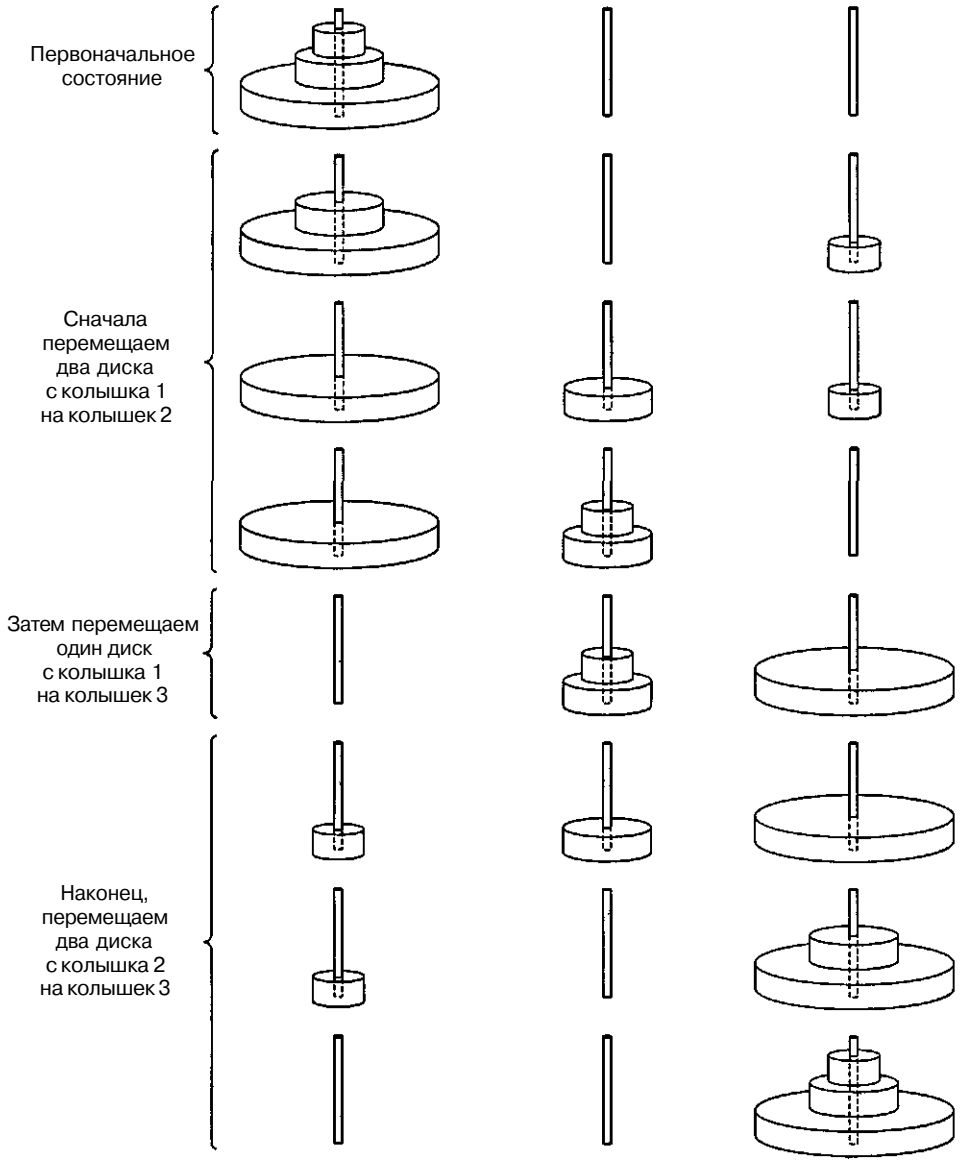


Рис. 5.26. Решение задачи «Ханойская башня» для трех дисков

Для рекурсивных процедур нам нужен стек, чтобы хранить параметры и локальные переменные для каждого вызова, как и в IJVM. Каждый раз при вызове процедуры на вершине стека новый стековый фрейм для процедуры. Текущий фрейм — это тот фрейм, который был создан последним. В наших примерах стек растет снизу вверх от малых адресов к большим, как и в IJVM.

Помимо указателя стека, который указывает на вершину стека, удобно иметь указатель фрейма (FP — Frame Pointer), который указывает на фиксированное

место во фрейме. Он может указывать на связующий указатель, как в IJVM, или на первую локальную переменную. На рис. 5.27 изображен стековый фрейм для машины с 32-битным словом. При первом вызове процедуры towers в стек помещаются n, i и j, а затем выполняется команда **CALL**, которая помещает в стек адрес возврата, 1012. Вызванная процедура сохраняет в стеке старое значение FP (1000) в ячейке 1016, а затем передвигает указатель стека для обозначения места хранения локальных переменных. При наличии только одной 32-битной локальной переменной (k) SP (Stack Pointer — указатель стека) увеличивается на 4 до 1020. На рис. 5.27, *а* показан результат всех этих действий.

Первое, что должна сделать процедура после того, как ее вызвали, — это сохранить предыдущее значение FP (так, чтобы его можно было восстановить при выходе из процедуры), скопировать значение SP в FP и, возможно, увеличить на одно слово, в зависимости от того, куда указывает FP нового фрейма. В этом примере FP указывает на первую локальную переменную, а в IJVM LV указывает на связующий указатель. Разные машины оперируют с указателем фрейма немного поразному, иногда помещая его в самый низ стекового фрейма, иногда — в вершину, а иногда — в середину, как на рис. 5.27. В этом отношении стоит сравнить рис. 5.27 с рис. 4.12, чтобы увидеть два разных способа обращения со связующим указателем. Возможны и другие способы. Но в любом случае обязательно должна быть возможность выйти из процедуры и восстановить предыдущее состояние стека.

Код, который сохраняет старый указатель фрейма, устанавливает новый указатель фрейма и увеличивает указатель стека, чтобы зарезервировать пространство для локальных переменных, называется **прологом процедуры.** При выходе из процедуры стек должен быть очищен, и этот процесс называется **эпилогом процедуры.** Одна из важнейших характеристик компьютера — насколько быстро он может совершать пролог и эпилог. Если они очень длинные и выполняются медленно, делать вызовы процедур будет невыгодно. Команды ENTER и LEAVE в машине Pentium II были разработаны для того, чтобы пролог и эпилог процедуры работали эффективно. Конечно, они содержат определенную модель обращения с указателем фрейма, и если компилятор имеет другую модель, их нельзя использовать.

А теперь вернемся к задаче «Ханойская башня». Каждый вызов процедуры добавляет новый фрейм к стеку, а каждый выход из процедуры удаляет фрейм из стека. Ниже мы проиллюстрируем, как используется стек при реализации рекурсивных процедур. Начнем с вызова

towers (3. 1, 3)

На рис. 5.27, *а* показано состояние стека сразу после вызова процедуры. Сначала процедура проверяет, равно ли n единице, а установив, что n=3, заполняет k и совершает вызов

towers (2. 1. 2)

Состояние стека после завершения этого вызова показано на рис. 5.27, *б.* После этого процедура начинается с начала (вызванная процедура всегда начинается с начала). На этот раз условие n=1 снова не подтверждается, поэтому процедура снова заполняет k и совершает вызов

towers (1. 1, 3)

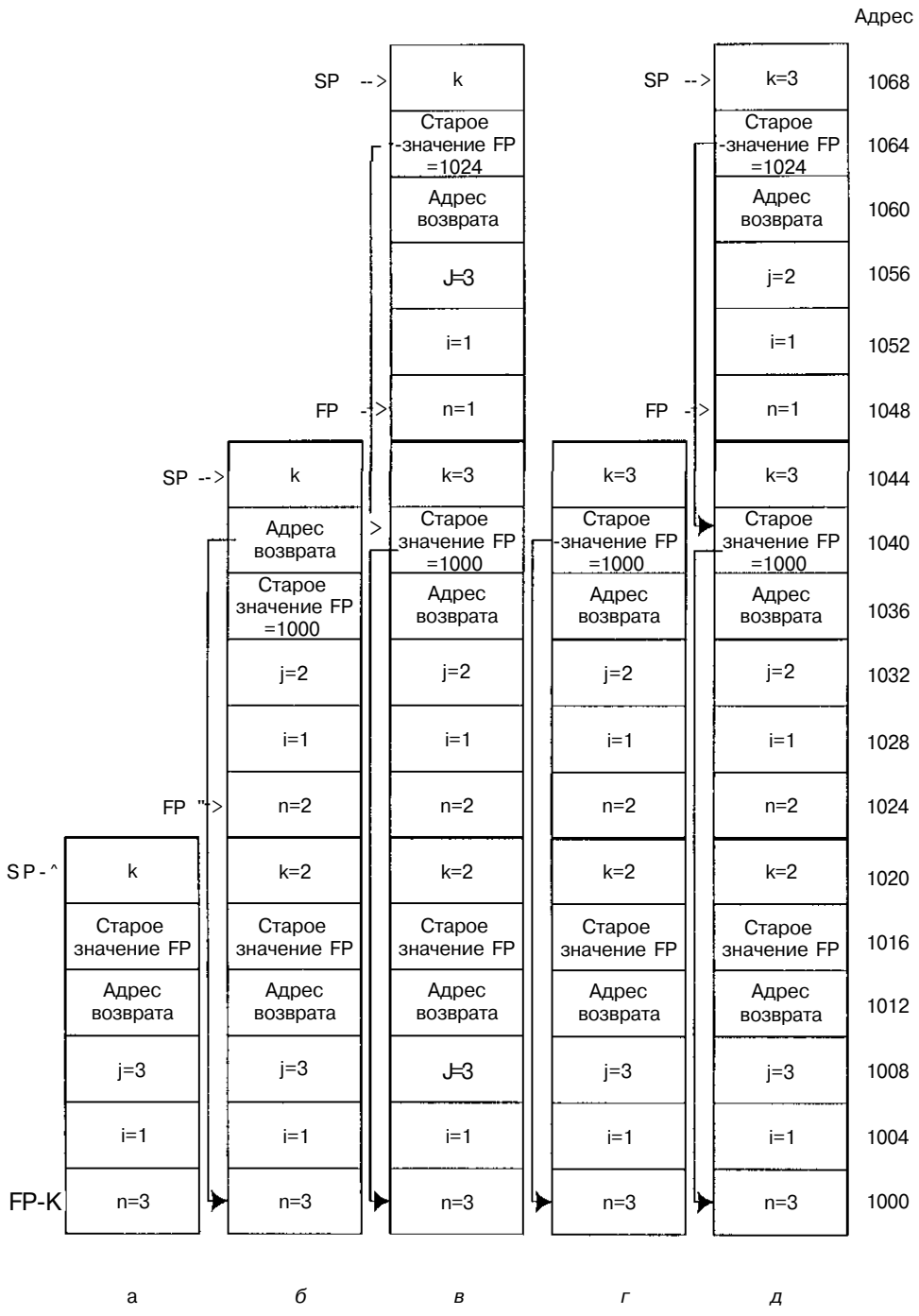


Рис. 5.27. Состояние стека во время выполнения программы листинга 5.6

Состояние стека после этого вызова показано на рис. 5.27, *в*. Счетчик команд указывает на начало процедуры. На этот раз условие подтверждается, и на экран выводится строка. Затем совершается выход из процедуры. Для этого удаляется один фрейм, а значения FP и SP переопределяются (см. рис. 5.27, *г)*. Затем процедура *продолжает выполняться в адресе возврата:*

towers (1. 1. 2)

Это добавляет новый фрейм в стек (см. рис. 5.27, д). Печатается еще одна строка. После выхода из процедуры фрейм удаляется из стека. Вызовы процедур продолжаются до тех пор, пока не завершится выполнение первой процедуры и пока фрейм, изображенный на рис. 5.27, *а,* не будет удален из стека. Чтобы вы лучше смогли понять, как работает рекурсия, вам нужно произвести полное выполнение процедуры

towers (3. 1. 3)

используя ручку и бумагу.

Сопрограммы

В обычной последовательности вызовов существует четкое различие между вызывающей процедурой и вызываемой процедурой. Рассмотрим процедуру А, которая вызывает процедуру В (рис. 5.28).

Процедура В работает какое-то время, затем возвращается к А. На первый взгляд может показаться, что эти ситуации симметричны, поскольку ни А, ни В не являются главной программой. И А, и В — это процедуры. (Процедуру А можно было бы назвать основной программой, но это в данном случае неуместно.) Более того, сначала управление передается от А к В (при вызове), а затем — от В к А (при возвращении).

Различие состоит в том, что когда управление переходит от А к В, процедура В начинает выполняться с самого начала; а при переходе из В обратно в А выполнение начинается не с начала процедуры А, а с того момента, за которым последовал вызов процедуры В. Если А работает некоторое время, а потом снова вызывает процедуру В, выполнение В снова начинается с самого начала, а не с того места, после которого произошло возвращение к процедуре А. Если процедура А вызывает процедуру В много раз, процедура В каждый раз начинается с начала, а процедура А уже никогда больше с начала не начинается.

Это различие отражается в способе передачи управления между А и В. Когда А вызывает В, она использует команду вызова процедуры, которая помещает адрес возврата (то есть адрес того выражения, которое последует за процедурой) в такое место, откуда его потом легко будет вытащить, например в вершину стека. Затем она помещает адрес процедуры В в счетчик команд, чтобы завершить вызов. Для выхода из процедуры В используется не команда вызова процедуры, а команда выхода из процедуры, которая просто выталкивает адрес возврата из стека и помещает его в счетчик команд.

Иногда нужно иметь две процедуры А и В, каждая из которых вызывает другую в качестве процедуры, как показано на рис. 5.29. При возврате из В к А про-

цедура B совершает переход к тому оператору, за которым последовал вызов процедуры B. Когда процедура A передает управление процедуре B, она возвращается не к самому началу B (за исключением первого раза), а к тому месту, на котором произошел предыдущий вызов A. Две процедуры, работающие подобным образом, называются **сопрограммами.**

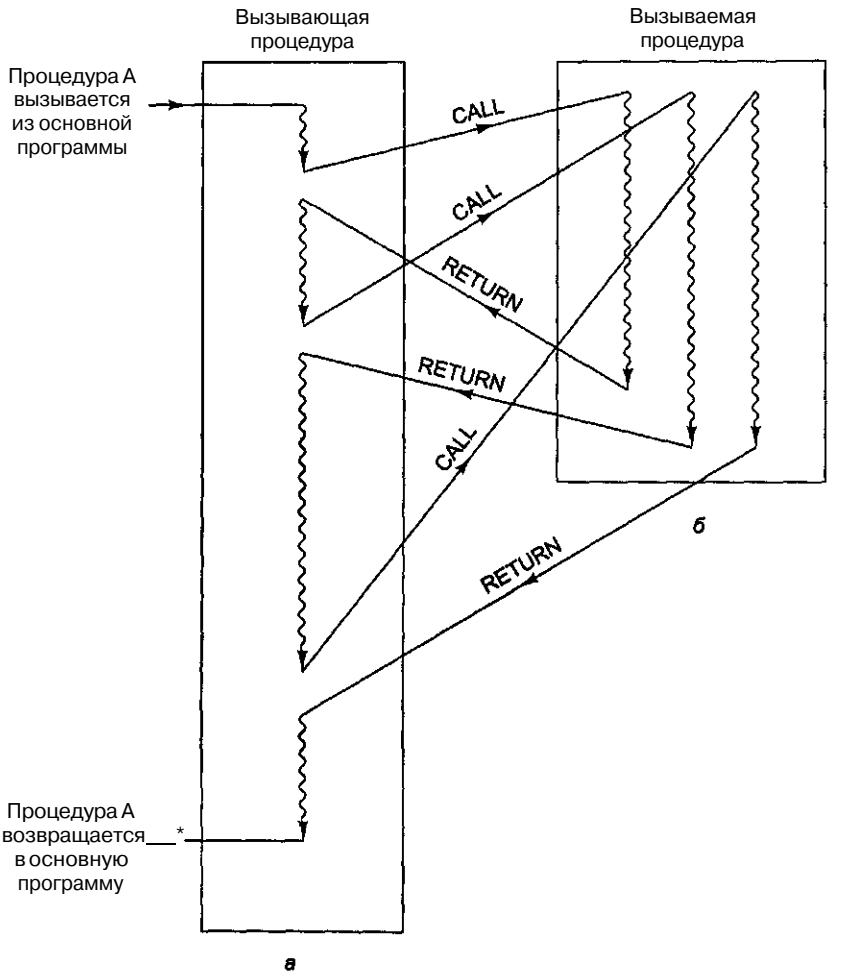


Рис. 5.28. Выполнение вызванной процедуры всегда начинается с самого начала этой процедуры

Сопрограммы обычно используются для того, чтобы производить параллельную обработку данных на одном процессоре. Каждая сопрограмма работает как бы параллельно с другими сопрограммами, как будто у нее есть собственный процессор. Такой подход упрощает программирование некоторых приложений. Он также полезен для проверки программного обеспечения, которое потом будет работать на мультипроцессоре.

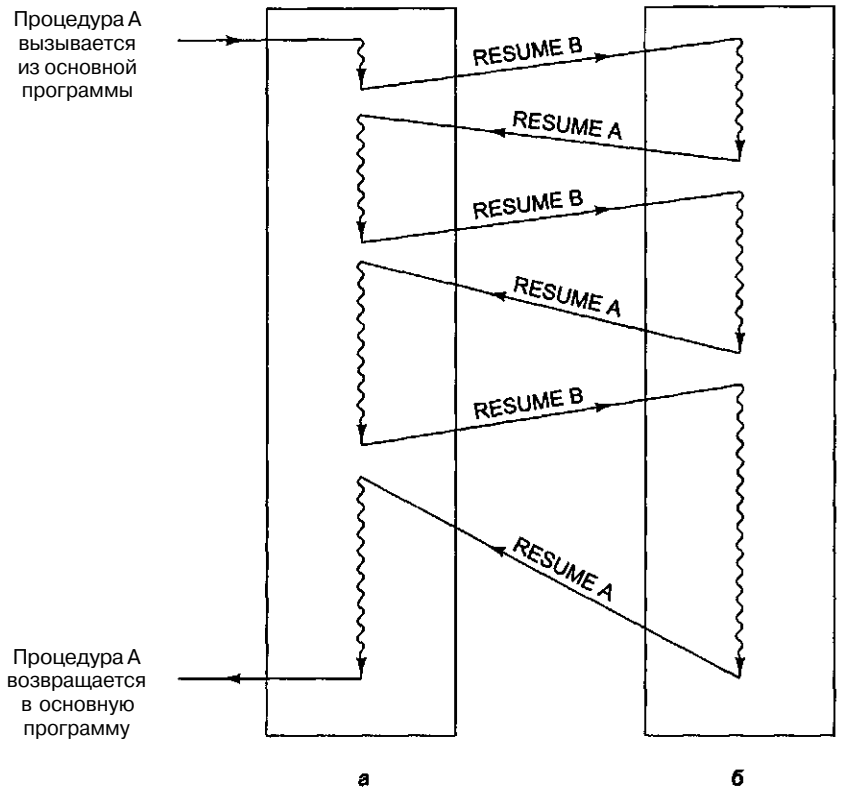


Рис. 5.29. После завершения сопрограммы выполнение начинается с того места, на котором оно завершилось в прошлый раз, а не с самого начала

Обычные команды CALL и RETURN не подходят для вызова сопрограмм, поскольку адрес для перехода берется из стека, как и при возврате, но, в отличие от возврата, при вызове сопрограммы адрес возврата помещается в определенном месте, чтобы в последующем к нему вернуться. Было бы неплохо, если бы существовала команда для замены вершины стека на счетчик команд. Эта команда сначала выталкивала бы старый адрес возврата из стека и помещала бы его во внутренний регистр, затем помещала бы счетчик команд в стек и, наконец, копировала бы содержание внутреннего регистра в счетчик команд. Поскольку одно слово выталкивается из стека, а другое помещается в стек, состояние указателя стека не меняется. Такая команда встречается очень редко, поэтому в большинстве случаев ее приходится моделировать из нескольких команд.

Ловушки

Ловушка (trap) — это особый тип вызова процедуры, который происходит при определенном условии. Обычно это очень важное, но редко встречающееся условие. Один из примеров такого условия — переполнение. В большинстве процессоров, если результат выполнения арифметической операции превышает самое боль-

шое допустимое число, срабатывает ловушка. Это значит, что поток управления переходит в какую-то фиксированную ячейку памяти, а не продолжается последовательно дальше. В этой фиксированной ячейке находится команда перехода к специальной процедуре **(обработчику системных прерываний),** которая выполняет какое-либо определенное действие, например печатает сообщение об ошибке. Если результат операции находится в пределах допустимого, ловушка не задействуется.

Важно то, что этот вид прерывания вызывается каким-то исключительным условием, вызванным самой программой и обнаруженным аппаратным обеспечением или микропрограммой. Есть и другой способ определения переполнения. Нужно иметь 1-битный регистр, который принимает значение всякий раз, когда происходит переполнение. Программист, который хочет проверить результат на переполнение, должен включить в программу явную команду «переход в случае установки бита переполнения» после каждой арифметической команды. Но это очень неудобно. А ловушки экономят время и память по сравнению с открытой проверкой под контролем программиста.

Ловушку можно реализовать путем открытой проверки, выполняемой микропрограммой (или аппаратным обеспечением). Если обнаружено переполнение, адрес ловушки загружается в счетчик команд. То, что является ловушкой на одном уровне, может находиться под контролем программы на более низком уровне. Проверка на уровне микропрограммы требует меньше времени, чем проверка под контролем программиста, поскольку она может выполняться одновременно с каким-либо другим действием. Кроме того, такая проверка экономит память, поскольку она должна присутствовать только в одном месте, например в основном цикле микропрограммы, независимо от того, сколько арифметических команд встречается в основной программе.

Наиболее распространенные условия, которые могут вызывать ловушки, — это переполнение и исчезновение значащих разрядов при операциях с плавающей точкой, переполнение при операциях с целыми числами, нарушения защиты, неопределяемый код операции, переполнение стека, попытка запустить несуществующее устройство ввода-вывода, попытка вызвать слово из ячейки с нечетным адресом и деление на 0.

Прерывания

Прерывания — это изменения в потоке управления, вызванные не самой программой, а чем-либо другим и обычно связанные с процессом ввода-вывода. Например, программа может приказать диску начать передачу информации и заставить диск произвести прерывание, как только передача данных завершится. Как и ловушка, прерывание останавливает работу программы и передает управление программе обработки прерываний, которая выполняет какое-то определенное действие. После завершения этого действия программа обработки прерываний передает управление прерванной программе. Она должна заново начать прерванный процесс в том же самом состоянии, в котором она находилась, когда произошло прерывание. Это значит, что прежнее состояние всех внутренних регистров (то есть состояние, которое было до прерывания) должно быть восстановлено.

Различие между ловушками и прерываниями в следующем: ловушки синхронны с программой, а прерывания асинхронны. Если программа перезапускается много раз с одним и тем же материалом на входе, ловушки каждый раз будут происходить в одном и том же месте, а прерывания могут меняться в зависимости от того, в какой момент человек нажимает возврат каретки. Причина воспроизводимости ловушек и невоспроизводимости прерываний состоит в том, что первые вызываются непосредственно самой программой, а прерывания вызываются программой косвенно.

Чтобы понять, как происходят прерывания, рассмотрим обычный пример: компьютеру нужно вывести на терминал строку символов. Программное обеспечение сначала собирает в буфер все символы, которые нужно вывести на экран, инициализирует глобальную переменную ptr, которая должна указывать на начало буфера, и устанавливает вторую глобальную переменную count, которая равна числу символов, выводимых на экран. Затем программное обеспечение проверяет, готов ли терминал, и если готов, то выводит на экран первый символ (например, используя регистры, которые показаны на рис. 5.30). Начав процесс ввода-вывода, центральный процессор освобождается и может запустить другую программу или сделать что-либо еще.

Через некоторое время символ отображается на экране. Теперь может начаться прерывание. Ниже перечислены основные шаги (в упрощенной форме).

Действия аппаратного обеспечения:

1. Контроллер устройства устанавливает линию прерывания на системной шине.
2. Когда центральный процессор готов к обработке прерывания, он устанавливает символ подтверждения прерывания на шине.
3. Когда контроллер устройства узнает, что сигнал прерывания был подтвержден, он помещает небольшое целое число на информационные линии, чтобы «представиться» (то есть показать, что это за устройство). Это число называется вектором прерываний[1].
4. Центральный процессор удаляет вектор прерывания с шины и временно его сохраняет.
5. Центральный процессор помещает в стек счетчик команд и слово состояния программы.
6. Затем центральный процессор определяет местонахождение нового счетчика команд, используя вектор прерывания в качестве индекса в таблице в нижней части памяти. Если, например, размер счетчика команд составляет 4 байта, тогда вектор прерываний п соответствует адресу 4п. Новый счетчик команд указывает на начало программы обслуживания прерываний для устройства, вызвавшего прерывание. Часто помимо этого загружается или изменяется слово состояния программы (например, чтобы блокировать дальнейшие прерывания).

[1] Автор не совсем прав: здесь речь должна идти о номере прерывания. Каждому типу прерывания соответствует свой номер. Термин «вектор прерываний» используется в случае, когда по номеру прерывания находится адрес программы обработки прерывания и этот адрес представляется не одним значением, а несколькими, то есть необходимо проинициализировать более одного регистра. Другими словами, адрес представляется не скалярной величиной, а многомерной, векторной. — *Примеч. научн. ред.*

Действия программного обеспечения:

1. Первое, что делает программа обработки прерываний, — сохраняет все нужные ей регистры таким образом, чтобы их можно было восстановить позднее. Их можно сохранить в стеке или в системной таблице.
2. Каждый вектор прерывания разделяется всеми устройствами данного типа, поэтому в данный момент еще не известно, какой терминал вызвал прерывание. Номер терминала можно узнать, считав значение какого-нибудь регистра.
3. Теперь можно считывать любую другую информацию о прерывании, например коды состояния.
4. Если происходит ошибка ввода-вывода, ее нужно обработать здесь.
5. Глобальные переменные ptr и count обновляются. Первая увеличивается на 1, чтобы показывать на следующий байт, а вторая уменьшается на 1, чтобы указать, что осталось вывести на 1 байт меньше. Если count все еще больше О, значит, еще не все символы выведены на экран. Тот символ, на который в данный момент указывает ptr, копируется в выходной буферный регистр.
6. В случае необходимости выдается специальный код, который сообщает устройству или контроллеру прерывания, что прерывание обработано.
7. Восстанавливаются все сохраненные регистры.
8. Выполнение команды RETURN FROM INTERRUPT (выход из прерывания): возвращение центрального процессора в то состояние, в котором он находился до прерывания. После этого компьютер продолжает работу с того места, в котором ее приостановил.

С прерываниями связано важное понятие **прозрачности.** Когда происходит прерывание, производятся какие-либо действия и запускаются какие-либо программы, но когда все закончено, компьютер должен вернуться точно в то же состояние, в котором он находился до прерывания. Программа обработки прерываний, обладающая таким свойством, называется прозрачной.

Если компьютер имеет только одно устройство ввода-вывода, тогда прерывания работают точно так, как мы только что описали. Однако большой компьютер может содержать много устройств ввода-вывода, причем несколько устройств могут работать одновременно, возможно, у разных пользователей. Существует некоторая вероятность, что во время работы программы обработки прерывания другое устройство ввода-вывода тоже захочет произвести свое прерывание.

Здесь существует два подхода. Первый подход — для всех программ обработки прерываний в первую очередь (даже до сохранения регистров) предотвратить последующие прерывания. При этом прерывания будут совершаться в строгой последовательности, но это может привести к проблемам с устройствами, которые не могут долго простаивать. Например, на коммуникационной линии со скоростью передачи 9600 битов в секунду символы поступают каждые 1042 микросекунды. Если первый символ еще не обработан, когда поступает второй, то данные могут потеряться.

Если компьютер имеет подобные устройства ввода-вывода, то лучше всего приписать каждому устройству определенный приоритет, высокий для более критич-

ных и низкий для менее критичных устройств. Центральный процессор тоже должен иметь приоритеты, которые определяются по одному из полей слова состояния программы. Если устройство с приоритетом п вызывает прерывание, программа обработки прерывания тоже должна работать с приоритетом п.

Если работает программа обработки прерываний с приоритетом п, любая попытка другого устройства с более низким приоритетом будет игнорироваться, пока программа обработки прерываний не завершится и пока центральный процессор не возвратится к выполнению программы более низкого приоритета. С другой стороны, прерывания, поступающие от устройств с более высоким приоритетом, должны происходить без задержек.

Поскольку сами программы обработки прерываний подвержены прерыванию, лучший способ строгого управления — сделать так, чтобы все прерывания были прозрачными. Рассмотрим простой пример с несколькими прерываниями. Компьютер имеет три устройства ввода-вывода: принтер, диск и линию RS232 с приоритетами 2,4 и 5 соответственно. Изначально (t=0; t — время) работает пользовательская программа. Вдруг при t= 10 принтер совершает прерывание. Запускается программа обработки прерывания принтера, как показано на рис. 5.30.

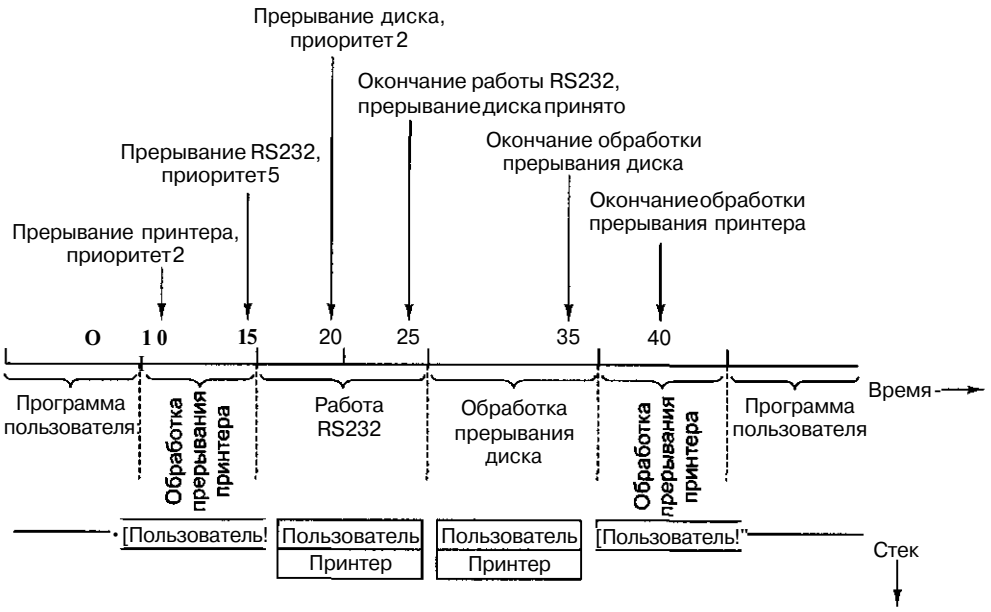


Рис. 5.30. Пример с несколькими прерываниями. Последовательность действий

При t=15 линия RS232 порождает сигнал прерывания. Так как линия RS232 имеет более высокий приоритет (5), чем принтер (2), прерывание происходит. Состояние машины, при котором работает программа обработки прерывания принтера, сохраняется в стеке, и начинается выполнение программы обработки прерывания RS232.

Немного позже, при t=20, диск завершает свою работу. Однако его приоритет (4) ниже, чем приоритет работающей в данный момент программы обработки прерыва-

ний (5), поэтому центральный процессор не подтверждает прием сигнала прерывания, и диск вынужден простаивать. При t=25 заканчивается программа RS232, и машина возвращается в то состояние, в котором она находилась до прерывания RS232, то есть в то состояние, когда работала программа обработки прерывания принтера с приоритетом 2. Как только центральный процессор переключается на приоритет 2, еще до того как будет выполнена первая команда, диск с приоритетом 4 совершает прерывание и запускается программа обработки прерываний диска. После ее завершения снова продолжается программа обработки прерываний принтера. Наконец, при t=40 все программы обработки прерываний завершаются и выполнение пользовательской программы начинается с того места, на котором она прервалась.

Со времен процессора 8088 все процессоры Intel имеют два уровня (приоритета) прерываний: маскируемые и немаскируемые прерывания. Немаскируемые прерывания обычно используются только для сообщения об очень серьезных ситуациях, например об ошибках четности в памяти. Все устройства ввода-вывода используют одно маскируемое прерывание.

Когда устройство ввода-вывода вызывает прерывание, центральный процессор использует вектор прерывания при индексировании таблицы из 256 элементов, чтобы найти адрес программы обработки прерываний. Элементы таблицы представляют собой 8-байтные дескрипторы сегмента. Таблица может начинаться в любом месте памяти. Глобальный регистр указывает на ее начало.

При наличии только одного уровня прерываний центральный процессор не может сделать так, чтобы устройство с более высоким приоритетом прерывало работу программы обработки прерываний с более низким приоритетом и чтобы устройство с более низким приоритетом не смогло прерывать выполнение программы обработки прерываний с более высоким приоритетом. Для решения этой проблемы центральные процессоры Intel обычно используют внешний контроллер прерываний (например, 8259A). При первом прерывании (например, с приоритетом п) работа процессора приостанавливается. Если после этого происходит еще одно прерывание с более высоким приоритетом, контроллер прерывания вызывает прерывание во второй раз. Если второе прерывание обладает более низким приоритетом, оно не реализуется до окончания первого. Чтобы эта система работала, контроллер прерывания должен каким-либо образом узнавать о завершении текущей программы обработки прерываний. Поэтому когда текущее прерывание полностью обработано, центральный процессор должен посылать специальную команду контроллеру прерываний.

Ханойская башня

Теперь, когда мы уже изучили уровень команд трех машин, нам нужно все обобщить. Давайте подробно рассмотрим тот же пример программы («Ханойская башня») для всех трех машин. В листинге 5.6 приведена версия этой программы на языке Java. В следующих разделах для решения этой задачи мы предложим программы на ассемблере.

Однако вместо того, чтобы давать трансляцию версии на языке Java, для машин Pentium II и UltraSPARC II мы представим трансляцию версии на языке C,

чтобы избежать проблем с вводом-выводом Java. Единственное различие — это замена оператора Java printf на стандартный оператор языка С

```
printfC"Переместить диск с %6 на %d\r\", i,j)
```

Синтаксис строки printf не важен (строка печатается буквально за исключением *d — это означает, что следующее целое число будет дано в десятичной системе счисления). Здесь важно только то, что процедура вызывается с тремя параметрами: форматирующей строкой и двумя целыми числами.

Мы использовали язык С для Pentium II и UltraSPARC II, поскольку библиотека ввода-вывода Java не доступна для этих машин, а библиотека С доступна. Для JVM мы будем использовать язык Java. Разница минимальна: всего один оператор вывода строки на экран.

Решение задачи «Ханойская башня» на ассемблере Pentium II

В листинге 5.7 приведен возможный вариант трансляции программы на языке С для компьютера Pentium П. Регистр EBP используется в качестве указателя фрейма. Первые два слова применяются для установления связи, поэтому первый параметр п (или N, поскольку регистр для макроассемблера не важен) находится в ячейке EBP+8, а за ним следуют параметры i и j в ячейках EBP+12 и EBP+16 соответственно. Локальная переменная к находится в EBP+20.

Листинг 5.7. Решение задачи «Ханойская башня» для машины Pentium II

```
             .586              ;компилируется для Pentium
.MODEL FLAT
PUBLIC _towers                 ;экспорт 'towers'
EXTERN _printf:NEAR            ;импорт printf
.CODE
_towers:   PUSH EBP            сохраняет EBP (указатель фрейма)
              MOV EBP. ESP     [устанавливает новый указатель фрейма над ESP
              CMP[EBP+8].1     ;if(n==1)
              JNE LI           ;переход, если п?1
              MOV EAX. [EBP+16] ;printf("...". i. j);
              PUSH EAX         ;сохранение параметров i. j и формата
              MOV EAX. [EBP+12] ;строка помещается в стек
              PUSH EAX         ;в обратном порядке. Таково требование языка С
              PUSH OFFSET FLAT:format : OFFSET FLAT - это адрес формата
              CALL _printf     ;вызов процедуры printf
              ADD ESP. 12      :удаление параметров из стека
              JMP Done         ;завершение
              MOV EAX.6        ;начало вычисления k=6-i-j
              SUB EAX. [EBP+12] :EAX=6-i
              SUB EAX. [EBP+16] ;EAX-6-i-j
              MOV [EBP+20], EAX ;k=EAX
              PUSH EAX         ;начало процедуры towers(n-l, п. к)
              MOV EAX. [EBP+12] ;EAX=i
              PUSH EAX         ;помещает в стек i
              MOV EAX. [EBP+8]  ;EAX=n
              DEC EAX;EAX=n-l
              PUSH EAX         ;помещает в стек n-1
              CALL _towers     ;вызов процедуры towers(n-1. i, 6-i-j)
```

```
            ADD ESP. 12        :удаление параметров из стека
            MOV EAX. [EBP+16] тачало процедуры towers (1, i, j)
            PUSH EAX           :помещает в стек j
            MOV EAX, [EBP+12] :EAX=i
            PUSH EAX           :помещает в стек i
            PUSH 1             шомещает в стек 1
            CALL _towers       ;вызывает процедуру towersCl, t, j)
            ADD ESP. 12        :удаляет параметры из стека
            MOV EAX. [EBP+12] .•начало процедуры towers(n-l. 6-i-j. i)
            PUSH EAX           .•помещает в стек i
            MOV EAX, [EBP+20] :EAX=k
            PUSH EAX           :помещает в стек к
            MOV EAX. [EBP+8]  :EAX = п
            DEC EAX:ПАХ-n-l
            PUSH EAX           ;помещает в стек п-1
            CALL _towers       :вызов процедуры towersCn-1, 6-i-j. i)
            ADD ESP. 12        :корректировка указателя  стека
Done:       LEAVE              ;под готовка к выходу
            RET 0              .•возврат к вызывающей программе
.DATA
format    DB "Переместить диск с %d на £d\n" [форматирующая строка
END
```

Процедура начинается с создания нового фрейма в конце старого. Для этого значение регистра ESP копируется в указатель фрейма EBP. Затем п сравнивается с 1, и если п>1, то совершается переход к оператору el se. Тогда программа then помещает в стек три значения: адрес форматирующей строки, i и j, и вызывает саму себя.

Параметры помещаются в стек в обратном порядке, поскольку это требуется для программ на языке С. Необходимо поместить указатель на форматирующую строку в вершину стека. Процедура printf имеет переменное число параметров, и если параметры будут помещаться в стек в прямом порядке, то процедура не сможет узнать, в каком месте стека находится форматирующая строка.

После вызова процедуры к регистру ESP прибавляется 12, чтобы удалить параметры из стека. На самом деле они не удаляются из памяти, но корректировка (изменение) регистра ESP делает их недоступными через обычные операции со стеком.

Выполнение части else начинается с L1. Здесь сначала вычисляется выражение 6-i-j, и полученное значение сохраняется в переменной к. Сохранение значения в переменной к избавляет от необходимости вычислять это во второй раз.

Затем процедура вызывает сама себя три раза, каждый раз с новыми параметрами. После каждого вызова стек освобождается.

Рекурсивные процедуры иногда приводят людей в замешательство. Но на самом деле они совсем несложные. Просто параметры помещаются в стек, и вызывается процедура.

Решение задачи «Ханойская башня» на ассемблере UltraSPARC II

А теперь рассмотрим то же самое для UltraSPARC П. Программа приведена в листинге 5.8. Поскольку программа для UltraSPARC II совершенно нечитаема даже после длительных тренировок, мы решили определить несколько символов, чтобы

прояснить дело. Чтобы такая программа работала, ее перед ассемблированием нужно пропустить через программу под названием cpp (препроцессор C). Здесь мы используем строчные буквы, поскольку ассемблер Pentium II требует этого (это на тот случай, если читатели захотят напечатать и запустить эту программу).

Листинг 5.8. Решение задачи «Ханойская башня» для UltraSPARC II

```
#define N £iO        /* N - это входной параметр 0 */
#define  M i l       /* I - это входной параметр 1 */
#define J £i2        /* J - это входной параметр 2 */
#define K «10        /* K - это локальная переменная 0 */
#define ParamO SoO   /* ParamO - это выходной параметр 0 */
#define Paraml Xol   /* Paraml - это выходной параметр 1 */
Idefine Param2 Яо2   /* Param2 - это выходной параметр 2 */
#define Scratch XII  /*примеч.: cpp использует запись комментариев как в языке C*/
          .proc 04
          .global towers
towers:   save *sp.-112. *sp
          cmp N, 1                      ! if(n= 1)
          bne Else                      ! if (n != 1) goto Else
          sethi £hi(format). ParamO     ! printf("Переместить диск с %d на %d\n". i, j)
          or ParamO. Xlo(format). ParamO ! ParamO = адрес форматирующей строки
          mov I. Paraml                 ! Paraml = i
          call printf                   ! вызов printf ДО установки параметра 2 (j)
          mov J. Param2                 ! пустая операция для установки параметра 2
          b Done                        ! завершение
          пор                           ! вставляет пустую операцию

Else:     mov 6. K                      ! начало вычисления к = 6 -i-j
          sub K.J.K                     ! k-6-j
          sub K.I,K                     !k=6-i-j

          add N, -1. Scratch   ! начало процедуры towers(n-l. i. k)
          mov Scratch, ParamO  ! Scratch = n-1
          mov I. Paraml        ! параметр 1 - i
          call towers          ! вызов процедуры towers ДО установки параметра 2 (k)
          mov K, Param2        ! пустая операция после вызова процедуры для установки
                                 параметра2

          mov 1, ParamO        ! начало процедуры towersd. i. j)
          mov I. Paraml        ! параметр1=1
          call towers          ! вызов процедуры towers ДО установки параметра 2 (j)
          mov J. Param2        ! параметр 2 = j

          mov Scratch. ParamO  ! начало процедуры towers(n-l. k. j)
          mov K. Paraml        ! параметр 1 « к
          call towers          ! вызов процедуры towers ДО установки параметра 2 (j)
          mov J. Param2        ! параметр 2 = j
Done:     ret                  ! выход из процедуры
          restore              ! вставка пустой команды после  ret для восстановления окон
format:   .asciz "Переместить диск с %d на %й\п"
```

По алгоритму версия UltraSPARC идентична версии Pentium II. В обоих случаях сначала проверяется n, и если n>1, то совершается переход к el se. Основные

сложности версии UltraSPARC II связаны с некоторыми свойствами архитектуры команд.

Сначала UlraSPARC II должен передать адрес форматирующей строки в printf, но машина не может просто переместить адрес в регистр, который содержит выходящий параметр, поскольку нельзя поместить 32-битную константу в регистр за одну команду. Для этого требуется выполнить две команды: SETHI и OR.

После вызова не нужно делать подстройку стека, поскольку регистровое окно корректируется командой RESTORE в конце процедуры. Возможность помещать выходящие параметры в регистры и не обращаться к памяти дает огромный выигрыш в производительности.

А теперь рассмотрим команду NOP, которая следует за Done. Это пустая операция. Эта команда всегда будет выполняться, даже если она следует за командой условного перехода. Сложность состоит в том, что процессор UltraSPARC II сильно конвейеризирован, и к тому моменту, когда аппаратное обеспечение обнаруживает команду перехода, следующая команда уже практически закончена. Добро пожаловать в прекрасный мир программирования RISC!

Эта особенность распространяется и на вызовы процедур. Рассмотрим первый вызов процедуры towers в части else. Процедура помещает n-1 в %oO, а i — в %ol, но совершает вызов процедуры towers до того, как поместит последний параметр в нужное место. На компьютере Pentium II вы сначала передаете параметры, а затем вызываете процедуру. А здесь вы сначала передаете часть параметров, затем вызываете процедуру, и только после этого передаете последний параметр. К тому моменту, когда машина осознает, что она имеет дело с командой CALL, следующую команду все равно приходится выполнять (из-за конвейеризации системы). А почему бы в этом случае не использовать пустую операцию, чтобы передать последний параметр? Даже если самая первая команда вызванной процедуры использует этот параметр, он уже будет на своем месте.

Наконец, рассмотрим часть команды Done. Здесь после команды RET тоже вставляется пустая операция. Эта пустая операция используется для команды RESTORE, которая увеличивает на 1 значение CWP, чтобы вернуть регистровое окно в прежнее состояние.

Решение задачи «Ханойская башня» на ассемблере для JVM

Соответствующая программа дана в листинге 5.9. Решение довольно простое, за исключением процесса ввода-вывода. Эта программа была порождена компилятором Java, переделана в символический язык ассемблера и обработана определенным образом для удобочитаемости. Компилятор JVM хранит три параметра n, i и j в локальных переменных 0, 1 и 2 соответственно. Локальная переменная k хранится в локальной переменной 3. Ко всем четырем локальным переменным можно обратиться с помощью 1-байтного кода операции, например ILOAD0. В результате двоичная версия этой программы в JVM получается очень короткой (всего 67 байтов).

Листинг 5.9. Решение задачи «Ханойская башня» для JVM

```
ILOADJ)          // лок. переменная 0 = п; помещает в стек п
ICONST_1         // помещает в стек 1
IFJCMPNE L I     //if(n!-l)gotoLI
GETSTATIC #13    // п – 1: эта команда обрабатывает выражение pnntln
NEW #7           // размещает буфер для строки, которую нужно создать
DUP              // дублирует указатель на буфер
LDC #2           // помещает в стек указатель на цепочку "перенести диск с"
INVOKESPECIAL #10 // копирует эту цепочку в буфер
ILOAD_1          // помещает в стек i
INVOKEVIRTUAL #11 // превращает i в цепочку и присоединяет к новому буферу
LDC #1           // помещает в стек указатель на цепочку "на"
INVOKEVIRTUAL #12 // присоединяет эту цепочку к буферу
ILOAD_2          // помещает в стек j
INVOKEVIRTUAL #11 // превращает j в цепочку и присоединяет ее к буферу
INVOKEVIRTUAL #15 // преобразование строки
INVOKEVIRTUAL #14 // вызов println
RETURN           // выход из процедуры towers
LI: BIPUSH6      // Часть Else: вычисление k = 6-i-j
ILOAD_1          // лок. переменная 1 = i; помещает в стек i
ISUB             // вершина стека = 6-i
IL0AD_2          // лок. переменная 2= j; помещает в стек j
ISUB             // вершина стека = 6-i-j
ISTORE_3         // лок. перем. 3 - k = 6-i-j: стек сейчас пуст
IL0ADJ)          // начало работы процедуры towers(n-l.i. k);помещает в стек п
ICONST_1         // помещает в стек 1
ISUB             // вершина стека = п-1
ILOAD_1          // помещает в стек i
IL0AD_3          // помещает в стек к
INVOKESTATIC #16 // вызывает процедуру towers(п-1. i. к)
ICONST_1         // начинается работа процедуры towersQ, i, j) помещает в стек 1
ILOAD_1          // помещает в стек i
IL0AD_2          // помещает в стек j
INVOKESTATIC #16 // вызов процедуры towersd. i. j)
ILOAD_0 "        // начало работы процедуры towers(n-l. k. j); помещает в стек п
ICONSTJ.         // помещает в стек 1
ISUB             // вершина стека = п-1
IL0AD_3          // помещает в стек к
IL0AD_2          // помещает в стек j
INVOKESTATIC #16 // вызов процедуры towers(n-l, k. j)
RETURN           // выход из процедуры towers
```

Сначала программа помещает в стек параметр п и константу 1, а затем сравнивает их с помощью команды IFICMPNE. Эта команда обратна команде IF_ICMPEQ, которая используется в машине IJVM. Она выталкивает из стека два операнда и совершает переход, если они различны.

Если они одинаковы, выполнение программы продолжается последовательно. Следующие 13 команд определяют буфер строки и строят в нем цепочку, которая затем передается в println для вывода на экран. После завершения печати совершается выход из процедуры.

Если говорить кратко, эти 13 команд размещают буфер строки в «кучу» и заполняют его. Команда GETSTATIC индексирует набор констант, чтобы получить слово 13, которое содержит указатель на дескриптор для буфера строки. Команда NEW использует этот дескриптор для размещения буфера строки в «куче». Следую-

щие 11 команд связывают две цепочки и два целых числа в одну цепочку в этом буфере и передают ее в println для вывода на экран.

Если n не равно 1, управление передается к L1. Переменная k вычисляется простым путем с использованием арифметических операций над числами в стеке. Затем совершаются три вызова один за другим.

Машина IJVM содержит ряд команд для вызова процедур. В данном случае компилятор использовал три разные команды. Все они содержат 2-байтный операнд, который индексирует набор констант, чтобы найти указатель на дескриптор, сообщающий все о вызываемой процедуре. Константы #10, #11 и т. д. — индексы в наборе констант.

Наличие нескольких типов команд для вызова процедур связано с оптимизацией. Команда INVOKESTATIC используется для вызова статических процедур, например towers. Команда INVOKESPECIAL применяется для вызова процедур инициализации, нестандартных процедур и процедур надкласса текущего класса. Наконец, команда INVOKEVIRTUAL используется для внутренних (библиотечных) вызовов.

Intel IA-64

Со временем увеличивать скорость работы IA-32 становилось все сложнее и сложнее. Единственным возможным решением этой проблемы стала разработка совершенно новой архитектуры команд. Новая архитектура, которая разрабатывалась совместно компаниями Intel и Hewlett Packard, получила название **IA-64.** Это полностью 64-битная машина от начала до конца. Появление полной серии процессоров, в которых реализуется эта архитектура, ожидается в ближайшие годы. Самым первым процессором этого типа был процессор **Merced** с высокой производительностью, хотя в будущем наверняка появится полный спектр процессоров разного уровня.

Поскольку все, что делает компания Intel, очень важно для компьютерной промышленности, мы подробно рассмотрим архитектуру IA-64. Однако ключевые идеи этой архитектуры уже очень хорошо известны многим исследователям, поэтому они могут отражаться в других разработках. Вообще, некоторые из них уже реализованы в разных формах в экспериментальных системах. В следующих разделах мы изложим, с какой проблемой столкнулась компания Intel, каким образом архитектура IA-64 помогла справиться с этой проблемой и как работают ключевые идеи этого проекта.

Проблема с Pentium II

Основная проблема заключалась в том, что IA-32 — это старая архитектура команд с совершенно не подходящими для современной техники свойствами. Это архитектура CISC с командами разной длины и огромным количеством различных форматов, которые трудно декодировать быстро и на лету. Современная техника лучше всего работает с архитектурами команд RISC с командами одной

длины и с кодом операции фиксированной длины, который легко декодировать. Команды IA-32 можно разбить на микрооперации типа RISC во время выполнения программы, но для этого требуется дополнительное аппаратное обеспечение (пространство на микросхеме), что занимает время и усложняет разработку. Это первый недостаток.

IA-32 — это архитектура, которая ориентирована на двухадресные команды. В настоящее время популярны архитектуры команд типа загрузка/сохранение, где обращение к памяти совершается только в тех случаях, когда нужно поместить операнды в регистры, а все вычисления выполняются с использованием трехадресных регистровых команд. Поскольку скорость работы процессора растет гораздо быстрее, чем скорость работы памяти, положение дел с IA-32 со временем все больше ухудшается. Это второй недостаток.

Архитектура IA-32 содержит небольшой и нерегулярный набор регистров. Из-за столь малого числа регистров общего назначения (четыре или шесть, в зависимости от того, как считать ESI и EDI) постоянно нужно записывать в память промежуточные результаты, и поэтому приходится делать дополнительные обращения к памяти, даже когда они по логике вещей не нужны. Это третий недостаток.

Из-за недостаточного числа регистров возникает множество ситуаций зависимостей, особенно WAR-зависимостей, поскольку промежуточные результаты нужно куда-то поместить, а дополнительных регистров нет. При недостатке регистров требуется переименование регистров в скрытые регистры. Во избежание слишком частых промахов кэш-памяти команды приходится выполнять не по порядку. Однако семантика архитектуры IA-32 определяет точные прерывания, поэтому команды, выполняемые не по порядку, должны записывать результаты в выходные регистры в строгом порядке. Для всего этого требуется очень сложное аппаратное обеспечение. Это четвертый недостаток.

Чтобы скорость работы была высокой, нужна сильно конвейеризированная система (12 стадий). Однако это значит, что для выполнения команды потребуется 11 циклов. Следовательно, становится существенным точное предсказание переходов, поскольку в конвейер должны попадать только нужные команды. Но даже при низком проценте неправильных предсказаний существенно снижается производительность. Это пятый недостаток.

Чтобы избежать проблем с неправильным прогнозированием переходов, процессору приходится осуществлять спекулятивное выполнение команд со всеми вытекающими отсюда последствиями. Это шестой недостаток.

Мы не будем перечислять недостатки дальше, поскольку уже сейчас ясно, что за ними кроется реальная проблема. И мы еще не упомянули, что 32-битные адреса архитектуры IA-32 ограничивают размер отдельных программ до 4 Гбайт, а это требование очень сложно выполнять на дорогостоящих серверах с высокой производительностью.

Ситуации с IA-32 можно сравнить с положением в небесной механике как раз перед появлением Коперника. В те времена в астрономии доминировала теория, что Земля является центром Вселенной и неподвижна, а планеты движутся вокруг нее. Однако новые наблюдения показывали все больше и больше несоответствий этой теории действительности, и в конце концов теория полностью разрушилась.

Компания Intel находится приблизительно в таком же положении. Огромное количество транзисторов в процессоре Pentium II предназначено для переделывания команд CISC в команды RISC, разрешения конфликтов, прогнозирования переходов, исправления неправильных предсказаний и решения многих других задач подобного рода, оставляя лишь незначительное число транзисторов на долю реальной работы, которая нужна пользователю. Поэтому компания Intel пришла к следующему выводу: нужно выбросить IA-32 и начать все заново (IA-64).

Модель IA-64: открытое параллельное выполнение команд

Первой реализацией архитектуры IA-32 был 64-битный процессор RISC (один из примеров этого процессора — UltraSPARC II). Поскольку архитектура IA-64 была разработана совместно с компанией Hewlett Packard, в ее основу, несомненно, легла архитектура PA-RISC. Merced — это двухрежимный процессор, который может выполнять и программы IA-32, и программы IA-64, но мы будем говорить только об архитектуре IA-64.

Архитектура IA-64 — это архитектура типа загрузка/сохранение с 64-битными адресами и 64-битными регистрами. Здесь имеется 64 регистра общего назначения, доступных для программ IA-64 (и дополнительные регистры для программ IA-32). Все команды имеют фиксированный формат: код операции, два 6-битных поля для указания входных регистров, одно 6-битное поле для указания выходного регистра и еще одно 6-битное поле, которое мы обсудим позже. Большинство команд берут два регистровых операнда, выполняют над ними какую-нибудь операцию и помещают результат в выходной регистр. Для параллельного выполнения различных операций существует много функциональных блоков. Как видим, пока ничего необычного в этой архитектуре нет. Большинство RISC-процессоров имеют сходную архитектуру.

А необычной здесь является идея о **пучке** связанных команд. Команды поступают группами по три штуки (рис. 5.31). Каждая такая группа называется пучком. Каждый 128-битный пучок содержит три 40-битные команды фиксированного формата и 8-битный шаблон. Пучки могут быть связаны вместе (при этом используется бит конца пучка), поэтому в одном пучке может присутствовать более трех команд. Формат содержит информацию о том, какие команды могут выполняться параллельно. При такой системе и при наличии большого числа регистров компилятор может выделять блоки команд и сообщать процессору, что эти команды можно выполнять параллельно. Таким образом, компилятор должен переупорядочивать команды, проверять, нет ли взаимозависимостей, проверять доступность функциональных блоков и т. д. вместо аппаратного обеспечения. Основная идея состоит в том, что работа упорядочивания и распределения команд RISC передается от аппаратного обеспечения компилятору. Именно поэтому эта технология называется **EPIC (Explicitly Parallel Instruction Computing — технология параллельной обработки команд с явным параллелизмом).**

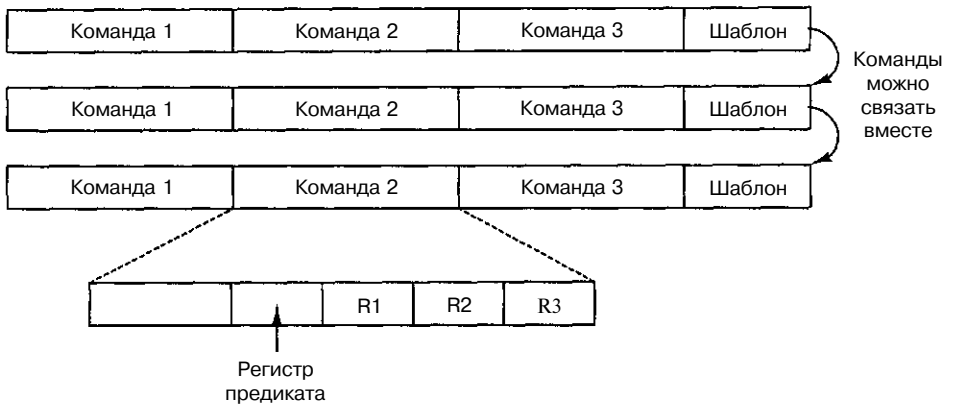


Рис. 5.31. Архитектура IA-64 основана на пучках из трех команд

Есть несколько причин, по которым следует совершать упорядочивание команд во время компиляции. Во-первых, поскольку теперь всю работу выполняет компилятор, аппаратное обеспечение можно сильно упростить, используя миллионы транзисторов для других полезных функций (например, можно увеличить кэш-память первого уровня). Во-вторых, для любой программы распределение должно производиться только один раз (во время компиляции). В-третьих, поскольку компилятор теперь выполняет всю работу, у поставщика программного обеспечения появится возможность использовать компилятор, который часами оптимизирует свою программу.

Идея пучков команд может быть использована при создании целого семейства процессоров. Процессоры с низкой производительностью могут запускать по одному пучку за цикл. Перед тем как выпустить новый пучок, центральный процессор должен дождаться завершения всех команд. Процессоры с высокой производительностью способны запускать несколько пучков за один цикл (по аналогии с современными суперскалярными процессорами). Кроме того, процессор с высокой производительностью может начать запускать команды из нового пучка еще до того, как закончится выполнение всех команд предыдущего пучка. Естественно, нужно все время проверять, доступны ли нужные регистры и функциональные блоки, но зато вовсе не надо проверять, не будут ли конфликтовать разные команды одного пучка, поскольку компилятор гарантирует, что этого не произойдет.

Предикация

Еще одна особенность архитектуры IA-64 — новый способ обработки условных переходов. Если бы была возможность избавиться от большинства из них, центральный процессор стал бы гораздо проще и работал бы гораздо быстрее. На первый взгляд может показаться, что устранить условные переходы невозможно, поскольку в программах всегда полно операторов if. Однако в архитектуре IA-64 используется специальная технология, названная **предикацией**[1], которая позволяет сильно сократить их число [10,63]. Ниже мы кратко опишем эту технологию.

[1] От англ. predicate — утверждение, предикат. — *Примеч. научн.ред.*

В современных машинах все команды являются безусловными в том смысле, что когда центральный процессор натыкается на команду, он просто ее выполняет. Здесь никогда не решается вопрос: «Выполнять или не выполнять?» Напротив, в архитектуре с предикацией команды содержат условия, которые сообщают, в каком случае нужно выполнять команду, а в каком — нет. Именно этот сдвиг от безусловных команд к командам с предикацией позволяет избавиться от многих условных переходов. Вместо того чтобы выбирать ту или иную последовательность безусловных команд, все команды сливаются в одну последовательность команд с предикацией, используя разные предикаты для различных команд.

Чтобы понять, как работает предикация, рассмотрим простой пример (листинги 5.10-5.12), в котором показано **условное выполнение** команд (условное выполнение — предтеча предикации). В листинге 5.10 мы видим оператор if. В листинге 5.11 мы видим трансляцию этого оператора в три команды: команду сравнения, команду условного перехода и команду перемещения.

Листинг 5.10. Оператор if

```
if (R1=0)
      R2=R3:
```

Листинг 5.11. Код на ассемблере для листинга 5.10

```
      CMP R1.0
      BNE L1
      MOV R2.R3
L1:
```

Листинг 5.12. Условная команда

```
CMOVZ R2.R3.R1
```

В листинге 5.12 мы избавились от условного перехода, используя новую команду CMOVZ, которая является условным перемещением. Эта команда проверяет, равен ли третий регистр R1 нулю. Если да, то команда копирует R3 в R2. Если нет, то команда не выполняет никаких действий.

Если у нас есть команда, которая может копировать данные, когда какой-либо регистр равен нулю, значит, у нас может быть и такая команда, которая копирует данные, если какой-нибудь регистр не равен нулю. Пусть это будет команда CMOVN. При наличии обеих команд мы уже на пути к полному условному выполнению. Представим оператор if с несколькими присваиваниями в части then и несколькими присваиваниями в части else. Весь этот кусок программы можно транслировать в код, который будет устанавливать какой-нибудь регистр на 0, если условие не выполнено, и на какое-нибудь другое значение, если условие выполнено. Следуя установке регистров, присваивания в части then можно скомпилировать в последовательность команд CMOVN, а присваивания в части else — в последовательность команд CMOVZ.

Все эти команды, регистровая установка, CMOVN и CMOVZ формируют единый основной блок без условных переходов. Команды можно даже переупорядочить при компиляции или во время выполнения программы. Единственное требование при этом состоит в том, чтобы условие было известно к тому моменту, когда условные команды уже нужно помещать в выходные регистры (то есть где-то

в конце конвейера). Простой пример куска программы с then и el se приведен в листингах 5.13-5.15.

Листинг 5.13. Оператор if

```
if(RI=0) {
R2=R3;
R4=R5;
} else {
R6=R7:
R8=R9:
}
```

Листинг 5.14. Код на ассемблере для листинга 5.13

```
       CMP R1.0
       BNE L1
       MOV R2.R3
       MOV R4.R5
       BR L2
LI: MOV R6.R7
       MOV R8.R9
L2:
```

Листинг 5.15. Условное выполнение

```
CMOVZ R2.R3.R1
CMOVZ R4.R5.R1
CMOVN R6.R7.R1
CMOVN R8.R9.R1
```

Мы показали только очень простые условные команды (взятые из Pentium II), но в архитектуре IA-64 все команды предикатны. Это значит, что выполнение каждой команды можно сделать условным. Дополнительное 6-битное поле, о котором мы упомянули выше, выбирает один из 64 1-битных предикатных регистров. Следовательно, оператор i f может быть скомпилирован в код, который устанавливает один из предикатных регистров на 1, если условие истинно, и на 0, если условие ложно. Одновременно с этим данное поле автоматически устанавливает другой предикатный регистр на обратное значение. При использовании предикации машинные команды, которые формируют операторы then и el se, будут сливаться в единый поток команд, первый из них — с использованием предиката, а второй — с использованием его обратного значения.

Листинги 5.16-5.18 показывают, как можно использовать предикацию для устранения переходов. Команда CMPEQ сравнивает два регистра и устанавливает предикатный регистр P4 на 1, если они равны, и на 0, если они не равны. Кроме того, команда устанавливает парный регистр, например P5, на обратное условие. Теперь команды частей i f и then можно поместить одну за другой, причем каждая из них связывается с каким-нибудь предикатным регистром (регистр указывается в угловых скобках). Сюда можно поместить любой код, при условии что каждая команда предсказывается правильно.

Листинг 5.16. Оператор if

```
if(RI==R2)
       R3=R4+R5:
Else
       R6=R4-R5
```

Листинг 5.17. Код на ассемблере для листинга 5.16

```
       CMP R1.R2
       BNE L1
       MOV R3.R4
       ADD R3.R5
       BR L2
LI: MOV R6.R4
       SUB R6.R5
L2:
```

Листинг 5.18. Выполнение с предикацией

```
CMPEQ R1.R2.P4
<P4> ADD R3.R4.R5
<P5> SUB R6.R4.R5
```

В архитектуре IA-64 эта идея доведена до логического конца: здесь с предикатными регистрами связаны и команды сравнения, и арифметические команды, а также некоторые другие команды, выполнение которых зависит от какого-либо предикатного регистра. Команды с предикацией могут помещаться в конвейер последовательно без каких-либо проблем и простаиваний. Поэтому они очень полезны.

В архитектуре IA-64 предикация происходит следующим образом. Каждая команда действительно выполняется, и в самом конце конвейера, когда уже нужно сохранять результат в выходной регистр, производится проверка, истинно ли предсказание. Если да, то результаты просто записываются в выходной регистр. Если предсказание ложно, то записи в выходной регистр не происходит. Подробно о предикации вы можете прочитать в книге [34].

Спекулятивная загрузка

Еще одна особенность IA-64 — наличие спекулятивной загрузки. Если команда LOAD спекулятивна и оказывается ложной, то вместо того чтобы вызвать исключение (exception), она просто останавливается и сообщает, что регистр, с которым она связана, недействителен. Если этот регистр будет использоваться позднее, то произойдет исключение (exception).

Компилятор должен перемещать команды LOAD в более ранние позиции относительно других команд с тем, чтобы они выполнялись еще до того, как они понадобятся. Поскольку выполнение этих команд начинается раньше, чем нужно, они могут завершиться еще до того, как потребуются результаты. Компилятор вставляет команду CHECK в том месте, где ему нужно получить значение определенного регистра. Если значение там уже есть, команда CHECK работает как NOP, и выполнение программы сразу продолжается дальше. Если значения в регистре еще нет, следующая команда должна простаивать.

Итак, машина с архитектурой IA-64 имеет несколько источников повышения скорости. Во-первых, это современная конвейеризированная трехадресная машина RISC типа загрузка/сохранение. Во-вторых, компилятор определяет, какие команды могут выполняться одновременно, не вступая в конфликт, и группирует эти команды в пучки. Таким образом, процессор может просто распределять пучок, не совершая никаких проверок. В-третьих, предикация позволяет сливать команды обоих переходов от оператора if, устраняя при этом условный переход, а также и само прогнозирование этого перехода. Наконец, спекулятивная загрузка позволя-

ет вызывать операнды заранее, и даже если позднее окажется, что эти операнды не нужны, ничего страшного не произойдет.

Проверка в реальных условиях

Если все эти нововведения функционируют, процессор Merced действительно будет чрезвычайно мощным. Однако здесь нужно высказать несколько предостережений. Во-первых, такая продвинутая машина никогда не создавалась раньше. История показывает, что грандиозные планы часто не удается осуществить по самым разным причинам.

Во-вторых, придется составлять компиляторы для IA-64. А составление хорошего компилятора для IA-64 — дело очень сложное. Кроме того, многочисленные исследования в параллельном программировании, проведенные за последние 30 лет, оказались не очень успешными. Если в программе есть какой-то параллелизм или если компилятор не может извлечь его, все пучки команд в архитектуре IA-64 будут короткими, а от этого большого увеличения производительности не произойдет.

В-третьих, для реализации этой идеи должна существовать полностью 64-битная операционная система. Windows 95 и Windows 98 такими системами не являются и вряд ли когда-нибудь будут. Это значит, что всем придется перейти на Windows NT или UNIX, и такой переход не будет безболезненным. Спустя 10 лет с момента появления 32-битного процессора 386 система Windows 95 все еще содержала множество 16-битных команд и никогда не использовала аппаратное обеспечение с сегментацией, которое около 10 лет включается во все процессоры Intel (сегментацию мы будем обсуждать в главе 6). Сколько времени потребуется на то, чтобы операционные системы стали полностью 64-битными, никто не знает.

В-четвертых, многие будут судить об архитектуре IA-64 по тому, как на ней будут работать старые 16-битные игры MS DOS. Когда появился Pentium Pro, он не пользовался особенным успехом, поскольку старые 16-битные программы работали на нем с такой же скоростью, как и на обычной машине Pentium. Поскольку старые 16-битные (и 32-битные) программы не будут использовать новые особенности процессоров IA-64, никакие предикатные регистры им не помогут. Людям придется приобретать средства наращивания для своего программного обеспечения, подходящие для новых компиляторов типа IA-64. Многие из них будут очень дорогими.

Наконец, другие производители (включая Intel) могут выпустить альтернативные варианты, которые будут давать высокую производительность, используя при этом более привычные архитектуры RISC, возможно, с большим количеством условных команд. Более высокоскоростные версии Pentium II также будут серьезным соперником для IA-64. Вероятно, пройдет очень много лет, прежде чем IA-64 будет доминировать на компьютерном рынке, подобно архитектуре IA-32.

Краткое содержание главы

Для большинства людей уровень архитектуры команд — это «машинный язык». На этом уровне машина имеет память с байтовой организацией или с пословной

организацией, состоящую из нескольких десятков мегабайтов и содержащую такие команды, как MOVE, ADD и BEQ.

В большинстве современных компьютеров память организована в виде последовательности байтов, при этом 4 или 8 байтов группируются в слова. Обычно в машине имеется от 8 до 32 регистров, каждый из которых содержит одно слово.

Команды обычно имеют 1,2 или 3 операнда, обращение к которым происходит с помощью различных способов адресации: непосредственной, прямой, регистровой, индексной и т. д. Команды обычно могут перемещать данные, выполнять унарные и бинарные операции (в том числе арифметические и логические), совершать переходы, вызывать процедуры, осуществлять циклы, а иногда и выполнять некоторые операции ввода-вывода. Типичные команды перемещают слово из памяти в регистр или наоборот, складывают, вычитают, умножают или делят два регистра или регистр и слово из памяти, или сравнивают два значения в регистрах или памяти. Обычно в компьютере содержится не более 200 команд.

Для осуществления передачи управления на втором уровне используется ряд элементарных действий: переходы, вызовы процедур, вызовы сопрограмм, ловушки и прерывания. Переходы нужны для того, чтобы остановить одну последовательность команд и начать новую. Процедуры нужны для того, чтобы изолировать какой-то блок программы, который можно вызывать из различных мест этой же программы. Сопрограммы позволяют двум потокам управления работать одновременно. Ловушки используются для сообщения об исключительных ситуациях (например, о переполнении). Прерывания позволяют осуществлять процесс ввода-вывода параллельно с основным вычислением, при этом центральный процессор получает сигнал, как только ввод-вывод завершен.

Задачу «Ханойская башня» можно решить с использованием рекурсии.

Наконец, архитектура IA-64 использует модель вычисления EPIC. Для повышения скорости работы в этой архитектуре предусмотрены предикация и спекулятивная загрузка. IA-64 может иметь значительное преимущество над Pentium II, но она возлагает на компилятор огромное бремя параллелизма.

Вопросы и задания

1. В Pentium II команды могут содержать любое число байтов, даже нечетное. В UltraSPARC II все команды содержат четное число байтов. В чем преимущество системы Pentium II?
2. Разработайте расширенный код операций, который позволяет закодировать в 36-битной команде следующее:
   - 7 команд с двумя 32-битными адресами и номером одного 3-битного регистра;
   - 500 команд с одним 15-битным адресом и номером одного 3-битного регистра;
   - 50 команд без адресов и регистров.

3. Можно ли разработать такой расширенный код операций, который позволял бы кодировать в 12-битной команде следующее:
   - 4 команды с тремя регистрами;
   - 255 команд с одним регистром;
   - 16 команд без регистров.

   (Размер регистра составляет 3 бита.)
4. В некоторой машине имеются 16-битные команды и 6-битные адреса. Одни команды содержат один адрес, другие — два. Если существует n двухадресных команд, то каково максимальное число одноадресных команд?
5. Имеется одноадресная машина с регистром-аккумулятором. Ниже приведены значения некоторых слов в памяти:
   - слово 20 содержит число 40;
   - слово 30 содержит число 50;
   - слово 40 содержит число 60;
   - слово 50 содержит число 70.

   Какие значения следующие команды загрузят в регистр-аккумулятор?
   - `LOAD IMMEDIATE 20`
   - `LOAD DIRECT 20`
   - `LOAD INDIRECT 20`
   - `LOAD IMMEDIATE 30`
   - `LOAD DIRECT 30`
   - `LOAD INDIRECT 30`.
6. Для каждого из четырех видов машин — безадресной, одноадресной, двухадресной и трехадресной — напишите программу вычисления следующего выражения:

   $X=(A+B\times C)/(D-E\times F)$.
7. В наличии имеются следующие команды:
   - безадресные:

     `PUSH M`

     `POP M`

     `ADD`

     `SUB`

     `MUL`

     `DIV`
   - одноадресные:

     `LOAD M`

     `STORE M`

     `ADD M`

SUB M

MUL M

DIV M

+ двухадресные:

MOV (X=Y)

ADD (X=X+Y)

SUB (X=X-Y)

MUL (X=X*Y)

DIV (X=X/Y)

• трехадресные:

MOV (X=Y)

ADD (X=Y+Z)

SUB (X=Y-Z)

MUL (X=Y*Z)

DIV (X=Y/Z).

8. M — это 16-битный адрес памяти, а X, Y и Z — это или 16-битные адреса, или 4-битные регистры. Безадресная машина использует стек, одноадресная машина использует регистр-аккумулятор, а оставшиеся две имеют 16 регистров и команды, которые оперируют со всеми комбинациями ячеек памяти и регистров. Команда SUB X, Y вычитает Y из X, а команда SUB X, Y, Z вычитает Z из Y и помещает результат в X. Если длина кодов операций равна 8 битам, а размеры команд кратны 4 битам, сколько битов нужно каждой машине для вычисления X?

9. Придумайте такой механизм адресации, который позволяет определять в 6-битном поле произвольный набор из 64 адресов, не обязательно смежных.

10. В чем недостаток самоизменяющихся программ, которые не были упомянуты в тексте?

11. Переделайте следующие формулы из инфиксной записи в обратную польскую запись:
    - A+B+C+D+E
    - (A+B) x (C+D)+E
    - (AxB)+(CxD)+E
    - (A-B) x (((C-DxE)/F)/G) xH

12. Переделайте следующие формулы из обратной польской записи в инфиксную запись:
    - AB+C+Dx
    - AB/CD/+
    - ABCDE+xx/
    - ABCDExF/+G-H/x+

13. Какие из следующих пар формул в обратной польской записи математически эквивалентны?
    - AB + C + и A B C + +
    - A B - C - и A B C - -
    - A B x C + и A B C + x
14. Напишите три формулы в обратной польской записи, которые нельзя переделать в инфиксную запись.
15. Переделайте следующие инфиксные логические формулы в обратной польской записи.
    - (A И B) ИЛИ C
    - (A ИЛИ B) И (A ИЛИ C)
    - (A И B) ИЛИ (C И D)
16. Переделайте следующую инфиксную формулу в обратную польскую запись и напишите код JVM, чтобы выполнить ее.

    (2x3+4)-(4/2+1)
17. Команда языка ассемблера

    MOV REG ADDR

    означает загрузку регистра из памяти компьютера Pentium II. Однако для UltraSPARC II для загрузки регистра из памяти нужно написать

    LOAD ADDR, REG

    Почему порядок операндов разный?
18. Сколько регистров содержится в машине, форматы команд которой даны на рис. 5.16?
19. В форматах команд на рис. 5.16 бит 23 используется для различения формата 1 и формата 2. Однако для определения формата 3 никакого специального бита не предусмотрено. Как аппаратное обеспечение узнает, что нужно использовать формат 3?
20. Обычно программа определяет местонахождение переменной X в пределах интервала от A до B. Если бы имелась трехадресная команда с операндами A, B и X, сколько битов кода условия было бы установлено этой командой?
21. Pentium II содержит бит кода условия, который следит за переносом бита 3 после выполнения арифметической операции. Зачем это нужно?
22. В UltraSPARC II нет такой команды, которая загружает в регистр 32-битное число. Вместо нее обычно используется последовательность из двух команд: SETHI и ADD. Существуют ли еще какие-нибудь способы загрузки 32-битного числа в регистр? Аргументируйте.
23. Один из ваших друзей стучится к вам в комнату в 3 часа ночи и радостно сообщает, что у него появилась замечательная идея: команда с двумя кодами операций. Что вы сделаете в этой ситуации: отправите своего друга получать патент или пошлете его обратно к чертежной доске?

24. В программировании очень распространены следующие формы проверки:

    if (n==0)
    if O>J).
    if (k<4).

    Предложите команду, которая будет проверять эти условия эффективно. Какие поля имеются в вашей команде?

25. Покажите для 16-битного двоичного числа 1001 0101 1100 0011:
    + Сдвиг вправо на 4 бита с заполнением нулями.
    • Сдвиг вправо на 4 бита с расширением по знаку.
    • Сдвиг влево на 4 бита.
    • Циклический сдвиг влево на 4 бита.
    • Циклический сдвиг вправо на 4 бита.
26. Как можно в машине, в которой нет команды CLR, очистить слово памяти?
27. Вычислите логическое выражение (А И В) ИЛИ С для:
    • A-1101 0000 1010 1101
    • B—1111 11110000 1111
    • C=0000 0000 0010 0000
28. Придумайте, как поменять местами две переменные А и В, не используя при этом третью переменную или регистр. Подсказка: подумайте о команде ИСКЛЮЧАЮЩЕЕ ИЛИ.
29. На некотором компьютере можно перемещать число из одного регистра в другой, сдвигать каждый из них влево на разное количество байтов и складывать полученные результаты за меньшее время, чем потребовалось бы для выполнения умножения. При каком условии эта последовательность команд будет полезна для вычисления произведения «константа х переменная»?
30. Разные машины имеют разную плотность команд (то есть разное число байтов, которое требуется для выполнения определенного вычисления). Транслируйте следующие три фрагмента программы на языке Java на ассемблер для Pentium II, UltraSPARC II и JVM. Затем посчитайте, сколько байтов требуется для выполнения каждого выражения для каждой машины (предполагается, что i и j — это локальные переменные памяти):
    • i-3;
    • i-j;
    • i-j-1;
31. В этой главе рассматривались команды цикла для работы с циклами for. Разработайте команду для обращения с циклами while.
32. Предположим, что ханойские монахи могут перемещать один диск за 1 минуту (они не торопятся закончить работу, поскольку в Ханое очень мало вакансий для людей с подобными навыками). Сколько времени им потребуется, чтобы решить задачу (то есть переместить все 64 диска)? Ответ дайте в годах.

33. Почему устройства ввода-вывода помещают вектор прерывания на шину? Разве нельзя вместо этого сохранить соответствующую информацию в таблице в памяти?
34. Компьютер для считывания информации с диска использует канал прямого доступа к памяти. Диск содержит 64 сектора по 512 байтов на дорожке. Время оборота диска 16 мс. Ширина шины 16 битов. Каждая передача шины занимает 500 не. В среднем для одной команды процессора требуется два цикла шины. Насколько скорость работы процессора замедляется из-за прямого доступа к памяти?
35. Почему программам обработки прерываний приписываются определенные приоритеты, а обычные процедуры приоритетов не имеют?
36. Архитектура IA-64 содержит необычайно большое число регистров (64). Связано ли столь большое количество регистров с использованием предикации? Если да, то каким образом? Если нет, то зачем тогда их так много?
37. В пятой главе обсуждалось понятие спекулятивной загрузки. Но о командах спекулятивного сохранения мы не упоминали. Почему? Может быть, они просто аналогичны спекулятивным загрузкам, или существует какая-то другая причина, по которой мы не стали о них говорить?
38. Когда нужно связать две локальные сети, между ними помещается мост, связанный с обеими сетями. Каждый передаваемый какой-либо сетью пакет вызывает прерывание на мосту, чтобы мост мог определить, нужно ли этот пакет пересылать. Предположим, что на обработку прерывания и проверку пакета требуется 250 мке, но пересылка этого пакета в случае необходимости совершается с использованием прямого доступа в память, поэтому центральный процессор не загружается. Если все пакеты вмещают 1 Кбайт, то какова максимальная скорость передачи данных на каждой из сетей?
39. На рис. 5.24 указатель фрейма указывает на первую локальную переменную. Какая информация нужна программе, чтобы выйти из процедуры и вернуться в исходное положение?
40. Напишите подпрограмму на языке ассемблера для превращения целого двоичного числа со знаком в код ASCII.
41. Напишите подпрограмму на языке ассемблера для превращения инфиксной формулы в обратную польскую запись.
42. «Ханойская башня» — это не единственная рекурсивная процедура, любимая многими компьютерщиками. Есть еще одна очень популярная рекурсивная процедура n!, где n!=n(n-1)! Подчиняется ограничивающему условию 01=1. Напишите на вашем любимом языке ассемблера процедуру для вычисления n!.
43. Попробуйте решить задачу «Ханойская башня» без использования рекурсии путем содержания стека в массиве. Предупреждаем, что, вероятно, вы не сможете найти решения.

Глава 6
Уровень операционной системы

Как мы уже говорили, современный компьютер состоит из ряда уровней, каждый из которых добавляет дополнительные функции к уровню, который находится под ним. Мы рассмотрели цифровой логический уровень, микроархитектурный уровень и уровень команд. Настало время перейти к следующему уровню — уровню операционной системы.

Операционная система — это программа, которая добавляет ряд команд и особенностей к тем, которые обеспечиваются уровнем команд. Обычно операционная система реализуется главным образом в программном обеспечении, но нет никаких веских причин, по которым ее нельзя было бы реализовать в аппаратном обеспечении (как микропрограммы). Уровень операционной системы показан на рис. 6.1.

Хотя и уровень операционной системы, и уровень команд абстрактны (в том смысле, что они не являются реальным аппаратным обеспечением), между ними есть важное различие. Набор команд уровня операционной системы — это полный набор команд, доступных для прикладных программистов. Он содержит практически все команды более низкого уровня, а также новые команды, которые добавляет операционная система. Эти новые команды называются **системными вызовами.** Они вызывают определенную службу операционной системы, в частности одну из ее команд. Обычный системный вызов считывает какие-нибудь данные из файла.

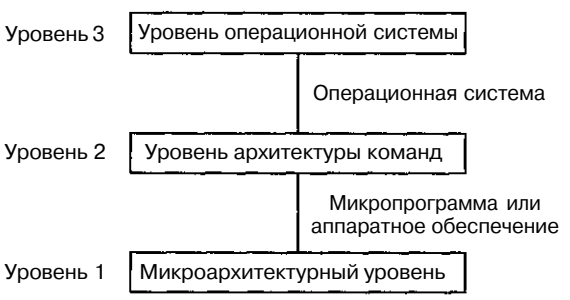


Рис. 6.1. Расположение уровня операционной системы

Уровень операционной системы всегда интерпретируется. Когда пользовательская программа выполняет команду операционной системы, например чтение данных из файла, операционная система выполняет эту команду шаг за шагом, точно

так же, как микропрограмма выполняет команду ADD. Однако когда программа выполняет команду уровня архитектуры команд, эта команда выполняется непосредственно микроархитектурным уровнем без участия операционной системы.

В этой книге мы можем лишь очень кратко в общих чертах рассказать вам об уровне операционной системы. Мы сосредоточимся на трех важных особенностях. Первая особенность — это виртуальная память. Виртуальная память используется многими операционными системами. Она позволяет создать впечатление, что у машины больше памяти, чем есть на самом деле. Вторая особенность — файл ввода-вывода. Это понятие более высокого уровня, чем команды ввода-вывода, которые мы рассматривали в предыдущей главе. Третья особенность — параллельная обработка (как несколько процессов могут выполняться, обмениваться информацией и синхронизироваться). Понятие процесса является очень важным, и мы подробно рассмотрим его ниже в этой главе. Под процессом можно понимать работающую программу и всю информацию об ее состоянии (о памяти, регистрах, счетчике команд, вводе-выводе и т. д.). После обсуждения этих основных характеристик мы покажем, как они применяются к операционным системам двух машин из трех наших примеров: Pentium II (Windows NT) и UltraSPARC II (UNIX). Поскольку picojava II обычно используется для встроенных систем, у этой машины нет операционной системы.

Виртуальная память

В первых компьютерах память была очнь мала по объему и к тому же дорого стоила. IBM-650, ведущий компьютер того времени (конец 50-х годов), содержал всего 2000 слов памяти. Один из первых 60 компиляторов, ALGOL, был написан для компьютера с размером памяти всего 1024 слова. Древняя система с разделением времени прекрасно работала на компьютере PDP-1, общий размер памяти которого составлял всего 4096 18-битных слов для операционной системы и пользовательских программ. В те времена программисты тратили очень много времени на то, чтобы впихнуть свои программы в крошечную память. Часто приходилось использовать алгоритм, который работает намного медленнее другого алгоритма, поскольку лучший алгоритм был слишком большим по размеру и программа, в которой использовался этот лучший алгоритм, могла не поместиться в память компьютера.

Традиционным решением этой проблемы было использование вспомогательной памяти (например, диска). Программист делил программу на несколько частей, так называемых **оверлеев,** каждый из которых помещался в память. Чтобы прогнать программу, сначала следовало считывать и запускать первый оверлей. Когда он завершался, нужно было считывать и запускать второй оверлей и т. д. Программист отвечал за разбиение программы на оверлеи и решал, в каком месте вспомогательной памяти должен был храниться каждый оверлей, контролировал передачу оверлеев между основной и вспомогательной памятью и вообще управлял всем этим процессом без какой-либо помощи со стороны компьютера.

Хотя эта технология широко использовалась на протяжении многих лет, она требовала длительной кропотливой работы, связанной с управлением оверлеями. В 1961 году группа исследователей из Манчестера (Англия) предложила метод автоматического выполнения процесса наложения, при котором программист мог вообще не знать об этом процессе [42]. Этот метод, который сейчас называется **виртуальной памятью,** имел очевидное преимущество, поскольку освобождал программиста от огромного количества нудной работы. Впервые этот метод был использован в ряде компьютеров, выпущенных в 60-е годы. К началу 70-х годов виртуальная память появилась в большинстве компьютеров. В настоящее время даже компьютеры[1] на одной микросхеме, в том числе Pentium II и UltraSPARC II, содержат очень сложные системы виртуальной памяти. Мы рассмотрим их ниже в этой главе.

Страничная организация памяти

Группа ученых из Манчестера выдвинула идею о разделении понятий адресного пространства и адресов памяти. Рассмотрим в качестве примера типичный компьютер того времени с 16-битным полем адреса в командах и 4096 словами памяти. Программа, работающая на таком компьютере, могла обращаться к 65536 словам памяти (поскольку адреса были 16-битными, а 2^{16}=65536). Обратите внимание, что число адресуемых слов зависит только от числа битов адреса и никак не связано с числом реально доступных слов в памяти. **Адресное пространство** такого компьютера состоит из чисел 0, 1, 2,..., 65535, так как это набор всех возможных адресов. Однако в действительности компьютер мог иметь гораздо меньше слов в памяти.

До изобретения виртуальной памяти приходилось проводить жесткое различие между теми адресами, которые меньше 4096, и теми, которые равны или больше 4096. Эти две части рассматривались как полезное адресное пространство и бесполезное адресное пространство соответственно (адреса выше 4095 были бесполезными, поскольку они не соответствовали реальным адресам памяти). Никакого различия между адресным пространством и адресами памяти не проводилось, поскольку между ними подразумевалось взаимно-однозначное соответствие.

Идея разделения понятий адресного пространства и адресов памяти состоит в следующем. В любой момент времени можно получить прямой доступ к 4096 словам памяти, но это не значит, что они непременно должны соответствовать адресам памяти от 0 до 4095. Например, мы могли бы сообщить компьютеру, что при обращении к адресу 4096 должно использоваться слово из памяти с адресом 0, при обращении к адресу 4097 — слово из памяти с адресом 1, при обращении к адресу **8191** — **слово** из памяти с адресом 4095 и т. д. Другими словами, мы определили отображение из адресного пространства в действительные адреса памяти (рис. 6.2).

[1] Строго говоря, сверхбольшие интегральные микросхемы, на которых сейчас располагают микропроцессоры, в том числе и упоминаемый автором процессор Pentium II, не являются компьютерами. Компьютер должен содержать помимо процессора память и контроллер управления ею, устройства ввода-вывода и соответствующие контроллеры для управления ими. — *Примеч. научн. ред.*

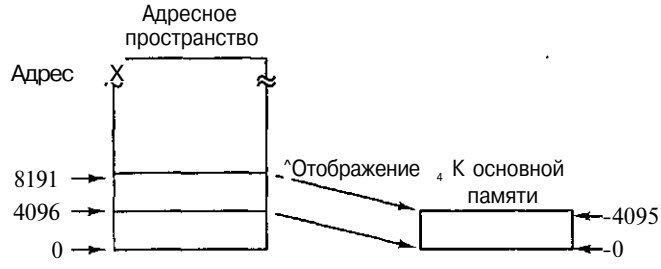


Рис. 6.2. Виртуальные адреса памяти с 4096 по 8191 отображаются в адреса основной памяти с 0 по 4095

Возникает интересный вопрос: а что произойдет, если программа совершит переход в один из адресов с 8192 по 12287? В машине без виртуальной памяти произойдет ошибка, на экране появится фраза «Несуществующий адрес памяти» и выполнение программы остановится. В машине с виртуальной памятью будет иметь место следующая последовательность шагов:

1. Слова с 4096 до 8191 будут размещены на диске.
2. Слова с 8192 до 12287 будут загружены в основную память.
3. Отображение адресов изменится: теперь адреса с 8192 до 12287 соответствуют ячейкам памяти с 0 по 4095.
4. Выполнение программы будет продолжаться, как будто ничего ужасного не случилось.

Такая технология автоматического наложения называется **страничной организацией памяти,** а куски программы, которые считываются с диска, называются **страницами.**

Возможен и другой, более сложный способ отображения адресов из адресного пространства в реальные адреса памяти. Адреса, к которым программа может обращаться, мы будем называть **виртуальным адресным пространством,** а реальные адреса памяти в аппаратном обеспечении — **физическим адресным пространством.** Схема распределения памяти или таблица страниц соотносит виртуальные адреса с физическими адресами. Предполагается, что на диске достаточно места для хранения полного виртуального адресного пространства (или, по крайней мере, той его части, которая используется в данный момент).

Программы пишутся так, как будто в основной памяти хватит места для размещения всего виртуального адресного пространства, даже если это не соответствует действительности. Программы могут загружать слова из виртуального адресного пространства или записывать слова в виртуальное адресное пространство, несмотря на то, что на самом деле физической памяти для этого не хватает. Программист может писать программы, даже не осознавая, что виртуальная память существует. Просто создается такое впечатление, что объем памяти данного компьютера достаточно велик.

Позднее мы сопоставим страничную организацию памяти с процессом сегментации, при котором программист должен знать о существовании сегментов. Еще

раз подчеркнем, что страничная организация памяти создает иллюзию большой линейной основной памяти такого же размера, как адресное пространство. В действительности основная память может быть меньше (или больше), чем виртуальное адресное пространство. То, что память большого размера просто моделируется с помощью страничной организации памяти, нельзя определить по программе (только с помощью контроля синхронизации). При обращении к любому адресу всегда появляются требующиеся данные или нужная команда. Поскольку программист может писать программы и при этом ничего не знать о существовании страничной организации памяти, этот механизм называют прозрачным.

Ситуация, когда программист использует какой-либо виртуальный механизм и даже не знает, как он работает, не нова. В архитектуры команд, например, часто включается команда MUL (команда умножения), даже если в аппаратном обеспечении нет специального устройства для умножения. Иллюзия, что машина может перемножать числа, поддерживается микропрограммой. Точно так же операционная система может создавать иллюзию, что все виртуальные адреса поддерживаются реальной памятью, даже если это неправда. Только разработчикам операционных систем и тем, кто изучает операционные системы, нужно знать, как создается такая иллюзия.

Реализация страничной организации памяти

Для виртуальной памяти требуется диск для хранения полной программы и всех данных. Естественно, при изменении копии должен меняться и оригинал.

Виртуальное адресное пространство разбивается на ряд страниц равного размера, обычно от 512 до 64 Кбайт, хотя иногда встречается 4 Мбайт. Размер страницы всегда должен быть степенью двойки. Физическое адресное пространство тоже разбивается на части равного размера таким образом, чтобы каждая такая часть основной памяти вмещала ровно одну страницу. Эти части основной памяти называются страничными кадрами. На рисунке 6.2 основная память содержит только один страничный кадр. На практике обычно имеется несколько тысяч страничных кадров.

На рисунке 6.3, *а* показан один из возможных вариантов разделения первых 64 K виртуального адресного пространства на страницы по 4 K (отметим, что сейчас мы говорим о 64 K и 4 K адресов). Адрес может быть байтом, но может быть словом в компьютере, в котором последовательно расположенные слова имеют последовательные адреса. Виртуальную память, изображенную на рис. 6.3, можно реализовать посредством таблицы страниц, в которой количество элементов равно количеству страниц в виртуальном адресном пространстве. Здесь для простоты мы показали только первые 16 элементов. Когда программа пытается обратиться к слову из первых 64 K виртуальной памяти, чтобы вызвать команду или данные или чтобы сохранить данные, сначала она порождает виртуальный адрес от 0 до 65532 (предполагается, что адреса слов должны делиться на 4). Для порождения этого адреса могут использоваться любые стандартные способы адресации, в том числе индексирование и косвенная адресация.

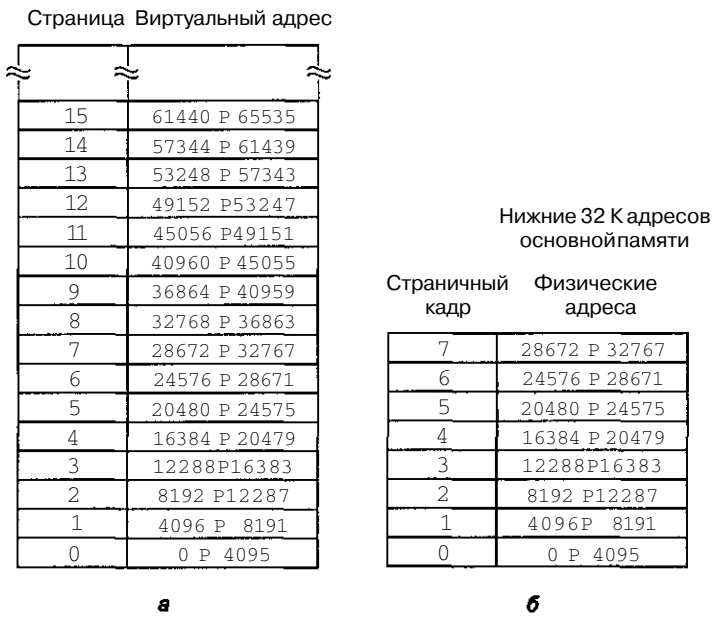


Рис. 6.3. Первые 64 К виртуального адресного пространства разделены на 16 страниц по 4 К каждая (а); 32 К основной памяти разделены на 8 страничных кадров по 4 К каждый (б)

На рис. 6.3, б изображена физическая память, состоящая из восьми страничных кадров по 4 К. Эту память можно ограничить до 32 К, поскольку: 1) это вся память машины (для процессора, встроенного в стиральную машину или микроволновую печь, этого достаточно), или 2) оставшаяся часть памяти занята другими программами.

А теперь рассмотрим, как можно 32-битный виртуальный адрес отобразить на физический адрес основной памяти. В конце концов, память воспринимает только реальные адреса и не воспринимает виртуальные, поэтому такое отображение должно быть сделано. Каждый компьютер с виртуальной памятью содержит устройство для осуществления отображения виртуальных адресов на физические. Это устройство называется **контроллером управления памятью (MMU - Memory Management Unit).** Он может находиться на микросхеме процессора или на отдельной микросхеме рядом с процессором. В нашем примере контроллер управления памятью отображает 32-битный виртуальный адрес в 15-битный физический адрес, поэтому ему требуется 32-битный входной регистр и 15-битный выходной регистр.

Чтобы понять, как работает контроллер управления памятью, рассмотрим пример на рис. 6.4. Когда в контроллер управления памятью поступает 32-битный виртуальный адрес, он разделяет этот адрес на 20-битный номер виртуальной страницы и 12-битное смещение внутри этой страницы (поскольку страницы в нашем примере по 4 К). Номер виртуальной страницы используется в качестве индекса в таблице страниц для нахождения нужной страницы. На рис. 6.4 номер виртуальной страницы равен 3, поэтому из таблицы выбирается элемент 3.

Сначала контроллер управления памятью проверяет, находится ли нужная страница в текущий момент в памяти. Поскольку у нас есть 2^{20} виртуальных страниц и

всего 8 страничных кадров, не все виртуальные страницы могут находиться в памяти одновременно. Контроллер управления памятью проверяет **бит** присутствия в данном элементе таблицы страниц. В нашем примере этот бит равен 1. Это значит, что страница в данный момент находится в памяти.

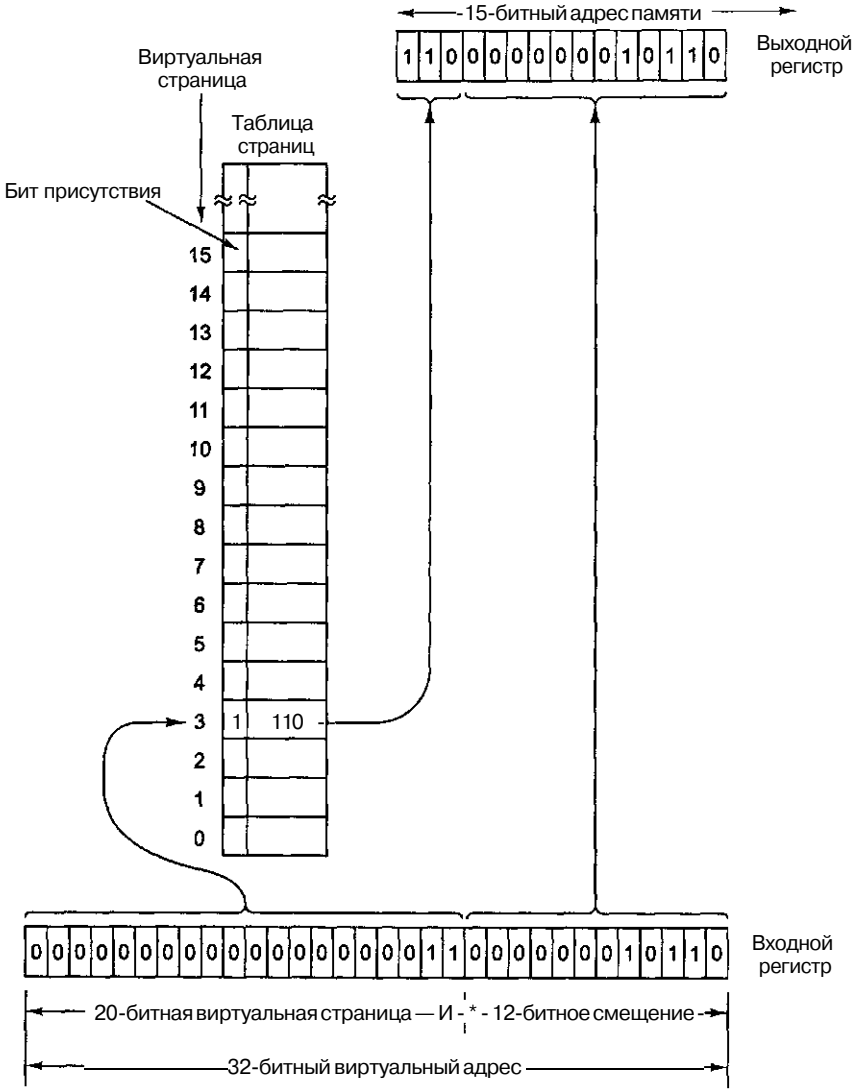


Рис. 6.4. Формирование адреса основной памяти из адреса виртуальной памяти

Далее из выбранного элемента таблицы нужно взять значение страничного кадра (в нашем примере — 6) и скопировать его в старшие три бита 15-битного выходного регистра. Нужно именно три бита, потому что в физической памяти находится 8 страничных кадров. Параллельно с этой операцией младшие 12 битов виртуального адреса (поле смещения страницы) копируются в младшие 12 битов выходного

регистра. Затем полученный 15-битный адрес отправляется в кэш-память или основную память для поиска.

На рисунке 6.5 показано возможное отображение виртуальных страниц в физические страничные кадры. Виртуальная страница 0 находится в страничном кадре 1. Виртуальная страница 1 находится в страничном кадре 0. Виртуальной страницы 2 нет в основной памяти. Виртуальная страница 3 находится в страничном кадре 2. Виртуальной страницы 4 нет в основной памяти. Виртуальная страница 5 находится в страничном кадре 6 и т. д.

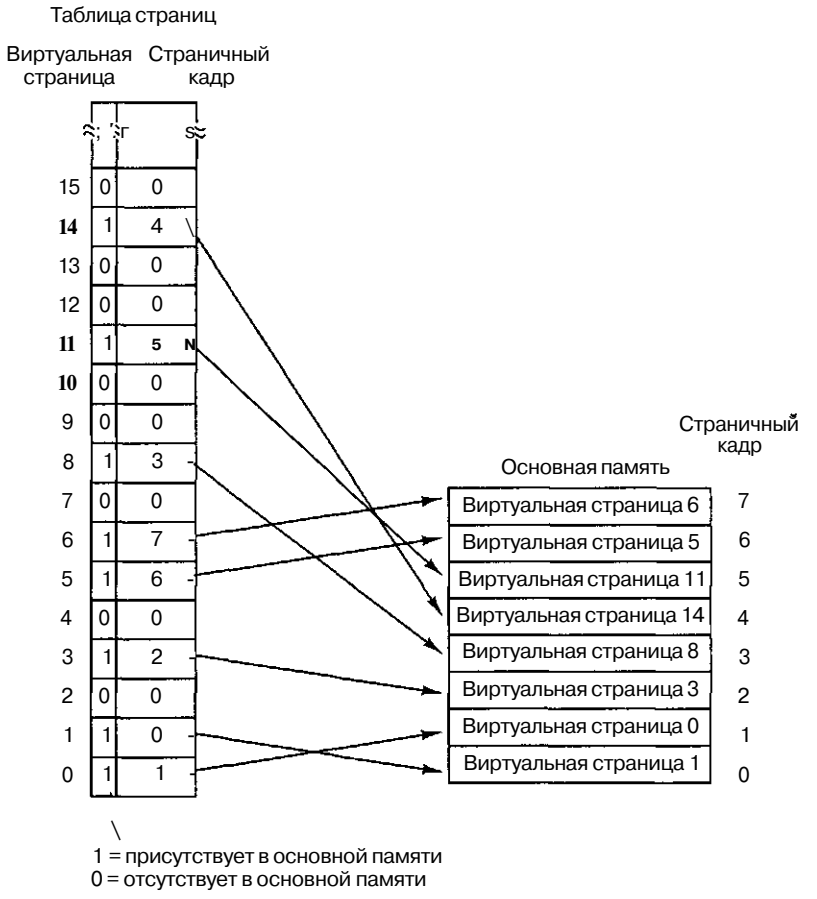


Рис. 6.5. Возможное отображение первых 16 виртуальных страниц в основную память, содержащую 8 страничных кадров

Вызов страниц по требованию и рабочее множество

Ранее предполагалось, что виртуальная страница, к которой происходит обращение, находится в основной памяти. Однако это предположение не всегда верно, поскольку в основной памяти недостаточно места для всех виртуальных страниц.

При обращении к адресу страницы, которой нет в основной памяти, происходит **ошибка из-за отсутствия страницы.** В случае такой ошибки операционная система должна считать нужную страницу с диска, ввести новый адрес физической памяти в таблицу страниц, а затем повторить команду, которая вызвала ошибку.

На машине с виртуальной памятью можно запустить программу даже в том случае, если ни одной части программы нет в основной памяти. Просто таблица страниц должна указывать, что абсолютно все виртуальные страницы находятся во вспомогательной памяти. Если центральный процессор пытается вызвать первую команду, он сразу получает ошибку из-за отсутствия страницы, в результате чего страница, содержащая первую команду, загружается в память и вносится в таблицу страниц. После этого начинается выполнение первой команды. Если первая команда содержит два адреса и оба эти адреса находятся на разных страницах, причем обе страницы не являются страницей, в которой находится команда, то произойдет еще две ошибки из-за отсуствия страницы и еще две страницы будут перенесены в основную память до завершения команды. Следующая команда может вызвать еще несколько ошибок и т. д.

Такой метод работы с виртуальной памятью называется **вызовом страниц по требованию.** Он похож на один из способов кормления младенцев: когда младенец кричит, вы его кормите (в противоположность кормлению по расписанию). При вызове страниц по требованию страницы переносятся в основную память только в случае необходимости, но не заранее.

Вопрос о том, стоит ли использовать вызов страниц по требованию или нет, имеет смысл только в самом начале при запуске программы. Когда программа проработает некоторое время, нужные страницы уже будут собраны в основной памяти. Если это компьютер с разделением времени и процессы откачиваются обратно, проработав примерно 100 миллисекунд, то каждая программа будет запускаться много раз. Для каждой программы распределение памяти уникально и при переключении с одной программы на другую меняется, поэтому в системах с разделением времени такой подход не годится.

Альтернативный подход основан на наблюдении, что большинство команд обращаются к адресному пространству не равномерно. Обычно большинство обращений относятся к небольшому числу страниц. При обращении к памяти можно вызвать команду, вызвать данные или сохранить данные. В каждый момент времени t существует набор страниц, которые использовались при последних **к** обращениях. Деннинг [29] назвал этот набор страниц **рабочим множеством.**

Поскольку рабочее множество обычно меняется очень медленно, можно, опираясь на последнее перед остановкой программы рабочее множество, предсказать, какие страницы понадобятся при новом запуске программы. Эти страницы можно загрузить заранее перед очередным запуском программы (предполагается, что предсказание будет точным).

Политика замещения страниц

В идеале набор страниц, которые постоянно используются программой (так называемое **рабочее множество),** можно хранить в памяти, чтобы сократить количество ошибок из-за отсутствия страниц. Однако программисты обычно не знают, какие страницы находятся в рабочем множестве, поэтому операционная система периодически должна показывать это множество. Если программа обращается

к странице, которая отсутствует в основной памяти, ее нужно вызвать с диска. Однако чтобы освободить для нее место, на диск нужно отправить какую-нибудь другую страницу. Следовательно, нужен алгоритм для определения, какую именно страницу необходимо убрать из памяти.

Выбирать такую страницу просто наугад нельзя. Если, например, выбрать страницу, содержащую команду, выполнение которой вызвало ошибку, то при попытке вызвать следующую команду произойдет еще одна ошибка из-за отсутствия страницы. Большинство операционных систем стараются предсказать, какие из страниц в памяти наименее полезны в том смысле, что их отсутствие не повлияет сильно на ход программы. Иными словами, вместо того чтобы удалять страницу, которая скоро понадобится, постарайтесь выбрать такую страницу, которая не будет нужна долгое время.

По одному из алгоритмов удаляется та страница, которая использовалась наиболее давно, поскольку вероятность того, что она будет в текущем рабочем множестве, очень мала. Этот алгоритм называется **LRU (Least Recently Used — алгоритм удаления наиболее давно использовавшихся элементов).** Хотя этот алгоритм работает достаточно хорошо, иногда возникают патологические ситуации. Ниже описана одна из них.

Представьте себе программу, выполняющую огромный цикл, который простирается на девять виртуальных страниц, а в физической памяти место есть только для восьми страниц. Когда программа перейдет к странице 7, в основной памяти будут находиться страницы с 0 по 7 (табл. 6.1). Затем совершается попытка вызвать команду из виртуальной страницы 8, что вызывает ошибку из-за отсутствия страницы. Нужно принять решение, какую страницу убрать. По алгоритму LRU будет выбрана виртуальная страница 0, поскольку она использовалась раньше всех. Виртуальная страница 0 удаляется, а нужная виртуальная страница помещается на ее место (табл. 6.2).

Таблица 6.1. Ситуация, в которой алгоритм LRU не действует (1)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница О

Таблица 6.2. Ситуация, в которой алгоритм LRU не действует (2)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница 8

Таблица 6.3. Ситуация, в которой алгоритм LRU не действует (3)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 0
Виртуальная страница 8

После выполнения команд из виртуальной страницы 8 программа возвращается к началу цикла, то есть к виртуальной странице 0. Этот шаг вызывает еще одну ошибку из-за отсутствия страницы. Только что выброшенную виртуальную страницу 0 приходится вызывать обратно. По алгоритму LRU удаляется страница 1 (табл. 6.3). Через некоторое время программа пытается вызвать команду из виртуальной страницы 1, что опять вызывает ошибку. Затем вызывается страница 1 и удаляется страница 2 и т. д.

Очевидно, что в этой ситуации алгоритм LRU совершенно не работает (другие алгоритмы тоже не работают при сходных обстоятельствах). Однако если расширить размер рабочего множества, число ошибок из-за отсутствия страниц станет минимальным.

Можно применять и другой алгоритм — **FIFO (First-in First-out — первым поступил, первым выводится).** FIFO удаляет ту страницу, которая раньше всех загружалась, независимо от того, когда в последний раз производилось обращение к этой странице. С каждым страничным кадром связан отдельный счетчик. Изначально все счетчики установлены на 0.

После каждой ошибки из-за отсутствия страниц счетчик каждой страницы, находящейся в памяти, увеличивается на 1, а счетчик только что вызванной страницы принимает значение 0. Когда нужно выбрать страницу для удаления, выбирается страница с самым большим значением счетчика. Поскольку она не загружалась в память очень давно, существует большая вероятность, что она больше не понадобится.

Если размер рабочего множества больше, чем число доступных страничных кадров, ни один алгоритм не дает хороших результатов, и ошибки из-за отсутствия страниц будут происходить часто. Если программа постоянно вызывает подобные ошибки, то говорят, что наблюдается **пробуксовка (thrashing).** Думаю, не нужно объяснять, что пробуксовка очень нежелательна. Если программа использует большое виртуальное адресное пространство, но имеет небольшое медленно изменяющееся рабочее множество, которое помещается в основную память, ничего страшного не произойдет. Это утверждение имеет силу, даже если программа использует в сто раз больше слов виртуальной памяти, чем их содержится в физической основной памяти.

Если страница, которую нужно удалить, не менялась с тех пор, как ее считали (а это вполне вероятно, если страница содержит программу, а не данные), то необязательно записывать ее обратно на диск, поскольку точная копия там уже существует. Однако если эта страница изменилась, то копия на диске уже ей не соответствует, и ее нужно туда переписать.

Если мы научимся определять, изменялась ли страница или не изменялась, то сможем избежать ненужных переписываний на диск и сэкономим много времени. На многих машинах в контроллере управления памятью содержится один бит для каждой страницы, который равен 0 при загрузке страницы и принимает значение 1, когда микропрограмма или аппаратное обеспечение изменяют эту страницу. По этому биту операционная система определяет, менялась данная страница или нет и нужно ли ее перезаписывать на диск или нет.

Размер страниц и фрагментация

Если пользовательская программа и данные время от времени заполняют ровно целое число страниц, то когда они находятся в памяти, свободного места там не будет. С другой стороны, если они не заполняют ровно целое число страниц, на последней странице останется неиспользованное пространство. Например, если программа и данные занимают 26 000 байтов на машине с 4096 байтами на страницу, то первые 6 страниц будут заполнены целиком, что в сумме даст 6x4096=24 576 байтов, а последняя страница будет содержать 26 000-24576=1424 байта. Поскольку в каждой странице имеется пространство для 4096 байтов, 2672 байта останутся свободными. Всякий раз, когда седьмая страница присутствует в памяти, эти байты будут занимать место в основной памяти, но при этом не будут выполнять никакой функции. Эта проблема называется внутренней фрагментацией (поскольку неиспользованное пространство является внутренним по отношению к какой-то странице).

Если размер страницы составляет п байтов, то среднее неиспользованное пространство в последней странице программы будет п/2 байтов — ситуация подсказывает, что нужно использовать страницы небольшого размера, чтобы свести к минимуму количество неиспользованного пространства. С другой стороны, если страницы маленького размера, то потребуется много страниц и большая таблица страниц. Если таблица страниц сохраняется в аппаратном обеспечении, то для хранения большой таблицы страниц нужно много регистров, что повышает стоимость компьютера. Кроме того, при запуске и остановке программы на загрузку и сохранение этих регистров потребуется больше времени.

Более того, маленькие страницы снижают эффективность пропускной способности диска. Поскольку перед началом передачи данных с диска приходится ждать примерно 10 млс (поиск + время вращения), выгоднее совершать большие передачи. При скорости передачи данных 10 Мбайт в секунду передача 8 Кбайт добавляет всего 0,7 млс (по сравнению с передачей 1 Кбайт).

Однако у маленьких страниц есть свои преимущества. Если рабочее множество состоит из большого количества маленьких отделенных друг от друга областей виртуального адресного пространства, при маленьком размере страницы будет реже возникать пробуксовка (режим интенсивной подкачки), чем при большом. Рассмотрим матрицу А 10 000x10 000, которая хранится в последовательных 8-байтныхсловах(А[1,1],А[2,1],А[3Д]ит. д.). При такой записи элементы ряда 1 (А[1,1], А[1,2], А[1,3] и т. д.) будут начинаться на расстоянии 80 000 байтов друг от друга. Программа, выполняющая вычисление над элементами этого ряда, будет использовать 10 000 областей, каждая из которых отделена от следующей 79 992 байтами. Если бы размер страницы составлял 8 Кбайт, то для хранения всех страниц понадобилось бы 80 Мбайт.

При размере страницы в 1 Кбайт для хранения всех страниц потребуется всего 10 Мбайт ОЗУ. При размере памяти в 32 Мбайт и размере страницы в 8 Кбайт программа войдет в режим интенсивной подкачки, а при размере страницы в 1 Кбайт этого не произойдет.

Сегментация

До сих пор мы говорили об одномерной виртуальной памяти, в которой виртуальные адреса идут один за другим от 0 до какого-то максимального адреса. По многим причинам гораздо удобнее использовать два или несколько отдельных виртуальных адресных пространств. Например, компилятор может иметь несколько таблиц, которые создаются в процессе компиляции:

1. Таблица символов, которая содержит имена и атрибуты переменных.
2. Исходный текст, сохраняемый для распечатки.
3. Таблица, содержащая все использующиеся целочисленные константы и константы с плавающей точкой.
4. Дерево, содержащее синтаксический анализ программы.
5. Стек, используемый для вызова процедур в компиляторе.

Каждая из первых четырех таблиц постоянно растет в процессе компиляции. Последняя таблица растет и уменьшается совершенно непредсказуемо. В одномерной памяти эти пять таблиц пришлось бы разместить в виртуальном адресном пространстве в виде смежных областей, как показано на рис. 6.6.

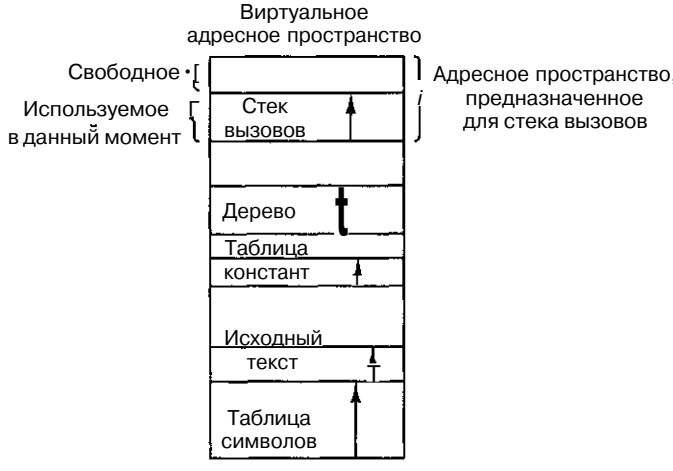


Рис. 6.6. В одномерном адресном пространстве, в котором содержатся постоянно увеличивающиеся таблицы, одна из таблиц может «врезаться» в другую

Посмотрим, что произойдет, если программа содержит очень большое число переменных. Область адресного пространства, предназначенная для таблицы символов, может переполниться, даже если в других таблицах полно свободного места. Компилятор, конечно, может сообщить, что он не способен продолжать работу

из-за большого количества переменных, но можно без этого и обойтись, поскольку в других таблицах много свободного места.

Компилятор может забирать свободное пространство у одних таблиц и передавать его другим таблицам, но это похоже на управление оверлеями вручную — некоторое неудобство в лучшем случае и долгая скучная работа в худшем.

На самом деле нужно просто освободить программиста от расширения и сокращения таблиц, подобно тому как виртальная память исключает необходимость следить за разбиением программы на оверлеи.

Для этого нужно создать много абсолютно независимых адресных пространств, которые называются **сегментами.** Каждый сегмент состоит из линейной последовательности адресов от 0 до какого-либо максимума. Длина каждого сегмента может быть любой от 0 до некоторого допустимого максимального значения. Разные сегменты могут иметь разную длину (обычно так и бывает). Более того, длина сегмента может меняться во время выполнения программы. Длина стекового сегмента может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться, когда что-либо выталкивается из стека.

Так как каждый сегмент основывает отдельное адресное пространство, разные сегменты могут увеличиваться или уменьшаться независимо друг от друга и не влияя друг на друга. Если стеку в определенном сегменте понадобилось больше адресного пространства (чтобы стек мог увеличиться), он может получить его, поскольку в его адресном пространстве больше не во что «врезаться». Естественно, сегмент может переполниться, но это происходит редко, поскольку сегменты очень большие. Чтобы определить адрес в двухмерной памяти, программа должна выдавать номер сегмента и адрес внутри сегмента. На рис. 6.7 изображена сегментированная память.

Мы подчеркиваем, что сегмент является логическим элементом, о котором программист знает и который он использует. Сегмент может содержать процедуру, массив, стек или ряд скалярных переменных, но обычно в него входит только один тип данных.

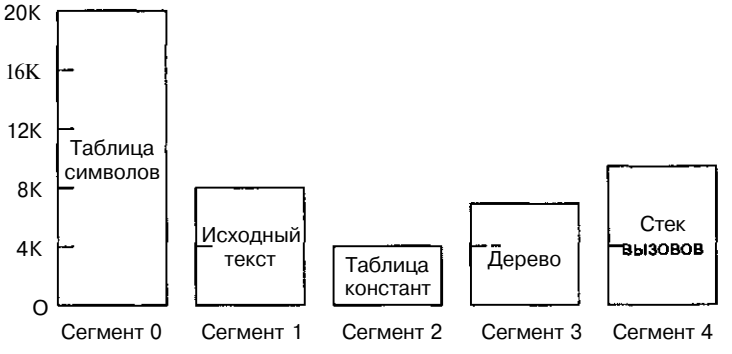


Рис. 6.7. Сегментированная память позволяет увеличивать и уменьшать каждую таблицу независимо от других таблиц

Сегментированная память имеет и другие преимущества помимо упрощения работы со структурами данных, которые постоянно уменьшаются или увеличиваются. Если каждая процедура занимает отдельный сегмент, в котором первый

адрес — это адрес 0, то связывание процедур, которые компилируются отдельно, сильно упрощается. Когда все процедуры программы скомпилированы и связаны, при вызове процедуры из сегмента п для обращения к слову 0 будет использоваться адрес (п, 0).

Если процедура в сегменте п впоследствии изменяется и перекомпилируется, то другие процедуры менять не нужно (поскольку ни один начальный адрес не был изменен), даже если новая версия больше по размеру, чем старая. В одномерной памяти процедуры обычно располагаются друг за другом, и между ними нет никакого адресного пространства. Следовательно, изменение размера одной процедуры может повлиять на начальный адрес других процедур. Это, в свою очередь, требует изменения всех процедур, которые вызывают любую из этих процедур, чтобы попадать в новые начальные адреса. Если в программу включено много процедур, этот процесс будет слишком накладным.

Сегментация облегчает разделение общих процедур или данных между несколькими программами. Если компьютер содержит несколько программ, работающих параллельно (это может быть реальная или смоделированная параллельная обработка), и если все эти программы используют определенные процедуры, снабжать каждую программу отдельной копией расточительно для памяти. А если сделать каждую процедуру отдельным сегментом, их легко можно будет разделять между несколькими программами, что исключит необходимость создавать физические копии каждой разделяемой процедуры. В результате мы сэкономим память.

Разные сегменты могут иметь разные виды защиты. Например, сегмент с процедурой можно определить как «только для выполнения», запретив тем самым считывание из него и запись в него. Для массива с плавающей точк разрешается только чтение и запись, но не выполнение и т. д. Такая защита часто помогает обнаружить ошибки в программе.

Вы должны понимать, почему защита имеет смысл только в сегментированной памяти и не имеет никакого смысла в одномерной (линейной) памяти. Обычно в сегменте не могут содержаться и процедура и стек одновременно (только что-нибудь одно). Поскольку каждый сегмент включает в себя объект только одного типа, он может использовать защиту, подходящую для этого типа. Страничная организация памяти и сегментация сравниваются в табл. 6.4.

Таблица 6.4. Сравнение страничной организации памяти и сегментации

Свойства	Страничная организация памяти	Сегментация
Должен ли программист знать об этом?	Нет	Да
Сколько линейных адресных пространств имеется?	1	Много
Может ли виртуальное адресное пространство увеличивать размер памяти?	Да	Да
Легко ли управлять таблицами с изменяющимися размерами?	Нет	Да
Зачем была придумана такая технология?	Чтобы смоделировать память большого размера	Чтобы обеспечить несколько адресных пространств

Содержимое страницы в каком-то смысле случайно. Программист может ничего не знать о страничной организации памяти. В принципе можно поместить несколько битов в каждый элемент таблицы страниц для доступа к нему, но чтобы использовать эту особенность, программисту придется следить за границами страницы в адресном пространстве. Дело в том, что страничное разбиение памяти было придумано для устранения подобных трудностей. Поскольку у пользователя создается иллюзия, что все сегменты постоянно находятся в основной памяти, к ним можно обращаться и не следить при этом за оверлеями.

Как реализуется сегментация

Сегментацию можно реализовать одним из двух способов. Это подкачка и разбиение на страницы. При первом подходе некоторый набор сегментов находится в памяти в данный момент. Если происходит обращение к сегменту, которого нет в данный момент в памяти, этот сегмент переносится в память. Если для него нет места в памяти, один или несколько сегментов нужно сначала записать на диск (если копия уже не находится там; в этом случае соответствующая копия просто удаляется из памяти). В каком-то смысле подкачка сегментов очень похожа на вызов страниц по требованию: сегменты загружаются и удаляются только в случае необходимости.

Однако сегментация существенно отличается от разбиения на страницы в следующем: размер страниц фиксирован, а размер сегментов — нет. На рис. 6.8, *а* показан пример физической памяти, в которой изначально содержится 5 сегментов. Посмотрим, что происходит, если сегмент 1 удаляется, а сегмент 7, который меньше по размеру, помещается на его место. В результате получится конфигурация, изображенная на рис. 6.8, *б*. Между сегментом 7 и сегментом 2 находится неиспользованная область («дырка»). Затем сегмент 4 меняется на сегмент 5 (рис. 6.8, в), а сегмент 3 замещается сегментом 6 (рис. 6.8, *г*). Через некоторое время память разделится на ряд областей, одни из которых будут содержать сегменты, а другие — неиспользованные области. Это называется **внешней фрагментацией** (поскольку неиспользованное пространство попадает не в сегменты, а в «дырки» между ними, то есть процесс происходит вне сегментов). Иногда внешнюю фрагментацию называют **поклеточной разбивкой.**

Посмотрите, что произойдет, если программа обратится к сегменту 3 в тот момент, когда память подвергается внешней фрагментации (рис. 6.8, г). Общее пространство «дырок» составляет 10 Кбайт, а это больше, чем нужно для сегмента 3, но так как это пространство разбито на маленькие кусочки, сегмент 3 туда загрузить нельзя. Вместо этого приходится сначала удалять другой сегмент.

Чтобы избежать такой ситуации, нужно сделать следующее. Каждый раз, когда появляется «дырка», нужно перемещать сегменты, следующие за «дыркой», ближе к адресу 0, устраняя таким образом эту «дырку» и оставляя большую «дырку» в конце. Есть и другой способ. Можно подождать, пока внешняя фрагментация не примет серьезный оборот (когда на долю «дырок» приходится больше определенного процента от всего объема памяти), и только после этого совершить уплотнение. На рис. 6.8, *д* показано, как память будет выглядеть после уплотнения. Цель уплотнения памяти — собрать все маленькие «дырки» в одну большую «дырку», в которую можно поместить один или несколько сегментов. Недостаток

уплотнения состоит в том, что на этот процесс тратится некоторое количество времени. Совершать уплотнение после появления каждой дырки невыгодно.

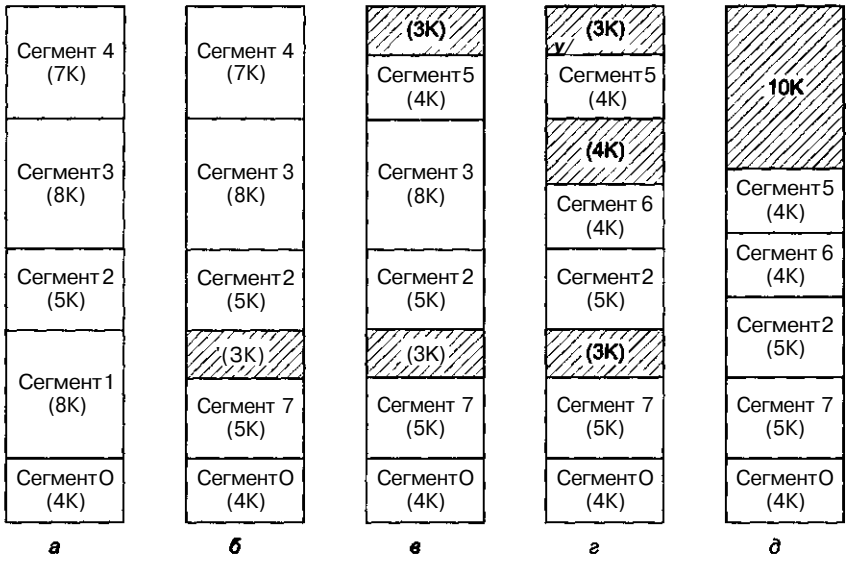


Рис. 6.8. Динамика внешней фрагментации (а, б, *в*, г); удаление внешней сегментации путем уплотнения (д)

Если на уплотнение памяти требуется слишком много времени, нужен специальный агоритм для определения, какую именно «дырку» лучше использовать для определенного сегмента. Для этого требуется список адресов и размеров всех «дырок». Популярный алгоритм **оптимальной подгонки** выбирает самую маленькую «дырку», в которую помещается нужный сегмент. Цель этого алгоритма — соотнести «дырки» и сегменты, чтобы избежать «отламывания» куска большой **«дырки»,** который может понадобиться позже для большого сегмента.

Другой популярный алгоритм по кругу просматривает список «дырок» и выбирает первую «дырку», которая по размеру подходит для данного сегмента. Естественно, это занимает меньше времени, чем проверка всего списка, чтобы найти оптимальную «дырку». Удивительно, но последний алгоритм гораздо лучше, чем алгоритм оптимальной подгонки, с точки зрения общей производительности, поскольку оптимальная подгонка порождает очень много маленьких неиспользованных дырок **[74].**

Оба алгоритма сокращают средний размер «дырки». Всякий раз, когда сегмент помещается в «дырку», которая больше, чем этот сегмент, что бывает практически всегда (точные попадания очень редки), «дырка» делится на две части. Одну часть занимает сегмент, а вторая часть — это новая «дырка». Новая «дырка» всегда меньше, чем старая. Без воссоздания больших «дырок» из маленьких оба алгоритма в конечном итоге будут наполнять память маленькими неиспользованными «дырками».

Опишем один из таких процессов. Всякий раз, когда сегмент удаляется из памяти, а одна или обе соседние области этого сегмента — «дырки», а не сегменты, смежные неиспользованные пространства можно слить в одну большую «дырку». Если из рис. 6.8, *г* удалить сегмент 5, то две соседние «дырки» и 4 K, которые использовались данным сегментом, будут слиты в одну «дырку» в 11 K.

В начале этого раздела мы говорили, что реализовать сегментацию можно двумя способами: подкачкой и разбиением на страницы. До сих пор речь шла о подкачке. При таком подходе по необходимости между памятью и диском перемещаются целые сегменты.

Второй способ реализации — разделить каждый сегмент на страницы фиксированного размера и вызывать их по требованию. В этом случае одни страницы сегмента могут находиться в памяти, а другие — на диске. Чтобы разбить сегмент на страницы, для каждого сегмента нужна отдельная таблица страниц. Поскольку сегмент представляет собой линейное адресное пространство, все средства разбиения на страницы, которые мы до сих пор рассматривали, применимы к любому сегменту. Единственное различие состоит в том, что каждый сегмент получает отдельную таблицу страниц.

MULTICS (Multiplexed Information and Computing Service — служба общей информации и вычислений) — это древняя операционная система, которая совмещала сегментацию с разбиением на страницы. Она была разработана Массачусетским технологическим институтом совместно с компаниями Bell Labs и General Electric [28, 106]. Адреса в MULTICS состоят из двух частей: номера сегмента и адреса внутри сегмента. Для каждого процесса существовал сегмент дескриптора, который содержал дескриптор для каждого сегмента. Когда аппаратное обеспечение получало виртуальный адрес, номер сегмента использовался в качестве индекса в сегмент дескриптора для нахождения дескриптора нужного сегмента (рис. 6.9). Дескриптор указывал на таблицу страниц, что позволяло разбивать на страницы каждый сегмент обычным способом. Для повышения производительности недавно используемые комбинации сегмента и страницы помещались в ассоциативную память из 16 элементов. Операционная система MULTICS уже давно не применяется, но виртуальная память всех процессоров Intel, начиная с 386-го, очень похожа на эту систему.

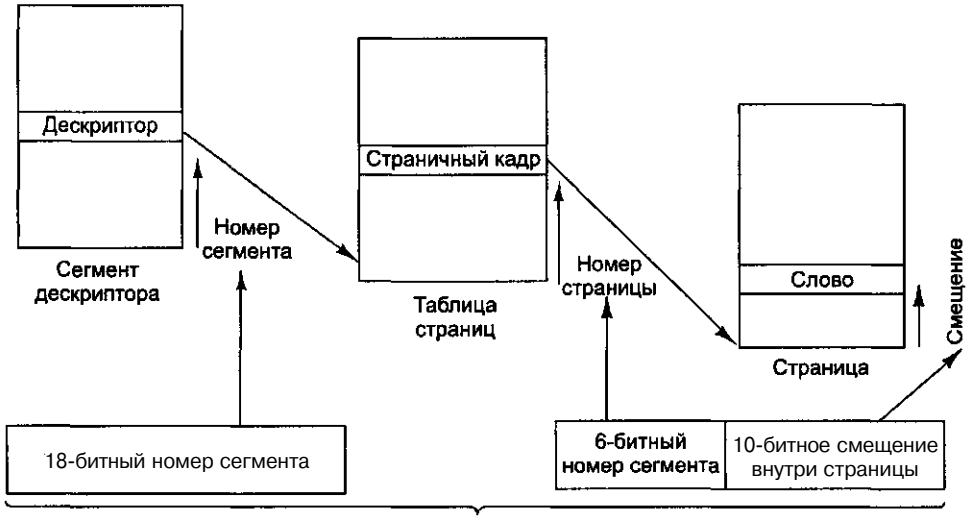


Рис. 6.9. Превращение адреса системы MULTICS, состоящего из двух частей, в адрес основной памяти

Виртуальная память в процессоре Pentium II

Pentium II имеет сложную систему виртуальной памяти, которая поддерживает вызов страниц по требованию, чистую сегментацию и сегментацию с разбиением на страницы. Виртуальная память состоит из двух таблиц: **LDT (Local Descriptor Table — локальная таблица дескрипторов)** и **GDT (Global Descriptor Table — глобальная таблица дескрипторов).** Каждая программа имеет свою собственную локальную таблицу дескрипторов, а единственная глобальная таблица дескрипторов разделяется всеми программами компьютера. Локальная таблица дескрипторов LDT описывает локальные сегменты каждой программы (ее код, данные, стек и т. д.), а глобальная таблица дескрипторов GDT описывает системные сегменты, в том числе саму операционную систему.

Как мы уже говорили в главе 5, чтобы получить доступ к сегменту, Pentium II сначала загружает селектор для сегмента в один из сегментных регистров. Во время выполнения программы регистр CS содержит селектор для сегмента кода, DS содержит селектор для сегмента данных и т. д. Каждый селектор представляет собой 16-битное число (рис. 6.10).

Один из битов селектора показывает, является сегмент локальным или глобальным (то есть в какой из двух таблиц он находится: в локальной таблице дескрипторов или в глобальной таблице дескрипторов). Еще 13 битов определяют номер элемента в локальной или глобальной таблице дескрипторов, поэтому объем каждой из этих таблиц ограничен до 8 K (2^{13}) сегментных дескрипторов. Оставшиеся два бита связаны с защитой. Мы опишем их позже.

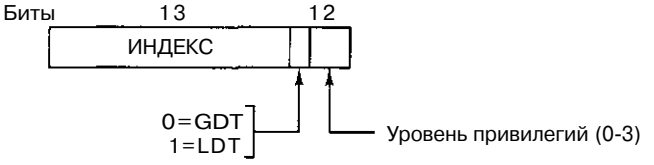


Рис. 6.10. Селектор в машине Pentium II

Дескриптор 0 недействителен и вызывает ловушку. Его можно загрузить в регистр сегмента, чтобы показать, что регистр сегмента в данный момент недоступен, но если попытаться использовать дескриптор 0, он вызовет ловушку.

Когда селектор загружается в сегментный регистр, соответствующий дескриптор вызывается из локальной таблицы дескрипторов или из глобальной таблицы дескрипторов и сохраняется во внутренних регистрах контроллера управления памятью, поэтому к нему можно быстро получить доступ. Дескриптор состоит из 8 байтов. Сюда входит базовый адрес сегмента, его размер и другая информация (рис. 6.11).

Формат селектора был выбран таким образом, чтобы упростить нахождение дескриптора. Сначала на основе бита 2 в селекторе выбирается локальная таблица дескрипторов LDT или глобальная таблица дескрипторов GDT. Затем селектор копируется во временный регистр контроллера управления памятью, а три младших бита принимают значение 0, в результате 13-битное число селектора умножается на 8. Наконец, к этому прибавляется адрес из локальной таблицы дескрипто-

ров или из глобальной таблицы дескрипторов (который хранится во внутренних регистрах контроллера управления памятью), и в результате получается указатель на дескриптор. Например, селектор 72 обращается к элементу 9 в глобальной таблице дескрипторов, который находится в ячейке с адресом GDT+72.

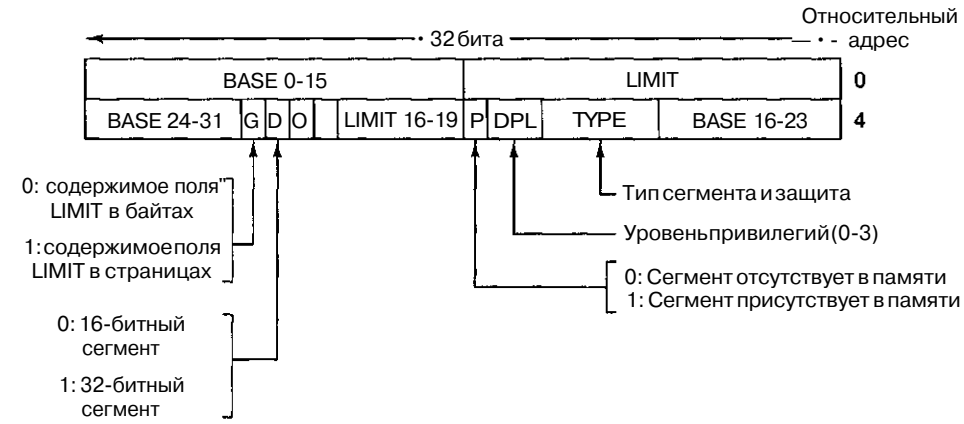


Рис. 6.11. Дескриптор сегмента кода в процессоре Pentium II. Сегменты данных практически ничем не различаются

Давайте проследим, каким образом пара (селектор, смещение) превращается в физический адрес. Как только аппаратное обеспечение определяет, какой именно регистр сегмента используется, оно может найти полный дескриптор, соответствующий данному селектору во внутренних регистрах. Если такого сегмента не существует (селектор 0) или в данный момент он не находится в памяти (P=0), вызывается системное прерывание (ловушка). В первом случае — это ошибка программирования; второй случай требует, чтобы операционная система вызвала нужный сегмент.

Затем аппаратное обеспечение проверяет, не выходит ли смещение за пределы сегмента. Если выходит, то снова происходит ловушка. По логике вещей в дескрипторе должно быть 32-битное поле для определения размера сегмента, но там в наличии имеется всего 20 битов, поэтому в данном случае используется совершенно другая схема. Если поле G (Granularity — степень детализации) равно 0, то поле LIMIT дает точный размер сегмента (до 1 Мбайт). Если поле G равно 1, то поле LIMIT указывает размер сегмента в страницах, а не в байтах. Размер страницы в компьютере Pentium II никогда не бывает меньше 4 Кбайт, поэтому 20 битов достаточно для сегментов до 2^{32} байтов.

Если сегмент находится в памяти, а смещение не выходит за границу сегмента, Pentium II прибавляет 32-битное поле BASE в дескрипторе к смещению, в результате чего получается **линейный адрес** (рис. 6.12). Поле BASE разбивается на три части и разносится по дескриптору, чтобы обеспечить совместимость с процессором 80286, в котором размер BASE составляет всего 24 бита. Поэтому каждый сегмент может начинаться с произвольного места в 32-битном адресном пространстве.

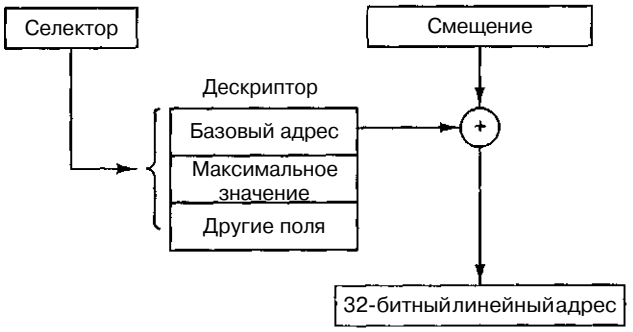


Рис. 6.12. Преобразование пары (селектор, смещение) в линейный адрес

Если разбиение на страницы блокировано (это определяется по биту в регистре глобального управления), линейный адрес интерпретируется как физический адрес и отправляется в память для чтения или записи. Таким образом, при блокировке разбиения на страницы мы имеем чистую схему сегментации, где каждый базовый адрес сегмента дан в его дескрипторе. Допускается перекрытие сегментов, поскольку было бы слишком утомительно тратить много времени на проверку, чтобы все сегменты были непересекающимися.

С другой стороны, если разбиение на страницы разрешено, линейный адрес интерпретируется как виртуальный адрес и отображается на физический адрес с использованием таблиц страниц, почти как в наших примерах. Единственная сложность состоит в том, что при 32-битном виртуальном адресе и страницах на 4 K сегмент может содержать 1 миллион страниц. Поэтому, чтобы сократить размер таблицы страниц для маленьких сегментов, применяется двухуровневое отображение.

Каждая работающая программа имеет специальную **таблицу страниц,** которая состоит из 1024 32-битных элементов. Ее адрес указывается глобальным регистром. Каждый элемент в этой таблице указывает на таблицу страниц, которая также содержит 1024 32-битных элементов. Элементы таблицы страниц указывают на страничные кадры. Схема изображена на рис. 6.13.

На рис. 6.13, ***а*** мы видим линейный адрес, разбитый на три поля: DIR, PAGE и OFF. Поле DIR используется в качестве индекса в директории страниц для нахождения указателя на нужную таблицу страниц. Поле PAGE используется в качестве индекса в таблице страниц для нахождения физического адреса страничного кадра. Наконец, поле OFF прибавляется к адресу страничного кадра, и получается физический адрес нужного байта или слова.

Размер каждого элемента таблицы страниц — 32 бита, 20 из которых содержат номер страничного кадра. Оставшиеся биты включают бит доступа и бит изменения, которые устанавливаются аппаратным обеспечением для помощи операционной системе, биты защиты и некоторые другие биты.

В каждой таблице страниц содержатся элементы для 1024 страничных кадров по 4 K каждый, поэтому одна таблица страниц работает с 4 Мбайт памяти. Сегмент короче 4 Мбайт будет иметь директорию страниц с одним элементом (указателем на его единственную таблицу страниц). Таким образом, непроизводительные издержки для коротких сегментов составляют всего две страницы, а не миллион страниц, как было бы в одноуровневой таблице страниц.

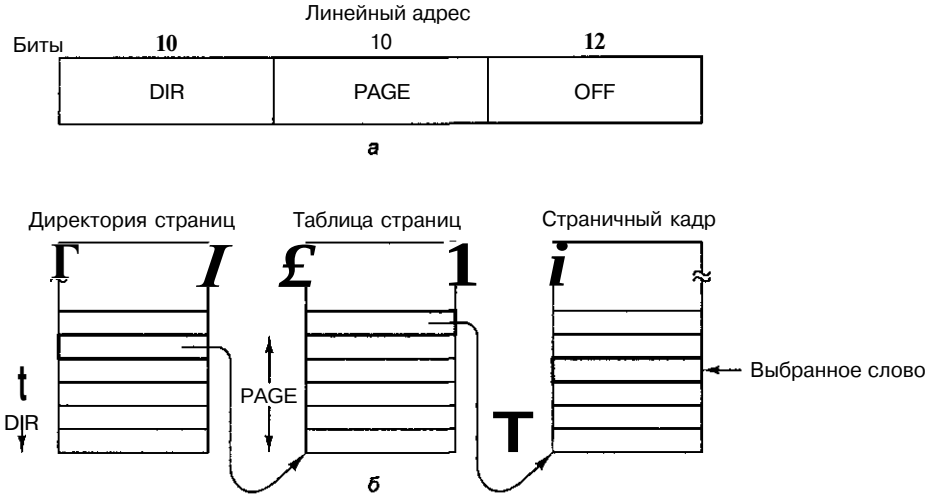


Рис. 6.13. Отображение линейного адреса на физический адрес

Чтобы избежать повторных обращений к памяти, устройство управления памятью Pentium II содержит специальную аппаратную поддержку для поиска недавно использовавшихся комбинаций DIR-PAGE и отображения их на физический адрес соответствующего страничного кадра. Шаги, показанные на рис. 6.13, выполняются только в том случае, если текущая комбинация не использовалась недавно.

При применении разбиения на страницы значение поля BASE в дескрипторе вполне может быть равно 0. Единственное, для чего нужно поле BASE, — это мотивировать небольшое смещение и использовать элемент в середине директории страниц, а не в начале. Поле BASE включено в дескриптор только для осуществления чистой сегментации (без разбиения на страницы), а также для обратной совместимости со старым процессором 80286, в котором не было разбиения на страницы.

Отметим, что если конкретное приложение не нуждается в сегментации и довольствуется единым 32-битным адресным пространством со страничной организацией, этого легко достичь. Все сегментные регистры могут быть заполнены одним и тем же селектором, дескриптор которого содержит поле BASE, равное 0, и поле LIMIT, установленное на максимальное значение. Смещение команды будет тогда линейным адресом с единственным адресным пространством — по сути, традиционное разбиение на страницы.

В завершение стоит сказать несколько слов о защите, поскольку это имеет непосредственное отношение к виртуальной памяти. Pentium II поддерживает 4 уровня защиты, где уровень 0 — самый привилегированный, а уровень 3 — наименее привилегированный. Они показаны на рис. 6.14. В каждый момент работающая программа находится на определенном уровне, указывающемся 2-битным полем в **PSW (Program Status Word — слово состояния программы)** — регистре аппаратного обеспечения, который содержит коды условия и другие биты состояния. Более того, каждый сегмент в системе также принадлежит к определенному уровню.

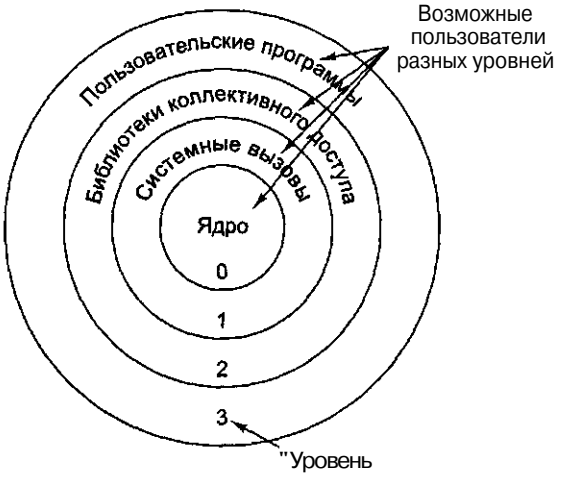


Рис. 6.14. Уровни защиты процессора Pentium II

Пока программа использует сегменты только своего собственного уровня, все идет нормально. Доступ к данным более высокого уровня разрешается. Доступ к данным более низкого уровня запрещен[1]: в этом случае происходит системное прерывание (ловушка). Допустим вызов процедур как более высокого, так и более низкого уровня, но при этом нужно вести строгий контроль. Для вызова процедуры из другого уровня команда CALL должна содержать селектор вместо адреса. Этот селектор обозначает дескриптор, который выдает адрес нужной процедуры. Таким образом, невозможно совершить переход в середину произвольного сегмента на другом уровне. Могут использоваться только официальные точки входа.

Рассмотрим рис. 6.14. На уровне 0 мы видим ядро операционной системы, которая контролирует процесс ввода-вывода, работу памяти и т. п. На уровне 1 находится обработчик системных вызовов. Пользовательские программы могут вызывать процедуры из этого уровня, но только строго определенные процедуры. Уровень 2 содержит библиотечные процедуры, которые могут разделяться несколькими работающими программами. Пользовательские программы могут вызывать эти процедуры, но не могут изменять их. На уровне 3 работают пользовательские программы, которые имеют самую низкую степень защиты. Система защиты Pentium II, как и схема управления памятью, в целом основана на идеях системы MULTICS.

Ловушки и прерывания используют механизм, сходный с описанным выше. Они тоже обращаются к дескрипторам, а не к абсолютным адресам, а эти дескрипторы указывают на процедуры, которые нужно выполнить. Поле FIELD на рис. 6.11 служит для различения сегментов кода, сегментов данных и различных типов логических элементов.

[1] Эти слова следует понимать следующим образом: автор называет более высоким уровнем те сегменты данных, у которых значение уровня привилегий больше, хотя на самом деле все обстоит иначе. Самым высоким уровнем привилегий считаются сегменты, имеющие уровень привилегий, равный О, а самый низкий уровень привилегий имеют сегменты со значением этого уровня, равным 3. — *Примеч. научн. ред.*

Виртуальная память UltraSPARC II

UltraSPARC II — это 64-разрядная машина, которая поддерживает виртуальную память со страничной организацией и с 64-битными виртуальными адресами. Тем не менее по разным причинам программы не могут использовать полное 64-битное виртуальное адресное пространство. Поддерживается только 64 бита, поэтому программы не могут превышать $1{,}8 \times 10^{13}$ байтов. Допустимая виртуальная память делится на 2 зоны по 2^{43} байтов каждая, одна из которых находится в верхней части виртуального адресного пространства, а другая — в нижней. Между ними находится «дырка», содержащая адреса, которые не используются. Попытка использовать их вызовет ошибку из-за отсутствия страницы.

Максимальная физическая память компьютера UltraSPARC II составляет 2^{41} байтов (2200 Гбайт). Поддерживается 4 размера страниц: 8 Кбайт, 64 Кбайт, 512 Кбайт и 4 Мбайт. Отображения этих четырех размеров показаны на рис. 6.15.

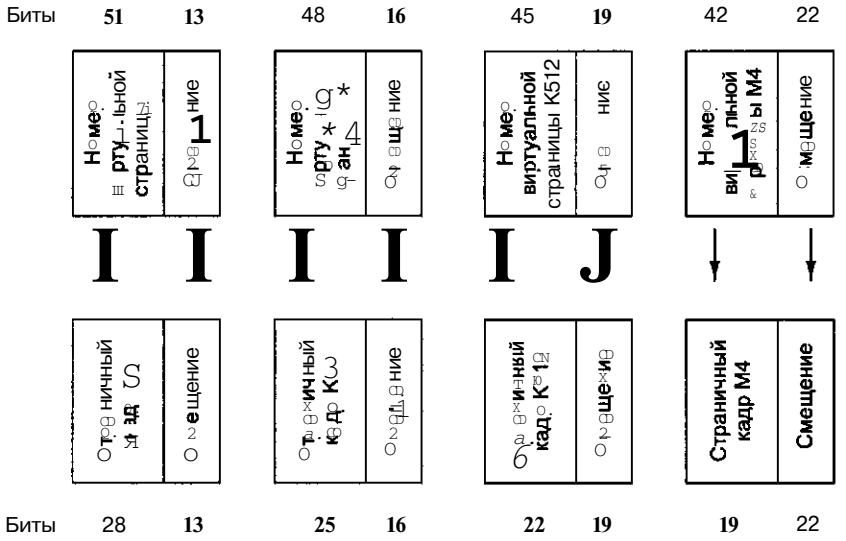


Рис. 6.15. Отображения виртуальных адресов в физические в машине UltraSPARC II

Из-за огромного виртуального адресного пространства обычная таблица страниц (как в Pentium II) не будет практичной. В UltraSPARC II применяется совершенно другой подход. Устройство управления памятью содержит таблицу, так называемый **TLB (Translation Lookaside Buffer — буфер быстрого преобразования адреса).** Эта таблица отображает номера виртуальных страниц в номера физических страничных кадров. Для страниц размером в 8 К существует 2^{31} номеров виртуальных страниц, то есть более двух миллиардов. Естественно, не все они могут быть отображены.

Поэтому TLB содержит только номера самых последних используемых виртуальных страниц. Страницы команд и страницы данных рассматриваются отдельно. Для каждой из этих категорий в TLB включены номера 64 последних виртуальных страниц. Каждый элемент этого буфера включает номер виртуальной страницы и соответствующий номер физического страничного кадра. Когда номер процесса

вызывает его контекст, виртуальный адрес в этом контексте передается в контроллер управления памятью, то он с помощью специальной схемы сравнивает номер виртуальной страницы со всеми элементами буфера быстрого преобразования адреса TLB для данного контекста одновременно. Если обнаруживается совпадение, номер страничного кадра в этом элементе буфера соединяется со смещением, взятым из виртуального адреса, чтобы получить 41-битный физический адрес и обработать некоторые флаги (например, биты защиты). Буфер быстрого преобразования адресаТЬВ изображен на рис. 6.16, *а.*

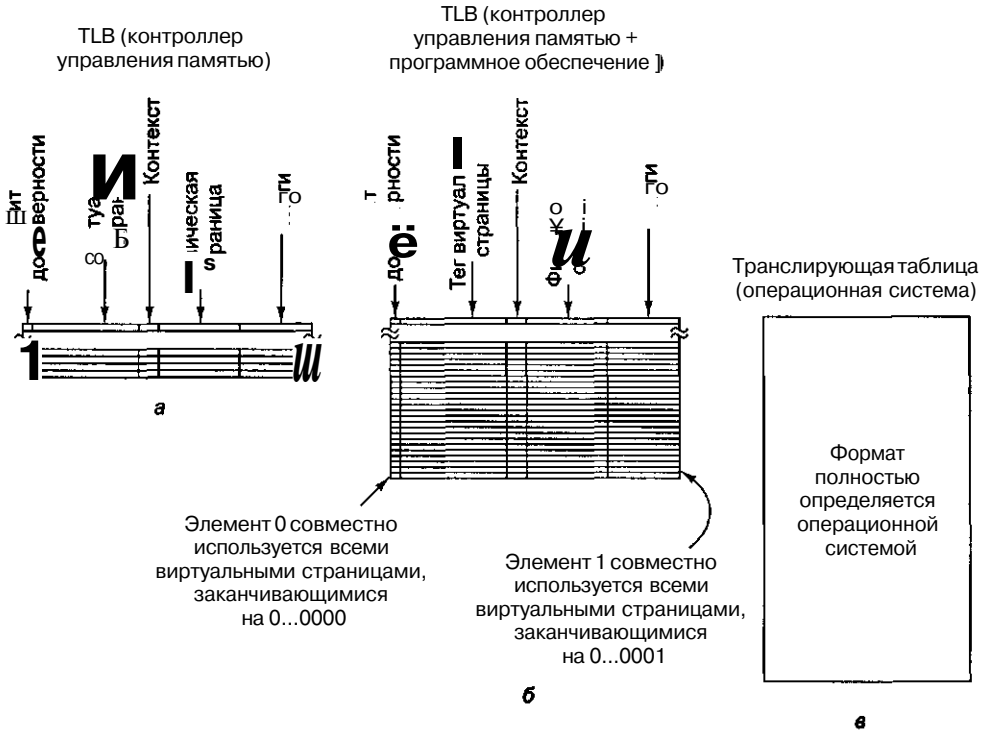


Рис. 6.16. Структуры данных, использующиеся для трансляции виртуального адреса на UltraSPARC II: буфер быстрого преобразования адреса TLB (а); буфер хранения преобразований (б); транслирующая таблица (в)

Если совпадение не обнаружилось, происходит **промах в TLB,** который вызывает ловушку в операционной системе. Обрабатывать ошибку должна сама операционная система. Отметим, что данный промах отличается от ошибки из-за отсутствия страницы. Промах буфера TLB может произойти, даже если нужная страница присутствует в памяти. Теоретически операционная система может сама загрузить новый элемент этого буфера для нужной виртуальной страницы. Однако для ускорения данной операции к этой работе подключается аппаратное обеспечение, если программное обеспечение взаимодействует с ним.

Операционная система должна сохранять наиболее часто используемые элементы буфера TLB в таблице под названием **буфер хранения преобразований**

(TSB — translation storage buffer). Эта таблица построена как кэш-память прямого отображения виртуальных страниц. Каждый 16-байтный элемент данной таблицы указывает на одну виртуальную страницу и содержит бит достоверности, номер контекста, тег виртуального адреса, номер физической страницы и несколько флаговых битов. Если размер кэш-памяти составляет, скажем, 8192 элемента, тогда все виртуальные страницы, у которых младшие 13 битов отображаются в 0000000000000, будут претендовать на элемент 0 в данной таблице. Точно так же все виртуальные страницы, у которых младшие биты отображаются в 0000000000001, претендуют на элемент 1 в этой таблице, как показано на рис. 6.16, *б*. Размер таблицы определяется программным обеспечением и передается в контроллер управления памятью через специальные регистры, доступные только для операционной системы.

При промахе буфера хранения преобразований операционная система проверяет, содержит ли соответствующий элемент буфера TLB нужную виртуальную страницу. Контроллер управления памятью вычисляет адрес этого элемента и помещает его в свой внутренний регистр, доступный для операционной системы. Если нужный элемент есть в таблице хранения преобразований, то какой-нибудь элемент удаляется из буфера TLB, а соответствующий элемент буфера хранения преобразований копируется туда. Аппаратное обеспечение с помощью алгоритма LRU выбирает, какой именно элемент нужно выкинуть.

Если нужной виртуальной страницы нет в кэш-памяти, операционная система использует другую таблицу для нахождения информации о странице, которая может находиться или не находиться в основной памяти. Таблица, которая применяется для этого поиска, называется **транслирующей таблицей.** Поскольку здесь аппаратное обеспечение не участвует в поиске элементов, операционная система может использовать любой формат. Например, она может хэшировать номер виртуальной страницы, разделив его на какое-либо число p, и использовать остаток для индексирования таблицы указателей, каждый из которых указывает на связный список виртуальных страниц, разделенных на p. Отметим, что эти элементы — не собственно страницы, а элементы таблицы TSB. Если поиск страницы в таблице трансляции привел к нахождению нужной страницы в памяти, то элемент TSB в кэш-памяти обновляется. Если в результате поиска обнаружилось, что нужной страницы нет в памяти, то происходит стандартная ошибка.

Сравним схемы разбиения на страницы в Pentium II и UltraSPARC И. Pentium II поддерживает чистую сегментацию, чистое разбиение на страницы и сегментацию в сочетании с разбиением на страницы. UltraSPARC II поддерживает только разбиение на страницы. Pentium II использует аппаратное обеспечение для перезагрузки элемента буфера TLB в случае промаха TLB. UltraSPARC II в случае такого промаха просто передает управление операционной системе.

Причина этого различия состоит в том, что Pentium II использует 32-битные сегменты, а такие маленькие сегменты (только 1 млн страниц) могут обрабатываться только с помощью страничных таблиц. Теоретически у Pentium II могли бы возникнуть проблемы, если бы программа использовала тысячи сегментов, но так как ни одна из версий Windows или UNIX не поддерживает более одного сегмента на процесс, никаких проблем не возникает. UltraSPARC II — 64-битная машина. Она может содержать до 2 млрд страниц, поэтому таблицы страниц не работают. В будущем все машины будут иметь 64-битные виртуальные адресные пространства, и схема UltraSPARC II станет нормой. Сравнение Pentium II, UltraSPARC II и четырех других схем виртуальной памяти можно найти в книге [66].

Виртуальная память и кэширование

На первый взгляд может показаться, что виртуальная память и кэширование никак не связаны, но на самом деле они сходны. При наличии виртуальной памяти вся программа хранится на диске и разбивается на страницы фиксированного размера. Некоторое подмножество этих страниц находится в основной памяти. Если программа главным образом использует страницы из основной памяти, то ошибки из-за отсутствия страницы будут встречаться редко и программа будет работать быстро. При кэшировании вся программа хранится в основной памяти и разбивается на блоки фиксированного размера. Некоторое подмножество этих блоков находится в кэш-памяти. Если программа главным образом использует блоки из кэш-памяти, то промахи кэш-памяти будут происходить редко и программа будет работать быстро. Как видим, виртуальная память и кэш-память идентичны, только работают они на разных уровнях иерархии.

Естественно, виртуальная память и кэш-память имеют некоторые различия. Промахи кэш-памяти обрабатываются аппаратным обеспечением, а ошибки из-за отсутствия страниц обрабатываются операционной системой. Блоки кэш-памяти обычно гораздо меньше страниц (например, 64 байта и 8 Кбайт). Кроме того, таблицы страниц индексируются по старшим битам виртуального адреса, а кэш-память индексируется по младшим битам адреса памяти. Тем не менее важно понимать, что различие здесь только в реализации.

Виртуальные команды ввода-вывода

Набор команд уровня архитектуры команд полностью отличается от набора команд микроархитектурного уровня. И сами операции, и форматы команд на этих двух уровнях различны. Наличие нескольких одинаковых команд случайно.

Набор команд уровня операционной системы содержит большую часть команд из уровня архитектуры команд, а также несколько новых очень важных команд. Некоторые ненужные команды в уровень операционной системы не включаются. Ввод-вывод — это одна из областей, в которых эти два уровня различаются очень сильно. Причина такого различия проста. Во-первых, пользователь, способный выполнять команды ввода-вывода уровня архитектуры команд, сможет считать конфиденциальную информацию, которая хранится где-нибудь в системе, и вообще будет представлять угрозу для самой системы. Во-вторых, обычные нормальные программисты не хотят осуществлять ввод-вывод на уровне команд, поскольку это слишком сложно и утомительно. Вместо этого для осуществления ввода-вывода нужно установить определенные поля и биты в ряде регистров устройств, затем подождать, пока операция закончится, и проверить, что произошло. Диски обычно содержат биты регистров устройств для обнаружения следующих ошибок.

1. Аппаратура диска не смогла выполнить позиционирование.
2. Несуществующий элемент памяти определен как буфер.
3. Процесс ввода-вывода с диска (на диск) начался до того, как закончился предыдущий.

4. Ошибка синхронизации при считывании.
5. Обращение к несуществующему диску.
6. Обращение к несуществующему цилиндру.
7. Обращение к несуществующему сектору.
8. Ошибка проверки записи после операции записи.

При наличии одной из этих ошибок устанавливается соответствующий бит в регистре устройства.

Файлы

Один из способов организации виртуального ввода-вывода — использование абстракции под названием файл. Файл состоит из последовательности байтов, записанных на устройство ввода-вывода. Если устройство ввода-вывода является устройством хранения информации (например, диск), то файл можно считать обратно. Если устройство не является устройством хранения информации (например, это принтер), то файл оттуда считать нельзя. На диске может храниться много файлов, в каждом из которых содержатся данные определенного типа, например картинка, крупноформатная таблица или текст. Файлы имеют разную длину и обладают разными свойствами. Эта абстракция позволяет легко организовать виртуальный ввод-вывод.

Для операционной системы файл является просто последовательностью байтов, как мы и описали выше. Ввод-вывод файла осуществляется с помощью системных вызовов для открытия, чтения, записи и закрытия файлов. Перед тем как считывать файл, его нужно открыть. Процесс открытия файла позволяет операционной системе найти файл на диске и передать в память информацию, необходимую для доступа к этому файлу.

После открытия файла его можно считывать. Системный вызов для считывания должен иметь как минимум следующие параметры:

1. Указание, какой именно открытый файл нужно считывать.
2. Указатель на буфер в памяти, в который нужно поместить данные.
3. Число считываемых байтов.

Данный системный вызов помещает требующиеся данные в буфер. Обычно он возвращает число считанных байтов. Это число может быть меньше того числа, которое запрашивалось изначально (например, нельзя считать 2000 байтов из файла размером 1000 байт).

С каждым открытым файлом связан указатель, который сообщает, какой байт будет считываться следующим. После команды read указатель дополняется числом считанных байтов, поэтому последовательные команды read считывают последовательные блоки данных из файла. Обычно этот указатель можно установить на особое значение, чтобы программы могли получать доступ к любой части файла. Когда программа закончила считывание файла, она может закрыть его и сообщить операционной системе, что она больше не будет использовать этот файл. Операционная система сможет освободить пространство в таблице, в которой хранилась информация об этом файле.

В операционных системах для универсальных вычислительных машин файл представляет собой более сложную структуру. Здесь файл может быть последовательностью **логических записей,** каждая из которых имеет строго определенную структуру. Например, логическая запись может представлять собой структуру данных, состоящую из пяти элементов: двух строк символов, «Имя» и «Начальник», двух целых чисел «Отдел» и «Комната», и одного логического числа «Пол женский». Некоторые операционные системы различают файлы, в которых все элементы имеют одинаковую структуру, и файлы, содержащие разные типы данных.

Основная виртуальная команда ввода считывает следующую запись из нужного файла и помещает ее в основную память, начиная с определенного адреса, как показано на рис. 6.17. Чтобы выполнить эту операцию, виртуальная команда должна получить сведения о том, какой файл считывать и куда в памяти поместить запись. Часто существуют параметры для чтения некоторой записи, которые определяются или по ее месту в файле, или по ее ключу.

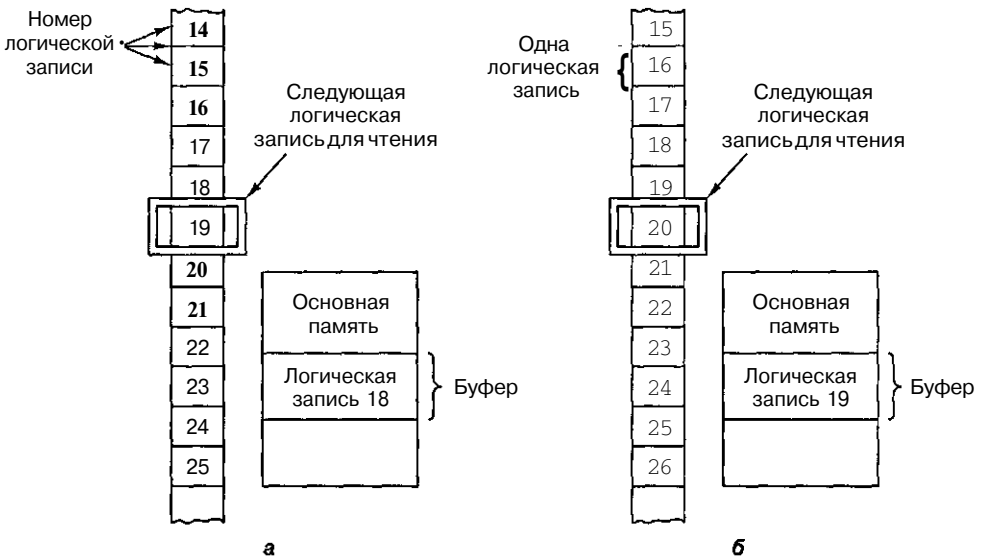


Рис. 6.17. Чтение файла, состоящего из логических записей: до чтения записи 19 (а); после чтения записи 19 (б)

Основная виртуальная команда вывода записывает логическую запись из памяти в файл. Последовательные команды **write** производят последовательные логические записи в файл.

Реализация виртуальных команд ввода-вывода

Чтобы понять, как реализуются виртуальные команды ввода-вывода, нужно изучить, как файлы организуются и хранятся. Основной вопрос здесь — распределение памяти. Единичным блоком может быть один сектор на диске, но чаще он состоит из нескольких последовательных секторов.

Еще одно фундаментальное свойство реализации системы файлов — хранится ли файл в последовательных блоках или нет. На рис. 6.18 изображен простой диск с одной поверхностью, состоящий из 5 дорожек по 12 секторов каждая. На рис. 6.18, а файл состоит из последовательных секторов. Последовательное расположение пяти блоков обычно применяется на компакт-дисках. На рис. 6.18, *б* файл занимает непоследовательные сектора. Такая схема является нормой для жестких дисков.

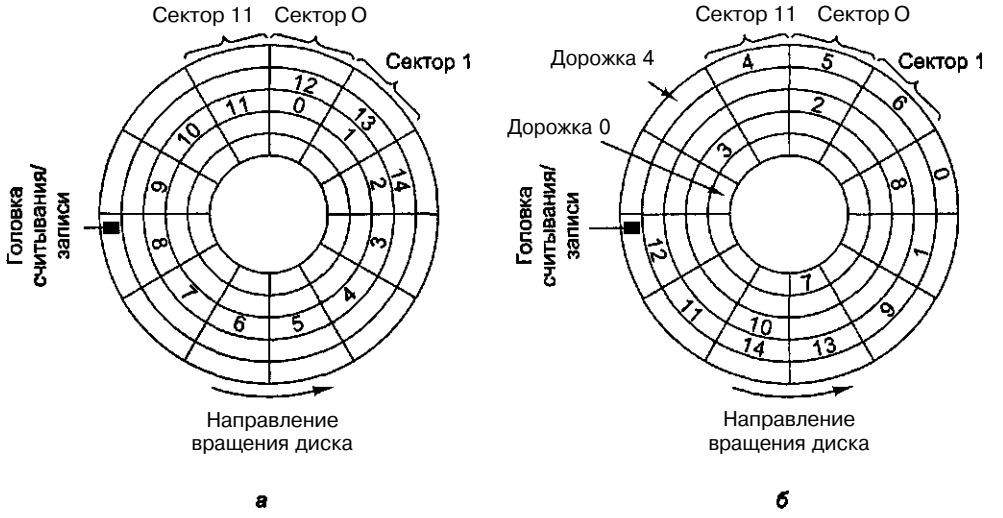


Рис. 6.18. Варианты расположения файла на диске' файл занимает последовательные сектора (а); файл занимает непоследовательные сектора (б)

Восприятие файла прикладным программистом сильно отличается от восприятия файла операционной системой. Программист воспринимает файл как линейную последовательность байтов или логических записей. Операционная система воспринимает файл как упорядоченную, хотя необязательно последовательную, совокупность единичных блоков на диске.

Для того чтобы операционная система могла доставить байт или логическую запись п из какого-то файла по требованию, она должна пользоваться каким-либо методом для определения местонахождения данных. Если файл расположен последовательно, операционная система должна знать только место начала файла, чтобы вычислить позицию нужного байта или логической записи.

Если файл расположен на диске непоследовательно, то невозможно только по начальной позиции файла вычислить позицию произвольного байта или логической записи в этом файле. Чтобы найти произвольный байт или логическую запись, нужна таблица (так называемый **индекс файла),** которая выдает единичные блоки и их физические адреса на диске. Индекс файла может быть организован либо в виде списка адресов блоков (такая схема используется в UNIX), либо в виде списка логических записей, для каждой из которых дается адрес на диске и смещение. Иногда каждая логическая запись имеет **ключ,** и программы могут обращаться к записи по этому ключу, а не по номеру логической записи. В последнем случае

каждый элемент таблицы должен содержать не только информацию о местонахождении записи на диске, но и ее ключ. Такая структура обычно применяется в универсальных вычислительных машинах.

Альтернативный метод нахождения блоков файла — организовать файл в виде связного списка. Каждый единичный блок содержит адрес следующего единичного блока. Для реализации этой схемы нужно в основной памяти иметь таблицу со всеми последующими адресами. Например, для диска с 64 K блоками операционная система может иметь в памяти таблицу из 64 K элементов, в каждом из которых дается индекс следующего единичного блока. Так, если файл занимает блоки 4,52 и 19, то элемент 4 в таблице будет содержать число 52, элемент 52 будет содержать число 19, а элемент 19 будет содержать специальный код (например, 0 или -1), который указывает на конец файла. Так работают системы файлов в MS DOS, Windows 95 и Windows 98. Windows NT поддерживает эту систему файлов, но кроме этого имеет свою собственную систему файлов, которая больше похожа на UNIX.

До сих пор мы обсуждали как последовательно расположенные, так и непоследовательно расположенные на диске файлы, но мы еще не объяснили, зачем нужны эти два типа расположения. Последовательно расположенными файлами легко управлять, но если максимальный размер файла не известен заранее, эту технологию нельзя использовать. Если файл начинается с сектора j и разрастается в последовательные сектора, он может наткнуться на другой файл в секторе к, и ему не хватит места на расширение. Если файл располагается непоследовательно, то таких проблем не возникает, поскольку следующие блоки можно поместить в другое место на диске. Если диск содержит ряд увеличивающихся файлов, конечные размеры которых неизвестны, записать их в последовательные блоки на диске практически невозможно. Иногда можно переместить существующий файл, но это очень накладно.

С другой стороны, если максимальный размер всех файлов известен заранее (например, это имеет место, когда создается компакт-диск), программа может заранее определить серии секторов, точно равных по длине каждому файлу. Если файлы по 1200, 700, 2000 и 900 секторов нужно поместить на компакт-диск, они просто могут начинаться с секторов 0,1200,1900 и 3900 соответственно (оглавление здесь не учитывается). Найти любую часть любого файла легко, поскольку известен первый сектор файла.

Чтобы распределить пространство на диске для файла, операционная система должна следить, какие блоки доступны, а какие уже заняты другими файлами. При записи на компакт-диск вычисление производится один раз и навсегда, а на жестком диске файлы постоянно записываются и удаляются. Один из способов — сохранить список всех «дырок» (неиспользованных пространств), где «дырка» — это любое число смежных единичных блоков. Этот список называется **списком свободной памяти.** На рис. 6.19, а изображен список свободной памяти для диска с рис. 6.18, *б*.

Альтернативный подход — сохранить битовое отображение, один бит на единичный блок, как показано на рис. 6.19, *б*. Бит со значением 1 показывает, что данный блок уже занят, а бит со значением 0 показывает, что данный блок свободен.

Дорожка	Сектор	Количество секторов в дырке
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

а

Дорожка	Сектор											
	0	1	0	1	0	1	0	1	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

б

Рис. 6.19. Два способа отслеживания свободных секторов: список свободной памяти (а); битовое отображение (б)

Первый подход позволяет легко находить дырку определенной длины. Однако у этого метода есть недостаток: по мере создания и уничтожения файлов длина списка будет меняться, а это нежелательно. Преимущество таблицы битов состоит в том, что она имеет постоянный размер. Кроме того, для изменения статуса единичного блока из свободного в занятый нужно поменять значение всего одного бита. Однако при таком подходе трудно найти блок данного размера. Оба метода требуют, чтобы при записи файла на диск или удалении файла с диска список размещения или таблица обновлялись.

Перед тем как закончить обсуждение вопроса о реализации системы файлов, нужно сказать пару слов о размере единичного блока. Здесь играют роль несколько факторов. Во-первых, время поиска и время, затрачиваемое на вращение диска, затормаживают доступ к диску. Если на нахождение начала блока тратится 10 млс, то гораздо выгоднее считать 8 Кбайт (это займет примерно 1 млс), чем 1 Кбайт (это займет примерно 0,125 миллисекунды), так как если считывать 8 Кбайт как 8 блоков по 1 Кбайт, нужно будет осуществлять поиск 8 раз. Для повышения производительности нужны большие единичные блоки.

Чем меньше размер единичного блока, тем больше их должно быть. Большое количество единичных блоков, в свою очередь, влечет за собой длинные индексы файлов и большие таблицы в памяти. Системе MS DOS пришлось перейти на многосекторные блоки по той причине, что дисковые адреса хранились в виде 16-битных чисел. Когда размер дисков стал превышать 64 K секторов, представить их можно было, только используя единичные блоки большего размера, поэтому число таких блоков не превышало 64 K. В первом выпуске системы Windows 95 возникла та же проблема, но в последующем выпуске уже использовались 32-битные числа. Система Windows 98 поддерживает оба размера.

Маленькие блоки тоже имеют свои преимущества. Дело в том, что файлы очень редко занимают ровно целое число единичных блоков. Следовательно, практически в каждом файле в последнем единичном блоке останется неиспользованное пространство. Если размер файла сильно превышает размер единичного блока, то в среднем неиспользованное пространство будет составлять половину блока. Чем

больше блок, тем больше остается неиспользованного пространства. Если средний размер файла намного меньше размера единичного блока, большая часть пространства на диске будет неиспользованной. Например, в системе MS DOS или в первой версии Windows 95 с разделом диска в 2 Гбайт размер единичного блока составляет 32 Кбайт, поэтому при записи на диск файла в 100 символов 32 668 байтов дискового пространства пропадут. С точки зрения распределения дискового пространства маленькие единичные блоки имеют преимущество над большими. В настоящее время самым важным фактором считается быстрота передачи данных, поэтому размер блоков постоянно увеличивается.

Команды управления директориями

Много лет назад программы и данные хранились на перфокартах. Поскольку размер программ увеличивался, а данных становилось все больше, такая форма хранения стала неудобной. Тогда возникла идея вместо перфокарт использовать для хранения программ и данных вспомогательную память (например, диск). Информация, доступная для компьютера без вмешательства человека, называется **неавтономной. Автономная** информация, напротив, требует вмешательства человека (например, нужно вставить компакт-диск).

Неавтономная информация хранится в файлах. Программы могут получить доступ к ней через программы ввода-вывода. Чтобы следить за информацией, записанной неавтономно, группировать ее в удобные блоки и защищать от незаконного использования, нужны дополнительные команды.

Обычно операционная система группирует неавтономные файлы в **директории.** На рис. 6.20 проиллюстрирован пример такой организации. Обеспечиваются системные вызовы, по крайней мере, для следующих функций:

1. Создание файла и введение его в директорию.
2. Стирание файла из директории.
3. Переименование файла.
4. Изменение статуса защиты файла.

Применяются различные схемы защиты. Например, владелец файлов может ввести секретный пароль для каждого файла. При попытке доступа к файлу программа должна выдавать пароль, который проверяется операционной системой. Доступ к файлу разрешается только в том случае, если пароль правильный. Для каждого файла можно создать список людей, программы которых могут получать доступ к данному файлу.

Все современные операционные системы позволяют хранить более одной директории. Каждая директория обычно сама представляет собой файл и как таковая может быть вставлена в другую директорию, в результате чего можно получить дерево директорий. Большое количество директорий особенно нужно программистам, которые работают над несколькими проектами. Они могут сгруппировать в одну директорию все файлы, связанные с одним проектом. Директории — это удобный способ делить файлы с членами своих рабочих групп.

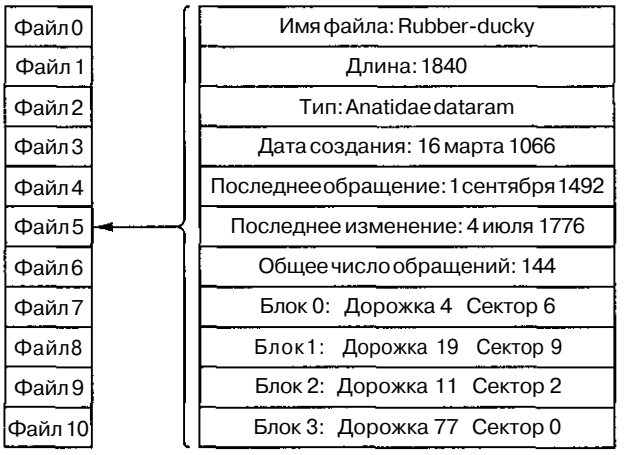


Рис. 6.20. Пользовательская директория (а); содержание одного из элементов директории (б)

Виртуальные команды для параллельной обработки

Некоторые вычисления удобно производить с помощью двух и более параллельных процессов (то есть как будто бы на разных процессорах). Другие вычисления можно поделить на части, которые затем выполняются параллельно, что сокращает общее время вычисления. Чтобы несколько процессов могли происходить параллельно, нужны специальные виртуальные команды. Мы будем их обсуждать в следующих разделах.

Интерес к параллельной обработке обусловлен и некоторыми законами физики. Согласно эйнштейновской теории относительности, скорость передачи электрических сигналов не может превышать скорость света, которая равна примерно 1 фут/нс в вакууме, а в медном проводе или оптическом волокне — еще меньше. Важно учитывать этот предел при разработке компьютеров. Например, если процессору нужны данные из основной памяти, которые находятся на расстоянии 1 фута, потребуется по крайней мере 1 не, чтобы запрос дошел до памяти, и еще 1 не, чтобы ответ венулся к центральному процессору. Следовательно, чтобы компьютеры могли передавать сигналы быстрее, они (компьютеры) должны быть совершенно крошечными. Альтернативный способ увеличения скорости компьютера — создание машины с несколькими процессорами. Компьютер, содержащий 1000 процессоров с временем цикла в 1 не, будет иметь такую же мощность, как процессор с временем цикла 0,001 не, но первое осуществить гораздо проще и дешевле.

В компьютере с несколькими процессорами каждый из нескольких взаимодействующих процессов можно приписать к одному определенному процессору, чтобы выполнять несколько действий одновременно. Если в компьютере имеется только один процессор, эффект параллельной обработки можно смоделировать. Для этого процессы будут выполняться по очереди, каждый понемножку. Иными словами, процессор будет разделяться между несколькими процессами.

На рис. 6.21 показана разница между реальной параллельной обработкой, когда присутствует несколько физических процессоров, и смоделированной параллельной обработкой, когда имеется всего один физический процессор. Даже если параллельная обработка моделируется, удобно считать, что каждому процессу приписывается его собственный виртуальный процессор. При моделированной параллельной обработке возникают те же проблемы, что и при реальной параллельной обработке.

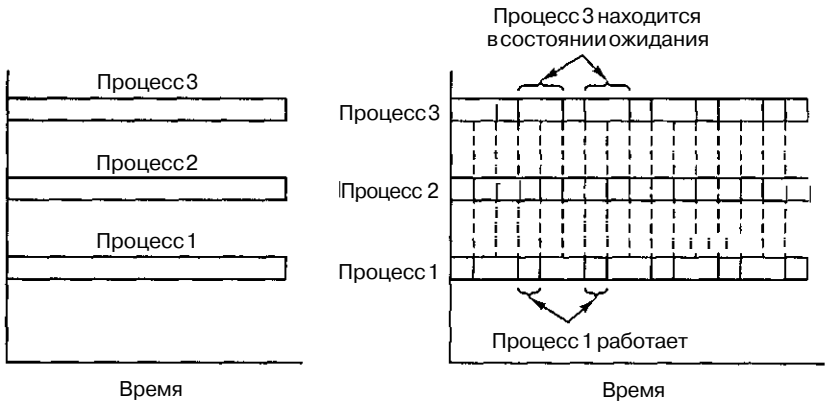


Рис. 6.21. Параллельная обработка с несколькими процессорами (а); моделирование параллельной обработки путем переключения одного процессора с одного процесса на другой (в данном случае всего три процесса) (б)

Формирование процесса

Программа должна работать как часть какого-либо процесса. Этот процесс, как и все другие процессы, характеризуется состоянием и адресным пространством, через которое можно получить доступ к программам и данным. Состояние включает как минимум счетчик команд, слово состояния программы, указатель стека и регистры общего назначения.

Большинство современных операционных систем позволяют формировать и прерывать процессы динамически. Для формирования нового процесса требуется системный вызов. Этот системный вызов может просто создать клон вызывающей программы или позволить исходному процессу определить начальное состояние нового процесса, то есть его программу, данные и начальный адрес.

В одних случаях исходный процесс сохраняет частичный или даже полный контроль над порожденным процессом. Виртуальные команды позволяют исходному процессу останавливать и снова запускать, проверять и завершать подчиненные процессы. В других случаях исходный процесс никак не контролирует порожденный процесс: после того как новый процесс сформирован, исходный процесс не может его остановить, запустить заново, проверить или завершить. Таким образом, эти два процесса работают независимо друг от друга.

Состояние гонок

Во многих случаях параллельные процессы должны взаимодействовать, и их работу нужно синхронизировать. В этом разделе мы рассмотрим синхронизацию и некоторые трудности, связанные с ней. Способы разрешения этих трудностей будут представлены в следующем разделе.

Рассмотрим два независимых процесса, процесс 1 и процесс 2, которые взаимодействуют через общий буфер в основной памяти. Для простоты мы будем называть процесс 1 **производителем** (producer), а процесс 2 — **потребителем** (consumer). Производитель выдает простые числа и помещает их в буфер по одному. Потребитель удаляет их из буфера по одному и печатает.

Эти два процесса работают параллельно с разной скоростью. Если производитель обнаруживает, что буфер заполнен, он переходит в режим ожидания, то есть временно приостанавливает операцию и ожидает сигнала от потребителя. Когда потребитель удаляет число из буфера, он посылает сигнал производителю, чтобы тот возобновил работу. Если потребитель обнаруживает, что буфер пуст, он приостанавливает работу. Когда производитель помещает число в пустой буфер, он посылает соответствующий сигнал потребителю.

В нашем примере для взаимодействия процессов мы будем использовать кольцевой буфер. Указатели *in* и *out* будут использоваться следующим образом: *in* указывает на следующее свободное слово (куда производитель поместит следующее простое число), а *out* указывает на следующее число, которое должен удалить потребитель.

Если *in=out,* буфер пуст, как показано на рис. 6.22, *а.* На рис. 6.22, б показана ситуация после того, как производитель породил несколько простых чисел. На рис. 6.22, *в* изображен буфер после того, как потребитель удалил из него несколько простых чисел для печати. На рис. 6.22, *г — е* представлены промежуточные стадии работы буфера. Буфер заполняется по кругу.

Если в буфер было отправлено слишком много чисел и он начал заполняться по второму кругу (снизу), а под *out* есть только одно слово (например, ii=52, а *out*=53), буфер заполнится. Последнее слово не используется; если бы оно использовалось, то не было бы возможности сообщить, что именно значит равенство *in=out* — полный буфер или пустой буфер.

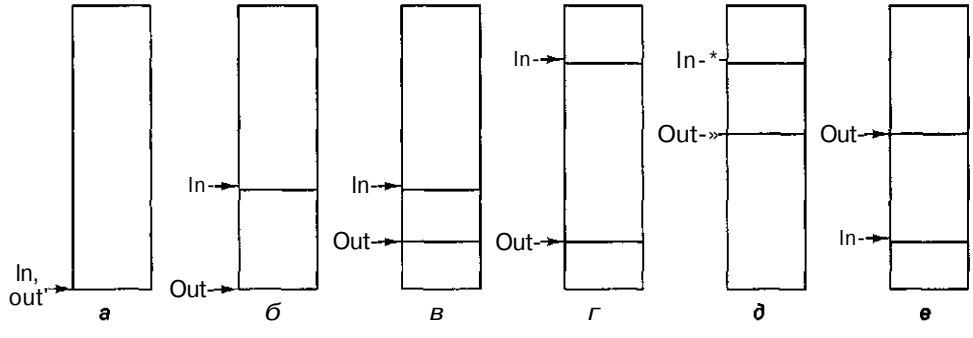


Рис. 6.22. Кольцевой буфер

В листинге 6.1 на языке Java показано решение задачи с производителем и потребителем.

Здесь используются три класса: *m,produser* и *consumer.* Класс ***m*** содержит некоторые константы, указатели буфера *in* и *out* и сам буфер, который в нашем примере вмещает 100 простых чисел (от buffer[O] до buffer[99]).

Для моделирования параллельных процессов в данном случае используются **потоки (threads).** У нас есть класс *producer* и класс *consumer,* которым приписываются значения переменных *p* и *c* соответственно. Каждый из этих классов образуется из базового класса *Thread c* процедурой *run.* Класс *run* содержит код для *thread.* Когда вызывается процедура *start* для объекта, образованного из *Thread,* запускается новый поток.

Каждый поток похож на процесс. Единственным различием является то, что все потоки в пределах одной программы на языке Java работают в одном адресном пространстве. Это позволяет им разделять один общий буфер. Если в компьютере имеется два и более процессоров, каждый поток может выполняться на другом процессоре, поэтому в данном случае имеет место реальный параллелизм. Если компьютер содержит только один процессор, потоки разделяются во времени на одном процессоре. Мы будем продолжать называть производителя и потребителя процессами (поскольку нас в данный момент интересуют параллельные процессы), хотя Java поддерживает только параллельные потоки, а не реальные параллельные процессы.

Функция *next* позволяет увеличивать значения *in* и *out,* при этом не нужно каждый раз записывать код, чтобы проверить условие циклического возврата. Если параметр в *next* равен 98 или указывает на более низкое значение, то возвращается следующее по порядку целое число. Если параметр равен 99, это значит, что мы наткнулись на конец буфера, поэтому возвращается 0.

Должен быть способ «усыплять» любой из процессов, если он не может продолжаться. Для этого разработчики Java включили в класс *Thread* специальные процедуры *suspend* (отключение) и *resume* (возобновление). Они используются в листинге 6.1.

А теперь рассмотрим саму программу для производителя и потребителя. Сначала производитель порождает новое простое число (шаг P1). Обратите внимание на строку mMAX_PRIME. Префикс т. здесь указывает, что имеется в виду MAX_PRIME, определенный в классе *m.* По той же причине этот префикс нужен для *in, out, buffer и next.*

Затем производитель проверяет (шаг P2), не находится ли *in* ниже *out.* Если да (например, iш=62 и *out=63),* то буфер заполнен и производитель вызывает процедуру *suspend* в P2. Если буфер не заполнен, туда помещается новое простое число (шаг P3) и значение *in* увеличивается (шаг P4). Если новое значение *in* на 1 больше значения *out* (шаг P5) (например, iш=17, *out=l6),* значит, *in* и *out* были равны перед тем, как увеличилось значение *in.* Производитель делает вывод, что буфер был пуст и что потребитель не функционировал (находился в режиме ожидания) и в данный момент тоже не функционирует. Поэтому производитель вызывает процедуру *resume,* чтобы возобновить работу потребителя (шаг P5). Наконец, производитель начинает искать следующее простое число.

Листинг 6.1. Параллельная обработка с состоянием гонок

```
public class m {
    final public static int BUF_SIZE = 100;                    // буфер от 0 до 99
    final public static long MAX_PRIME=100000000000L;          //остановиться здесь
    public static int in = 0, out = 0.                         // указатели на данные
    public static long buffer[ ] - new long[BUF_SIZE];         //простые числа хранятся здесь
    public static producer p.                                  //имя производителя
    public static consumer c;                                  //имя потребителя

    public static void mam(Stnng args[ ]){                     // основной класс
          p = new producer );                                  //создание производителя
          c = new consumer ),                                  //создание потребителя
          p startO:                                            //запуск производителя
          c startO.                                            //запуск потребителя
       }
       //Это утилита для циклического увеличения m и out
       public static int next(int k) {if (k < BUF_SIZE - 1) return(k+l). else return(O):}
}
class producer extends Thread {                                //класс производителя
       public void runO {                                      //код производителя
          long prime = 2;                                      // временная переменная

          while (prime < m.MAX_PRIME) {
             prime = next_pnme(pnme):                          //шаг P1
             if (m next(m.m) == m.out) suspendO:               //шаг P2
             m buffer[m. in] = prime;                          //шаг P3
             m in = m.next(m in).                              //шаг P4
             if (m next(m out) = m in) m c.resumeO.            //шаг P5
          }
       }

       private long next_pnme(long pnme){  ..}     //функция, вычисляющая следующее число

}
class consumer extends Thread {                                //класс потребителя
       public void run() {                                     // код потребителя
          long emirp = 2;                                      // временная переменная
          while (emirp < m MAX_PRIME) {
             if (m in == m out) suspend0:                      //шагС1
             emirp = m buffer[m.out]:                          //шагС2
             m.out = m next(m out);                            //шагС3
             if (m.out – m.next(m next(m.in))) m p.resume0;    //шагС4
             System out print!n(emirp).                        //шагС5
          }
       }
}
```

Программа потребителя по структуре очень проста. Сначала производится проверка (шаг С1), чтобы узнать, пуст ли буфер. Если он пуст, то потребителю ничего не нужно делать, поэтому он отключается. Если буфер не пуст, то потребитель удаляет из него следующее число для печати (шаг С2) и увеличивает значение *out*. Если после этого *out* стало на две позиции выше *in,* значит, до этого *out* было на одну позицию выше *in*. Так как *in=out-* — это условие заполненности буфера, значит, производитель не работает и потребитель должен вызвать процедуру *resume*. После этого число выводится на печать, и весь цикл повторяется снова.

К сожалению, такая программа содержит ошибку (рис. 6.23). Напомним, что эти два процесса работают асинхронно и с разными скоростями, которые, к тому же, могут меняться. Рассмотрим случай, когда в буфере осталось только одно число в элементе 21, и in=22, а *out=2i* (см. рис. 6.23, *а).* Производитель на шаге Р1 ищет простое число, а потребитель на шаге С5 печатает число из позиции 20. Потребитель заканчивает печатать число, совершает проверку на шаге С1 и забирает последнее число из буфера на шаге С2. Затем он увеличивает значение *out.* В этот момент и *in* и out равны 22. Потребитель печатает число, а затем переходит к шагу С1, на котором он вызывает *in* и *out* из памяти, чтобы сравнить их (рис. 6.23, б).

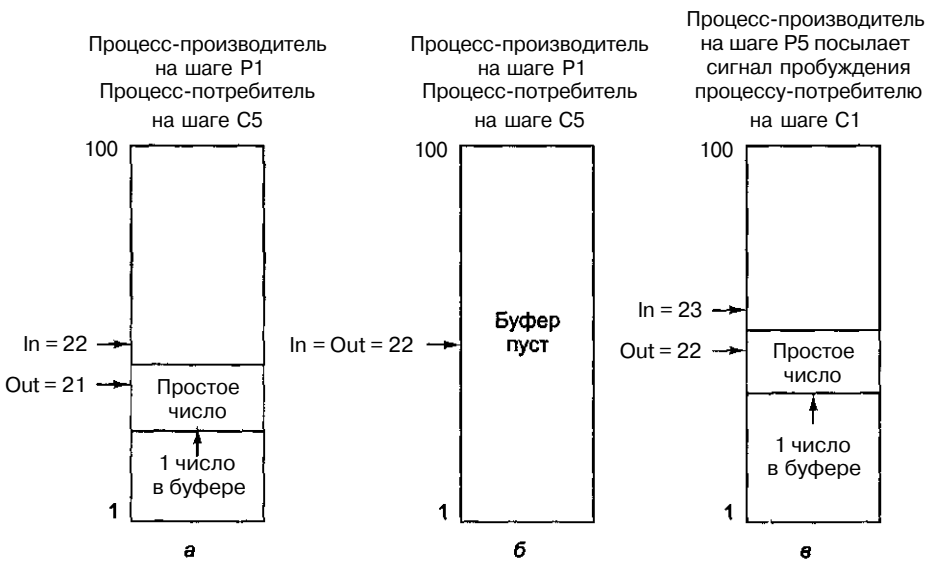


Рис. 6.23. Ситуация, при которой механизм взаимодействия производителя ипотребителя не работает

В этот момент, после того как потребитель вызвал *in* и *out,* но еще до того как он сравнил их, производитель находит следующее простое число. Он помещает это простое число в буфер на шаге Р3 и увеличивает *in* на шаге Р4. Теперь in=23, а out=22. На шаге Р5 производитель обнаруживает, что *in=*next(out).* Иными словами, *in* на единицу больше *out,* а это значит, что в буфере в данный момент находится один элемент.

Исходя из этого, производитель делает неверный вывод, что потребитель отключен, и вызывает процедуру *resume* (рис. 6.23, *в).* На самом деле потребитель все это время продолжал работать, поэтому вызов процедуры *resume* оказался ложным. Затем производитель начинает искать следующее простое число.

В этот момент потребитель продолжает работать. Он уже вызвал *in* и *out* из памяти, перед тем как производитель поместил последнее число в буфер. Так как in=22 и *out=22,* потребитель отключается. К этому моменту производитель находит следующее простое число. Он проверяет указатели и обнаруживает, что in=24, а *out=22.* Из этого он делает заключение, что в буфере находится 2 числа (что соответствует действительности) и что потребитель функционирует (что неверно).

Производитель продолжает цикл. В конце концов он заполняет буфер и отключается. Теперь оба процесса отключены и будут находиться в таком состоянии до скончания веков.

Сложность здесь в том, что между моментом, когда потребитель вызывает *in* и *out,* и моментом, когда он отключается, производитель, обнаружив, что *in=out+\,* и предположив, что потребитель отключен (хотя на самом деле он еще продолжает функционировать), вызывает процедуру *resume,* чего не нужно делать, поскольку потребитель функционирует. Такая ситуация называется **состоянием гонок,** поскольку успех процедуры зависит от того, кто выиграет гонку по проверке *in* и *out,* после того как значение *out* увеличилось.

Проблема состояния гонок хорошо известна. Она была настолько серьезна, что через несколько лет после появления Java компания Sun изменила класс *Thread* и убрала вызовы процедур *suspend* и *resume,* поскольку они очень часто приводили к состоянию гонок. Предложенное решение было основано на изменении языка, но поскольку мы изучаем операционные системы, мы обсудим другое решение, которое используется во многих операционных системах, в том числе UNIX и NT.

Синхронизация процесса с использованием семафоров

Проблему состояния гонок можно разрешить по крайней мере двумя способами. Первый способ — снабдить каждый процесс специальным битом ожидания пробуждения. Если процесс, который функционирует в данный момент, получает сигнал «пробуждения», то этот бит устанавливается. Если процесс отключается в тот момент, когда этот бит установлен, он немедленно перезапускается, а бит сбрасывается. Данный бит сохраняет сигнал пробуждения для будущего использования.

Этот метод решает проблему состояния гонок только в том случае, если у нас есть всего 2 процесса. В общем случае при наличии n процессов он не работает. Конечно, каждому процессу можно приписать n -1 таких битов ожидания пробуждения, но это неудобно.

Дейкстра [31] предложил другое решение этой проблемы. Где-то в памяти находятся две переменные, которые могут содержать неотрицательные целые числа. Эти переменные называются **семафорами.** Операционная система предоставляет два системных вызова, *up* и *down,* которые оперируют семафорами. *Up* прибавляет 1 к семафору, а *down* отнимает 1 от семафора. Если операция *down* совершается над семафором, значение которого больше 0, этот семафор уменьшается на 1 и процесс продолжается. Если значение семафора равно 0, то операция *down* не может завершиться. Тогда данный процесс отключается до тех пор, пока другой процесс не выполнит операцию *up* над этим семафором.

Команда *up* проверяет, не равен ли семафор нулю. Если он равен 0 и другой процесс находится в режиме ожидания, то семафор увеличивается на 1. После этого процесс, который «спит», может завершить операцию *down,* установив семафор на 0. Теперь оба процесса могут продолжать работу. Если семафор не равен 0, команда *up* просто увеличивает его на 1. Семафор позволяет сохранять сигналы пробуждения, так что они не пропадут зря. У семафорных команд есть одно важное свойство: если один из процессов начал выполнять команду над семафором, то

другой процесс не может получить доступ к этому семафору до тех пор, пока первый не завершит выполнение команды или не будет приостановлен при попытке выполнить команду *down* над 0. В таблице 6.5 изложены важные свойства системных вызовов *up* и *down*.

Таблица 6.5. Результаты операций над семафором

Команда	Семафор = 0	Семафор > 0
Up	Семафор = семафор + 1; если другой процесс пытается совершить команду down над этим семафором, теперь он сможет это сделать и продолжить свою работу	Семафор = семафор +1
Down	Процесс останавливается до тех пор, пока другой процесс не выполнит операцию up над этим семафором	Семафор = семафор -1

Как мы уже сказали, в языке Java предусмотрено свое решение проблемы состояния гонок, но мы сейчас обсуждаем операционные системы. Следовательно, нам нужно каким-либо образом выразить использование семафоров в языке Java. Мы предположим, что были написаны две процедуры, *up* и *down,* которые совершают системные вызовы *up* и *down* соответственно. Используя в качестве параметров обычные целые числа, мы сможем выразить применение семафоров в программах на языке Java.

В листинге 6.2 показано, как можно устранить состояние гонок с помощью семафоров. В класс *m* добавляются два семафора: *available,* который изначально равен 100 (это размер буфера), *w.filled,* который изначально равен 0. Производитель начинает работу с шага P1, а потребитель — с шага C1. Выполнение процедуры *down* над семафором *filled* сразу же приостанавливает работу потребителя. Когда производитель находит первое простое число, он вызывает процедуру *down* с *available* в качестве параметра, устанавливая *available* на 99. На шаге P5 он вызывает процедуру *up* с параметром *filled,* устанавливая *filled* на 1. Это действие освобождает потребителя, который теперь может завершить вызов процедуры *down.* После этого *filled* принимает значение 0, и оба процесса продолжают работу.

А теперь давайте еще раз рассмотрим состояние гонок. В определенный момент iш=22, а *out=2\,* производитель находится на шаге P1, а потребитель — на шаге C5. Потребитель завершает действие и переходит к шагу C1, который вызывает процедуру *down,* чтобы выполнить ее над семафором *filled,* который до вызова имел значение 1, а после вызова принял значение 0. Затем потребитель берет последнее число из буфера и выполняет процедуру *up* над *available,* после чего *available* принимает значение 100. Потребитель печатает это число и переходит к шагу C1.

Как раз перед тем, как потребитель может вызвать процедуру *down,* производитель находит следующее простое число и быстро выполняет шаги P2, P3 и P4.

В этот *MOMemfilled=0.* Производитель собирается выполнить над ним команду up, а потребитель собирается вызвать процедуру *up.* Если потребитель выполнит команду первым, то он будет приостановлен до тех пор, пока производитель не освободит его (вызвав процедуру *up).* Если же первым будет производитель, то семафор примет значение 1 и потребитель вообще не будет приостановлен. В обоих случаях сигнал пробуждения не пропадет. Именно для этого мы и ввели в программу семафоры.

Операции над семафорами неделимы. Если операция над семафором уже началась, то никакой другой процесс не может использовать этот семафор до тех пор, пока первый процесс не завершит операцию или пока он не будет приостановлен. Более того, при наличии семафоров сигналы пробуждения не пропадают. А вот операторы if в листинге 6.1 делимы. Между проверкой условия и выполнением нужного действия другой процесс может послать сигнал пробуждения.

В сущности, проблема синхронизации была устранена путем введения неделимых системных прерываний *up* и *down*. Чтобы эти операции были неделимыми, операционная система должна запретить двум и более процессам использовать один семафор одновременно. Если был сделан системный вызов *up* или *down,* ни один другой код пользователя не будет запущен, пока данный вызов не завершится. Для этого обычно во время выполнения операций над семафорами вводится запрет на прерывания.

Технология с использованием семафоров работает для произвольного количества процессов. Несколько процессов могут «спать», не завершив системный вызов *down* на одном и том же семафоре. Когда какой-нибудь другой процесс выполнит процедуру *up* на том же семафоре, один из ждущих процессов может завершить вызов *down* и продолжить работу. Семафор сохраняет значение 0, и другие процессы продолжают ждать.

Поясним это на другом примере. Представьте себе 20 баскетбольных команд. Они играют 10 партий (процессов). Каждая игра происходит на отдельном поле. Имеется большая корзина (семафор) для баскетбольных мячей. К сожалению, есть только 7 мячей. В каждый момент в корзине находится от 0 до 7 мячей (семафор принимает значение от 0 до 7). Помещение мяча в корзину — это операция *up,* поскольку она увеличивает значение семафора. Извлечение мяча из корзины — это операция *down,* поскольку она уменьшает значение семафора.

В самом начале один игрок от каждого поля посылается к корзине за мячом. Семерым из них удается получить мяч (завершить операцию *down);* оставшиеся трое вынуждены ждать мяч. Их игры временно приостановлены. В конце концов одна из партий заканчивается и мяч возвращается в корзину (выполняется операция *up).* Эта операция позволяет одному из трех оставшихся игроков получить мяч (закончить незавершенную операцию *down)* и продолжить игру. Оставшиеся две партии остаются приостановленными до тех пор, пока еще два мяча не возвратятся в корзину. Когда эти два мяча положат в корзину (то есть будет выполнено еще две операции *up),* можно будут продолжить последние две партии.

Листинг 6.2. Параллельная обработка с использованием семафоров

```
public class m {
    final public static int BUF_SIZE = 100.              //буфер от 0 до 99
    final public static long MAX_PRIME=100000000000L.    //остановиться здесь
    public static int in = 0. out = 0.                   //указатели на данные
    public static long buffer[ ] = new long[BUF_SIZE].   //здесь хранятся простые числа

    public static producer p                             //имя произво/щеля
    public static consumer c                             //имя потребитШя
    public static int filled = 0, available = 100.       //семафоры
```

```
   public static vend mainCStnng args[ ]) {          //основной клласс
      p = new producer0;                             //создание производителя
      c = new consumer0;                             //создание потребителя
      p.start0:                                      //запуск производителя
      c start0:                                      //запуск потребителя
   }
   // Это утилита для циклического увеличения in и out
   public static int nextdnt k) {if (k < BUF_SIZE - 1) return(k+1): else return(0).}
}
class producer extends Thread {                      //класс производителя
   native void updnt s); native void downdnt s);     //процедуры над семафорами
      public void run() {                            //код производителя
         long prime - 2;                             //временная переменная

         while (prime < m.MAX_PRIME) {
            prime = next_pnme(pnme):                 //шаг P1
            downtm.available);                       //шаг P2
            m buffer[m in] = prime;                  //шаг P3
            m.in = m next(m.in);                     //шаг P4
            up(m.filled);                            //шаг P5
      }
   }

private long next_pnme(long pnme){ ...}              //функция, которая выдает
                                                       следующее число
}

class consumer extends Thread {                      //класс потребителя
      native void updnt s); native void downdnt s);  //процедуры над семафорами
      public void runC) {                            //код потребителя
         long emirp = 2,                             //временная переменная

         while (emirp < m MAX_PRIME) {
            down(m.filled),                          //шаг C1
            emirp = m.buffer[m.out];                 //шаг C2
            m.out = m next(m out);                   //шаг C3
            up(m. available):                        //шаг C4
            System.out.print!n(emirp);               //шаг C5
         }
      }
}
```

Примеры операционных систем

В этом разделе мы продолжим обсуждать Pentium II и Ultra SPARC П. Мы рассмотрим операционные системы, которые используются на этих процессорах.

Для Pentium II мы возьмем Windows NT (для краткости мы будем называть эту операционную систему NT); для UltraSPARC II — UNIX. Мы начнем наш разговор с UNIX, поскольку эта система гораздо проще NT. Кроме того, операционная система UNIX была разработана раньше и сильно повлияла на NT, поэтому такой порядок изложения будет более осмысленным.

Введение

В этом разделе мы дадим краткий обзор двух операционных систем (UNIX и NT). При этом мы обратим особое внимание на историю, структуру и системные вызовы.

UNIX

Операционная система UNIX была разработана в компании Bell Labs в начале 70-х годов. Первая версия была написана Кеном Томпсоном (Ken Thompson) на ассемблере для мини-компьютера PDP-7. Затем была написана вторая версия для компьютера PDP-11, уже на языке C. Ее автором был Деннис Ритчи (Dennis Ritchie). В 1974 году Ритчи и Томпсон опубликовали очень важную работу о системе UNIX [120]. За эту работу они были награждены престижной премией Тьюринга Ассоциации вычислительной техники (Ritchie, 1984; Thompson, 1984). После публикации этой работы многие университеты попросили у Bell Labs копию UNIX. Поскольку материнская компания Bell Labs, AT&T была в тот момент регулируемой монополией и ей не разрешалось принимать участие в компьютерном бизнесе, университеты смогли приобрести операционную систему UNIX за небольшую плату.

PDP-11 использовались практически во всех компьютерных научных отделах университетов, и операционные системы, которые пришли туда вместе с PDP-11, не нравились ни профессорам, ни студентам. UNIX быстро заполнил эту нишу. Эта операционная система была снабжена исходными текстами, поэтому люди могли до бесконечности исправлять ее.

Одним из первых университетов, которые приобрели систему UNIX, был Калифорнийский университет в Беркли. Поскольку имелась в наличии полная исходная программа, в Беркли сумели существенно преобразовать эту систему. Среди изменений было портирование этой системы на мини-компьютер VAX, создание виртуальной памяти со страничной организацией, расширение имен файлов с 14 символов до 255, а также включение сетевого протокола TCP/IP, который сейчас используется в Интернете (во многом благодря тому факту, что он был в системе Berkeley UNIX).

Пока в Беркли производились все эти изменения, компания AT&T самостоятельно продолжала разработку UNIX, в результате чего в 1982 году появилась System III, а в 1984 — System V. В конце 80-х годов широко использовались две разные и совершенно несовместимые версии UNIX: Berkeley UNIX и System V. Такое положение, да еще и отсутствие стандартов на форматы программ в двоичном коде сильно препятствовало коммерческому успеху системы UNIX. Поставщики программного обеспечения не могли писать программы для UNIX, ведь не было никакой гарантии, что эти программы будут работать на любой версии UNIX (как было сделано с MS DOS). После долгих споров комиссия стандартов в Институте инженеров по электротехнике и электронике выпустила стандарт **POSIX** (Portable Operating System-IX — интерфейс переносной операционной системы[1]). POSIX также известен как стандарт P1003. Позднее он стал международным стандартом.

[1] POSIX — Portable Operating System Interface for Computer Environments — платформенно-независимый системный интерфейс. — *Примеч. научн. ред.*

Стандарт POSIX разделен на несколько частей, каждая из которых покрывает отдельную область системы UNIX. Первая часть P1003.1 определяет системные вызовы; вторая часть P1003.2 определяет основные обслуживающие программы и т. д. Стандарт P1003.1 определяет около 60 системных вызовов, которые должны поддерживаться всеми соответствующими системами. Это вызовы для чтения и записи файлов, создания новых процессов и т. д. Сейчас практически все системы UNIX поддерживают системные вызовы P1003.1. Однако многие системы UNIX поддерживают и дополнительные системные вызовы, в частности те, которые определены в System V или в Berkeley UNIX. Обычно к набору POSIX добавляется до 100 системных вызовов. Операционная система для машины UltraSPARC II основана на System V. Она называется **Solaris.** Она поддерживает и многие вызовы из системы Berkeley.

В табл. 6.6 приведены категории системных вызовов. Системные вызовы управления файлами и директориями — это самые большие и самые важные категории. Большинство из них относятся к стандарту P1003.1. Остальные происходят из системы System V.

Таблица 6.6. Системные вызовы UNIX

Категория	Примеры
Управление файлами	Открытие, чтение, запись, закрытие и блокировка файлов
Управление директориями	Создание и удаление директорий; перемещение файлов по директориям
Управление процессами	Порождение, завершение, отслеживание процессов и передача сигналов
Управление памятью	Разделение общей памяти между процессами; защита страниц
Вызов/установка параметров	Идентификация пользователя, группы, процесса; установка приоритетов
Даты и периоды времени	Указание на время доступа к файлам; использование датчика временных интервалов; рабочий профиль программы
Работа в сети	Установка/принятие соединения; отправка/получение сообщения
Прочее	Учет использования ресурсов; ограничение на доступный объем памяти; перезагрузка системы

Сфера использования сетей в большей степени относится к Berkeley UNIX, а не к System V. В Беркли было введено понятие сокет (конечный пункт сетевой связи). Четырехпроводные стенные розетки, к которым можно подсоединять телефоны, послужили в качестве модели этого понятия. Процесс в системе UNIX может создать сокет, присоединиться к нему и установить связь с сокетом на удаленном компьютере. По этой связи можно пересылать данные в обоих направлениях, обычно с использованием протокола TCP/IP. Поскольку технология сетевой связи десятилетиями применялась в системе UNIX, значительное число серверов в Интернете используют именно UNIX.

Существует много разных вариантов системы UNIX, и каждая из них чем-то отличается от всех остальных, поэтому структуру данной ог рационной системы описать трудно. Но схема, изображенная на рис. 6.24, применима к большинству

из них. Внизу находятся драйверы устройств, которые защищают систему файлов от аппаратного обеспечения. Изначально каждый драйвер устройства был написан отдельно от всех других и представлял собой независимую единицу. Это привело к многочисленным дублированиям, поскольку многие драйверы должны иметь дело с управлением потоками, исправлением ошибок, приоритетами, отделением данных от команд и т. д. По этой причине Деннис Ритчи изобрел структуру под названием **поток** для написания драйверов в модулях. При наличии потока можно установить двустороннюю связь между пользовательским процессом и устройством аппаратного обеспечения и вставить между ними один или несколько модулей. Пользовательский процесс передает данные в поток, а затем эти данные обрабатываются или передаются дальше каждым модулем до тех пор, пока они не дойдут до аппаратного обеспечения. При передаче данных от аппаратного обеспечения происходит обратный процесс.

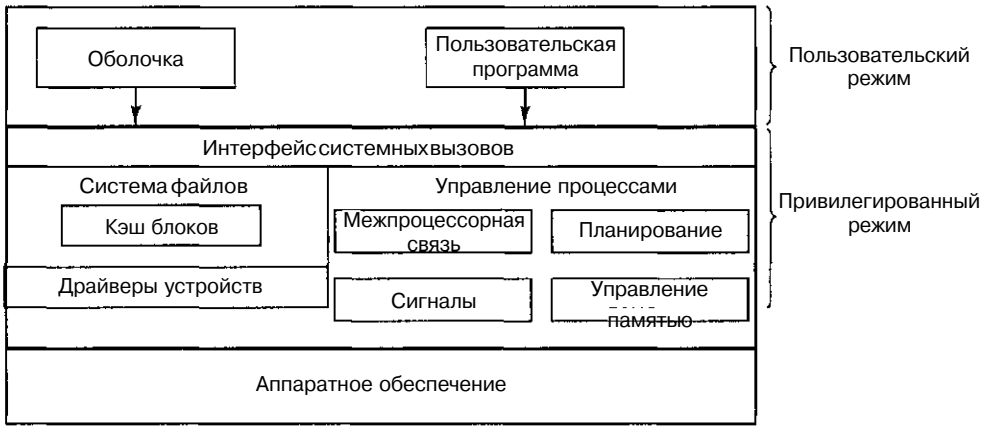


Рис. 6.24. Структура типичной системы UNIX

Над драйверами устройств находится система управления файлами. Она управляет именами файлов, директориями, расположением блоков на диске, защитой и выполняет многие другие функции. В системе файлов имеется так называемый **кэш блоков** для хранения недавно считанных с диска блоков, на случай если они понадобятся еще раз. Некоторые системы файлов использовались на протяжении многих лет. Среди них можно назвать быструю файловую систему Berkeley [95] и журналирующие файловые системы [121].

Еще одна часть ядра системы UNIX — структура управления процессами. Она выполняет различные функции, в том числе управляет межпроцессорной связью, которая позволяет разным процессам взаимодействовать друг с другом и синхронизирует работу процессов, чтобы избежать состояния гонок. Для этого существует ряд механизмов. Код управления процессами также управляет планированием работы процессов, которое основано на приоритетах. Кроме того, здесь обрабатываются сигналы прерываний. Наконец, здесь же происходит управление памятью. Большинство систем UNIX поддерживают виртуальную память с подкачкой страниц по требованию, иногда с некоторыми дополнительными особенностями (например, несколько процессов могут разделять общие области адресного пространства).

UNIX изначально должен был быть маленькой системой, чтобы достигнуть увеличения надежности и высокой производительности. Первые версии UNIX были полностью текстовыми и использовали терминалы, которые могли отображать 24 или 25 строк по 80 символов ASCII-кодов. Пользовательским интерфейсом управляла программа, так называемая **оболочка,** которая предоставляла интерфейс командной строки. Поскольку оболочка не являлась частью ядра, было легко добавлять новые оболочки в UNIX, и с течением времени было придумано несколько черезвычайно сложных оболочек.

Позднее, когда появились графические терминалы, в Массачусетском технологическом институте для UNIX была разработана система **X Windows.** Еще позже полностью доработанный **графический интерфейс пользователя** под названием **Motif** был установлен поверх X Windows. Поскольку требовалось сохранить маленькое ядро, практически весь код системы X Windows и Motif работают в пользовательском режиме вне ядра.

Windows NT

Первая машина IBM PC, выпущенная в 1981 году, была оснащена 16-битной операционной системой индивидуального пользования, работающей в реальном режиме, с командной строкой. Она называлась MS-DOS 1.0. Эта операционная система состояла из находящейся в памяти программы на 8 Кбайт. Через два года появилась более мощная система на 24 Кбайт — MS-DOS 2.0. Она содержала процессор командной строки (оболочку) с рядом особенностей, заимствованных из системы UNIX. В 1984 году компания IBM выпустила машину PC/AT с операционной системой MS-DOS 3.0, размер которой к тому моменту составлял 36 Кбайт. С годами у системы MS-DOS появлялись все новые и новые особенности, но она при этом оставалась системой с командной строкой.

Вдохновленная успехом Apple Macintosh, компания Microsoft решила создать графический пользовательский интерфейс, который она назвала **Windows.** Первые три версии Windows, включая систему Windows 3.x, были не настоящими операционными системами, а графическими пользовательскими интерфейсами на базе MS-DOS. Все программы работали в одном и том же адресном пространстве, и ошибка в любой из них могла привести к остановке всей системы.

В 1995 году появилась система Windows 95, но это не устранило MS-DOS, хотя MS-DOS уже представляла собой новую версию 7.0. Windows 95 и MS-DOS 7.0 в совокупности включали в себя особенности развитой операционной системы, в том числе виртуальную память, управление процессами и мультипрограммирование. Однако операционная система Windows 95 не была полностью 32-битной программой. Она содержала большие куски старого 16-битного кода и все еще использовала файловую систему MS-DOS со всеми ограничениями. Единственным изменением в системе файлов было добавление длинных имен файлов (ранее в MS-DOS длина имен файлов была не более 8+3 символов).

Даже при выпуске Windows 98 в 1998 году система MS-DOS все еще присутствовала (на этот раз версия 7.1) и включала 16-битный код. Система Windows 98 не очень отличалась от Windows 95, хотя часть функций перешла от MS-DOS к Windows и формат дисков, подходящий для дисков большего размера, стал стандартным. Основным различием был пользовательский интерфейс, который объединил рабочий стол, Интернет, телевидение и сделал систему более закрытой. Именно

это и привлекло внимание судебного департамента США, который тогда подал на компанию Microsoft в суд, обвинив ее в незаконном монополизме.

Во время всех этих преобразований компания Microsoft разрабатывала совершенно новую 32-битную операционную систему, которая была написана заново с нуля. Эта новая система называлась **Windows New Technology** (новая технология) или **Windows NT**[1]. Изначально предполагалось, что она заменит все другие операционные системы компьютеров на базе процессоров Intel, но она очень медленно распространялась и позднее была переориентирована на более дорогостоящие компьютеры. Постепенно она стала пользоваться популярностью и в других кругах.

NT продается в двух вариантах: для серверов и для рабочих станций. Эти две версии практически идентичны и выработаны из одного исходного кода. Первая версия предназначена для локальных файловых серверов и серверов для печати и имеет более сложные особенности управления, чем версия для рабочих станций, которая предназначена для настольных вычислений одного пользователя. Существует особый вариант версии для серверов, предназначенный для больших сайтов. Различные версии настраиваются по-разному, и каждая из них оптимизирована для ожидаемого окружения. Во всем остальном эти версии сходны. Практически все выполняемые файлы идентичны для всех версий. Система NT сама определяет свою версию по специальной переменной во внутренней структуре данных (системный реестр). Пользователям запрещено изменять эту переменную и таким образом превращать дешевую версию для рабочей станции в более дорогостоящую версию для сервера или в версию для предприятия. В дальнейшем мы не будем заострять внимание на различиях.

MS-DOS и все предыдущие версии Windows были рассчитаны на одного пользователя. NT поддерживает мультипрограммирование, поэтому на одной и той же машине в одно и то же время могут работать несколько пользователей[2]. Например, сетевой сервер позволяет нескольким пользователям входить в систему по сети одновременно, причем каждый из них получает доступ к своим собственным файлам.

NT представляет собой реальную 32-битную операционную систему с мультипрограммированием. Она поддерживает несколько пользовательских процессов, каждый из которых имеет в своем распоряжении полное 32-битное виртуальное адресное пространство с подкачкой страниц по требованию. Кроме того, сама система написана как 32-битный код.

NT, в отличие от Windows 95, имеет модульную структуру. Она состоит из небольшого ядра, которое работает в привилегированном режиме, и нескольких обслуживающих процессов, работающих в пользовательском режиме. Пользо-

---

[1] Работы над операционной системой, получившей впоследствии название Windows NT, начались в рамках совместного проекта по созданию новой 32-битной версии операционной системы OS/2, который одно время осуществляли компании IBM и Microsoft после ряда неудач с предыдущей 16-битной версией системы OS/2. Этот проект, ориентированный на возможности микропроцессора 5386, начался в 1989 году, однако уже в следующем году пути этих компаний разошлись, и Microsoft, продолжившая работу над системой OS/2 v.3.0, затем дала ей имя Windows NT, желая этим показать и использование единого графического интерфейса со своими популярными одноименными оболочками, и дистанцирование от компании IBM. — *Примеч. научн. ред.*

[2] Необходимо заметить, что, в отличие от UNIX, Windows NT не позволяет нескольким пользователям одновременно работать с компьютером, поскольку это однотерминальная система, тогда как UNIX — это мультитерминальная операционная система. Однако по сети с ней могут одновременно взаимодействовать несколько пользователей, работающих на своих компьютерах. — *Примеч. научн. ред.*

вательские процессы взаимодействуют с обслуживающими процессами с применением модели клиент—сервер: клиент посылает запрос серверу, а сервер выполняет работу и отправляет результат клиенту. Модульная структура позволяет переносить систему NT на некоторые компьютеры не из семейства Intel (DEC Alpha, IBM Power PC и SGI MIPS). Однако из соображений повышения производительности начиная с NT 4.0 большая часть системы была перенесена обратно в ядро.

Можно до бесконечности долго рассказывать, какова структура NT и каков ее интерфейс. Поскольку нас в первую очередь интересует виртуальная машина, представленная различными операционными системами, мы кратко расскажем о структуре системы, а затем перейдем к интерфейсу.

Структура NT показана на рис. 6.25. Она состоит из ряда модулей, которые расположены по уровням. Их совместная работа реализует операционную систему. Каждый модуль выполняет определенную функцию и имеет определенный интерфейс с другими модулями. Практически все модули написаны на языке C, хотя часть графического интерфейса написана на C++, а кое-что из самых нижних уровней — на ассемблере.

В самом низу расположен **уровень аппаратных абстракций.** Он должен снабжать операционную систему абстрактными аппаратными устройствами, лишенными всех недостатков, которых у реального аппаратного обеспечения в избытке. К моделируемым устройствам относятся кэш-память вне кристалла, тактовые генераторы, шины ввода-вывода, контроллеры прерываний и контроллеры прямого доступа к памяти. Если эти устройства представить перед операционной системой в идеализированном виде, то это упростит перенос NT на другие аппаратные платформы, поскольку большая часть необходимых изменений концентрируется в одном месте.

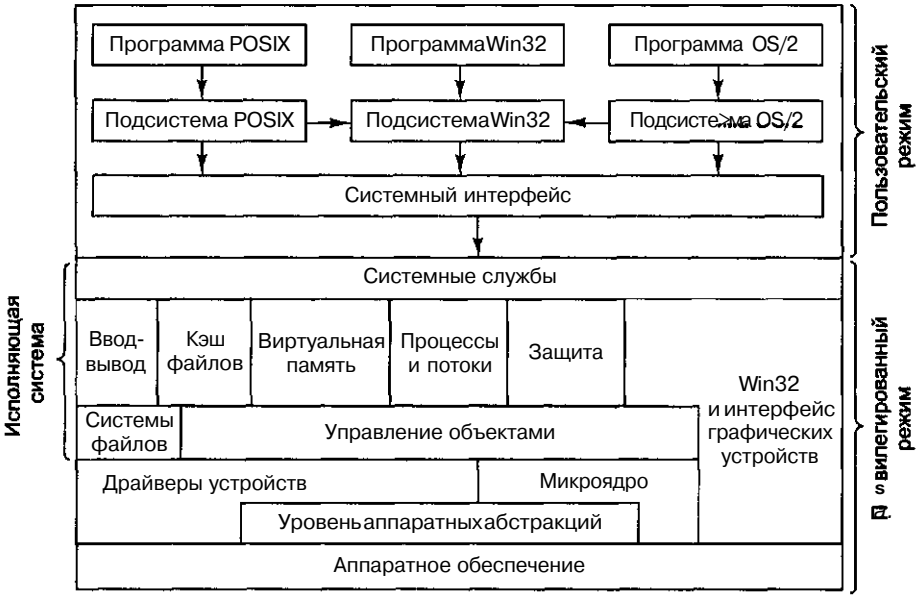


Рис. 6.25. Структура Windows NT

Над уровнем аппаратных абстракций расположен уровень, содержащий микроядро и драйверы устройств. Микроядро и все драйверы устройств имеют прямой доступ к аппаратному обеспечению, поскольку они содержат зависимый от аппаратного обеспечения код.

Микроядро поддерживает примитивные объекты ядра, обработку прерываний, ловушек, исключений, синхронизацию процессов, синхронизацию работы процессоров в многопроцессорных системах и управление временем. Основная задача этого уровня — сделать остальную часть операционной системы полностью независимой от аппаратного обеспечения и, следовательно, высокомобильной. Микроядро постоянно находится в основной памяти и никуда не передается, хотя оно временно может передать управление прерываниям ввода-вывода.

Каждый **драйвер устройств** может управлять одним или несколькими устройствами ввода-вывода. Кроме того, драйвер устройств может выполнять какие-то функции, не связанные с конкретным устройством, например шифровку потока данных или даже обеспечение доступа к структурам данных ядра. Так как пользователи имеют возможность устанавливать новые драйверы устройств, они могут повлиять на ядро и испортить всю систему. По этой причине драйверы нужно писать с особой осторожностью.

Над микроядром и драйверами устройств находится исполняющая система. **Исполняющая система** — независимая архитектура, поэтому ее можно переносить на другие машины. Она состоит из трех уровней.

Самый нижний уровень содержит файловые системы и диспетчер объектов. **Файловые системы** управляют использованием файлов и директорий. **Диспетчер объектов** управляет объектами, известными ядру (процессами, потоками, директориями, семафорами, устройствами ввода-вывода, тактовыми генераторами и т. п.). Эта программа также управляет пространством имен, куда можно помещать новые объекты, чтобы обращаться к ним позже в случае необходимости.

Следующий уровень состоит из 6 основных частей, как показано на рис. 6.25. **Диспетчер ввода-вывода** обеспечивает структуру для управления устройствами ввода-вывода, а также общими службами ввода-вывода. Диспетчер ввода-вывода использует службы файловой системы, которая, в свою очередь, использует драйверы устройств, а также службы диспетчера объектов.

Диспетчер кэш-памяти хранит в памяти блоки с диска, которые недавно использовались, чтобы повысить скорость доступа к ним, если они понадобятся снова. Диспетчер кэш-памяти должен вычислять, какие блоки могут понадобиться снова, а какие — нет. Можно конфигурировать NT с несколькими системами файлов. В этом случае диспетчер кэш-памяти работает на все системы файлов, поэтому отдельный диспетчер для каждой из них не нужен. Если требуется какой-либо блок диска, диспетчеру кэш-памяти посылается сигнал выдать этот блок. Если данного блока нет, диспетчер вызывает соответствующую систему файлов, чтобы получить этот блок. Поскольку файлы могут быть отображены на адресные пространства процессов, диспетчер кэш-памяти должен взаимодействовать с модулем управления виртуальной памятью, чтобы обеспечить необходимую согласованность.

Модуль управления виртуальной памятью реализует архитектуру виртуальной памяти с подкачкой страниц по требованию. Он управляет отображением виртуальных страниц на физические страничные кадры. Он вводит дополнительные

правила защиты, которые ограничивают доступ каждого процесса только к тем страницам, которые принадлежат его адресному пространству. Он также обрабатывает некоторые системные вызовы, которые связаны с виртуальной памятью.

Диспетчер процессов и потоков управляет процессами и потоками, в том числе их созданием и удалением.

Диспетчер безопасности предоставляет механизмы безопасности NT, которые удовлетворяют требованиям Оранжевой книги департамента защиты США. В Оранжевой книге определяется огромное количество правил, которым должна удовлетворять система, начиная с пароля и заканчивая тем, что виртуальные страницы должны обнуляться перед повторным использованием.

Интерфейс графических устройств управляет изображением на мониторе и принтерами. Он обеспечивает системные вызовы, которые позволяют пользовательским программам записывать информацию на монитор или принтеры независимо от устройств. Он также содержит диспетчер окон и драйверы аппаратных устройств. В версиях NT до NT 4.0 он находился в пользовательском пространстве, но производительность при этом оставляла желать лучшего, поэтому компания Microsoft перенесла его в ядро. Модуль Win32 также управляет многими системными вызовами. Изначально он тоже располагался в пользовательском пространстве, но позднее был перемещен в ядро с целью повышения производительности.

Самый верхний уровень исполняющей системы — **системные** службы. Этот уровень обеспечивает интерфейс с исполняющей системой. Он принимает системные вызовы NT и вызывает другие части исполняющей системы для выполнения.

Вне ядра находятся пользовательские программы и **подсистемы окружения.** Необходимость подсистем окружения объясняется тем, что пользовательские программы не способны непосредственно осуществлять системные вызовы. Поэтому каждая такая подсистема экспортирует определенный набор вызовов функций, которые могут использовать пользовательские программы. На рисунке 6.25 показаны 3 подсистемы окружения: Win32 (для программ NT и Windows 95/98), POSIX (для программ UNIX) и OS/2 для программ OS/2[1].

Приложения Windows используют функции **подсистемы Win32** и взаимодействуют с подсистемой Win32, чтобы совершать системные вызовы. **Подсистема Win32** принимает вызовы функций Win32 и использует модуль **системного интерфейса,** чтобы системные вызовы NT могли выполнять их.

Подсистема POSIX обеспечивает поддержку для приложений UNIX. Она поддерживает только стандарт P1003.1. Это закрытая подсистема. Ее приложения не могут использовать приспособления подсистемы Win32, что сильно ограничивает ее возможности. На практике перенос любой программы UNIX на NT с использованием этой подсистемы почти невозможен. Ее включили в NT только потому, что правительство США потребовало, чтобы операционные системы на компьютерах в правительстве соответствовали стандарту P1003.1. Эта подсистема не является самодостаточной, поэтому для своей работы она использует подсистему Win32, но при этом не передает полный интерфейс Win32 своим пользовательским программам.

[1] Бытует заблуждение, что Windows NT может выполнять различные программы, разработанные для OS/2. Однако на самом деле эта подсистема NT позволяет выполнять только те немногие 16-битные программы OS/2, которые работают исключительно в текстовом режиме. 32-битные программы OS/2 система Windows NT выполнять не может. — *Примеч. научн. ред.*

Функции подсистемы OS/2 тоже ограничены. Возможно, в будущих версиях ее уже не будет. Она также использует подсистему Win32. Существует и подсистема MS DOS (она не показана на рисунке).

Перейдем к обсуждению служб, которые предлагает операционная система NT. Ее интерфейс — это основное средство связи программиста с системой. К сожалению, компания Microsoft не опубликовала полный список системных вызовов NT, и кроме того, она меняет их от выпуска к выпуску. При таких обстоятельствах практически невозможно написание программ, которые непосредственно совершают системные вызовы.

Зато компания Microsoft определила набор вызовов **Win32 API (Application Programming Interface — прикладной программный интерфейс).** Это библиотечные процедуры, которые либо совершают системные вызовы, чтобы выполнить определенные действия, либо в некоторых случаях выполняют некоторые действия прямо в библиотечной процедуре пользовательского пространства или подсистеме Win32. Вызовы Win32 API не меняются при создании новых версий.

Однако, кроме этого, существуют вызовы NT API, которые могут менятся в новых версиях NT. Так как вызовы Win32 API задокументированы и более стабильны, мы сосредоточим наше внимание именно на них, а не на системных вызовах NT.

В системах Win32 API и UNIX применяются совершенно разные подходы. В UNIX все системные вызовы общеизвестны и формируют минимальный интерфейс: удаление хотя бы одного из них изменит функционирование операционной системы. Подсистема Win32 обеспечивает очень полный интерфейс. Здесь часто одно и то же действие можно выполнить тремя или четырьмя разными способами. Кроме того, Win32 включает в себя много функций, которые не являются системными вызовами (например, копирование целого файла).

Многие вызовы Win32 API создают объекты ядра того или иного типа (файлы, процессы, потоки, каналы и т. п.). Каждый вызов, создающий объект ядра, возвращает вызывающей программе результат, который называется **идентификатором (handle).** Этот идентификатор впоследствии может использоваться для выполнения операций над объектом. Для каждого процесса существует свой идентификатор. Он не может передаваться другому процессу и использоваться там (дескрипторы файла в UNIX тоже нельзя передавать другому процессу). Однако при определенных обстоятельствах можно продублировать идентификатор, передать его другим процессам и разрешить им доступ к объектам, которые принадлежат другим процессам. Каждый объект имеет связанный с ним дескриптор защиты, который сообщает, кому разрешено или запрещено совершать те или иные операции над объектом.

Операционную систему NT иногда называют объектно-ориентированной, поскольку оперировать с объектами ядра можно только с помощью вызова процедур (функций API) по их идентификаторам. С другой стороны, она не обладает такими основными свойствами объектно-ориентированной системы, как наследование и полиморфизм.

Win32 API имеется и в системе Windows 95/98 (а также в операционной систме Windows CE), правда, с некоторыми исключениями. Например, Windows 95/98 не имеет защиты, поэтому те вызовы API, которые связаны с защитой, просто возвращают код ошибки. Кроме того, для имен файлов в NT используется набор

символов Unicode, которого нет в Windows 95/98. Существуют различия в параметрах для некоторых вызовов API. В системе NT, например, все координаты экрана являются 32-битными числами, а в системе Windows 95/98 используются только младшие 16 битов (для совместимости с Windows 3.1). Существование набора вызовов Win32 API на нескольких разных операционных системах упрощает перенос программ между ними, но при этом кое-что удаляется из основной системы вызовов. Различия между Windows 95/98 и NT изложены в табл. 6.7.

Таблица 6.7. Некоторые различия между версиями Windows

Характеристика	Windows 95/98	NT 5.0
Win32 API	Да	Да
Полностью 32-битная система	Нет	Да
Защита	Нет	Да
Отображение защищенных файлов	Нет	Да
Отдельное адресное пространство для каждой программы MS-DOS	Нет	Да
Plug and Play	Да	Да
Unicode	Нет	Да
Процессор	Intel 80x86	80x86, Alpha
Многопроцессорная поддержка	Нет	Да
Реентерабельная программа (допускающая повторное вхождение) внутри операционной системы	Нет	Да
Пользователь может сам написать некоторые важные части операционной системы	Да	Нет

Примеры виртуальной памяти

В этом разделе мы поговорим о виртуальной памяти в системах UNIX и NT. С точки зрения программиста они во многом сходны.

Виртуальная память UNIX

Модель памяти в системе UNIX довольно проста. Каждый процесс имеет три сегмента: код, данные и стек, как показано на рис. 6.26. В машине с линейным адресным пространством код обычно располагается в нижней части памяти, а за ним следуют данные. Стек помещается в верхней части памяти. Размер кода фиксирован, а данные и стек могут увеличиваться или уменьшаться. Такую модель легко реализовать практически на любой машине. Она используется в операционной системе Solaris.

Более того, если машина содержит страничную память, то все адресное пространство может быть разбито на страницы, а пользовательские программы этого не знают. Единственное, что им будет известно, — то, что размер программы может превышать размер физической памяти машины. Системы UNIX, у которых нет страничной организации памяти, обычно перекачивают целые процессы между памятью и диском, чтобы сколь угодно большое число процессов работало в режиме разделения времени.

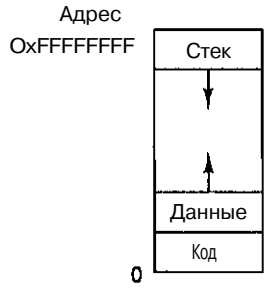


Рис. 6.26. Адресное пространство одного процесса UNIX

Описание, данное выше (виртуальная память с подкачкой страниц по требованию), в целом подходит для Berkeley UNIX. Однако Sysytem V (и Solaris) имеет некоторые особенности, позволяющие пользователям управлять виртуальной памятью. Самой важной является способность процесса отображать файл или часть файла на часть его адресного пространства. Например, если файл в 12 Кбайт отображается на виртуальный адрес 144 K, то в ячейке с адресом 144 K будет находиться первое слово этого файла. Таким образом, можно осуществлять ввод-вывод файла без применения системных вызовов. Поскольку размер некоторых файлов может превышать размер виртуального адресного пространства, можно отображать не весь файл, а только его часть. Чтобы осуществить отображение, сначала нужно открыть файл и получить дескриптор *файла fd* (file descriptor). Дескриптор используется для идентификации файла, который нужно отобразить. Затем процесс совершает вызов

```
paddr=nmap(virtual_address.1ength.protection,f1 ags.fd,fI1e_offset)
```

который отображает *length,* начиная с *file_offset* в файле, в виртуальное адресное пространство, начиная с *virtualjxddress.* Параметр *flags* требует, чтобы система выбрала виртуальный адрес, который затем возвращается в *paddr.* Отображаемая область должна содержать целое число страниц и должна быть выровнена в границах страницы. Параметр *protection* определяет разрешение на чтение, запись и выполнение (в любой комбинации). Отображение можно в дальнейшем удалить с помощью команды unmap.

В один и тот же файл можно одновременно отображать несколько процессов. Есть два варианта разделения общих страниц. В первом случае разделяются все страницы, поэтому записи, призводимые одним процессом, видны всем другим процессам. Эта возможность обеспечивает между процессами тракт с высокой пропускной способностью. Во втором случае страницы разделяются всеми процессами до тех пор, пока какой-нибудь процесс не изменит их. Как только один из процессов пытается произвести запись в страницу, он получает ошибку защиты, в результате чего операционная система выдает ему копию этой страницы, в которую можно производить запись. Такая схема используется в том случае, когда для каждого из нескольких процессов нужно создать иллюзию, что только он отображается в файл.

Виртуальная память Windows NT

В NT каждый пользовательский процесс имеет свое собственное виртуальное адресное пространство. Длина виртуального адреса составляет 32 бита, поэтому каждый процесс имеет 4 Гбайт виртуального адресного пространства. Нижние

2 Гбайт предназначены для кода и данных процесса; верхние 2 Гбайт разрешают ограниченный доступ к памяти ядра. Исключение составляют версии NT для предприятий, в которых разделение памяти может быть другим: 3 Гбайт — для пользователя и 1 Гбайт — для ядра. Виртуальное адресное пространство с подкачкой страниц по требованию содержит страницы фиксированного размера (4 Кбайт на машине Pentium II).

Каждая виртуальная страница может находиться в одном из трех состояний: она может быть **свободной** (free), **зарезервированной** (reserved) и **выделенной** (committed). Свободная страница в текущий момент не используется, а обращение к ней вызывает ошибку из-за отсутствия страницы. Когда процесс начинается, все его страницы находятся в свободном состоянии до тех пор, пока программа и начальные данные не будут отображены в свое адресное пространство. Если код или данные отображены в страницу, то такая страница является выделенной. Обращение к выделенной странице будет успешным, если страница находится в основной памяти. Если страница отсутствует в основной памяти, то произойдет ошибка и операционной системе придется вызывать нужную страницу с диска. Виртуальная страница может находиться и в зарезервированном состоянии. Это значит, что эта страница недоступна для отображения до тех пор, пока резервирование не будет отменено. Помимо атрибутов состояния страницы имеют и другие атрибуты (например, указывающие на возможность чтения, записи и выполнения). Верхние 64 Кбайт и нижние 64 Кбайт памяти всегда свободны, чтобы можно было отыскивать ошибки указателей (неинициализированные указатели часто равны 0 или -1).

Каждая выделенная страница имеет теневую страницу на диске, где она хранится при отсутствии ее в основной памяти. Свободные и зарезервированные страницы не имеют теневых страниц, поэтому обращения к ним вызывают ошибки из-за отсутствия страницы (система не может вызвать страницу с диска, если этой страницы нет на диске). Теневые страницы на диске сгруппированы в один или несколько страничных файлов. Операционная система следит, в какую часть какого страничного файла отображается каждая виртуальная страница. Файлы с текстами программ имеют теневые страницы; для страниц данных используются специальные страничные файлы.

NT, как и System V, позволяет отображать файлы прямо в области виртуального адресного пространства. Если файл был отображен в адресное пространство, его можно считывать или записывать путем обычных обращений к памяти.

Отображаемые в память файлы реализуются так же, как другие выделенные страницы, только теневые страницы могут находиться в файле на диске, а не в страничном файле. В результате, когда файл отображается, версия в памяти может не совпадать с версией на диске (из-за последних записей в виртуальное адресное пространство). Однако когда отображение файла удаляется, версия на диске обновляется.

NT позволяет отображать два и более процессов в одном файле одновременно, возможно, в разных виртуальных адресах. Путем считывания слов из памяти и записи слов в память процессы могут взаимодействовать друг с другом и передавать данные в обоих направлениях с достаточно высокой скоростью, поскольку никакого копирования не требуется. Разные процессы могут обладать разными разрешениями на доступ. Все процессы, использующие отображенный файл, разделяют

одни и те же страницы, поэтому изменения, произведенные одним из процессов, видны всем остальным процессам, даже если файл на диске еще не был обновлен.

Win32 API содержит ряд функций, которые позволяют процессу открыто управлять виртуальной памятью. Самые важные из этих функций приведены в табл. 6.8. Все они работают в области, состоящей либо из одной страницы, либо из двух или более страниц, последовательно расположенных в виртуальном адресном пространстве.

Таблица 6.8. Основ, ые функции API для управления виртуальной памятью в системе Windows NT

Функция API	Значение
VirtualAlloc	Резервация или выделение области
VirtualFree	Освобождение области или снятие выделения
VirtualProtect	Изменение типа защиты на чтение/запись/выполнение
VirtualQuery	Запрос о состоянии области
VirtualLock	Делает область памяти резидентной (то есть запрещает разбиение на страницы в ней)
VirtualUnlock	Снимает запрет на разбиение на страницы
CreateFileMapping	Создает объект отображения файла и иногда приписывает ему имя
MapViewOfFile	Отображает файл или часть файла в адресное пространство
UnmapViewOfFile	Удаляет отображенный файл из адресного пространства
OpenFileMapping	Открывает ранее созданный объект отображения файла

Первые четыре функции очевидны. Следующие две функции позволяют процессу делать резидентной область памяти размером до 30 страниц и отменять это действие. Это качество может понадобиться программам, работающим в режиме реального времени. Операционная систма устанавливает определенный предел, чтобы процессы не становились слишком поглощающими. В системе NT также имеются функции API, которые позволяют процессу получать доступ к виртуальной памяти другого процесса (они не указаны в табл. 6.8).

Последние 4 функции API предназначены для управления отображаемыми в память файлами. Чтобы отобразить файл, сначала нужно создать объект отображения файла с помощью функции *CreateFileMapping*. Эта функция возвращает идентификатор (handle) объекту отображения файла и иногда еще и вводит в систему файлов имя для него, чтобы другой процесс мог использовать объект. Две функции отображают файлы и удаляют отображение соответственно. Следующая функция нужна для того, чтобы отобразить файл, который в данный момент отображен другим процессом. Таким образом, два и более процессов могут разделять области своих адресных пространств.

Эти функции API являются основными. На них строится вся остальная система управления памятью. Например, существуют функции API для размещения и освобождения структур данных в одной или нескольких «кучах». «Кучи» используются для хранения структур данных, которые создаются и разрушаются. «Кучи» не занимаются «сборкой мусора», поэтому пользовательское программное обеспечение само должно освобождать блоки виртуальной памяти, которые уже не нуж-

ны. («Сборка мусора» — это автоматическое удаление неиспользуемых структур данных.) «Куча» в NT сходна с функцией *malloc* в системах UNIX, но в NT, в отличие от UNIX, может быть несколько независимых «куч».

Примеры виртуального ввода-вывода

Основной задачей любой операционной системы является предоставление служб для пользовательских программ. Главным образом, это службы ввода-вывода для чтения и записи файлов. И UNIX, и NT предлагают широкий спектр служб ввода-вывода. Для большинства системных вызовов UNIX в NT имеется эквивалентный вызов, но обратное не верно, поскольку NT содержит гораздо больше вызовов и каждый из них гораздо сложнее соответствующего вызова в UNIX.

Виртуальный ввод-вывод в системе UNIX

Система UNIX пользовалась большой популярностью во многом благодаря своей простоте, которая, в свою очередь, является прямым результатом организации системы файлов. Обычный файл представляет собой линейную последовательность 8-битных байтов[1] от 0 до максимум 2^{32}-1 байтов. Сама операционная система не сообщает структуру записей в файлах, хотя многие пользовательские программы рассматривают текстовые файлы в коде ASCII как последовательности строк, каждая из которых завершается переводом строки.

С каждым открытым файлом связан указатель на следующий байт, который нужно считать или записать. Системные вызовы *read* и *write* считывают и записывают данные, начиная с позиции, которую определяет указатель. После операции оба вызова перемещают указатель в другую позицию, передвигая его ровно на столько байтов, сколько было считано или записано. Возможен и случайный доступ к файлам, когда указатель файла устанавливается на определенное значение.

Кроме обычных файлов, система поддерживает специальные файлы, которые используются для доступа к устройствам ввода-вывода. С каждым устройством ввода-вывода обычно связан один или несколько специальных файлов. Считывая информацию из этих файлов и записывая информацию в эти файлы, программа может считывать информацию с устройства ввода-вывода и записывать информацию на устройство ввода-вывода. Так происходит работа с дисками, принтерами, терминалами и многими другими устройствами.

Основные системные вызовы для файлов в UNIX приведены в табл. 6.9. Вызов *creat* (без *e* на конце) используется для создания нового файла. В настоящее время он не является обязательным, поскольку вызов *open* тоже может создавать новый файл. Вызов *unlink* удаляет файл (предполагается, что файл находится только в одной директории).

Вызов *open* используется для открытия существующих файлов, а также для создания новых. Флаг *mode* сообщает, как его открыть (для чтения, для записи и т. д.). Вызов возвращает небольшое целое число, которое называется дескриптором файла. Дескриптор файла определяет файл в последующих вызовах. Сам про-

[1] Для многих сейчас слова про 8-битные байты могут показаться странными, однако на самом деле раньше байт мог быть и 5-битным, и 7-битным, и 8-битным. Теперь мы по умолчанию считаем байт состоящим из 8 битов. — *Примеч. научн. ред.*

цесс ввода-вывода осуществляется с помощью процедур *read* и *write,* каждая из которых содержит дескриптор файла (он указывает, какой файл использовать), буфер для данных и число байтов, которое сообщает, какое количество данных нужно передать. Вызов *Iseek* используется для перемещения указателя файла, что делает возможным случайный доступ к файлам.

Stat выдает информацию о файле (размер, время последнего доступа, имя владельца и т. п.). *Chmod* изменяет режим защиты файла (например, разрешает или, наоборот, запрещает каким-нибудь пользователям читать его). Наконец, *fcntl* выполняет различные действия над файлами, например блокирование и разблокирование.

Таблица 6.9. Основные системные вызовы UNIX

Системный вызов	Значение
creat(name, mode)	Создает файл; *mode* определяет тип защиты
unlink(name)	Удаляет файл (предполагается, что есть только 1 связь)
open(name, mode)	Открывает или создает файл и возвращает дескриптор файла
close(fd)	Закрывает файл
read (fd, buffer, count)	Считывает байты в количестве *count* в *buffer*
write(fd, buffer, count)	Записывает в файл *count* байтов из *buffer*
lseek(fd, offset, w)	Перемещает указатель файла на *offset* и *w*
stat(name, buffer)	Возвращает информацию о файле
chmod(name, mode)	Изменяет тип защиты файла
fcntl(fd, cmd, j)	Производит различные операции управления (например, блокирует файл или его часть)

В листинге 6.3 показано, как происходит процесс ввода-вывода. Эта программа минимальна и не включает в себя проверку ошибок. Перед тем как войти в цикл, программа открывает существующий файл *data* и создает новый файл *newf.* Каждый вызов возвращает дескриптор файла *infd* и *outfd* соответственно. Следующий параметр в обоих вызовах — биты защиты, которые определяют, что файлы нужно считать и записать соответственно. Оба вызова возвращают дескриптор файла. Если не удалось произвести *open* или *creat,* то возвращается отрицательный дескриптор файла, который сообщает, что вызов не удался.

Листинг 6.3. Фрагмент программы для копирования файла с использованием системных вызовов UNIX. Этот фрагмент написан на языке C, поскольку в языке Java не показываются системные вызовы низкого уровня, а нам нужно их показать

```
/* Открытие дескрипторов файла. */
infd = openC'data", 0);
outfd = creat("newf", ProtectionBits);
/* Цикл копирования. */
do{
       count = readCinfd. buffer, bytes);
       if (count > 0) write(outfd, buffer, count):
} while (count > 0):

/* Закрытие файлов.*/
close(infd):
ciose(outfd);
```

Вызов *read* имеет три параметра: дескриптор файла, буфер и число байтов. Данный вызов должен считать нужное число байтов из указанного файла в буфер. Число считанных байтов помещается в *count. Count* может быть меньше, чем *bytes,* если файл был слишком коротким. Вызов *write* копирует считанные байты в выходной файл. Цикл продолжается до тех пор, пока входной файл не будет прочитан полностью. Тогда цикл завершается, а оба файла закрываются.

Дескрипторы файлов в системе UNIX представляют собой небольшие целые числа (обычно до 20). Дескрипторы файлов 0, 1 и 2 соответствуют **стандартному** вводу, **стандартному выводу** и **стандартной ошибке** соответственно. Обычно первый из них обращается к клавиатуре, а второй и третий — к дисплею, но пользователь может перенаправить их к файлам. Многие программы UNIX получают входные данные из стандартного устройства ввода и записывают выходные данные в стандартное устройство вывода. Такие программы называются фильтрами.

С системой файлов тесно связана система директорий. Каждый пользователь может иметь несколько директорий, а каждая директория может содержать файлы и поддиректории. Система UNIX обычно конфигурируется с главной директорией, так называемым **корневым каталогом,** который содержит поддиректории *bin* (для часто используемых программ), *dev* (для специальных файлов устройств ввода-вывода), *lib* (для библиотек) и *usr* (для пользовательских директорий, как показано на рис. 6.27). В нашем примере директория *usr* содержит поддиректории *ast* и *jim*. Директория *ast* включает в себя два файла *(data* и/оо.с) и поддиректорию *bin,* в которую входят 4 игры.

Чтобы назвать файл, нужно указать его путь из корневого каталога. Путь содержит список всех директорий от корневого каталога к файлу, для разделения директорий используется слэш. Например, путь к файлу *game2* будет таким: /usr/ast/bin/game2. Путь, который начинается с корневого каталога, называется **абсолютным путем.**

В каждый момент времени каждая работающая программа имеет текущий **каталог.** Путь может быть связан с текущим каталогом. В этом случае в начале пути слэш не ставится (чтобы отличить такой путь от абсолютного пути). Такой путь называется **относительным** путем. Если /usr/ast — текущий каталог, то можно получить доступ к файлу game3, используя путь bin/game3. Пользователь может создать связь с чужим файлом, используя для этого системный вызов *link*. В нашем примере пути /usr/ast/bin/game3 и /usr/jim/jotto приводят к одному и тому же файлу. Не разрешается применять связи к директориям, чтобы предотвратить циклы в системе директорий. Вызовы *open* и *creat* используют и абсолютные, и относительные пути.

Основные вызовы для оперирования с директориями в системе UNIX приведены в табл. 6.10. *Mkdir* создает новую директорию, а *rmdir* удаляет существующую пустую директорию. Следующие три вызова применяются для чтения элементов директорий. Первый открывает директорию, второй считывает элементы из нее, а третий закрывает директорию. *Chdir* изменяет текущую директорию.

Link создает элемент директории, который указывает на уже существующий файл. Например, элемент /usr/jim/jotto можно создать с помощью вызова

```
Iink("/usr/ast/bin/game3", "/usr/jim/jotto")
```

или с помощью эквивалентного вызова, используя относительные пути, которые зависят от текущей директории. *Unlink* удаляет элемент директории. Если файл

имеет только одну связь, он удаляется. Если он имеет две и более связей, то он не удаляется. Не имеет никакого значения, была ли удаленная связь изначально созданной, или это копия. Вызов

```
unlink("/usr/ast/bin/game3")
```

делает файл *game3* доступным только через путь /usr/jim/jotto. Вызовы *link* и *unlink* могут использоваться для перемещения файлов из одной директории в другую.

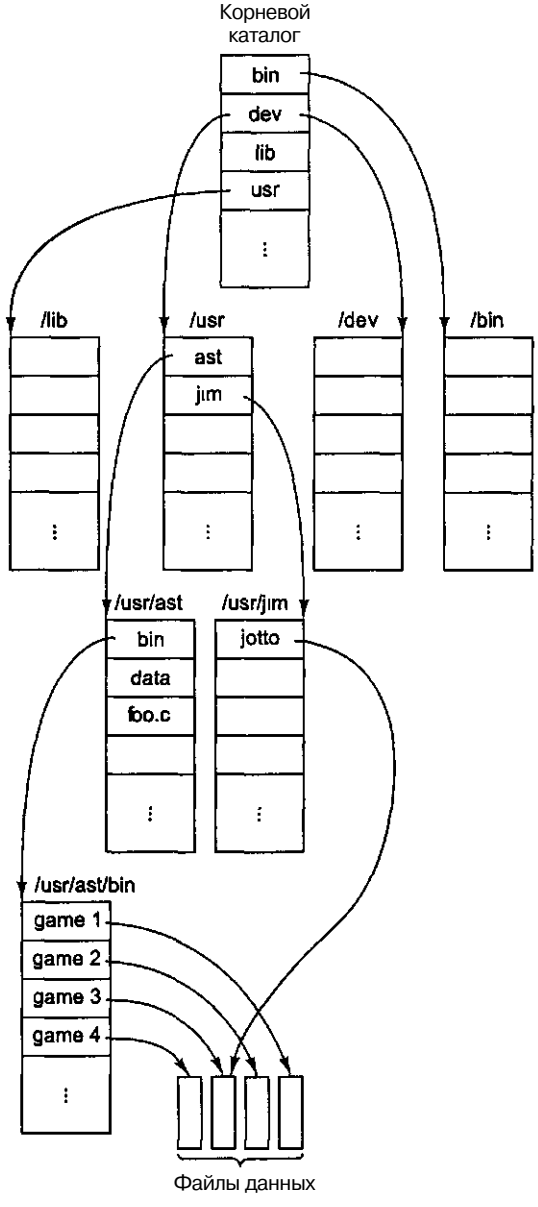


Рис. 6.27. Часть системы директорий в операционной системе UNIX

С каждым файлом (а также с каждой директорией, поскольку директория — это тоже файл) связано битовое отображение, которое сообщает, кому разрешен доступ к этому файлу. Отображение содержит три поля RWX (Read, Write, eXecute — чтение, запись, выполнение). Первое из них контролирует разрешение на чтение, запись и выполнение файлов для их владельца, второе — для других пользователей из группы владельца, а третье — для любых пользователей. Поля RWX R-X —X означают, что владелец файла может читать этот файл, записывать что-либо в него и выполнять его (очевидно, файл является выполняемой программой, иначе не было бы разрешения на его выполнение), другие члены группы могут читать и выполнять его, а посторонние люди — только выполнять. Таким образом, посторонние люди могут использовать эту программу, но не могут ее украсть (скопировать), поскольку им запрещено чтение. Отнесение пользователей к тем или иным группам осуществляется системным администратором, которого обычно называют **привилегированным пользователем.** Привилегированный пользователь имеет право действовать вопреки механизму защиты и считывать, записывать и выполнять любой файл.

Таблица 6.10. Основные вызовы для работы с директориями в системе UNIX

Системный вызов	Значение
mkdir(name, mode)	Создает новую директорию
rmdir(name)	Удаляет пустую директорию
Opendir(name)	Открывает директорию для чтения
readdir(dirpointer)	Читает следующий элемент директории
Closedir(dirpointer)	Закрывает директорию
chdir(dirname)	Изменяет текущий каталог на *dirname*
link(name1, name2)	Создает элемент директории *пате2,* указывающий на *пате 1*
unlink(name)	Удаляет *пате* из директории

Теперь рассмотрим, как файлы и директории реализованы в системе UNIX. Более детальное описание см. в [152]. С каждым файлом (и с каждой директорией, поскольку директория — это тоже файл) связан блок информации в 64 байта, который называется **индексным дескриптором (i-node).** I-node сообщает, кто владеет файлом, что разрешено делать с файлом, где найти данные и т. п. Индексные дескрипторы для файлов расположены или последовательно в начале диска, или, если диск разделен на группы цилиндров, — в начале цилиндра. Индексные дескрипторы снабжены последовательными номерами. Таким образом, система UNIX может обнаружить i-node просто путем вычисления его адреса на диске.

Элемент директории состоит из двух частей: имени файла и номера индексного дескриптора. Когда программа выполняет команду

```
openC'foo.c",  0),
```

система ищет текущий каталог для файла «foo.c», чтобы найти номер индексного дескриптора для этого файла. Обнаружив номер индексного дескриптора, программа может считать его и узнать всю информацию об этом файле.

При большей длине пути файла основные шаги, изложенные выше, повторяются несколько раз, пока не будет пройден весь путь. Например, чтобы найти номер индексного дескриптора для пути /usr/ast/data, система сначала ищет корне-

вой каталог для элемента *usr*. Обнаружив индексный дескриптор для *usr,* она может прочитать этот файл (директория в системе UNIX — это тоже файл). В этом файле она ищет элемент *ast* и находит номер индексного дескриптора для файла /usr/ast. Считав информацию о местонахождении директории /usr/ast, система может обнаружить элемент для *data* и, следовательно, номер индексного дескриптора для /usr/ast/data. Найдя номер индексного дескриптора для этого файла, система может узнать все об этом файле.

Формат, содержание и размещение индексных дескрипторов несколько различаются в разных системах (особенно когда идет речь о сети), но следующие характеристики присущи практически каждому дескриптору:

1. Тип файла, 9 битов защиты RWX и некоторые другие биты.
2. Число связей с файлом (число элементов директорий).
3. Идентификатор владельца.
4. Группа владельца.
5. Длина файла в байтах.
6. Тринадцать адресов на диске.
7. Время, когда файл читали в последний раз.
8. Время, когда последний раз производилась запись в файл.
9. Время, когда в последний раз менялся индексный дескриптор.

Типы файлов бывают следующие: обычные файл, директории и два вида особых файлов для устройств ввода-вывода с блочной структурой и неструктурированных устройств ввода-вывода соответственно. Число связей и идентификатор владельца мы уже обсуждали. Длина файла выражается 32-битным целым числом, которое показывает самый старший байт файла. Вполне возможно создать файл, перенести указатель на позицию 1 000 000 и записать 1 байт. В результате получится файл длиной 1 000 001. Тем не менее этот файл не требует сохранения всех отсутствующих байтов.

Первые 10 адресов на диске указывают на блоки данных. Если размер блока — 1024 байта, то можно работать с файлами размером до 10 240 байтов. Адрес 11 указывает на блок **косвенной** адресации, который содержит 256 адресов диска. Здесь можно работать с файлами размером до 10 240+256x1024=272 384 байта. Для файлов еще большего размера существует адрес 12, который указывает на 256 блоков косвенной адресации. Здесь допустимый размер файлов составляет 272384+256x256x1024=67 381 248 байтов. Если и эта схема **блока двойной косвенной адресации** слишком мала, то используется адрес 13. Он указывает на блок тройной косвенной адресации, который содержит адреса 256 блоков двойной косвенной адресации. Используя прямую, косвенную, двойную косвенную и тройную косвенную адресацию, можно обращаться к 16 843 018 блокам. Это значит, что максимально возможный размер файла составляет 17 247 250 432 байта. Поскольку размер указателей файлов ограничен до 32 битов, реальный верхний предел на размер файла составляет 4 294 967 295 байтов. Свободные блоки диска хранятся в связном списке. Если нужен новый блок, он берется из списка. В результате получается, что блоки каждого файла беспорядочно раскиданы по всему диску.

Чтобы повысить скорость ввода-вывода с диска, нужно сделать следующее. После открытия файла его индексный дескриптор копируется в таблицу в основной памяти и хранится там, пока файл остается открытым. Кроме того, в памяти находится набор блоков, к которым недавно производилось обращение. Так как большинство файлов считывается последовательно, часто при обращении к файлу требуется тот же блок, что и при предыдущем обращении. Чтобы увеличить скорость, система считывает следущий блок в файл еще до того, как к нему произведено обращение. Все эти моменты скрыты от пользователя. Когда пользователь выдает вызов *read,* работа программы приостанавливается, пока требуемые данные не появятся в буфере.

Зная все это, мы теперь можем рассмотреть, как происходит процесс ввода-вывода. *Open* заставляет систему искать директории по определенному пути. Если поиск успешен, то индексный дескриптор считывается во внутреннюю таблицу. Вызовы *read* и *write* требуют, чтобы система вычислила номер блока из текущей позиции файла. Адреса первых 10 блоков диска всегда находятся в основной памяти (в индексном дескрипторе); для остальных блоков сначала требуется считать один или несколько блоков косвенной адресации. *Lseek* просто меняет текущую позицию указателя и не производит никакого ввода-вывода.

Link смотрит на свой первый аргумент, чтобы обнаружить номер индексного дескриптора. Затем он создает элемент директории для второго аргумента и помещает номер индексного дескриптора первого файла в этот элемент директории. Наконец, он увеличивает число связей в индексном дескрипторе на 1. *Unlink* удаляет элемент директории и уменьшает число связей в индексном дескрипторе. Если это число равно 0, файл удаляется и все блоки помещаются в список свободных блоков.

Виртуальный ввод-вывод в Windows NT

NT поддерживает несколько файловых систем, самые важные из которых — **NTFS (NT File Sysytem — файловая система Windows NT)** и **FAT (File Allocation Table — таблица размещения файлов).** Первая была разработана специально для NT. Вторая является старой файловой системой для MS-DOS, которая также используется в Windows 95/98 (хотя и с длинными именами файлов). Поскольку система FAT устарела, ниже мы рассмотрим только файловую систему NTFS. FAT32 начала использоваться с NT 5.O. Она подерживалась и в более позних версиях Windows 95 и Windows 98.

В файловой системе NT длина имени файла может быть до 255 символов. Имена файлов написаны в коде Unicode, благодаря чему люди в разных странах, где не используется латинский алфавит, могут писать имена файлов на их родном языке. В файловой системе NT заглавные и строчные буквы в именах файлов считаются разными (то есть foo отличается от FOO). К сожалению, в системе Win32 API заглавные и строчные буквы в именах файлов и директорий не различаются, поэтому это преимущество теряется для программ, которые используют Win32.

Как и в системе UNIX, файл представляет собой линейную последовательность байтов, максимальная длина $2^{N} - 1$. Указатели тоже существуют, но их длина не 32, а 64 бита, чтобы можно было поддерживать максимальную длину файла. Вызовы функций в Win32 API для манипуляций с директориями и файлами в целом схожи с вызовами функций в системе UNIX, но большинство из них имеют больше

параметров, и модель защиты другая. При открытии файла возвращается идентификатор (handle), который затем используется для чтения и записи файла. В отличие от системы UNIX, идентификаторы не являются маленькими целыми числами, а стандартный ввод, стандартный вывод и стандартная ошибка не определяются заранее как 0,1 и 2 (исключение составляет пультовый режим работы). Основные функции Win32 API для управления файлами приведены в табл. 6.11.

Таблица 6.11. Основные функции Win32 API для ввода-вывода файлов. Во второй колонке дается эквивалент из UNIX

Функция API	UNIX	Значение
CreateFile	open	Создает файл или открывает существующий файл; возвращает идентификатор
DeleteFile	unlink	Удаляет существующий файл
CloseHandle	close	Закрывает файл
ReadFile	read	Считывает данные из файла
WriteFile	write	Записывает данные в файл
SetFilePointer	lseek	Устанавливает указатель файла на определенное место в файле
SetFileAttributes	stat	Возвращает свойства файла
LockFile	Fcntl	Блокирует область файла, чтобы обеспечить взаимное исключение доступа
UnlockFile	Fcntl	Снимает блокировку с ранее заблокированной области файла

Рассмотрим эти вызовы. *CreateFile* используется для создания нового файла и возвращает идентификатор (handle) для него. Эта функция применяется и для открытия уже существующего файла, поскольку в системе API нет функции *open*. Мы не будем приводить параметры функций API, поскольку их очень много. Например, *CreateFile* имеет семь параметров:

1. Указатель на имя файла, который нужно создать или открыть.
2. Флаги, которые сообщают, какие действия разрешено производить с файлом: читать, записывать или и то и другое.
3. Флаги, которые сообщают, могут ли несколько процессов открывать файл одновременно.
4. Указатель на дескриптор безопасности, который сообщает, кто имеет доступ к файлу.
5. Флаги, которые сообщают, что нужно делать, если файл существует или не существует.
6. Флаги, связанные с атрибутами архивации, компрессии и т. д.
7. Идентификатор файла (handle), атрибуты которого нужно клонировать для нового файла.

Следующие шесть функций API сходны с соотвествующими функциями в системе UNIX. Последние две позволяют блокировать и разблокировать область файла, чтобы обеспечить взаимное исключение доступа.

Используя эти функции API, можно написать процедуру для копирования файла, аналогичную процедуре в листинге 6.3. Такая процедура (без проверки оши-

бок) приведена в листинге 6.4. На практике программу для копирования файла писать не нужно, поскольку существует функция *CopyFile*.

Листинг 6.4. Фрагмент программы для копирования файла с применением функции API из системы Windows NT. Фрагмент написан на языке C, язык Java не показывает системные вызовы низкого уровня, а нам нужно было продемонстрировать их

```
/* Открытие файлов для ввода и вывода. */
inhandle - CreateFileCdata". GENERIC_READ. 0, NULL.    OPENJXISTING, 0. NULL);
outhandle = CreateFileC'newF. GENERIC_WRITE. 0. NULL. CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL.
NULL);
/* Копирование файла. */
do{
        s = ReadFile(inhandle. buffer. BUF_SIZE. «count. NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count. Socnt. NULL);
while (s > 0 && count > 0);
/* Закрытие файлов. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

NT поддерживает иерархическую систему файлов, сходную с системой файлов UNIX. Однако в качестве разделителя здесь используется не /, а \ (заимствовано из MS-DOS). Здесь тоже существует понятие текущего каталога, а пути могут быть абсолютными и относительными. Однако между NT и UNIX есть одно существенное различие. UNIX позволяет монтировать в одно дерево системы файлов с разных дисков и машин, скрывая таким образом структуру диска от программного обеспечения. NT 4.0 не имеет этого качества, поэтому абсолютные имена файлов должны начинаться с буквы, которая указывает на диск (например, C:\windows\system\foo.dll). Свойство монтирования систем файлов появилось с NT 5.O.

Основные функции для работы с директориями приведены в табл. 6.12 (также вместе с эквивалентами из UNIX). Думаем, что раскрывать их значение не требуется.

Отметим, что NT 4.0 не поддерживает связи файлов. На уровне графического рабочего стола поддерживаются клавишные комбинации быстрого вызова, но эти структуры не имели соответствий в самой системе файлов. Прикладная программа не могла войти в файл во второй директории, не скопировав весь файл. Начиная с NT 5.0 к системе файлов были добавлены файловые связи.

Таблица 6.12. Основные функции Min32 API для работы с директориями. Во втором столбце даны эквиваленты из UNIX, если они существуют

Функция API	UNIX	Значение
CreateDirectiry	mkdir	Создает новую директорию
RemoveDirectory	rmdir	Удаляет пустую директорию
FindFirstFile	opendir	Инициализирует чтение элементов директории
FindNextFile	readdir	Читает следующий элемент директории
MoveFile		Перемещает файл из одной директории в другую
SetCurrentDirectory	chdir	Меняет текущую директорию

NT имеет более сложный механизм защиты, чем в UNIX. Когда пользователь входит в систему, его процесс получает **маркер доступа** от операционной системы.

Маркер доступа содержит **идентификатор безопасности** (SID — **Security ID),** список групп, к которым принадлежит пользователь, имеющиеся привилегии и некоторую другую информацию. Маркер доступа концентрирует всю информацию о защите в одном легко доступном месте. Все процессы, созданные этим процессом, наследуют этот же маркер доступа.

Дескриптор защиты — это один из параметров, который дается при создании любого объекта. Дескриптор защиты содержит список элементов, который называется **списком контроля доступа (ACL — Access Control List).** Каждый элемент разрешает или запрещает совершать определенный набор операций над объектом какому-либо отдельному человеку или группе. Например, файл может содержать дескриптор защиты, который определяет, что Иванов не имеет доступа к файлу вообще, Петров может читать файл, Сидоров может читать и записывать файл, а все члены группы XYZ могут прочитать только размер файла.

Если процесс пытается выполнить какую-либо операцию над объектом с использованием идентификатора (handle), который он получил при открытии объекта, диспетчер безопасности получает маркер доступа данного процесса и начинает перебирать элементы списка контроля доступа по порядку. Как только он находит элемент, который соответствует нужному пользователю или одной из групп, информация о разрешении или запрещении доступа, найденная там, принимается в качестве заданной. По этой причине элементы, запрещающие доступ, обычно помещаются в список контроля доступа перед элементами, разрешающими доступ (чтобы пользователь, у которого нет доступа, не смог получить его незаконно, будучи членом одной из групп, которой доступ разрешен). Дескриптор защиты также содержит информацию, используемую для аудита доступов к объекту.

А теперь рассмотрим, как файлы и директории реализуются в NT. Каждый диск разделен на тома, такие же как разделы диска в UNIX. Каждый том содержит файлы, битовые отображения директорий и другие структуры данных. Каждый том организован в виде линейной последовательности **кластеров.** Размер кластера фиксирован для каждого тома. Он может быть от 512 байтов до 64 Кбайт, в зависимости от размера тома. Обращение к кластеру осуществляется по смещению от начала тома. При этом используются 64-битные числа.

Основной структурой данных в каждом томе является **MFT (Master File Table — главная файловая таблица),** в которой содержится элемент для каждого файла и директории в томе. Эти элементы аналогичны элементам индексного дескриптора (i-node) в системе UNIX. Главная файловая таблица является файлом и может быть помещена в любое место в пределах тома. Это устраняет проблему, возникающую при наличии испорченных блоков на диске в середине индексных дескрипторов.

Главная файловая таблица показана на рис. 6.28. Она начинается с заголовка, в котором дается информация о томе (указатели на корневой каталог, файл загрузки, список лиц, пользующихся свободным доступом и т. д.). Затем идет по одному элементу на каждый файл или директорию (1 Кбайт за исключением тех случаев, когда размер кластера составляет 2 Кбайт и более). Каждый элемент содержит все метаданные (административную информацию) о файле или директории. Допускается несколько форматов, один из которых изображен на рис. 6.28.

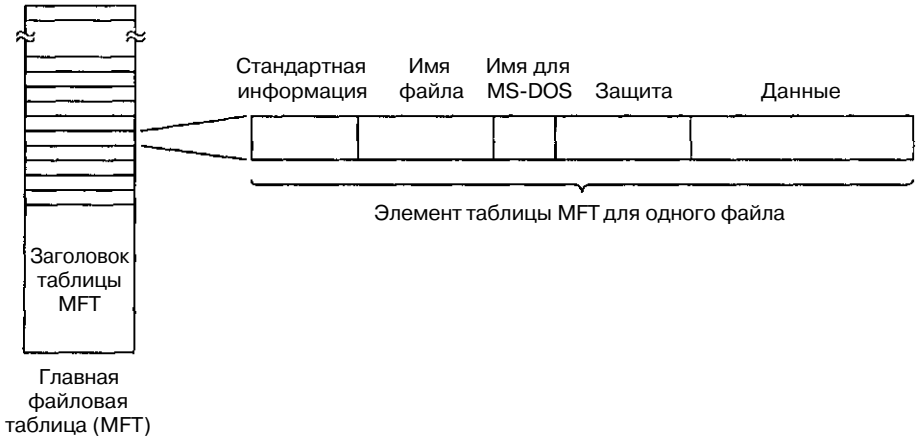


Рис. 6.28. Главная файловая таблица в системе Windows NT

Поле стандартой информации содержит информацию об отметках времени, необходимых стандартах POSIX, о числе связей, о битах «только для чтения» и битах архивирования и т. д. Это поле имеет фиксированную длину и является обязательным. Имя файла может иметь любую длину до 255 символов Unicode. Чтобы такие файлы стали доступны для старых 16-битных программ, они могут снабжаться дополнительным именем MS-DOS, состоящим максимум из 8 символов, за которыми следует точка и расширение из 3 символов. Если действительное имя файла соответствует правилу наименования в MS-DOS (8+3), то второе имя для MS-DOS не используется.

Далее следует информация о защите. Во всех версиях, вплоть до NT 4.0, в поле защиты содержался дескриптор защиты. Начиная с NT 5.0, вся информация о защите помещается в один файл, а поле защиты просто указывает на соответствующую часть этого файла.

Для файлов небольшого размера сами данные этих файлов содержатся в элементе главной файловой таблицы, что упрощает их вызов, — для этого не требуется обращаться к диску. Данная идея получила название **непосредственный файл** (immediate file) [100]. Для файлов большого размера это поле содержит указатели на кластеры, в которых содержатся данные или (что более распространено) блоки последовательных кластеров, так что номер кластера и его длина могут представлять произвольное количество данных. Если элемент главной файловой таблицы недостаточно велик для хранения нужной информации, к нему можно привязать один или несколько дополнительных элементов.

Максимальный размер файла составляет 2^{64} байтов. Поясним, что собой представляет файл на 2^{64} байтов. Представим, что файл был написан в двоичной системе, а каждый 0 или 1 занимает 1 мм пространства. Длина листинга на 2^{67} мм составила бы 15 световых лет. Этого хватило бы для того, чтобы выйти за пределы Солнечной системы, достичь Альфа Центавра и вернуться обратно.

Файловая система NTFS имеет много других интересных особенностей, в том числе возможность компрессии данных и отказоустойчивость с применением атомарных транзакций. Дополнительную информацию об этой системе можно найти в [137].

Примеры управления процессами

Системы NT и UNIX позволяют разделить работу на несколько процессов, которые выполняются параллельно и взаимодействуют друг с другом, как в примере с производителем и потребителем, который мы обсуждали ранее. В этом разделе мы поговорим о том, как происходит управление процессами в обеих системах. Обе системы поддерживают параллелизм в пределах одного процесса с использованием потоков, и об этом мы тоже расскажем.

Управление процессами в системе UNIX

В любой момент процесс в системе UNIX может создать подпроцесс, который является его точной копией. Для этого выполняется системный вызов *fork*. Изначальный процесс называется **порождающим процессом,** а новый — **порожденным процессом.** Два процесса, полученные в результате выполнения операции *fork,* абсолютно идентичны и даже разделяют одни и те же файловые дескрипторы. Каждый из этих двух процессов выполняет свою работу независимо от другого.

Часто порожденный процесс определенным образом дезориентирует дескрипторы файлов, а затем выполняет системный вызов *exec,* который замещает его программу и данные программой и данными из выполняемого файла, определенного в качестве параметра к вызову *exec*. Например, если пользователь печатает команду xyz, то интерпретатор команд (оболочка) выполняет операцию *fork,* создавая таким образом порожденный процесс. А этот процесс выполняет процедуру *exec,* чтобы запустить программу *xyz*.

Эти два процесса работают параллельно, но иногда порождающий процесс должен по каким-либо причинам ждать, чтобы порожденный процесс завершил свою работу, и только после этого продолжать выполнение тех или иных действий. В этом случае порождающий процесс выполняет системный вызов *wait* или *waitpid,* в результате чего он временно приостанавливается и ждет, пока порожденный процесс не выполнит системный вызов *exit*.

Процессы могут выполнять процедуру *fork* сколь угодно часто, в результате чего получается целое дерево процессов. Посмотрите на рис. 6.29. Здесь процесс А выполнил процедуру *fork* дважды и породил два новых процесса, В и С. Затем процесс В тоже выполнил процедуру *fork* дважды, а процесс С выполнил ее один раз. Таким образом, получилось дерево из шести процессов.

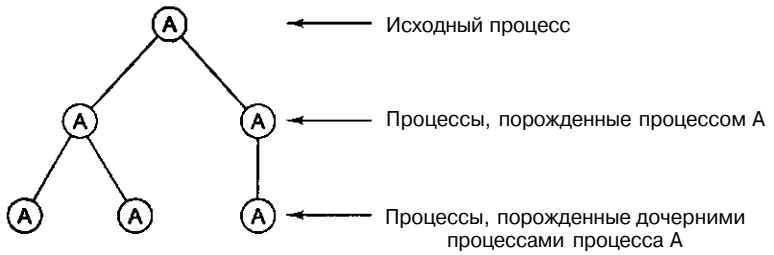


Рис. 6.29. Дерево процессов в системе UNIX

Процессы в системе UNIX могут взаимодействовать друг с другом через специальную информационную структуру, которая называется **каналом.** Канал пред-

ставляет собой вид буфера, в который один процесс записывает поток данных, а другой процесс забирает эти данные оттуда. Байты всегда возвращаются из канала в том порядке, в котором они были записаны. Случайный доступ невозможен. Каналы не сохраняют границы между отрезками данных, поэтому если один процесс записал в канал 4 отрезка по 128 байтов, а другой процесс считывает данные по 512 байтов, то этот второй процесс получит все данные сразу без указания на то, что они были записаны в несколько заходов.

В системах System V и Solaris применяется другой метод взаимодействия процессов. Здесь используются так называемые **очереди сообщений.** Процесс может создать новую очередь сообщений или открыть уже существующую с помощью вызова *msgget.* Для отправки сообщений используется вызов *msgsnd,* а для получения — *msgrecv*. Сообщения, отправленные таким способом, отличаются от данных, помещаемых в канал. Во-первых, границы сообщений сохраняются, а в канал передается просто поток данных. Во-вторых, сообщения имеют приоритеты, поэтому срочные сообщения идут перед всеми остальными. В-третьих, сообщения типизированы, и вызов *msgrecv* может определять их тип, если это необходимо.

Два и более процесса могут разделять общую область своих адресных пространств. UNIX управляет этой разделенной памятью путем отображения одних и тех же страниц в виртуальное адресное пространство всех разделенных процессов. В результате запись в общую область, произведенная одним из процессов, будет видна всем остальным процессам. Этот механизм обеспечивает очень высокую пропускную способность между процессами. Системные вызовы, включенные в разделенную память, идут по алфавиту (как *shmat* и *shmop).*

Еще одна особенность System V и Solaris — наличие семафоров. Принципы их работы мы уже описывали, когда говорили о производителе и потребителе.

Системы UNIX могут поддерживать несколько потоков управления в пределах одного процесса. Эти потоки управления обычно называют просто **потоками.** Они похожи на процессы, которые делят общее адресное пространство и все объекты, связанные **с** этим адресным пространством (дескрипторы файлов, переменные окружения и т. д.). Однако каждый поток имеет свой собственный счетчик команд, свои собственные регистры и свой собственный стек. Если один из потоков приостанавливается (например, пока не завершится процесс ввода-вывода), другие потоки в том же процессе могут продолжать работу. Дна потока в одном процессе, которые действуют как процесс-производитель и процесс-потребитель, сходны с двумя однопоточными процессами, которые разделяют сегмент памяти, содержащий буфер, хотя не идентичны им. Во втором случае каждый процесс имеет свои собственные дескрипторы файлов и т. д., тогда как в первом случае все **эти** элементы общие.

В каких случаях могут понадобиться потоки? **Рассмотрим** сервер World Wide Web. Такой сервер может хранить в основной памяти кэш часто используемых web-страниц. Если нужная страница находится в кэш-памяти, то она выдается немедленно. Если нет, то она вызывается с диска. К сожалению, на это требуется довольно длительное время (обычно 20 млс), и на это время процесс блокируется и не может обслуживать новые поступающие запросы, даже если эти web-страницы находятся в кэш-памяти.

По этой причине было принято решение сделать несколько потоков в одном процессе, которые разделяют общую кэш-память web-страниц. Если один из потоков блокируется, новые запросы могут обрабатываться другими потоками. Предотвратить блокировку процессов можно и без использования потоков. Для этого потребуется иметь несколько процессов, но тогда нужно будет продублировать кэш, а это несколько расточительно, поскольку размер памяти ограничен.

Стандарт системы UNIX для потоков называется pthreads. Он определяется стандартом POSIX (P1003.1C) и содержит вызовы для управления потоками и их синхронизации. Управляется ли потоками ядро, или они полностью находятся в пользовательском пространстве, в стандарте не определено. Наиболее распространенные вызовы для работы с потоками приведены в табл. 6.13.

Таблица 6.13. Основные вызовы для потоков, определенные в стандарте POSIX

Вызов потока	Значение
pthread_create	Создает новый поток в адресном пространстве вызывающейпроцедуры
pthread_exit	Завершает поток
pthreadjoin	Ждет завершения потока
pthread_mutex_init	Создает новый мьютекс
pthread_mutex_destroy	Удаляет мьютекс
pthread_mutex_lock	Блокирует мьютекс
pthread_mutex_unlock	Снимает блокировку с мьютекса
pthread_cond_init	Создает переменную условия
pthread_cond_destroy	Удаляет переменную условия
pthread_cond_wait	Ждет переменную условия
pthread_cond_signal	Снимает блокировку с одного из потоков, который ждет переменнуюусловия

Давайте рассмотрим эти вызовы. Первый вызов, *pthread_create,* создает новый поток. После выполнения этой процедуры в адресном пространстве появляется на один поток больше. Поток, который выполнил свою работу, вызывает*pthread_exit.* Если потоку нужно подождать, пока другой поток не окончит свою работу, он вызывает *pthreadjoin.* Если этот другой поток уже закончил свою работу, вызов *pthreadjoin* немедленно завершается.

Потоки можно синхронизировать с помощью специальных объектов, которые называются **мьютексами.** Обычно мьютекс управляет каким-либо ресурсом (например, буфером, разделенным между двумя потоками). Для того чтобы в конкретный момент времени только один поток мог получать доступ к общему ресурсу, потоки дожны запирать мьютекс перед использованием ресурса и отпирать его после завершения работы с ним. Таким образом можно избежать состояния гонок, поскольку этому протоколу подчиняются все потоки. Мьютексы похожи на бинарные семафоры (то есть семафоры, которые могут принимать только два значения: 0 или 1). Эти объекты получили название «мьютексы» (mutexes), поскольку они используются для обеспечения взаимного исключения доступа к какому-либо из ресурсов (MUTual Exclusion по-английски значит «взаимное исключение»).

Мьютексы можно создавать и разрушать с помощью вызовов *pthreadjnutexjnit* и *pthread_mutex_destroy* соответственно. Мьютекс может находиться в одном из двух состояний: блокированном и неблокированном. Если потоку нужно установить блокировку на незапертый мьютекс, он использует *pthread'_mutex_lock,* а затем продолжает работу. Однако если поток пытается запереть мьютекс, который уже заперт, он блокируется. Когда поток, который в данный момент использует общий ресурс, завершит работу с этим ресурсом, он должен разблокировать соответствующий мьютекс с помощью *pthread_mutex_unlock.*

Мьютексы предназначены для кратковременной блокировки (например, для защиты общей переменной). Они не предназначены для длительной синхронизации (например, для ожидания, пока освободится накопитель на магнитной ленте). Для длительной синхронизации существуют **переменные условия (condition variables).** Эти переменные создаются и удаляются с помощью вызовов *pthread_cond_init* и *pthread_cond__destroy* соответственно.

Если, например, поток обнаружил, что накопитель на магнитной ленте, который ему нужен, в данный момент занят, этот поток совершает *pthread_cond_wait* над переменной условия. Когда поток, который использует накопитель на магнитной ленте, завершает свою работу с этим устройством (а это может произойти через несколько часов), он посылает сигнал *pthread_cond_signal,* чтобы разблокировать ровно один поток, который ожидает эту переменную условия. Если ни один поток не ожидает эту переменную, сигнал пропадает. У переменных условия (condition variables) нет счетчика, как у семафоров. Отметим, что над потоками, мьютексами и переменными условия можно выполнять и некоторые другие операции.

Управление процессами в Windows NT

NT поддерживает несколько процессов, которые могут взаимодействовать и синхронизироваться. Каждый процесс содержит по крайней мере один поток, который, в свою очередь, содержит по крайней мере одну **нить** (fiber) (это легковесный поток). Процессы, потоки и нити в совокупности обеспечивают набор средств для поддержания параллелизма и в машинах с одним процессором, и в многопроцессорных системах.

Новые процессы создаются с помощью функции API *CreateProcess.* Эта функция имеет 10 параметров, каждый из которых имеет множество опций. Ясно, что такая разработка гораздо сложнее соответствующей схемы в UNIX, *где fork* вообще не имеет параметров, а *exec* имеет всего три параметра: указатели на имя файла, который нужно выполнить, на массив параметров командной строки и на строки описания конфигурации. Ниже изложены 10 параметров функции *CreateProcess:*

1. Указатель на имя выполняемого файла.
2. Сама командная строка.
3. Указатель на дескриптор защиты для данного процесса.
4. Указатель на дескриптор защиты для внутреннего потока.
5. Бит, который сообщает, наследует ли новый процесс идентификаторы (handles) исходного процесса.
6. Различные флаги (например, ошибка, приоритет, отладка, консоль).
7. Указатель на строки описания конфигурации.

8. Указатель на имя текущего каталога нового процесса.
9. Указатель на структуру, которая описывает исходное окно экрана.
10. Указатель на структуру, которая возвращает 18 значений вызывающей процедуре.

В NT нет никакой иерархии типа порождающий—порожденный. Все процессы создаются равными. Но поскольку одним из 18 параметров, возвращаемых исходному процессу, является идентификатор (handle) для нового процесса (а это дает возможность контролировать новый процесс), здесь существует внутренняя иерархия с точки зрения того, что определенные процессы содержат идентификаторы других процессов. Эти идентификаторы нельзя просто непосредственно передавать другим процессам, но процесс может сделать определенный идентификатор доступным для другого процесса, а затем передать ему этот идентификатор, так что внутренняя иерархия процессов не может сохраняться долго.

Каждый процесс в NT создается с одним потоком, но позднее этот процесс может создать еще несколько потоков. Создание потока проще, чем создание процесса, поскольку *CreateThread* имеет всего 6 параметров вместо 10: дескриптор защиты, размер стека, начальный адрес, определяемый пользователем параметр, начальное состояние потока (готов к работе или блокирован) и идентификатор потока.

Создание потоков производится ядром (то есть потоки реализуются не в пользовательском пространстве, как в некоторых других системах).

Когда ядро совершает распределение, оно вызывает не только процесс, который должен запускаться следующим, но и поток в этом процессе. Это значит, что ядро всегда знает, какие потоки блокированы, а какие — нет. Так как потоки являются объектами ядра, они имеют дескрипторы защиты и идентификаторы. Поскольку идентификатор разрешается передавать другому процессу, можно сделать так, чтобы один процесс управлял потоками в другом процессе. Эта особенность может понадобиться для программ отладки.

Создавать потоки в NT довольно расточительно, поскольку для создания потока требуется войти в ядро, а затем выйти из него. Чтобы избежать этого, в NT предусмотрены **нити** (fibers), которые похожи на потоки, но распределяются в пользовательском пространстве программой, которая их создает. Каждый поток может иметь несколько нитей, точно так же как процесс может иметь несколько потоков, только в данном случае, когда нить блокируется, она встает в очередь заблокированных нитей и выбирает другую нить для работы в своем потоке. Ядро не знает об этом переходе, поскольку поток все равно продолжает работать, даже если сначала действовала одна нить, а затем другая. Ядро управляет процессами и потоками, но не управляет нитями. Нити могут пригодиться, например, в том случае, когда программы, которые управляют своими собственными потоками, переносятся в NT.

Процессы могут взаимодействовать друг с другом разными способами: через каналы, именованные каналы, почтовые ящики, сокеты (sockets), удаленные вызовы процедур и общие файлы. Каналы бывают двух видов: байтовые каналы и каналы сообщений. Тип выбирается во время создания. Байтовые каналы работают так же, как в системе UNIX. Каналы сообщений сохраняют границы сообщений, поэтому четыре записи по 128 байтов будут прочитаны как четыре сообщения по 128 байтов (а не как одно сообщение на 512 байтов, как в случае с байтовыми каналами). Кроме того, существуют именованные каналы, которые тоже бывают двух видов. Именованные каналы могут использоваться в сети, а обычные каналы — нет.

Почтовые ящики есть только в NT (в системе UNIX их нет). **Они** во многом похожи на каналы, хотя не во всем. Во-первых, они односторонние, а каналы — двусторонние. Их можно использовать в сети, но они не гарантируют доставку. Наконец, они позволяют отправлять сообщение нескольким получателям, а не только одному.

Сокеты похожи на каналы, но они обычно связывают процессы на разных машинах. Однако их можно применять и для связи процессов на одной машине. Вообще говоря, связь через сокет ненамного выгоднее связи через обычный или именованный канал.

Удаленные вызовы процедур позволяют процессу A приказать процессу B совершить вызов процедуры в адресном пространстве B от имени A и возвратить результат процессу A. Здесь существуют различные ограничения на параметры. Например, не имеет никакого смысла передача указателя другому процессу.

Наконец, процессы могут разделять общую память путем отображения одновременно в один и тот же файл. Тогда все записи, произведенные одним процессом, появляются в адресном пространстве других процессов. Применяя такой механизм, можно легко реализовать разделенный буфер, который мы описывали в примере с процессом-производителем и процессом-потребителем.

NT предоставляет множество механизмов синхронизации (семафоры, мьютексы, критические секции и события). Все эти механизмы работают с потоками, но не с процессами, поэтому когда поток блокируется на семафоре, это никак не влияет на другие потоки в этом процессе — они просто продолжают работать.

Семафор создается с помощью функции API *CreateSemaphore,* которая может установить его на определенное значение и определить его максимальное значение. Семафоры являются объектами ядра, поэтому они имеют дескрипторы защиты и идентификаторы (handles). Идентификатор для семафора может дублироваться с помощью *DuplicateHandle* и передаваться другому процессу, поэтому на одном семафоре можно синхронизировать несколько процессов. Присутствуют функции *up* и *down,* хотя они имеют особые названия: *ReleaseSemaphore (up)* и *WaitForSingleObject(down).* Можно определить для функции *WaitForSingleObject* предел на время простоя, и тогда вызывающий поток в конце концов может быть разблокирован, даже если семафор сохраняет значение 0.

Мьютексы тоже являются объектами ядра, но они проще семафоров, поскольку у них нет счетчиков. Они, по сути, представляют собой объекты с функциями API для блокирования *(WaitForSingleObject)* и разблокирования *(ReleaseMutex).* Идентификаторы мьютексов, как и идентификаторы семафоров, можно дублировать и передавать другим процессам, так что потоки из разных процессов могут иметь доступ к одному и тому же мьютексу.

Третий механизм синхронизации основан на **критических секциях,** которые сходны с мьютексами, за исключением локальности по отношению к адресному пространству исходного потока. Поскольку критические секции не являются объектами ядра, у них нет идентификаторов (handles) и дескрипторов защиты и их нельзя передавать другим процессам. Блокировка и разблокировка осуществляются с помощью *EnterCriticalSection* и *LeaveCriticalSection* соответственно. Так как эти функции API выполняются целиком в пользовательском пространстве, они работают гораздо быстрее, чем мьютексы.

Последний механизм связан с использованием объектов ядра, которые называются **событиями.** Если потоку нужно дождаться какого-то события, он вызывает *WaitForSingleObject.* Можно разблокировать один ожидающий поток с помощью *SetEvent* или все ожидающие потоки с помощью *PulseEvent.* Существует несколько видов событий, которые имеют несколько опций.

События, мьютексы и семафоры можно назвать определенным образом и хранить в системе файлов как именованные каналы. Можно синхронизировать работу двух и более процессов путем открытия одного и того же события, мьютекса или семафора. В этом случае им не нужно создавать объект, а затем дублировать идентификаторы, хотя такой подход тоже возможен.

Краткое содержание главы

Операционную систему можно считать интерпретатором определенных особенностей архитектуры, которых нет на уровне команд. Главными среди них являются виртуальная память, виртуальные команды ввода-вывода и средства параллельной обработки.

Виртуальная память нужна для того, чтобы позволить прграммам использовать больше адресного пространства, чем есть у машины на самом деле, или чтобы обеспечить удобный механизм для защиты и разделения памяти. Виртуальную память можно реализовать в виде чистого разбиения на страницы, чистой сегментации или того и другого вместе. При страничной организации памяти адресное пространтсво разбивается на равные по размеру виртуальные страницы. Одни из них отображаются на физические страничные кадры, другие — нет. Отсылка к отображенной странице преобразуется контроллером управления памятью в правильный физический адрес. Обращение к неотображенной странице вызывает ошибку из-за отсутствия страницы. Pentium II и UltraSPARC II имеют сложные контроллеры управления памятью, которые поддерживают виртуальную память и страничную организацию.

Самой важной абстракцией ввода-вывода на этом уровне является файл. Файл состоит из последовательности байтов или логических записей, которые можно читать и записывать, не зная при этом о том, как работают диски и другие устройства ввода-вывода. Доступ к файлам может осуществляться последовательно, непоследовательно по номеру записи и непоследовательно по ключу. Для группировки файлов используются директории. Файлы могут храниться в последовательных секторах, а могут быть разбросаны по диску. В последнем случае требуются специальные структуры данных для нахождения всех блоков файла. Чтобы следить за свободным пространством на диске, можно использовать список или битовое отображение.

В системах часто поддерживается параллельная обработка. Для этого путем разделения времени в одном процессоре моделируется несколько процессов. Неконтролируемое взаимодействие различных процессов может привести к состоянию гонок. Чтобы избежать их, вводятся специальные средства синхронизации. Самыми простыми из них являются семафоры.

UNIX и NT являются сложными операционными системами. Обе системы поддерживают страничную организацию памяти и отображаемые в память файлы.

Кроме того, они поддерживают иерархические системы файлов, в которых файлы состоят из последовательности байтов. Наконец, обе системы поддерживают процессы и потоки и обеспечивают механизмы их синхронизации.

Вопросы и задания

1. Почему операционная система интепретирует только некоторые команды третьего уровня, тогда как микропрограмма интерпретирует все команды этого уровня (уровня архитектуры команд)?
2. Машина содержит 32-битное виртуальное адресное пространство с побайтовой адресацией. Размер страницы составляет 8 Кбайт. Сколько существует страниц виртуального адресного пространства?
3. Виртуальная память содержит 8 виртуальных страниц и 4 физических страничных кадра. Размер страницы составляет 1024 слова. Ниже приведена таблица страниц:

Виртуальная страница	Страничный кадр
0	3
1	1
2	Нет в основной памяти
3	Нет в основной памяти
4	2
5	Нет в основной памяти
6	0
7	Нет в основной памяти

   1. Создайте список виртуальных адресов, обращение к которым будет вызывать ошибку из-за отсутствия страницы.
   2. Каковы физические адреса для 0, 3728, 1023, 1024,1025, 7800 и 4096?
4. Компьютер имеет 16 страниц виртуального адресного пространства и только 4 страничных кадра. Изначально память пуста. Программа обращается к виртуальным страницам в следующем порядке:

   0,7,2,7,5,8,9,2,4

   а. Какие из обращений вызовут ошибку с алгоритмом LRU?

   б. Какие из обращений вызовут ошибку с алгоритмом FIFO?
5. В разделе «Политика замещения страниц» был предложен алогоритм замещения страниц FIFO. Разработайте более эффективный алгоритм. *Подсказка:* можно обновлять счетчик во вновь загружаемой странице, оставляя все другие.
6. В системах со страничной организацией памяти, которые мы обсуждали в этой главе, обработчик ошибок, происходящих из-за отсутствия страниц, был частью уровня архитектуры команд и, следовательно, не присутствовал в адресном пространстве операционной системы. На практике же этот обработчик

занимает некоторые страницы и может быть удален при определенных обстоятельствах (например, в соответствии с политикой замещения страниц). Что бы случилось, если бы обработчика ошибок не было в наличии в тот момент, когда произошла ошибка? Как можно было бы разрешить эту проблему?

7. Не все компьютеры содержат специальный бит, который автоматически устанавливается, когда производится запись в страницу. Однако нужно каким-то образом следить, какие страницы были изменены, чтобы не пришлось записывать все страницы обратно на диск после их использования. Если предположить, что каждая страница имеет специальные биты для разрешения чтения, записи и выполнения, то как операционная система может следить, какие страницы изменялись, а какие — нет?
8. Сегментированная память содержит страничные сегменты. Каждый виртуальный адрес содержит 2-битный номер сегмента, 2-битный номер страницы и 11-битное смещение внутри страницы. Основная память содержит 32 Кбайт, которые разделены на страницы по 2 Кбайт. Каждый сегмент разрешается либо только читать, либо читать и выполнять, либо читать и записывать, либо читать, записывать и выполнять. Таблицы страниц с указанием на защиту приведены ниже:

Сегмент 0		**Сегмент 1**		**Сегмент 2**	**Сегмент 3**	
Только для чтения		Чтение/выполнение		Чтение/запись/ выполнение	Чтение/запись	
Виртуальная страница	Страничный кадр	Виртуальная страница	Страничный кадр		Виртуальная страница	Страничный кадр
0	9	0	На диске	Таблицы	0	14
1	3	1	0	страниц нет	1	1
2	На диске	2	15	в основной	2	6
3	12	3	8	памяти	3	На диске

Вычислите физический адрес для каждого из нижеперечисленных доступов к виртуальной памяти. Если происходит ошибка, скажите, какого она типа.

Доступ	Сегмент	Страница	Смещение внутри страницы
1. Вызов данных	0	1	1
2. Вызов данных	1	1	10
3. Вызов данных	3	3	2047
4. Сохранение данных	0	1	4
5. Сохранение данных	3	1	2
6. Сохранени данных	3	0	14
7. Переход	1	3	100
8. Вызов данных	0	2	50
9. Вызов данных	2	0	5
10. Переход	3	0	60

9. Некоторые компьютеры позволяют осуществлять ввод-вывод непосредственно в пользовательское пространство. Например, программа может начать передачу данных с диска в буфер внутри пользовательского процесса. Вызовет ли это какие-либо проблемы, если для реализации виртуальной памяти используется уплотнение? Аргументируйте.
10. Операционные системы, в которых допускаются проецируемые в память файлы, всегда требуют, чтобы файлы были отображены в границах страниц. Например, если у нас есть страницы по 4 K, файл может быть отображен, начиная с виртуального адреса 4096, но не с виртуального адреса 5000. Зачем это нужно?
11. При загрузке сегментного регистра в Pentium II вызывается соответствующий дескриптор, который загружается в невидимую часть сегментного регистра. Как вы думаете, почему разработчики Intel решили это сделать?
12. Программа в компьютере Pentium II обращается к локальному сегменту 10 со смещением 8000. Поле BASE сегмента 10 в локальной таблице дескрипторов содержит число 10000. Какой элемент таблицы страниц использует Pentium II? Каков номер страницы? Каково смещение?
13. Рассмотрите возможные алгоритмы для удаления сегментов в сегментированной памяти без страничной организации.
14. Сравните внутреннюю фрагментацию с внешней фрагментацией. Что можно сделать, чтобы улучшить каждую из них?
15. Супермаркеты часто сталкиваются с проблемой, сходной с замещением страниц в системах с виртуальной памятью. В супермаркетах есть фиксированная площадь пространства на полках, куда требуется помещать все больше и больше различных товаров. Если поступил новый важный продукт, например питание для собак очень высокого качества, какой-либо другой продукт нужно убрать, чтобы освободить место для нового продукта. Мы знаем два алгоритма: LRU и FIFO. Какой из них вы бы предпочли?
16. Почему блоки кэш-памяти всегда намного меньше, чем страницы в виртуальной памяти (бывает даже, что в 100 раз меньше)?
17. Почему многие системы файлов требуют, чтобы файл перед прочтением явным образом открывался с помощью системного вызова *open?*
18. Сравните применение битового отображения и списка неиспользованных пространств для слежения за свободным пространством на диске. Диск состоит из 800 цилиндров, на каждом из которых расположено 5 дорожек по 32 сектора. Сколько понадобится «дырок», чтобы список «дырок» (список свободной памяти) стал больше, чем битовое отображение? Предполагается, что единичный блок — это сектор и что для «дырки» требуется 32-битный элемент таблицы.
19. Чтобы сделать некоторые прогнозы относительно производительности диска, нужно иметь модель распределения памяти. Предположим, что диск рассматривается как линейное адресное пространство из N» 1 секторов. Здесь сначала идет последовательность блоков данных, затем неиспользованное пространство, затем другая последовательность блоков данных и т. д. Эмпирические измерения показывают, что вероятностные распределения для

длин данных и неиспользованных пространств одинаковы, причем для каждого из них вероятность быть *i* секторов составляет 2^{-i}. Каково при этом ожидаемое число «дырок» на диске?

20. На определенной машине программа может создавать столько файлов, сколько ей нужно, и все файлы могут увеличиваться в размерах во время выполнения программы, причем операционная система не получает никаких дополнительных данных об их конечном размере. Как вы думаете, хранятся ли файлы в последовательных секторах? Поясните.

21. Рассмотрим один метод реализации команд для работы с семафорами. Всякий раз, когда центральный процессор собирается совершить команду up или down над семафором (семафор — это целочисленная переменная в памяти), сначала он устанавливает приоритет центрального процессора таким образом, чтобы блокировать все прерывания. Затем он вызывает из памяти семафор, изменяет его и в соответствии с этим совершает переход. После этого он снова снимает запрет с прерываний. Будет ли этот метод работать, если:

    а. Существует один центральный процессор, который переключается между процессами каждые 100 миллисекунд?

    б. Два центральных процессора разделяют общую память, в которой расположен семафор?

22. Компания, разрабатывающая операционные системы, получает жалобы от своих клиентов по поводу последней разработки, которая поддерживает операции с семафорами. Клиенты решили, что аморально со стороны процессов приостанавливать свою работу (то есть спать на работе). Чтобы угодить своим клиентам, компания решила добавить третью операцию, *peek*. Эта операция просто проверяет семафор, но не изменяет его и не блокирует процесс. Таким образом, программы сначала проверяют, можно ли делать над семафором операцию *down*. Будет ли эта идея работать, если семафор используют три и более процессов? А если два процесса?

23. Составьте таблицу, в которой в виде функции от времени от 0 до 1000 миллисекунд показано, какие из трех процессов P1, P2 и P3 работают, а какие блокированы. Все три процесса выполняют команды up и down над одним и тем же семафором. Если два процесса блокированы и совершается команда up, то запускается процесс с меньшим номером, то есть P1 имеет преимущество над P2 и P3 и т. д. Изначально все три процесса работают, а значение семафора равно 1.

    При t=100 P1 совершает down.

    При t=200 PI совершает down.

    При t=300 PI совершает up.

    При t=400 PI совершает down.

    При t=500 PI совершает down.

    При t=600 PI совершает up.

    При t=700 PI совершает down.

    При t=800 PI совершает up.

    При t=900 PI совершает up.

24. В системе бронирования билетов на авиарейсы необходимо быть уверенным в том, что пока один процесс использует файл, никакой другой процесс не может использовать этот же файл. В противном случае два разных процесса, которые работают на два разных агентства по продаже билетов, могут продать последнее оставшееся место двум пассажирам. Разработайте метод синхронизации с использованием семафоров, чтобы точно знать, что только один процесс в конкретный момент времени может получать доступ к файлу (предполагается, что процессы подчиняются правилам).
25. Чтобы сделать возможной реализацию семафоров на компьютере с несколькими процессорами, которые разделяют общую память, разработчики включают в машину команду для проверки и блокирования. Команда TSL X проверяет ячейку X. Если содержание равно 0, семафоры устанавливаются на 1 за один неделимый цикл памяти, а следующая команда пропускается. Если содержание ячейки не равно О, TSL работает как пустая операция. Используя TSL, можно написать процедуры *lock* и *unlock* со следующими свойствами: *lock{x)* проверяет, заперт ли *x*. Если нет, эта процедура запирает *x* и возвращает управление; *unlock* отменяет существующую блокировку. Если *x* уже заперт, процедура просто ждет, пока он не освободится, и только после этого запирает *x* и возвращает управление. Если все процессы запирают таблицу семафоров перед ее использованием, то в определенный момент времени только один процесс может производить операции с переменными и указателями, что предотвращает состояние гонок. Напишите процедуры *lock* и *unlock* на ассемблере.
26. Каково будет значение *in* и *out* для кольцевого буфера длиной в 65 слов после каждой из следующих операций? Изначально значения *in* и *out* равны 0.
    а. 22 слова помещаются в буфер;
    б. 9 слов удаляются из буфера;
    в. 40 слов помещаются в буфер;
    г. 17 слов удаляются из буфера;
    д. 12 слов помещаются в буфер;
    е. 45 слов удаляются из буфера;
    ж. 8 слов помещаются в буфер;
    з. 11 слов удаляются из буфера.
27. Предположим, что одна из версий UNIX использует 2 K блоков на диске и хранит 512 адресов диска на каждый блок косвенной адресации (обычной косвенной адресации, двойной и тройной). Каков будет максимальный размер файла? Предполагается, что размер указателей файла составляет 64 бита.
28. Предположим, что системный вызов UNIX

    ```
    unlink("/usr/ast/bin/game3")
    ```

    был выполнен в контексте рис. 6.27. Опишите подробно, какие изменения произойдут в системе директорий.
29. Представьте, что вам нужно реализовать систему UNIX на микрокомпьютере, где основной памяти недостаточно. После долгой работы она все еще не вполне влезает в память, и вы выбираете системный вызов наугад, чтобы

пожертвовать им для общего блага. Вы выбрали системный вызов pipe, который создает каналы для передачи потоков байтов от одного процесса к другому. Возможно ли после этого как-то изменить ввод-вывод? Что вы можете сказать о конвейерах? Рассмотрите проблемы и возможные решения.

30. Комиссия по защите дескрипторов файлов выдвинула протест против системы UNIX, потому что когда эта система возвращает дескриптор файла, она всегда возвращает самый маленький номер, который в данный момент не используется. Следовательно, едва ли когда-нибудь будут использоваться дескрипторы файлов с большими номерами. Комиссия настаивает на том, чтобы система возвращала дескриптор с самым маленьким номером из тех, которые еще не использовались программой, а не из тех, которые не используются в данный момент. Комиссия утверждает, что эту идею легко реализовать, это не повлияет на существующие программы и, кроме того, это будет гораздо справедливее по отношению к дескрипторам. А что вы думаете по этому поводу?

31. В системе NT можно составить список управления доступом таким образом, чтобы один пользователь не имел доступа ни к одному из файлов, а все остальные имели полный доступ к ним. Как это можно реализовать?

32. Опишите два способа программирования работы процессора-производителя и процессора-потребителя с использованием общих буферов и семафоров в NT. Подумайте о том, как можно реализовать разделенный буфер в каждом из двух случаев.

33. Работу алгоритмов замещения страниц обычно проверяют путем моделирования. Предположим, что вам нужно написать моделирующую программу для виртуальной памяти со страничной организацией для машины, содержащей 64 страницы по 1 Кбайт. Программа должна поддерживать одну таблицу из 64 элементов, один элемент на страницу. Каждый элемент таблицы содержит номер физической страницы, который соответствует данной виртуальной странице. Моделирующая программа должна считывать файл, содержащий виртуальные адреса в десятичной системе счисления, по одному адресу на строку. Если соответствующая страница находится в памяти, просто записывайте наличие страницы. Если ее нет в памяти, вызовите процедуру замещения страниц, чтобы выбрать страницу, которую можно выкинуть (то есть элемент таблицы, который нужно переписать), и записывайте отсутствие страницы. Никакой передачи страниц не происходит. Создайте файл, состоящий из непоследовательных адресов, и проверьте производительность работы двух алгоритмов: LRU и FIFO. А теперь создайте файл адресов, в котором x процентов адресов находятся на 4 байта выше, чем предыдущие. Проведите тесты для различных значений x и сообщите о полученных результатах.

34. Напишите программу для UNIX или NT, которая на входе получает имя директории. Программа должна печатать список файлов этой директории, каждый файл на отдельной строке, а после имени файла должен печататься размер файла. Имена файлов должны располагаться в том порядке, в котором они появляются в директории. Неиспользованные слоты в директории должны выводиться с пометой (неиспользованный).

Глава 7
Уровень языка ассемблера

В четвертой, пятой и шестой главах мы обсуждали три уровня, которые имеются в большинстве современных компьютеров. В этой главе речь пойдет о еще одном уровне, который также присутствует практически во всех современных машинах. Это уровень языка ассемблера. Уровень языка ассемблера существенно отличается от трех предыдущих, поскольку он реализуется с помощью трансляции, а не с помощью интерпретации.

Программы, которые преобразуют пользовательские программы, написанные на каком-либо определенном языке, в другой язык, называются **трансляторами.** Язык, на котором изначально написана программа, называется **входным языком,** а язык, на который транслируется эта программа, называется **выходным языком.** Входной язык и выходной язык определяют уровни. Если имеется процессор, который может выполнять программы, написанные на входном языке, то нет необходимости транслировать исходную программу на другой язык.

Трансляция используется в том случае, если есть аппаратный или программный процессор для выходного языка и нет процессора для входного языка. Если трансляция выполнена правильно, то оттранслированная программа будет давать точно такие же результаты, что и исходная программа (если бы существовал подходящий для нее процессор). Следовательно, можно организовать новый уровень, который сначала будет транслировать программы на выходной уровень, а затем выполнять полученные программы.

Важно понимать разницу между трансляцией и интерпретацией[1]. При трансляции исходная программа на входном языке не выполняется сразу. Сначала она преобразуется в эквивалентную программу, так называемую **объектную программу,** или **исполняемую двоичную программу,** которая выполняется только после завершения трансляции. При трансляции нужно пройти следующие два шага:

1. Создание эквивалентной программы на выходном языке.
2. Выполнение полученной программы.

Эти два шага выполняются не одновременно. Второй шаг начинается только после завершения первого. В интерпретации есть только один шаг: выполнение исходной программы. Никакой эквивалентной программы порождать не нужно, хотя иногда исходная программа преобразуется в промежуточную форму (например, в код Java) для упрощения интерпретации.

[1] В отечественной литературе принято и интерпретацию, и компиляцию (именно компиляцию автор здесь называет трансляцией) называть трансляцией. Другими словами, трансляторы могут быть либо компиляторами, либо интерпретаторами. — *Примеч. научн. ред.*

Во время выполнения объектной программы задействовано только три уровня: микроархитектурный уровень, уровень команд и уровень операционной системы. Следовательно, во время работы программы в памяти компьютера можно найти три программы: пользовательскую объектную программу, операционную систему и микропрограмму (если она есть). Никаких следов исходной программы не остается. Таким образом, число уровней, присутствующих при выполнении программы, может отличаться от числа уровней, присутствующих до трансляции. Следует отметить, что хотя мы определяем уровень по командам и языковым конструкциям, доступным программистам этого уровня (а не по технологии реализации), некоторые авторы иногда проводят различие между уровнями, реализованными интерпретаторами, и уровнями, реализованными при трансляции.

Введение в язык ассемблера

Трансляторы можно разделить на две группы в зависимости от отношения между входным и выходным языком. Если входной язык является символической репрезентацией числового машинного языка, то транслятор называется **ассемблером,** а входной язык называется **языком ассемблера.** Если входной язык является языком высокого уровня (например, Java или C), а выходной язык является либо числовым машинным языком, либо символической репрезентацией последнего, то транслятор называется **компилятором.**

Что такое язык ассемблера?

Язык ассемблера — это язык, в котором каждое высказывание соответствует ровно одной машинной команде. Иными словами, существует взаимно однозначное соответствие между машинными командами и операторами в программе на языке ассемблера. Если каждая строка в программе на языке ассемблера содержит ровно один оператор и каждое машинное слово содержит ровно одну команду, то программа на языке ассемблера в n строк произведет программу на машинном языке из n слов.

Мы используем язык ассемблера, а не программируем на машинном языке (в шестнадцатеричной системе счисления), поскольку на языке ассемблера программировать гораздо проще. Использовать символьные имена и адреса вместо двоичных и восьмеричных намного удобнее. Многие могут запомнить, что обозначениями для сложения (add), вычитания (subtract), умножения (multiply) и деления (divide) служат команды ADD, SUB, MUL и DIV, но мало кто может запомнить соответствующие числа, которые использует машина. Программисту на языке ассемблера нужно знать только символические названия, поскольку ассемблер транслирует их в машинные команды.

Это утверждение касается и адресов. Программист на языке ассемблера может дать имена ячейкам памяти, и уже ассемблер должен будет выдавать правильные числа. Программист на машинном языке всегда должен работать с числовыми номерами адресов. Сейчас уже нет программистов, которые пишут программы на машинном языке, хотя несколько десятилетий назад до изобретения ассемблеров программы именно так и писались.

Язык ассемблера имеет несколько особенностей, отличающих его от языков высокого уровня. Во-первых, это взаимно однозначное соответствие между высказываниями языка ассемблера и машинными командами (об этом мы уже говорили). Во-вторых, программист на языке ассемблера имеет доступ ко всем объектам и командам, присутствующим на целевой машине. У программистов на языках высокого уровня такого доступа нет. Например, если целевая машина содержит бит переполнения, программа на языке ассемблера может проверить его, а программа на языке Java не может. Программа на языке ассемблера может выполнить любую команду из набора команд целевой машины, а программа на языке высокого уровня не может. Короче говоря, все, что можно сделать в машинном языке, можно сделать и на языке ассемблера, но многие команды, регистры и другие объекты недоступны для программиста, пишущего программы на языке высокого уровня. Языки для системного программирования (например C) часто занимают промежуточное положение. Они обладают синтаксисом языка высокого уровня, но при этом с точки зрения возможностей доступа ближе к языку ассемблера.

Наконец, программа на языке ассемблера может работать только на компьютерах одного семейства, а программа, написанная на языке высокого уровня, потенциально может работать на разных машинах. Возможность переносить программное обеспечение с одной машины на другую очень важна для многих прикладных программ.

Зачем нужен язык ассемблера?

Язык ассемблера довольно труден. Написание программы на языке ассемблера занимает гораздо больше времени, чем написание той же программы на языке высокого уровня. Кроме того, очень много времени занимает отладка.

Но зачем же тогда вообще писать программы на языке ассемблера? Есть две причины: производительность и доступ к машине. Во-первых, профессиональный программист языка ассемблера может составить гораздо меньшую по размеру программу, которая будет работать гораздо быстрее, чем программа, написанная на языке высокого уровня. Для некоторых программ скорость и размер весьма важны. Многие встроенные прикладные программы, например программы в кредитных карточках, сотовых телефонах, драйверах устройств, а также процедуры BIOS попадают в эту категорию.

Во-вторых, некоторым процедурам требуется полный доступ к аппаратному обеспечению, что обычно невозможно сделать на языке высокого уровня. В эту категорию попадают прерывания и обработчики прерываний в операционных системах, а также контроллеры устройств во встроенных системах, работающих в режиме реального времени.

Первая причина (достижение высокой производительности) является более важной, поэтому мы рассмотрим ее подробнее. В большинстве программ лишь небольшой процент всего кода отвечает за большой процент времени выполнения программы. Обычно 1% программы отвечает за 50% времени выполнения, а 10% программы отвечает за 90% времени выполнения.

Предположим, что для написания программы на языке высокого уровня требуется 10 человеко-лет и что полученной программе требуется 100 секунд, чтобы

выполнить некоторую типичную контрольную задачу. (**Контрольная задача** — это программа проверки, которая используется для сравнения компьютеров, компиляторов и т. п.). Написание всей программы на языке ассемблера может занять 50 человеко-лет. Полученная в результате программа будет выполнять контрольную задачу примерно за 33 секунды, поскольку хороший программист может оказаться в три раза умнее компилятора (хотя об этом можно спорить бесконечно). Ситуация проиллюстрирована в табл. 7.1.

Так как только крошечная часть программы отвечает за большую часть времени выполнения этой программы, возможен другой подход. Сначала программа пишется на языке высокого уровня. Затем проводится ряд измерений, чтобы определить, какие части программы отвечают за большую часть времени выполнения. Для таких измерений обычно используется системный тактовый генератор. С его помощью можно узнать, сколько времени затрачивается на каждую процедуру, сколько раз выполняется каждый цикл и т. п.

Предположим, что 10% программы отвечает за 90% времени ее выполнения. Это значит, что из всех 100 секунд работы 90 секунд проводится в этих 10%, а 10 секунд — в оставшихся 90% программы. Эти 10% программы можно усовершенствовать, переписав их на язык ассемблера. Этот процесс называется **настройкой** (tuning). Он проиллюстрирован в табл. 7.1. На переделку основных процедур потребуется еще 5 лет, но время выполнения программы сократится с 90 секунд до 30 секунд.

Таблица 7.1. Сравнение программирования на языке ассемблера и на языке высокого уровня (с настройкой и без настройки)

	Количество человеко-лет, затрачиваемых на написание программы	**Время выполнения программы в секундах**
Язык ассемблера	**50**	**33**
Язык высокого уровня	**10**	100
Смешанный подход до настройки		
Критические 10%	**1**	**90**
Остальные 90%	**9**	**10**
Всего	10	100
Смешанный подход после настройки		
Критические 10%	6	30
Остальные 90%	9	10
Всего	15	40

Сравним этот смешанный подход, в котором используется и язык ассемблера, и язык высокого уровня, с подходом, в котором применяется только язык ассемблера (табл. 7.1). При втором подходе программа работает примерно на 20% быстрее (33 секунды против 40 секунд), но более чем за тройную цену (50 человеко-лет против 15). Более того, у смешанного подхода есть еще одно преимущество: гораздо проще переделать в код ассемблера уже отлаженную процедуру, написанную на языке высокого уровня, чем писать процедуру на языке ассемблера с нуля.

Отметим, что если бы написание программы занимало только 1 год, соотношение между смешанным подходом и подходом, при котором используется только язык ассемблера, составляло бы 4:1 в пользу смешанного подхода.

Программист, который использует язык высокого уровня, не занят перемещением битов и может так решить задачу, так построить программу, что в конце концов достигнет действительно большого увеличения производительности. А программисты, пишущие программы на языке ассемблера, обычно стараются так построить команды, чтобы сэкономить несколько циклов, поэтому у них такой ситуации возникнуть не может.

Расскажем о двух экспериментах, проведенных во время разработки системы MULTICS. Грехем [49] описал процедуру PL/I, за три месяца переделанную в новую версию, которая была в 26 раз меньше и работала в 50 раз быстрее, чем исходная. Он описал еще одну процедуру PL/L, которая получилась в 20 раз меньше исходной и работала в 40 раз быстрее, чем исходная, после двух месяцев работы. Корбато [27] описал процедуру PL/I, размер кода которой был сокращен с 50 000 слов до 1000 слов менее чем за месяц, а контроллер уменьшен с 65 000 до 30 000 слов с увеличением производительности в 8 раз за 4 месяца. Здесь важно понимать, что у программистов языков высокого уровня глобальный подход к тому, что они делают, поэтому они гораздо быстрее могут разработать лучший алгоритм.

Однако, несмотря на все это, существует по крайней мере 4 веские причины для изучения языка ассемблера. Во-первых, желательно уметь писать программы на языке ассемблера, поскольку успех или неудача большого проекта может зависеть от того, можно ли повысить производительность какой-то важной процедуры в 2 или 3 раза.

Во-вторых, язык ассемблера может быть единственным возможным выходом из-за недостатка памяти. Кредитные карты, например, содержат центральный процессор, но у них нет мегабайта памяти и жесткого диска. Однако они должны выполнять сложные вычисления при наличии ограниченных ресурсов. Процессоры, встроенные в электроприборы, часто имеют минимальное количество памяти, поскольку они должны быть достаточно дешевыми. Различные электронные устройства, работающие на батарейках, обычно содержат очень маленькую память, поэтому здесь тоже нужен эффективный код.

В-третьих, компилятор должен либо давать на выходе программу, которая используется ассемблером, либо самостоятельно выполнять процесс ассемблирования. Таким образом, понимание языка ассемблера существенно для понимания того, как работает компьютер. И вообще, кто-то ведь должен писать компилятор и его ассемблер.

Наконец, изучение языка ассемблера дает прекрасное представление о реальной машине. Для тех, кто изучает архитектуру компьютеров, написание программы на языке ассемблера — единственный способ узнать, что собой представляет машина на архитектурном уровне.

Формат оператора в языке ассемблера

Хотя структура оператора в языке ассемблера отражает структуру соответствующей машинной команды, языки ассемблера для разных машин и разных уровней во многом сходны друг с другом, что позволяет говорить о языке ассемблера вооб-

ще. В таблицах 7.2-7.4 показаны фрагменты программ на языке ассемблера для Pentium II, Motorola 680x0 и (Ultra)SPARC. Все эти программы выполняют вычисление N=I+J. Во всех трех примерах операторы над пропуском в таблице выполняют вычисление. Операторы под пропуском — это указания ассемблеру зарезервировать память для переменных I, J и N. Последние не являются символьными репрезентациями машинных команд.

Таблица 7.2. Вычисление выражения N=I+J в Pentium II

Метка	Код операции	Операнды	Комментарии
FORMULA:	MOV	EAX.I	; регистр EAX=1
	ADD	EAX.J	; регистр EAX=I+J
	MOV	N.EAX	; N=I+J
I	DW	3	; резервируем 4 байта и устанавливаем значение 3
J	DW	4	; резервируем 4 байта и устанавливаем значение 4
N	DW	O	; резервируем 4 байта и устанавливаем значение O

Таблица 7.3. Вычисление выражения N=I+J в Motorola 680x0

Метка	Код операции	Операнды	Комментарии
FORMULA:	MOVE.L	I,DO	; регистр D0=I
	ADD.L	J,D0	; регистр D0=I+J
	MOVE.L	D0,N	; N=I+J
I	DC.L	3	; резервируем 4 байта и устанавливаем значение 3
J	DC.L	4	; резервируем 4 байта и устанавливаем значение 4
N	DC.L	0	; резервируем 4 байта и устанавливаем значение 0

Таблица 7.4. Вычисление выражения N=I+J в SPARC

Метка	Код операции	Операнды	Комментарии
FORMULA:	SETHI	%HI(I),%R1	! R1 = старшие биты адреса I
	LD	[%R1+%LO(I)],%R1	! R1=I
	SETHI	%HI(J),%R2	! R2 = старшие биты адреса J
	LD	[%R2+%LO(J)],%R2	!R2=J
	NOP		! ждем прибытия J из памяти
	ADD	%R1,%R2,%R2	!R2=R1+R2
	SETHI	%HI(N),%R1	! R1 = старшие биты адреса N
	ST	%R2, [%R1+%LO(N)]	
I:	.WORD3	3	! резервируем 4 байта и устанавливаем знач. 3
J:	.WORD 4	4	! резервируем 4 байта и устанавливаем знач. 4
N:	.WORD 0	0	! резервируем 4 байта и устанавливаем знач. 0

Для компьютеров семейства Intel существует несколько ассемблеров, которые отличаются друг от друга по синтаксису. В этой книге мы будем использовать язык ассемблера Microsoft MASM. Мы будем говорить о процессоре Pentium II, но все, что мы будем обсуждать, применимо и к процессорам 386,486, Pentium и Pentium Pro. Для процессора SPARC мы будем использовать ассемблер Sun. Все это также применимо к более ранним 32-битным версиям. В книге коды операций и регистры всегда обозначаются прописными буквами, причем не только в ассемблере для Pentium II, как это обычно принято, но и в ассемблере Sun, где по соглашению используются строчные буквы.

Высказывания языка ассемблера состоят из четырех полей: поля метки, поля операции, поля операндов и поля комментариев. Метки используются для того, чтобы обеспечить символические имена для адресов памяти. Они нужны для того, чтобы можно было совершить переход к командам. Они также нужны для слов с данными, чтобы по символическому имени можно было получить доступ к тому месту, где они хранятся. Если высказывание снабжено меткой, то эта метка обычно располагается в колонке 1.

В каждом из трех примеров есть 4 метки: FORMULA, I, J и N. Отметим, что в языках ассемблера для SPARC после каждой метки нужно ставить двоеточие, а для Motorola — нет. В компьютерах Intel двоеточия ставятся только после меток команд, но не после меток данных. Данное различие вовсе не является фундаментальным. Разработчики разных ассемблеров имеют разные вкусы. Архитектура машины никак не определяет тот или иной выбор. Единственное преимущество двоеточия состоит в том, что метку можно писать на отдельной строке, а код операции — на следующей строке в колонке 1. Это упрощает работу компилятора: без двоеточия нельзя было бы отличить метку на отдельной строке от кода операции на отдельной строке.

В некоторых ассемблерах длина метки ограничена до 6 или 8 символов. А в большинстве языков высокого уровня длина имен произвольна. Длинные и хорошо подобранные имена упрощают чтение и понимание программы другими людьми.

В каждой машине содержится несколько регистров, но всем им даны совершенно разные названия. Регистры в Pentium II называются EAX, EBX, ECX и т. д. Регистры в Motorola называются D0, D1, D2. Регистры в машине SPARC имеют несколько названий. Здесь для их обозначения мы будем использовать %R1 и %R2.

В поле кода операции содержится либо символическая аббревиатура этого кода (если высказывание является символической репрезентацией машинной команды), либо команда для самого ассемблера. Выбор имени — дело вкуса, и поэтому разные разработчики языков ассемблера называют их по-разному. Разработчики ассемблера Intel решили использовать обозначение MOV и для загрузки регистра из памяти, и для сохранения регистра в память. Разработчики ассемблера Motorola выбрали обозначение MOVE для обеих операций. А разработчики ассемблера SPARC решили использовать LD для первой операции и ST для второй. Очевидно, что выбор названий в данном случае никак не связан с архитектурой машины.

Напротив, необходимость использовать две машинные команды для доступа к памяти объясняется устройством архитектуры SPARC, поскольку виртуальные адреса могут быть 32-битными (как в SPARC Version 8) и 44-битными (как в SPARC

Version 9), а команды могут содержать максимум 22 бита данных. Следовательно, чтобы передать все биты полного виртуального адреса, всегда требуется две команды. Команда

SETHI янкп.та

обнуляет старшие 32 бита и младшие 10 битов 64-битного регистра R1, а затем помещает старшие 22 бита 32-битного адреса переменной I в регистр R1 в битовые позиции с 10 по 31. Следующая команда

юсш+шхш.да

складывает R1 и младшие 10 битов адреса I (в результате чего получается полный адрес I), вызывает данное слово из памяти и помещает его в регистр R1.

Процессоры семейства Pentium, 680x0 и SPARC — все допускают операнды разной длины (типа byte (байт), word (слово) и long). Каким образом ассемблер определит, какую длину использовать? И опять разработчики ассемблера приняли разные решения. В Pentium II регистры разной длины имеют разные названия. Так, для перемещения 32-битных элементов используется название EAX, для 16-битных — AX, а для 8-битных — AL и AH. Разработчики ассемблера Motorola решили прибавлять к каждому коду операции суффикс .L для типа long, .W — для типа word и .B для типа byte. В SPARC для операндов разной длины используются разные коды операций (например, для загрузки байта, полуслова (halfword) и слова в 64-битный регистр используются коды операций LDSB, LDSH и LDSW соответственно). Как видите, разработка языка произвольна.

Три ассемблера, которые мы рассматриваем, различаются по способу резервирования пространства для данных. Разработчики языка ассемблера для Intel выбрали DW (Define Word — определить слово). Позднее был введен альтернативный вариант .WORD. В Motorola используется DC (Define Constant — определить константу). Разработчики SPARC с самого начала предпочли .WORD. И слова различия произвольны.

В поле операндов определяются адреса и регистры, которые являются операндами для машинной команды. В поле операндов команды целочисленного сложения сообщается, что и к чему нужно прибавить. Поле операндов команд перехода определяет, куда нужно совершить переход. Операндами могут быть регистры, константы, ячейки памяти и т. д.

В поле комментариев приводятся пояснения о действиях программы. Они могут понадобиться программистам, которые будут использовать и переделывать чужую программу, или программисту, который изначально писал программу и возвратился к работе над ней через год. Программа на ассемблере без таких комментариев совершенно непонятна программистам (даже автору этой программы). Комментарии нужны только человеку. Они никак не влияют на работу программы.

Директивы

Программа на языке ассемблера должна не только определять, какие машинные команды нужно выполнить, но и содержать команды, которые должен выполнять сам ассемблер (например, потребовать от него определить местонахождение какой-либо сохраненной информации или выдать новую страницу листинга). Команды для ассемблера называются **псевдокомандами** или **директивами ассем-**

блера. Мы уже видели одну типичную псевдокоманду DW (см. табл. 7.2). В табл. 7.5 приведены некоторые другие псевдокоманды (директивы). Они взяты из ассемблера MASM для семейства Intel.

Таблица 7.5. Некоторые директивы ассемблера MASM

Директива	Значение
SEGMENT	Начинает новый сегмент (текста, данных и т.п.) с определенными атрибутами
ENDS	Завершает текущий сегмент
ALIGN	Контролирует выравнивание следующей команды или данных
EQU	Определяет новый символ, равный данному выражению
DB	Выделяет память для одного или нескольких байтов
DD	Выделяет память для одного или нескольких 16-битных полуслов
DW	Выделяет память для одного или нескольких 32-битных слов
DQ	Выделяет память для одного или нескольких 64-битных двойных слов
PROC	Начинает процедуру
ENDP	Завершает процедуру
MACRO	Начинает макроопределение
ENDM	Завершает макроопределение
PUBLIC	Экспортирует имя, определенное в данном модуле
EXTERN	Импортирует имя из другого модуля
INCLUDE	Вызывает другой файл и включает его в текущий файл
IF	Начинает условную компоновку программы на основе данного выражения
ELSE	Начинает условную компоновку программы, если условие IF над директивой не выполнено
ENDIF	Завершает условную компоновку программы
COMMENT	Определяет новый отделитель комментариев
PAGE	Совершает принудительный обрыв страницы в листинге
END	Завершает программу ассемблирования

Директива SEGMENT начинает новый сегмент, а директива ENDS завершает его. Разрешается начинать текстовый сегмент, затем начинать сегмент данных, затем переходить обратно к текстовому сегменту и т. д.

Директива ALIGN переводит следующую строку (обычно данные) в адрес, который делим на аргумент данной директивы. Например, если текущий сегмент уже содержит 61 байт данных, тогда следующим адресом после **ALIGN 4** будет адрес 64.

Директива EQU дает символическое название некоторому выражению. Например, после записи

```
BASE EQU 1000
```

символ BASE можно использовать вместо 1000. Выражение, которое следует за EQU, может содержать несколько символов, соединенных арифметическими и другими операторами, например:

```
LIMIT EQU 4 * BASE + 2000
```

Большинство ассемблеров, в том числе MASM, требуют, чтобы символ был определен в программе до появления в некотором выражении.

Следующие 4 директивы DB, DB, Ш v^ ръетсределэдэт тосштсъ для одшлтам.нескольких переменных размером 1, 2,4 и 8 байтов соответственно. Например,

```
TABLE D8 11. 23. 49
```

выделяет пространство для 3 байтов и присваивает им начальные значения 11, 23 и 49 соответственно. Эта директива, кроме того, определяет символ TABLE, равный тому адресу, где хранится число 11.

Директивы PROC и ENDP определяют начало и конец процедур языка ассемблера. Процедуры в языке ассемблера выполняют ту же функцию, что и в языках программирования высокого уровня. Директивы MACRO и ENDM определяют начало и конец макроса. О макросах мы будем говорить ниже.

Далее идут директивы PUBLIC и EXTERN. Программы часто пишут в виде совокупности файлов. Часто процедуре, находящейся в одном файле, нужно вызвать процедуру или получить доступ к данным, определенным в другом файле. Чтобы такие отсылки между файлами стали возможными, обозначение (имя), которое нужно сделать доступным для других файлов, экспортируется с помощью директивы PUBLIC. Чтобы ассемблер не ругался по поводу использования символа, который не определен в данном файле, этот символ может быть объявлен внешним (EXTERN), это сообщит ассемблеру, что символ определен в каком-то другом файле. Символы, которые не определены ни в одной из этих директив, используются только в пределах одного файла. Поэтому даже если символ F00 используется в нескольких файлах, это не вызовет никакого конфликта, поскольку этот символ локален по отношению к каждому файлу.

Директива INCLUDE приказывает ассемблеру вызвать другой файл и включить его в текущий файл. Такие включенные файлы часто содержат определения, макросы и другие элементы, необходимые для разных файлов.

Многие языки ассемблера, в том числе MASM, поддерживают условную компоновку программы. Например, программа

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:  DW32
ELSE
WSIZE:  DW 16
ENDIF
```

выделяет в памяти одно 32-битное слово и вызывает его адрес WSIZE. Этому слову придается одно из значений: либо 32, либо 16 в зависимости от значения WORDSIZE (в данном случае 16). Такая конструкция может использоваться в программах для 16-битных машин (как 8088) или для 32-битных машин (как Pentium II). Если в начале и в конце машинозависимого кода поставить IF и ENDIF, а затем изменить одно определение, WORDSIZE, программу можно автоматически установить на один из двух размеров. Применяя такой подход, можно сохранять одну такую исходную программу для нескольких разных машин. В большинстве случаев все машинозависимые определения, такие как WORDSIZE, сохраняются в одном файле, причем для разных машин должны быть разные файлы. Путем включения файла с нужными определениями программу можно легко перекомпилировать на разные машины.

Директива COMMENT позволяет пользователю изменять символ комментария на что-либо отличное от точки с запятой. Директива PAGE используется для управления листингом программы. Наконец, директива END отмечает конец программы.

В ассемблере MASM есть еще много директив. Другие ассемблеры для Pentium II содержат другой набор директив, поскольку они определяются не в соответствии с архитектурой машины, а по желанию разработчиков ассемблера.

Макросы

Программистам на языке ассемблера часто приходится повторять одни и те же цепочки команд по несколько раз. Проще всего писать нужные команды всякий раз, когда они требуются. Но если последовательность достаточно длинная или если ее нужно повторять очень много раз, то это становится утомительным.

Альтернативный подход — оформить эту последовательность в процедуру и вызывать ее в случае необходимости. У такой стратегии тоже есть свои недостатки, поскольку в этом случае каждый раз придется выполнять специальную команду вызова процедуры и команду возврата. Если последовательности команд короткие (например, всего две команды), но используются часто, то вызов процедуры может сильно снизить скорость работы программы. Макросы являются простым и эффективным решением этой проблемы.

Макроопределение, макровызов и макрорасширение

Макроопределение — это способ дать имя куску текста. После того как макрос был определен, программист может вместо куска программы писать имя макроса. В сущности, макрос — это обозначение куска текста. В листинге 7.1 приведена программа на языке ассемблера для Pentium II, которая дважды меняет местами содержимое переменных p и q. Эти последовательности команд можно определить как макросы (листинг 7.2). После определения макроса каждое имя SWAP в программе замещается следующими четырьмя строками:

```
MOV EAX.P
MOV EBX.Q
MOV Q.EAX
MOV P.EBX
```

Программист определил SWAP как обозначение для этих четырех операторов.

Хотя разные языки ассемблера используют немного разные записи для определения макросов, все они состоят из одних и тех же базовых частей:

1. Заголовок макроса, в котором дается имя определяемого макроса.
2. Текст, в котором приводится тело макроса.
3. Директива, которая завершает определение (например, ENDM).

Когда ассемблер наталкивается на макроопределение в программе, он сохраняет его в таблице макроопределений для последующего использования. Всякий раз, когда в программе в качестве кода операции появляется макрос (в нашем примере SWAP), ассемблер замещает его телом макроса. Использование имени макроса в качестве кода операции называется макровызовом, а его замещение телом макроса называется макрорасширением.

Листинг 7.1. Код на языке ассемблера, в котором переменные p и q дважды меняются местами (без использования макроса)

```
MOV EAX.P
MOV EBX.Q
MOV Q.EAX
MOV P.EBX

MOV EAX.P
MOV EBX.Q
MOV Q.EAX
MOV P.EBX
```

Листинг 7.2. Тот же код с использованием макроса

```
SWAP    MACRO
        MOV EAX.P
        MOV EBX.Q
        MOV Q.EAX
        MOV P.EBX
        ENDM

        SWAP

        SWAP
```

Макрорасширение происходит во время процесса ассемблирования, а не во время выполнения программы. Этот момент очень важен. Программы, приведенные в листингах 7.1 и 7.2, произведут один и тот же машинный код. По программе на машинном языке невозможно определить, использовались ли макросы при ее порождении. После завершения макрорасширения ассемблер отбрасывает макрорасширения. В полученной программе никаких признаков макросов не остается.

Макровызовы не следует путать с вызовами процедур. Основное различие состоит в том, что макровызов — это команда ассемблеру заменить имя макроса телом макроса. Вызов процедуры — это машинная команда, которая вставлена в объектную программу и которая позднее будет выполнена, чтобы вызвать процедуру. В табл. 7.6 сравниваются макровызовы и вызовы процедур.

Таблица 7.6. Сравнение макровызовов и вызовов процедур

	Макровызов	**Вызов процедуры**
Когда совершается вызов программы?	Во время ассемблирования	Во время выполнения
Вставляется ли тело макроса или процедуры в объектную программу каждый раз, когда совершается вызов?	Да	Нет
Команда вызова процедуры вставляется в объектную программу, а затем выполняется?	Нет	Да
Нужно ли после вызова использовать команду возврата?	Нет	Да
Сколько копий тела макровызова или процедуры появляется в объектной программе?	Одна на макровызов	1

Можно считать, что процесс ассемблирования осуществляется в два прохода. На первом проходе сохраняются все макроопределения, а макровызовы расширяются. На втором проходе обрабатывается полученный в результате текст. Иными словами, исходная программа считывается, а затем трансформируется в другую программу, из которой удалены все макроопределения и в которой каждый макровызов замещен телом макроса. Полученная программа без макросов затем поступает в ассемблер.

Важно иметь в виду, что программа представляет собой цепочку символов. Это могут быть буквы, цифры, пробелы, знаки пунктуации и «возврат каретки» (переход на новую строку). Макрорасширение состоит в замене определенных подцепочек из этой цепочки другими цепочками. Макросредства — это способ манипулирования цепочками символов безотносительно их значений.

Макросы с параметрами

Макросредства, описанные ранее, можно использовать для сокращения программ, в которых часто повторяется точно одна и та же последовательность команд. Однако очень часто программа содержит несколько похожих, но не идентичных последовательностей команд (листинг 7.3). Здесь первая последовательность меняет местами P и Q, а вторая последовательность меняет местами R и S.

Листинг 7.3. Почти идентичные последовательности команд без использования макроса

```
MOV EAX.P
MOV EBX.Q
MOV Q.EAX
MOV P.EBX

MOV EAX.R
MOV EBX.S
MOV S.EAX
MOV R.EBX
```

Листинг 7.4. Те же последовательности с использованием макроса

```
CHANGE    MACRO P1.P2
              MOV  EAX.P1
              MOV  EBX.P2
              MOV  P2.EAX
              MOV  PI.EBX
              ENDM

              CHANGE  P.Q

              CHANGE  R.S
```

Для работы с такими почти идентичными последовательностями предусмотрены макроопределения, которые обеспечивают **формальные параметры,** и макровызовы, которые обеспечивают **фактические параметры.** Когда макрос расширяется, каждый формальный параметр, который появляется в теле макроса, замещается соответствующим фактическим параметром. Фактические параметры помещаются в поле операндов макровызова. В листинге 7.4. представлена программа из листинга 7.3, в которую включен макрос с двумя параметрами. Символы P1

и P2 — это формальные параметры. Во время расширения макроса каждый символ P1 внутри тела макроса замещается первым фактическим параметром, а символ P2 замещается вторым фактическим параметром. В макровызове

```
CHANGE P.Q
```

P — это первый фактический параметр, а Q — это второй фактический параметр. Таким образом, программы в листингах 7.3 и 7.4 идентичны.

Расширенные возможности

Большинство макропроцессоров содержат целый ряд расширенных особенностей, которые упрощают работу программиста на языке ассемблера. В этом разделе мы рассмотрим несколько расширенных особенностей MASM. Во всех ассемблерах есть одна проблема: дублирование меток. Предположим, что макрос содержит команду условного перехода и метку, к которой совершается переход. Если макрос вызывается два и более раз, метка будет дублироваться, что вызовет ошибку. Поэтому программист должен приписывать каждому вызову в качестве параметра отдельную метку. Другое решение (оно применяется в MASM) — объявлять метку локальной (LOCAL), при этом ассемблер автоматически будет порождать другую метку при каждом расширении макроса. В некоторых ассемблерах номерные метки автоматически считаются локальными.

MASM и большинство других ассемблеров позволяют определять макросы внутри других макросов. Эта особенность очень полезна в сочетании с условной компоновкой программы. Обычно один и тот же макрос определятся в обеих частях оператора IF:

```
MI  MACRO
IF WORDSIZE GT 16 M2     MACRO
...
ENDM
ELSE
M2  MACRO
...
ENDM
ENDIF
tNDM
```

В любом случае макрос M2 б> дет определен, но определение зависит от того, на какой машине ассемблируется программа: на 16-битной или на 32-битной. Если Ml не вызывается, макрос M2 вообще не будет определен.

Наконец, одни макросы могут вызывать другие макросы, в том числе самих себя. Если макрос рекурсивный, то есть вызывает самого себя, он должен передавать самому себе параметр, который изменяется при каждом расширении, а также проверять этот параметр и завершать рекурсию, когда параметр достигает определенного значения. В противном случае получится бесконечный цикл.

Реализация макросредств в ассемблере

Для реализации макросов ассемблер должен уметь выполнять две функции: сохранять макроопределения и расширять макровызовы. Мы рассмотрим эти функции по очереди.

Ассемблер должен сохранять таблицу всех имен макросов, в которой каждое имя сопровождается указателем на определение этого макроса, чтобы его можно было получить в случае необходимости. В одних ассемблерах предусмотрена отдельная таблица для имен макросов, а другие содержат общую таблицу, в которой находятся не только имена макросов, но и все машинные команды и директивы.

Когда встречается макроопределение, создается новый элемент таблицы с именем макроса, числом параметров и указателем на другую таблицу — таблицу макроопределений, где будет храниться тело макроса. Список формальных параметров тоже создается в это время. Затем считывается тело макроса и сохраняется в таблице макроопределений. Формальные параметры, которые встречаются в теле цикла, указываются специальным символом.

Ниже приведен пример внутреннего представления макроса CHANGE. В качестве символа возврата каретки используется точка с запятой, а в качестве символа формального параметра — амперсант.

```
MOV EAX.&P1;MOV EBX.&P2:M0V &P2EAX;M0V &P1.EBX:
```

В таблице макроопределений тело макроса представляет собой просто цепочку символов.

Во время первого прохода ассемблирования отыскиваются коды операций, а макросы расширяются. Всякий раз, когда встречается макроопределение, оно сохраняется в таблице макросов. При вызове макроса ассемблер временно приостанавливает чтение входных данных из входного устройства и начинает считывать сохраненное тело макроса. Формальные параметры, извлеченные из тела макроса, замещаются фактическими параметрами, которые предоставляются вызовом. Амперсант перед параметрами позволяет ассемблеру узнавать их.

Процесс ассемблирования

В следующих разделах мы опишем, как работает ассемблер. И хотя на каждой машине есть свой определенный ассемблер, отличный от других, процесс ассемблирования по сути один и тот же.

Двухпроходной ассемблер

Поскольку программа на языке ассемблера состоит из ряда операторов, на первый взгляд может показаться, что ассемблер сначала должен читать оператор, затем транслировать его на машинный язык и, наконец, переносить полученный машинный язык в файл, а соответствующий кусок листинга — в другой файл. Этот процесс будет повторяться до тех пор, пока вся программа не будет оттранслирована. Но, к сожалению, такая стратегия не работает.

Рассмотрим ситуацию, где первый оператор — переход к L. Ассемблер не может ассемблировать это оператор, пока не будет знать адрес L. L может находиться где-нибудь в конце программы, и тогда ассемблер не сможет найти этот адрес, не прочитав всю программу. Эта проблема называется проблемой опережающей

ссылки, поскольку символ L используется еще до того, как он определен (то есть было сделано обращение к символу, определение которого появится позднее).

Опережающие ссылки можно разрешать двумя способами. Во-первых, ассемблер действительно может прочитать программу дважды. Каждое прочтение исходной программы называется **проходом,** а транслятор, который читает исходную программу дважды, называется **двухпроходным транслятором.** На первом проходе собираются и сохраняются в таблице все определения символов, в том числе метки. К тому времени как начнется второй проход, значения символов уже известны, поэтому никакой опережающей ссылки не будет, и каждый оператор можно читать и ассемблировать. При этом требуется дополнительный проход по исходной программе, но зато такая стратегия относительно проста.

При втором подходе программа на языке ассемблера читается один раз и преобразуется в промежуточную форму, и эта промежуточная форма сохраняется в таблице в памяти. Затем совершает второй проход, но уже не по исходной программе, а по таблице. Если физической памяти (или виртуальной памяти) достаточно для этого подхода, то будет сэкономлено время, затрачиваемое на процесс ввода-вывода. Если требуется вывести листинг, тогда нужно сохранить полностью исходное выражение, включая комментарии. Если листинг не нужен, то промежуточную форму можно сократить, оставив только голые команды.

Еще одна задача первого прохода — сохранить все макроопределения и расширить вызовы по мере их появления. Следовательно, в одном проходе происходит и определение символов, и расширение макросов.

Первый проход

Главная функция первого прохода — построить **таблицу символов,** в которой содержатся значения всех имен. Символом может быть либо метка, либо значение, которому с помощью директивы приписывается определенное имя:

```
BUFSIZE EQU 8192
```

Приписывая значение символьному имени в поле метки команды, ассемблер должен знать, какой адрес будет иметь эта команда во время выполнения программы. Для этого ассемблер во время процесса ассемблирования сохраняет **счетчик адреса команд (ILC — Instruction Location Counter)** (специальную переменную). Эта переменная устанавливается на 0 в начале первого прохода и увеличивается после каждой обработанной команды на длину этой команды (табл. 7.7.). Пример написан для Pentium П. Мы не будем давать примеры для SPARC и Motorola, поскольку различия между языками ассемблера не очень важны и одного примера будет достаточно. Кроме того, язык ассемблера для SPARC неудобочитаем.

При первом проходе в большинстве ассемблеров используется по крайней мере 3 таблицы: таблица символьных имен, таблица директив и таблица кодов операций. В случае необходимости используется еще литеральная таблица. Таблица символьных имен содержит один элемент для каждого имени, как показано в табл. 7.8. Символьные имена либо используются в качестве меток, либо явным образом определяются (например, с помощью EQU). В каждом элементе таблицы символьных

имен содержится само имя (или указатель на него), его численное значение и иногда некоторая дополнительная информация. Она может включать:

1. Длину поля данных, связанного с символом.
2. Биты перераспределения памяти (которые показывают, изменяется ли значение символа, если программа загружается не в том адресе, в котором предполагал ассемблер).
3. Сведения о том, можно ли получить доступ к символу извне процедуры.

Таблица 7.7. Счетчик адреса команд используется для слежения за адресами команд. В данном примере операторы до MARIA занимают 100 байтов

Метка	Код операции	Операнды	Комментарии	Длина	Счетчик адреса команд
MARIA:	MOV	EAX, I	EAX=I	5	100
	MOV	EBX, J	EBX=J	6	105
ROBERTA:	MOV	ECX, K	ECX=K	6	111
	IMUL	EAX, EAX	EAX=I*I	2	117
	IMUL	EBX, EBX	EBX=J*J	3	119
	IMUL	ECX, ECX	ECX=K*K	3	122
MARILYN:	ADD	EAX, EBX	EAX=I*I+J*J	2	125
	ADD	EAX, ECX	EAX=I*I+J*J+K*K	2	127
STEPHANY:	JMP	DONE	Переход к DONE	5	129

Таблица 7.8. Таблица символьных имен для программы из табл. 7.7.

Символьное имя	Значение	Прочая информация
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

В таблице кодов операций предусмотрен по крайней мере один элемент для каждого символического кода операции в языке ассемблера (табл. 7.9). В каждом элементе таблицы содержится символический код операции, два операнда, числовое значение кода операции, длина команды и номер типа, по которому можно определить, к какой группе относится код операции (коды операций делятся на группы в зависимости от числа и типа операндов).

Таблица 7.9. Некоторые элементы таблицы кодов операций для ассемблера Pentium II

Код операции	Первый операнд	Второй операнд	Шестнадцатеричный код	Длина команды	Класс команды
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

В качестве примера рассмотрим код операции ADD. Если команда ADD в качестве первого операнда содержит регистр EAX, а в качестве второго — 32-битную константу (immed32), то используется код операции 0x05, а длина команды составляет 5 байтов. Если используется команда ADD с двумя регистрами в качестве операндов, то длина команды составляет 2 байта, а код операции будет равен 0x01. Все комбинации кодов операций и операндов, которые соответствуют данному правилу, будут отнесены к классу 19 и будут обрабатываться так же, как команда ADD с двумя регистрами в качестве операндов. Класс команд обозначает процедуру, которая вызывается для обработки всех команд данного типа.

В некоторых ассемблерах можно писать команды с применением непосредственной адресации, даже если соответствующей команды не существует в выходном языке. Такие команды с «псевдонепосредственными» адресами обрабатываются следующим образом. Ассемблер назначает участок памяти для непосредственного операнда в конце программы и порождает команду, которая обращается к нему. Например, универсальная вычислительная машина IBM 3090 не имеет команд с непосредственными адресами. Тем не менее программист может написать команду

```
L 14.=F'5'
```

для загрузки в регистр 14 константы 5 размером в полное слово. Таким образом, программисту не нужно писать директиву, чтобы разместить слово в памяти, придать ему значение 5, дать ему метку, а затем использовать эту метку в команде L. Константы, для которых ассемблер автоматически резервирует память, называются **литералами.** Литералы упрощают читаемость и понимание программы, делая значение константы очевидным в исходном операторе. При первом проходе ассемблер должен создать таблицу из всех литералов, которые используются в программе. Все три компьютера, которые мы взяли в качестве примеров, содержат команды с непосредственными адресами, поэтому их ассемблеры не обеспечивают литералы. Команды с непосредственными адресами в настоящее время считаются обычными, но раньше они рассматривались как нечто совершенно необычное. Вероятно, широкое распространение литералов внушило разработчикам, что непосредственная адресация — это очень хорошая идея. Если нужны литералы, то во время ассемблирования сохраняется таблица литералов, в которой появляется новый элемент всякий раз, когда встречается литерал. После первого прохода таблица сортируется и продублированные элементы удаляются.

В листинге 7.5 показана процедура, которая лежит в основе первого прохода ассемблера. Названия процедур были выбраны таким образом, чтобы была ясна их суть. Листинг 7.5 представляет собой хорошую отправную точку для изучения. Он достаточно короткий, он легок для понимания, и из него видно, каким должен быть следующий шаг — это написание процедур, которые используются в данном листинге.

Листинг 7.5. Первый проход простого ассемблера

```
public static void pass_one() {
// Эта процедура - первый проход ассемблера
   boolean more_input=true;                  //флаг, который останавливает первый проход
   String line, symbol, literal, opcode;     //поля команды
   int location_counter, length, value, type; //переменные
   final int END_STATEMENT = -2;             //сигналы конца ввода
```

```
    location_counter = 0;     //ассемблирование первой команды в ячейке 0
    imtialize_tables(),       //общая инициализация

while (more_input) {                //more_input получает значение «ложь» с помощью END
       line = read_next_line();      //считывание строки
          length =0;                 //# байт в команде
          type =0.                   //тип команды

       if (line_isjiot_coniment(line)) {
          symbol = check_for_symbol(line),    //Содержит ли строка метку?
          if (symbol !- null)                //если да, то записывается символ и значение
             enter_new_symbol(symbol. 1ocation_counter),
          literal = check_for_literal(line).  //Содержит ли строка литерал?
          if (literal != null)               //если да, то он вводится в таблицу
             enter_new_literal(1itera1);
                                              //Теперь определяем тип кода операции.
                                              //-1 значит недопустимый код операции.
          opcode = extract_opcode(line).      //определяем место кода операции
          type =search_opcode_table(opcode).  //находим формат, например. OP REG1.REG2
          if (type < 0)                      //Если это не код операции, является
                                              //ли это директивой?
             type = search_pseudo_table(opcode).
          switch(type) {                      //определяем длину команды
             case l.length=get_length_of_typel (line), break,
             case 2 Iength=get_length_of_type2(line); break.
                                              //другие случаи
       }
    }
    wnte_temp_file(type, opcode, length, line),  //информация для второго прохода
    location_counter = location_counter + length, //обновление счетчика адреса команд
    if (type == END_STATEMENT) {                  //завершился ли ввод?
       morejinput - false.                       //если да. то выполняем служебные действия-
       rewind_temp_for_pass_two().                //перематываем файл обратно
       sort_literal_table().                      //сортируем таблицу литералов
       remove_redundant_literals();               //и удаляем из нее дубликаты
       }
    }
}
```

Одни процедуры будут относительно короткими, например *check_jor_symbol,* которая просто выдает соответствующее обозначение в виде цепочки символов, если таковое имеется, и выдает ноль, если его нет. Другие процедуры, например *get_length_of_type1* и *get_length_ofjtype2,* могут быть достаточно длинными и могут сами вызывать другие процедуры. Естественно, на практике типов будет не два, а больше, и это будет зависеть от языка, который ассемблируется, и от того, сколько типов команд предусмотрено в этом языке.

Структурирование программ имеет и другие преимущества помимо простоты программирования. Если ассемблер пишется группой людей, разнообразные процедуры могут быть разбиты на куски между программистами. Все подробности получения входных данных спрятаны в процедуре *read_next_line.* Если эти детали нужно изменить (например, из-за изменений в операционной системе), то это повлияет только на одну подчиненную процедуру, и никаких изменений в самой процедуре *passjone* делать не нужно.

По мере чтения программы во время первого прохода ассемблер должен анализировать каждую строку, чтобы найти код операции (например, ADD), определить

ее тип (набор операндов) и вычислить длину команды. Эта информация понадобится при втором проходе, поэтому ее лучше записать, чтобы не анализировать строку во второй раз. Однако переписывание входного файла потребует больше операций ввода-вывода. Что лучше — увеличить количество операций ввода-вывода, чтобы меньше времени тратить на анализ строк, или сократить количество операций ввода-вывода и потратить больше времени на анализ, зависит от скорости работы центрального процессора и диска, эффективности файловой системы и некоторых других факторов. В нашем примере мы запишем временный файл, который будет содержать тип, код операции, длину и саму входную цепочку. Именно это цепочка и будет считываться при втором проходе, и читать файл по второму разу будет не нужно.

После прочтения директивы END первый проход завершается. В этот момент можно сохранить таблицу символьных имен и таблицу литералов, если это необходимо. В таблице литералов можно произвести сортировку и удалить продублированные литералы.

Второй проход

Задача второго прохода — произвести объектную программу и напечатать протокол ассемблирования (если нужно). Кроме того, при втором проходе должна выводиться информация, необходимая компоновщику для связывания процедур, которые ассемблировались в разное время, в один выполняемый файл. В листинге 7.6 показана процедура для второго прохода.

Листинг 7.6. Второй проход простого ассемблера

```
public static void pass_two() {
                                    //Эта процедура - второй проход ассемблера
   boolean morejnput = true:        //флаг, который останавливает второй проход
   String line, opcode;             //поля команды
   int location_counter, length, type: //переменные
    final int END_STATEMENT = -2:   //сигналы конца ввода
    final int MAX_CODE =16;         //максимальное количество байтов в команде
    byte code[] = new byte[MAX_CODE]; //количество байтов в команде в порожденном коде

   location_counter = 0;            //ассемблирование первой команды в адресе 0

   while (morejnput) {              //morejnput устанавливается на «ложь» с помощью END
         type = readj:ype():        //считывание поля типа следующей строки
opcode = read_opcode();             //считывание поля кода операции следующей строки
length = readJength0;               //считывание поля длины в следующей строке
line = readJine0;                   //считывание самой входной строки
if (type != 0) {                    //тип 0 указывает на строки комментария
   switch(type) {                   //порождение выходного кода

      case 1:evalj:ypel(opcode, length, line, code): break;
             case 2: eval_type2(opcode, length, line, code); break;
                                    //Другие случаи
         }
      }
   write_output(code):              // запись двоичного кода
   writejisting(code. line);        // вывод на печать одной строки
   location_counter = location_counter + length; //обновление счетчика адреса команд
   if (type == END_STATEMENT) {     // завершен ли ввод?
```

```
        more_input = false;              // если да, то выполняем служебные операции
        finishjjpO;                      // завершение
    }
  }
}
```

Процедура второго прохода более или менее сходна с процедурой первого прохода: строки считываются по одной и обрабатываются тоже по одной. Поскольку мы записали в начале каждой строки тип, код операции и длину (во временном файле), все они считываются, и таким образом, нам не нужно проводить анализ строк во второй раз. Основная работа по порождению кода выполняется процедурами *eval_type1, eval_type2* и т. д. Каждая из них обрабатывает определенную модель (например, код операции и два регистра-операнда). Полученный в результате двоичный код команды сохраняется в переменной *code*. Затем совершается контрольное считывание. Желательно, чтобы процедура *write_code* просто сохраняла в буфере накопленный двоичный код и записывала файл на диск большими порциями, чтобы сократить рабочую нагрузку на диск.

Исходный оператор и выходной (объектный) код, полученный из него (в шестнадцатеричной системе), можно напечатать или поместить в буфер, чтобы напечатать потом. После переустановки счетчика адреса команды вызывается следующий оператор.

До настоящего момента предполагалось, что исходная программа не содержит никаких ошибок. Но любой человек, который когда-нибудь писал программы на каком-либо языке, знает, насколько это предположение не соответствует действительности. Наиболее распространенные ошибки приведены ниже:

1. Используемый символ не определен.
2. Символ был определен более одного раза.
3. Имя в поле кода операции не является допустимым кодом операции.
4. Код операции не снабжен достаточным количеством операндов.
5. У кода операции слишком много операндов.
6. Восьмеричное число содержит 8 или 9.
7. Недопустимое применение регистра (например, переход к регистру).
8. Отсутствует оператор END.

Программисты весьма изобретательны по части новых ошибок. Ошибки с неопределенным символом часто возникают из-за опечаток. Хороший ассемблер может вычислить, какой из всех определенных символов в большей степени соответствует неопределенному, и подставить его. Для исправления других ошибок ничего кардинального предложить нельзя. Лучшее, что может сделать ассемблер при обнаружении оператора с ошибкой, — это вывести сообщение об ошибке на экран и попробовать продолжить процесс ассемблирования.

Таблица символов

Во время первого прохода ассемблер аккумулирует всю информацию о символах и их значениях. Эту информацию он должен сохранить в таблице символьных имен, к которой будет обращаться при втором проходе. Таблицу символьных имен можно организовать несколькими способами. Некоторые из них мы опишем ниже.

При применении любого из этих способов мы пытаемся смоделировать **ассоциативную память,** которая представляет собой набор пар (символьное имя, значение). По имени ассоциативная память должна выдавать его значение.

Проще всего реализовать таблицу символьных имен в виде массива пар, где первый элемент является именем (или указателем на имя), а второй — значением (или указателем на него). Если нужно найти какой-нибудь символ, то таблица символьных имен просто последовательно просматривается, пока не будет найдено соответствие. Такой метод довольно легко запрограммировать, но он медленно работает, поскольку в среднем при каждом поиске придется просматривать половину таблицы.

Другой способ организации — отсортировать таблицу по именам и для поиска имен использовать алгоритм **двоичного поиска.** В соответствии с этим алгоритмом средний элемент таблицы сравнивается с символьным именем. Если нужное имя по алфавиту идет раньше среднего элемента, значит, оно находится в первой половине таблицы. Если символьное имя по алфавиту идет после среднего элемента, значит, оно находится во второй части таблицы. Если нужное имя совпадает со средним элементом, то поиск на этом завершается.

Предположим, что средний элемент таблицы не равен символу, который мы ищем. Мы уже знаем, в какой половине таблицы он находится. Алгоритм двоичного поиска можно применить к соответствующей половине. В результате мы либо получим совпадение, либо определим нужную четверть таблицы. Таким образом, в таблице из n элементов нужный символ можно найти примерно за lo&n попыток. Очевидно, что такой алгоритм работает быстрее, чем просто последовательный просмотр таблицы, но при этом элементы таблицы нужно сохранять в алфавитном порядке.

Совершенно другой подход — **хэш-кодирование.** Для этого подхода требуется хэш-функция, которая отображает символы (имена) в целые числа в промежутке от 0 до k-1. Такой функцией может быть функция перемножения кодов ASCII всех символов в имени. Можно перемножить все коды ASCII символов с игнорированием переполнения, а затем взять значение по модулю **k** или разделить полученное значение на простое число. Фактически подойдет любая входная функция, которая дает равномерное распределение значений.

Символьные имена можно хранить в таблице, состоящей из **k** участков, от 0 до k-1. Все пары (символьное имя, значение), в которых имя соответствует i, сохраняются в связном списке, на который указывает слот i в хэш-таблице. Если в хэш-таблице содержится n символьных имен и k слотов, то в среднем длина списка будет n/k. Если мы выберем k, приблизительно равное n, то на нахождение нужного символьного имени в среднем потребуется всего один поиск. Путем корректировки **k** мы можем сократить размер таблицы, но при этом скорость поиска снизится. Хэш-код показан на рис. 7.1.

Связывание и загрузка

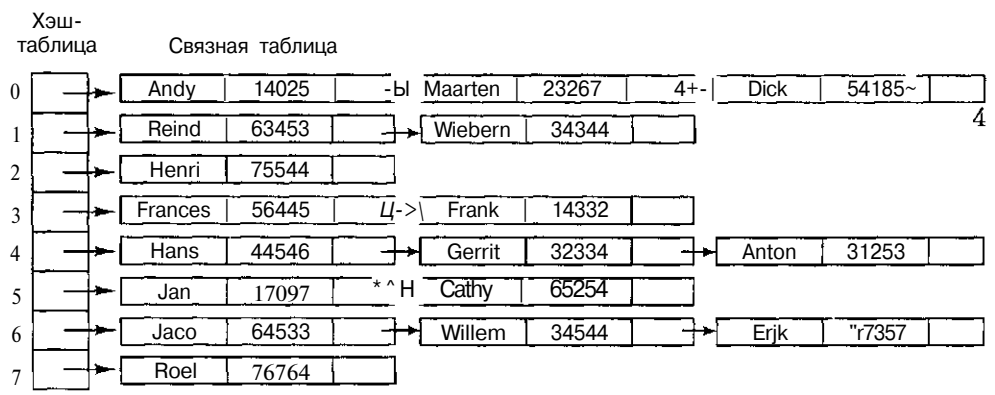
Большинство программ содержат более одной процедуры. Компиляторы и ассемблеры транслируют одну процедуру и помещают полученный на выходе результат на диск. Перед запуском программы должны быть найдены и связаны все оттран-

слированные процедуры. Если виртуальной памяти нет, связанная программа должна загружаться в основную память. Программы, которые выполняют эти функции, называются по-разному: **компоновщиками, связывающими загрузчиками** и **редакторами связей.** Для полной трансляции исходной программы требуется два шага, как показано на рис. 7.2:

1. Компиляция или ассемблирование исходных процедур.
2. Связывание объектных модулей.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebern	34344	1

а



б

Рис. 7.1. Хэш-кодирование: символьные имена, значения и хэш-коды, образованные от символьных имен (а); хэш-таблица из 8 элементов со связным списком символьных имен и значений *(б)*

Первый шаг выполняется ассемблером или компилятором, а второй — компоновщиком.

Трансляция исходной процедуры в объектном модуле — это переход на другой уровень, поскольку исходный язык и выходной язык имеют разные команды и запись. Однако при связывании перехода на другой уровень не происходит, поскольку программы на входе и на выходе компоновщика предназначены для одной и той же виртуальной машины. Задача компоновщика — собрать все процедуры, которые транслировались раздельно, и связать их вместе, чтобы в результате получился **исполняемый двоичный код.** В системах MS-DOS, Windows 95/98 и NT объектные модули имеют расширение .obj, а исполняемые двоичные программы — расширение .exe. В системе UNIX объектные модули имеют расширение .o, а исполняемые двоичные программы не имеют расширения.

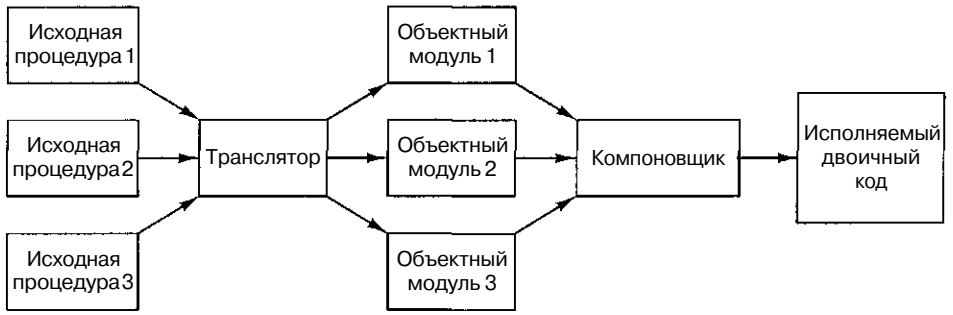


Рис. 7.2. Для получения исполняемой двоичной программы из совокупности оттранслированных независимо друг от друга процедур используется компоновщик

Компиляторы и ассемблеры транслируют каждую исходную процедуру как отдельную единицу. На это есть веская причина. Если компилятор или ассемблер считывал бы целый ряд исходных процедур и сразу переводил бы их в готовую программу на машинном языке, то при изменении одного оператора в исходной процедуре потребовалось бы заново транслировать все исходные процедуры.

Если каждая процедура транслируется по отдельности, как показано на рис. 7.2, то транслировать заново нужно будет только одну измененную процедуру, хотя понадобится заново связать все объектные модули. Однако связывание происходит гораздо быстрее, чем трансляция, поэтому выполнение этих двух шагов (трансляции и связывания) сэкономит время при доработке программы. Это особенно важно для программ, которые содержат сотни или тысячи модулей.

Задачи компоновщика

В начале первого прохода ассемблирования счетчик адреса команды устанавливается на 0. Этот шаг эквивалентен предположению, что объектный модуль во время выполнения будет находиться в ячейке с адресом 0. На рис. 7.3 показаны 4 объектных модуля для типичной машины. В этом примере каждый модуль начинается с команды перехода BRANCH к команде MOVE в том же модуле.

Чтобы запустить программу, компоновщик помещает объектные модули в основную память, формируя отображение исполняемого двоичного кода (рис. 7.4, *а*). Цель — создать точное отображение виртуального адресного пространства ис-

полняемой программы внутри компоновщика и разместить все объектные модули в соответствующих адресах. Если физической или виртуальной памяти не достаточно для формирования отображения, то можно использовать файл на диске. Обычно небольшой раздел памяти, начинающийся с нулевого адреса, используется для векторов прерывания, взаимодействия с операционной системой, обнаружения неинициализированных указателей и других целей, поэтому программы обычно начинаются не с нулевого адреса, а выше. В нашем примере программы начинаются с адреса 100.

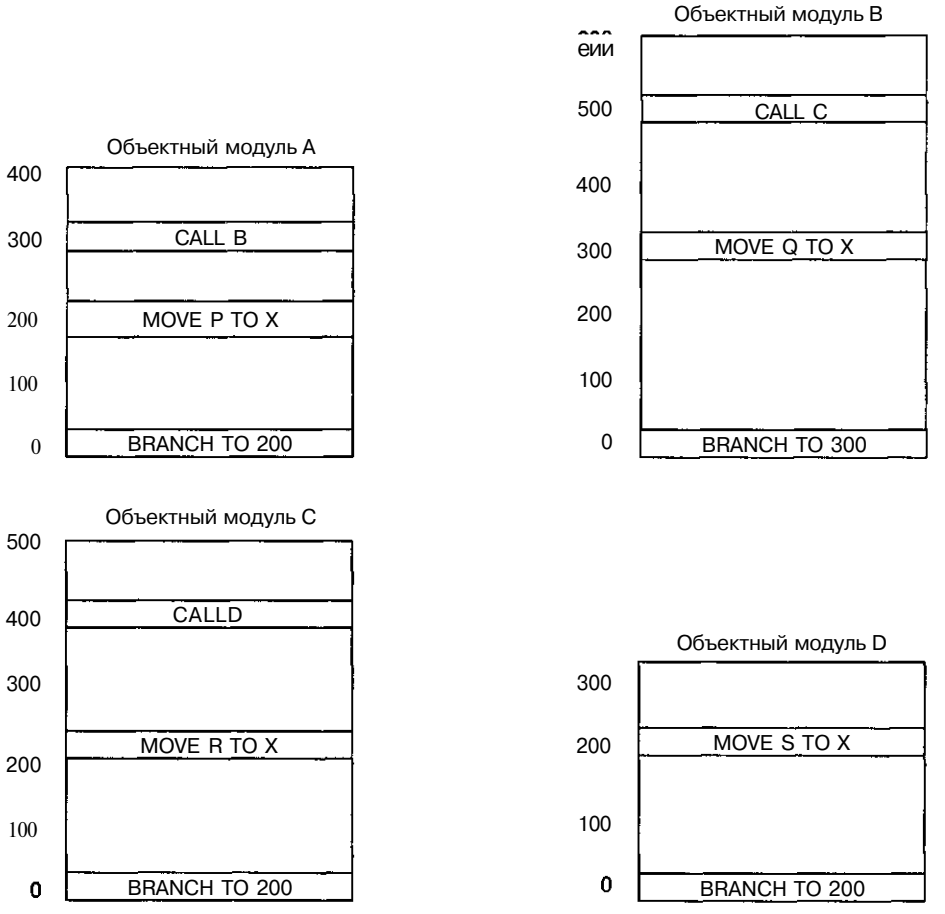


Рис. 7.3. Каждый модуль имеет свое собственное адресное пространство, начинающееся с нуля

Посмотрите на рис. 7.4, *а*. Хотя программа уже загружена в отображение исполняемого двоичного файла, она еще не готова для выполнения. Посмотрим, что произойдет, если выполнение программы начнется с команды в начале модуля A. Программа не совершит перехода к команде MOVE, поскольку эта команда находится в ячейке с адресом 300. Фактически все команды обращения к памяти не будут выполнены по той же причине.

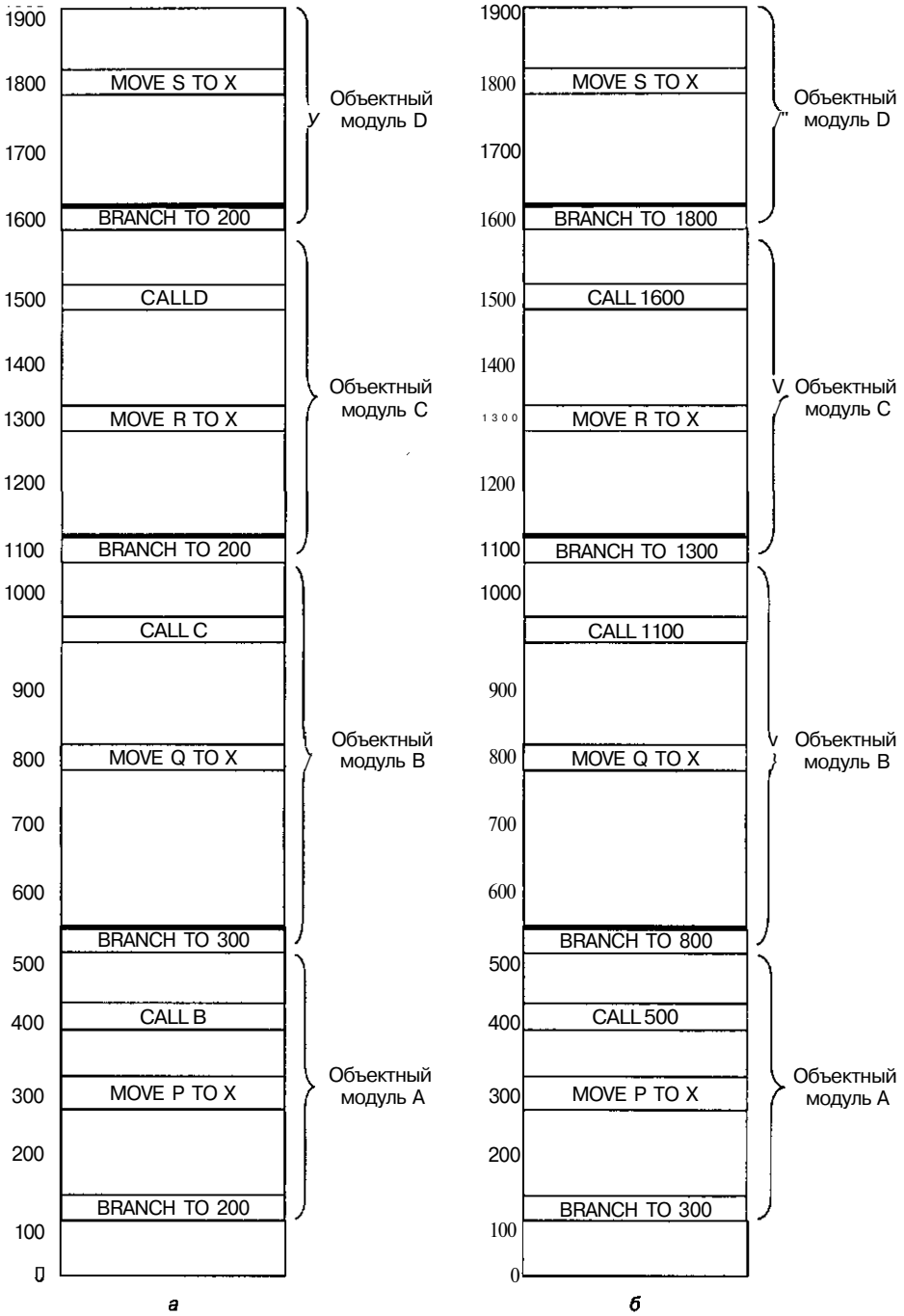


Рис. 7.4. Объектные модули после размещения в двоичном отображении, но до перераспределения памяти и связывания (а); те же объектные модули после связывания и перераспределения памяти (б). В результате получается исполняемая двоичная программа, которую можно запускать

Здесь возникает **проблема перераспределения памяти,** поскольку каждый объектный модуль на рис. 7.3 занимает отдельное адресное пространство. В машине с сегментированным адресным пространством (например, в Pentium II) каждый объектный модуль теоретически может иметь свое собственное адресное пространство, если его поместить в отдельный сегмент. Однако для Pentium II только система OS/2 поддерживает такую структуру[1]. Все версии Windows и UNIX поддерживают только одно линейное адресное пространство, поэтому все объектные модули должны быть слиты вместе в одно адресное пространство.

Более того, команды вызова процедур (см. рис. 7.4, *а)* вообще не будут работать. В ячейке с адресом 400 программист намеревается вызвать объектный модуль B, но поскольку каждая процедура транслируется отдельно, ассемблер не может определить, какой адрес вставлять в команду CALL B. Адрес объектного модуля B не известен до времени связывания. Такая проблема называется проблемой **внешней ссылки.** Обе проблемы решаются с помощью компоновщика.

Компоновщик сливает отдельные адресные пространства объектных модулей в единое линейное адресное пространство. Для этого совершаются следующие шаги:

1. Компоновщик строит таблицу объектных модулей и их длин.
2. На основе этой таблицы он приписывает начальные адреса каждому объектному модулю.
3. Компоновщик находит все команды, которые обращаются к памяти, и прибавляет к каждой из них **константу перемещения,** которая равна начальному адресу этого модуля.
4. Компоновщик находит все команды, которые обращаются к процедурам, и вставляет в них адрес этих процедур.

Ниже показана таблица объектных модулей рис. 7.4, построенная на первом шаге. В ней дается имя, длина и начальный адрес каждого модуля.

Модуль	Длина	Начальный адрес
A	400	100
B	600	500
C	500	1100
D	300	1600

На рисунке 7.4, б показано, как адресное пространство выглядит после выполнения компоновщиком всех шагов.

Структура объектного модуля

Объектные модули обычно состоят из шести частей (рис. 7.5). В первой части содержится имя модуля, некоторая информация, необходимая компоновщику (например, длины различных частей модуля), а иногда дата ассемблирования.

[1] Необходимо отметить, что сегментный способ организации был использован только в первой версии OS/2, которая была 16-битовой и разрабатывалась для 286-го микропроцессора. Поэтому относить эту систему к Pentium II представляется не вполне правильно. Начиная с 1993 года все последующие версии OS/2 были 32-битовыми и, как и остальные современные операционные системы, перестали поддерживать сегментирование, а стали использовать только страничный механизм. — *Примеч. научн.ред.*

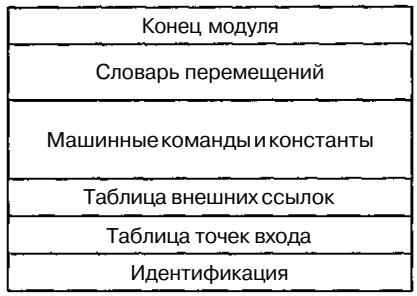


Рис. 7.5. Внутренняя структура объектного модуля

Вторая часть объектного модуля — это список символов, определенных в модуле, вместе с их значениями. К этим символам могут обращаться другие модули. Например, если модуль состоит из процедуры *bigbug,* то элемент таблицы будет содержать цепочку символов «bigbug», за которой будет следовать соответствующий адрес. Программист на языке ассемблера с помощью директивы PUBLIC указывает, какие символьные имена считаются **точками входа.**

Третья часть объектного модуля состоит из списка символьных имен, которые используются в этом модуле, но определены в других модулях. Здесь также имеется список, который показывает, какие именно символьные имена используются теми или иными машинными командами. Второй список нужен для того, чтобы компоновщик мог вставить правильные адреса в команды, которые используют внешние имена. Процедура может вызывать другие независимо транслируемые процедуры, объявив имена вызываемых процедур внешними. Программист на языке ассемблера с помощью директивы EXTERN указывает, какие символы нужно объявить **внешними.** В некоторых компьютерах точки входа и внешние ссылки объединены в одной таблице.

Третья часть объектного модуля — это машинные команды и константы. Это единственная часть объектного модуля, которая будет загружаться в память для выполнения. Остальные 5 частей используются компоновщиком, а затем отбрасываются еще до начала выполнения программы.

Пятая часть объектного модуля — это словарь перемещений. К командам, которые содержат адреса памяти, должна прибавляться константа перемещения (см. рис. 7.4). Компоновщик сам не может определить, какие слова в четвертой части содержат машинные команды, а какие — константы. Поэтому в этой таблице содержится информация о том, какие адреса нужно переместить. Это может быть битовая таблица, где на каждый бит приходится потенциально перемещаемый адрес, либо явный список адресов, которые нужно переместить.

Шестая часть содержит указание на конец модуля, а иногда — контрольную сумму для определения ошибок, сделанных во время чтения модуля, и адрес, с которого нужно начинать выполнение.

Большинству компоновщиков требуется два прохода. На первом проходе компоновщик считывает все объектные модули и строит таблицу имен и длин модулей и глобальную таблицу символов, которая состоит из всех точек входа и внешних ссылок. На втором проходе модули считываются, перемещаются в памяти и связываются.

Время принятия решения и динамическое перераспределение памяти

В мультипрограммной системе программу можно считать в основную память, запустить ее на некоторое время, записать на диск, а затем снова считать в основную память для выполнения. В большой системе с большим количеством программ трудно быть уверенным, что программа считывается каждый раз в одно и то же место в памяти.

На рис. 7.6 показано, что произойдет, если уже перемещенная программа (см. рис. 7.4, *б)* будет загружена в адрес 400, а не в адрес 100, куда ее изначально поместил компоновщик. Все адреса памяти будут неправильными. Более того, информация о перемещении уже давно удалена. Даже если эта информация была бы доступна, перемещать все адреса при каждой перекачке программы было бы неудобно.

Проблема перемещения программ, уже связанных и размещенных в памяти, близко связана с моментом времени, в который совершается финальное связывание символических имен с абсолютными адресами физической памяти. В программе содержатся символические имена для адресов памяти (например, BR L). Время, в которое определяется адрес в основной памяти, соответствующий L, называется **временем принятия решения.** Существует по крайней мере шесть вариантов для времени принятия решения относительно привязок:

1. Когда пишется программа.
2. Когда программа транслируется.
3. Когда программа компонуется, но еще до загрузки.
4. Когда программа загружается.
5. Когда загружается базовый регистр, который используется для адресации.
6. Когда выполняется команда, содержащая адрес.

Если команда, содержащая адрес памяти, перемещается после связывания, этот адрес будет неправильным (предполагается, что объект, на который происходит ссылка, тоже перемещен). Если транслятор производит исполняемый двоичный код, то связывание происходит во время трансляции и программа должна быть запущена в адресе, в котором этого ожидает транслятор. При применении метода, описанного в предыдущем разделе, во время связывания символические имена соотносятся с абсолютными адресами, и именно по этой причине перемещать программы после связывания нельзя (см. рис. 7.6).

Здесь возникают два вопроса. Первый — когда символические имена связываются с виртуальными адресами, а второй — когда виртуальные адреса связываются с физическими адресами? Только после двух этих операций процесс связывания можно считать завершенным. Когда компоновщик связывает отдельные адресные пространства объектных модулей в единое линейное адресное пространство, он фактически создает виртуальное адресное пространство. Перемещение в памяти и связывание нужно для связи символических имен с определенными виртуальными адресами. Это наблюдение верно независимо от того, используется виртуальная память или нет.

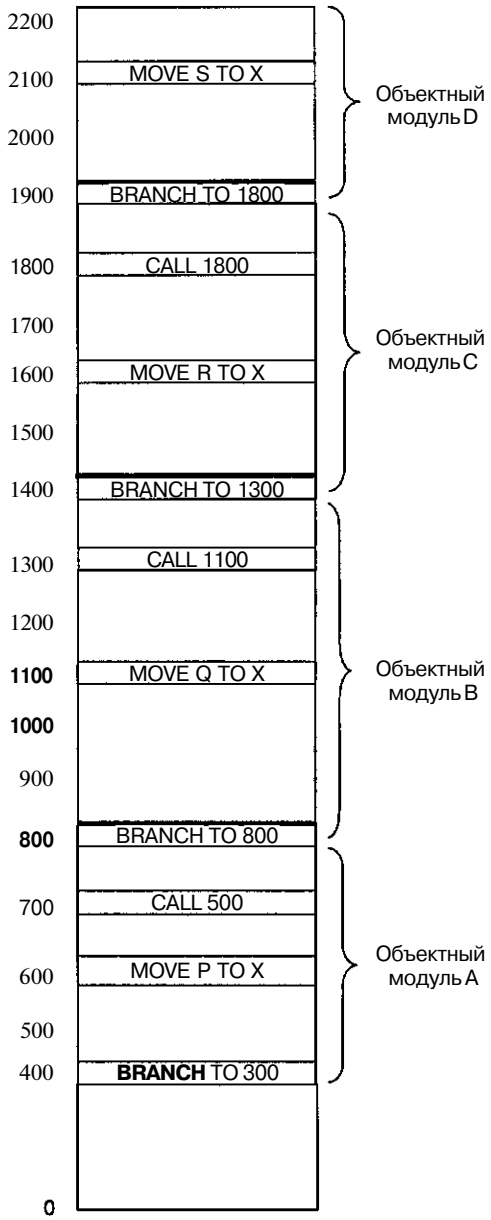


Рис. 7.6. Двоичная программа с рис. 7.4, б, передвинутая вверх на 300 адресов. Многие команды теперь обращаются к неправильным адресам памяти

Предположим, что адресное пространство, изображенное на рис. 7.4, *б,* было разбито на страницы. Ясно, что виртуальные адреса, соответствующие символическим именам A, B, C и D, уже определены, хотя их физические адреса будут зависеть от содержания таблицы страниц. Исполняемая двоичная программа представляет собой связывание символических имен с виртуальными адресами.

Любой механизм, который позволяет легко изменять отображение виртуальных адресов на адреса основной физической памяти, будет облегчать перемещение программы в основной памяти, даже если они уже связаны с виртуальным адресным пространством. Одним из таких механизмов является разбиение на страницы. Если программа перемещается в основной памяти, нужно изменить только ее таблицу страниц, но не саму программу.

Второй механизм — использование регистра перемещения. Компьютер CDC 6600 и его последователи содержали такой регистр. В машинах, в которых используется эта технология перемещения, регистр всегда указывает на физический адрес начала текущей программы. Аппаратное обеспечение прибавляет регистр перемещения ко всем адресам памяти, прежде чем отправить их в память. Весь процесс перемещения является «прозрачным» для каждой пользовательской программы. Пользовательские программы даже не подозревают, что этот процесс происходит. Если программа перемещается, операционная система должна обновить регистр перемещения. Такой механизм менее обычен, чем разбиение на страницы, поскольку перемещаться должна вся программа целиком (однако если есть отдельные регистры для перемещения кода и перемещения данных, как, например, в процессоре Intel 8088, то в этом случае программу нужно перемещать как два компонента).

Третий механизм можно использовать в машинах, которые могут обращаться к памяти относительно счетчика команд. Всякий раз, когда программа перемещается в основной памяти, нужно обновлять только счетчик команд. Программа, все обращения к памяти которой либо связаны со счетчиком команд, либо абсолютны (например, обращения к регистрам устройств ввода-вывода в абсолютных адресах), называется **позиционно-независимой программой.** Позиционно-независимую процедуру можно поместить в любом месте виртуального адресного пространства без настройки адресов.

Динамическое связывание

Стратегия связывания, которую мы обсуждали в разделе «Задачи компоновщика», имеет одну особенность: связь со всеми процедурами, нужными программе, устанавливается до начала работы программы. Однако если мы будем устанавливать все связи до начала работы программы в компьютере с виртуальной памятью, то мы не используем всех возможностей виртуальной памяти. Многие программы содержат процедуры, которые вызываются только при определенных обстоятельствах. Например, компиляторы содержат процедуры для компиляции редко используемых операторов, а также процедуры для исправления ошибок, которые встречаются редко.

Более гибкий способ связывания отдельно скомпилированных процедур — установление связи с каждой процедурой в тот момент, когда она впервые вызывается. Этот процесс называется **динамическим связыванием.** Впервые он был применен в системе MULTICS. В следующих разделах мы рассмотрим применение динамического связывания в нескольких системах.

Динамическое связывание в системе MULTICS

В системе MULTICS с каждой программой соотносится сегмент, так называемый **сегмент связи,** содержащий один блок информации для каждой процедуры, кото-

рая может быть вызвана. Этот блок информации начинается со слова, зарезервированного для виртуального адреса процедуры, а за ним следует имя процедуры, которое сохраняется в виде цепочки символов.

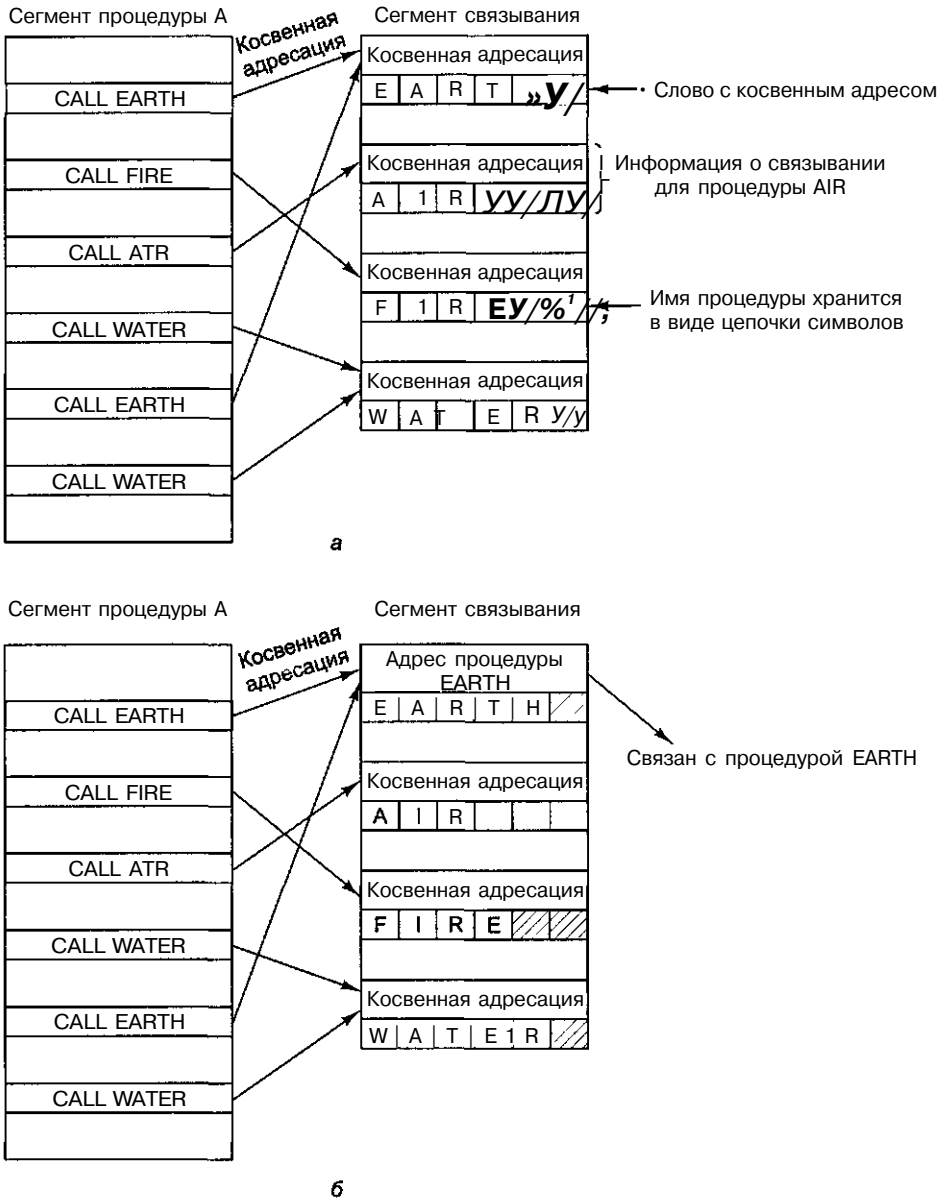


Рис. 7.7. Динамическое связывание: до вызова процедуры EARTH (а); после того как процедура EARTH была вызвана и связана (б)

При применении динамического связывания вызовы процедур во входном языке транслируются в команды, которые с помощью косвенной адресации обраща-

ются к первому слову соответствующего блока, как показано на рис. 7.7, *а*. Компилятор заполняет это слово либо недействительным адресом, либо специальным набором битов, который вызывает системное прерывание (ловушку).

Когда вызывается процедура в другом сегменте, попытка косвенно обратиться к недействительному слову вызывает системное прерывание компоновщика. Затем компоновщик находит цепочку символов в слове, которое следует за недействительным адресом, и начинает искать пользовательскую директорию для скомпилированной процедуры с таким именем. Затем этой процедуре приписывается виртуальный адрес (обычно в ее собственном сегменте), и этот виртуальный адрес записывается поверх недействительного адреса, как показано на рис. 7.7, *б*. После этого команда, которая вызвала ошибку, выполняется заново, что позволяет программе продолжать работу с того места, где она находилась до системного прерывания.

Все последующие обращения к этой процедуре будут выполняться без ошибок, поскольку слово с косвенным адресом теперь содержит действительный виртуальный адрес. Следовательно, компоновщик вызывается только тогда, когда некоторая процедура вызывается впервые. После этого вызывать компоновщик уже не нужно.

Динамическое связывание в системе Windows

Все версии операционной системы Windows, в том числе NT, поддерживают динамическое связывание. При динамическом связывании используется специальный файловый формат, который называется **DLL (Dynamic Link Library — динамически подключаемая библиотека).** Динамически подключаемые библиотеки могут содержать процедуры, данные или и то и другое вместе. Обычно они используются для того, чтобы два и более процессов могли разделять процедуры и данные библиотеки. Большинство файлов DDL имеют расширение **.dll,** но встречаются и другие расширения, например .drv (для библиотек драйверов — driver libraries) и .fon (для библиотек шрифтов — font libraries).

Самая распространенная форма динамически подключаемой библиотеки — библиотека, состоящая из набора процедур, которые могут загружаться в память и к которым имеют доступ несколько процессов одновременно. На рис. 7.8 показаны два процесса, которые разделяют файл DLL, содержащий 4 процедуры, A, B, C и D. Программа 1 использует процедуру A; программа 2 использует процедуру C, хотя они вполне могли бы использовать одну и ту же процедуру.

Файл DLL строится компоновщиком из коллекции входных файлов. Построение файла DDL очень похоже на построение исполняемого двоичного кода, только при создании файла DLL компоновщику передается специальный флаг, который сообщает ему, что создается именно файл DLL. Файлы DLL обычно конструируются из набора библиотечных процедур, которые могут понадобиться нескольким процессорам. Типичными примерами файлов DLL являются процедуры сопряжения с библиотекой системных вызовов Windows и большими графическими библиотеками. Применяя файлы DDL, мы можем сэкономить пространство в памяти и на диске. Если какая-то библиотека была связана с каждой программой, использующей ее, то она будет появляться во многих исполняемых двоичных программах в памяти и на диске, а забивать пространство такими дубликатами неэко-

номно. Если мы будем использовать файлы DLL, то каждая библиотека будет появляться один раз на диске и один раз в памяти.

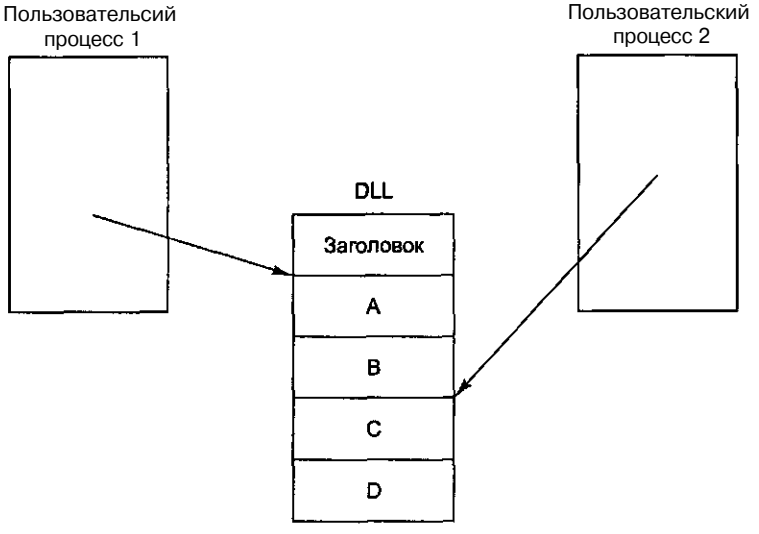


Рис. 7.8. Два процесса используют один файл DLL

Этот подход, кроме того, упрощает обновление библиотечных процедур и позволяет осуществлять обновление процедур, даже после того как программы, использующие их, были скомпилированы и связаны. Для коммерческих программных пакетов, где пользователям обычно недоступна входная программа, использование файлов DLL означает, что поставщик программного обеспечения может обнаруживать ошибки в библиотеках и исправлять положение, просто распространяя новые файлы DLL по Интернету, причем при этом не требуется производить никаких изменений в основных бинарных программах.

Основное различие между файлом DLL и исполняемой двоичной программой состоит в том, что файл DLL не может запускаться и работать сам по себе (поскольку у него нет ведущей программы). Он также содержит совершенно другую информацию в заголовке. Кроме того, файл DLL имеет несколько дополнительных процедур, не связанных с процедурами в библиотеке. Например, существует одна процедура, которая вызывается автоматически всякий раз, когда новый процесс связывается с файлом DLL, и еще одна процедура, которая вызывается автоматически всякий раз, когда связь процесса с файлом DLL отменяется. Эти процедуры могут выделять и освобождать память или управлять другими ресурсами, которые необходимы файлу DLL.

Программа может установить связь с файлом DLL двумя способами: с помощью неявного связывания и с помощью явного связывания. При **неявном связывании** пользовательская программа статически связывается со специальным файлом, так называемой **библиотекой импорта,** которая образована обслуживающей программой (утилитой), извлекающей определенную информацию из файла DLL. Библиотека импорта обеспечивает связующий элемент, который позволяет пользовательской программе получать доступ к файлу DLL. Пользовательская програм-

ма может быть связана с несколькими библиотеками импорта. Когда программа, которая применяет неявное связывание, загружается в память для выполнения, система Windows проверяет, какие файлы DLL использует эта программа и все ли эти файлы уже находятся в памяти. Те файлы, которых еще нет в памяти, загружаются туда немедленно (но необязательно целиком, поскольку они разбиты на страницы). Затем производятся некоторые изменения в структурах данных в библиотеках импорта так, чтобы можно было определить местоположение вызываемых процедур (это похоже на изменения, показанные на рис. 7.7). Их тоже нужно отобразить в виртуальное адресное пространство программы. С этого момента пользовательскую программу можно запускать. Теперь она может вызывать процедуры в файлах DLL, как будто они статически связаны с ней.

Альтернативой неявного связывания является **явное связывание.** Такой подход не требует библиотек импорта, и при нем не нужно загружать файлы DLL одновременно с пользовательской программой. Вместо этого пользовательская программа делает явный вызов прямо во время работы, чтобы установить связь с файлом DLL, а затем совершает дополнительные вызовы, чтобы получить адреса процедур, которые ей требуются. Когда все это сделано, программа совершает финальный вызов, чтобы разорвать связь с файлом DLL Когда последний процесс разрывает связь с файлом DLL, этот файл может быть удален из памяти.

Важно осознавать, что процедура в файле DLL не имеет отличительных особенностей (как поток или процесс). Она работает в потоке вызывающей программы и для своих локальных переменных использует стек вызывающей программы. Она может содержать специфичные для процесса статические данные (а также разделенные данные) и в остальном работает как статически связанная процедура. Единственным существенным отличием является способ установления связи.

Динамическое связывание в системе UNIX

В системе UNIX используется механизм, по сути сходный с файлами DLL в Windows. Это библиотека коллективного доступа. Как и файл DLL, библиотека коллективного доступа представляет собой архивный файл, содержащий несколько процедур или модулей данных, которые присутствуют в памяти во время работы программы и одновременно могут быть связаны с несколькими процессами. Стандартная библиотека C и большинство сетевых программ являются библиотеками коллективного доступа.

Система UNIX поддерживает только неявное связывание, поэтому библиотека коллективного доступа состоит из двух частей: **главной библиотеки** (host library), которая статически связана с исполняемым файлом, и **целевой библиотеки** (target library), которая вызывается во время работы программы. Несмотря на некоторые различия в деталях, по существу это то же, что файлы DLL.

Краткое содержание главы

Хотя большинство программ можно и нужно писать на языках высокого уровня, существуют такие ситуации, в которых необходимо применять язык ассемблера, по крайней мере отчасти. Это программы для компьютеров с недостаточным коли-

чеством ресурсов (например, кредитные карточки, различные приборы и портативные цифровые устройства). Программа на языке ассемблера — это символическая репрезентация программы на машинном языке. Она транслируется на машинный язык специальной программой, которая называется ассемблером.

Если для успеха какого-либо аппарата требуется быстрое выполнение программы, то лучше всего сначала написать программу на языке высокого уровня, затем путем измерений установить, выполнение какой части программы занимает большую часть времени, и переписать на языке ассемблера только эту часть программы. Практика показывает, что часто небольшая часть всей программы занимает большую часть всего времени выполнения этой программы.

Во многих ассемблерах предусмотрены макросы, которые позволяют программистам давать символические имена целым последовательностям команд. Обычно эти макросы могут быть параметризированы прямым путем. Макросы реализуются с помощью алгоритма обработки строковых литералов.

Большинство ассемблеров двухпроходные. Во время первого прохода строится таблица символов для меток, литералов и объявляемых идентификаторов. Символьные имена можно либо не сортировать и искать путем последовательного просмотра таблицы, либо сначала сортировать, а потом применять двоичный поиск, либо хэшировать. Если символьные имена не нужно удалять во время первого прохода, то хэширование — это лучший метод. Во время второго прохода происходит порождение программы. Одни директивы выполняются при первом проходе, а другие — при втором.

Программы, которые ассемблируются независимо друг от друга, можно связать вместе и получить исполняемую двоичную программу, которую можно запускать. Эту работу выполняет компоновщик. Его задачи — это перемещение в памяти и связывание имен. Динамическое связывание — это технология, при которой определенные процедуры не связываются до тех пор, пока они не будут вызваны. Библиотеки коллективного пользования в системе UNIX и файлы DLL (динамически подсоединяемые библиотеки) в системе Windows используют технологию динамического связывания.

Вопросы и задания

1. 1% определенной программы отвечает за 50% времени выполнения этой программы. Сравните следующие три стратегии с точки зрения времени программирования и времени выполнения. Предположим, что для написания программы на языке С потребуется 100 человеко-месяцев, а программу на языке ассемблера написать в 10 раз труднее, но зато она работает в 4 раза эффективнее.
   1. Вся программа написана на языке С.
   2. Вся программа написана на ассемблере.
   3. Программа сначала написана на С, а затем нужный 1% программы переписан на ассемблере.
2. Для двухпроходных ассемблеров существуют определенные соглашения. Подходят ли они для компиляторов?

3. Придумайте, как программисты на языке ассемблера могут определять синонимы для кодов операций. Как это можно реализовать?
4. Все ассемблеры для процессоров Intel имеют в качестве первого операнда адрес назначения, а в качестве второго — исходный адрес. Какие проблемы могут возникнуть при другом подходе?
5. Можно ли следующую программу ассемблировать в два прохода? *Примечание:* EQU — это директива, которая приравнивает метку и выражение в поле операнда.

```
A EQU B
B EQU C
C EQU D
O EQU 4
```

6. Одна компания планирует разработать ассемблер для компьютера с 40-битным словом. Чтобы снизить стоимость, менеджер проекта, доктор Скрудж, решил ограничить длину символьных имен, чтобы каждое имя можно было хранить в одном слове. Скрудж объявил, что символьные имена могут состоять только из букв, причем буква Q запрещена. Какова максимальная длина символьного имени? Опишите вашу схему кодировки.
7. Чем отличается команда от директивы?
8. Чем отличается счетчик адреса команд от счетчика команд? А существует ли вообще между ними различие? Ведь и тот и другой следят за следующей командой в программе.
9. Какой будет таблица символов (имен) после обработки следующих операторов ассемблера для Pentium II (первому оператору приписан адрес 1000)?

```
EVEREST: POP BX (1 байт)
   K2: PUSH BP (1 байт)
   WHITNEY: MOV BP.SP (2 байта)
   MCKINLEY: PUSH X (3 байта)
   FUJI: PUSH SI (1 байт)
   KIBO: SUB SI.300 (3 байта)
```

10. Можете ли вы представить себе обстоятельства, при которых метка совпадет с кодом операции (например, может ли быть MOV меткой)? Аргументируйте.
11. Какие шаги нужно совершить, чтобы, используя двоичный поиск, найти элемент «Berkeley» в следующем списке: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, Yellow Springs. Когда будете вычислять средний элемент в списке из четного числа элементов, возьмите элемент, который идет сразу после среднего индекса.
12. Можно ли использовать двоичный поиск в таблице, в которой содержится простое число элементов?
13. Вычислите хэш-код для каждого из следующих символьных имен. Для этого сложите буквы (A=1, B=2 и т. д.) и возьмите результат по модулю размера хэш-таблицы. Хэш-таблица содержит 19 слотов (от 0 до 18).

    els, jan, jelle, maaike

Образует ли каждое символьное имя уникальное значение хэш-функции? Если нет, то как можно разрешить эту коллизию?

14. Метод хэш-кодирования, описанный в тексте, связывает все элементы, имеющие один хэш-код, в связном списке. Альтернативный метод — иметь только одну таблицу из n слотов, в которой в каждом слоте имеется пространство для одного ключа и его значения (или для указателей на них). Если алгоритм хэширования порождает слот, который уже заполнен, производится вторая попытка с использованием того же алгоритма хэширования. Если и на этот раз слот заполнен, алгоритм используется снова и т. д. Так продолжается до тех пор, пока не будет найден пустой слот. Если доля слотов, которые уже заполнены, составляет R, сколько попыток в среднем понадобится для того, чтобы ввести в таблицу новый символ?

15. Вероятно, когда-нибудь в будущем на одну микросхему можно будет помещать тысячи идентичных процессоров, каждый из которых содержит несколько слов локальной памяти. Если все процессоры могут считывать и записывать три общих регистра, то как можно реализовать ассоциативную память?

16. Pentium II имеет сегментированную архитектуру. Сегменты независимы. Ассемблер для этой машины может содержать директиву SEG N, которая помещает последующий код и данные в сегмент N. Повлияет ли такая схема на счетчик адреса команды?

17. Программы часто связаны с многочисленными файлами DLL (динамически подсоединяемыми библиотеками). А не будет ли более эффективным просто поместить все процедуры в один большой файл DLL, а затем установить связь с ним?

18. Можно ли отобразить файл DLL в виртуальные адресные пространства двух процессов с разными виртуальными адресами? Если да, то какие проблемы при этом возникают? Можно ли их разрешить? Если нет, то что можно сделать, чтобы устранить их?

19. Опишем один из способов связывания. Перед сканированием библиотеки компоновщик составляет список необходимых процедур, то есть имен, которые в связываемых модулях определены как внешние (EXTERN). Затем компоновщик последовательно просматривает всю библиотеку, извлекая каждую процедуру, которая находится в списке нужных имен. Будет ли работать такая схема? Если нет, то почему, и как это можно исправить?

20. Может ли регистр использоваться в качестве фактического параметра в макровызове? А константа? Если да, то почему. Если нет, то почему.

21. Вам нужно реализовать макроассемблер. Из эстетических соображений ваш начальник решил, что макроопределения не должны предшествовать вызовам макросов. Как повлияет это решение на реализацию?

22. Подумайте, как можно поместить макроассемблер в бесконечный цикл.

23. Компоновщик считывает 5 модулей, длины которых составляют 200, 800, 600, 500 и 700 слов соответственно. Если они загружаются в этом порядке, то каковы константы перемещения?

24. Напишите модуль таблицы символов, состоящий из двух процедур: *enterisymbol, value)* и *lookup(symbol, value)*. Первый вводит новые символьные имена в таблицу, а второй ищет их в таблице. Используйте какую-либо хэш-кодировку.
25. Напишите простой ассемблер для компьютера Mic-1, о котором мы говорили в главе 4. Помимо оперирования машинными командами обеспечьте возможность приписывать константы символьным именам во время ассемблирования, а также способ ассемблировать константу в машинное слово.
26. Добавьте макросы к ассемблеру, который вы должны были написать, выполняя предыдущее задание.

Глава 8
Архитектуры компьютеров параллельного действия

Скорость работы компьютеров становится все выше, но и требования, предъявляемые к ним, тоже постоянно растут. Астрономы хотят воспроизвести всю историю Вселенной с момента большого взрыва до самого конца. Фармацевты хотели бы разрабатывать новые лекарственные препараты для конкретных заболеваний с помощью компьютеров, не принося в жертву целые легионы крыс. Разработчики летательных аппаратов могли бы прийти к более эффективным результатам, если бы всю работу за них выполняли компьютеры, и тогда им не нужно было бы конструировать аэродинамическую трубу. Если говорить коротко, насколько бы мощными ни были компьютеры, этого никогда не хватает для решения многих задач (особенно научных, технических и промышленных).

Скорость работы тактовых генераторов постоянно повышается, но скорость коммутации нельзя увеличивать бесконечно. Главной проблемой остается скорость света, и заставить протоны и электроны перемещаться быстрее невозможно. Из-за высокой теплоотдачи компьютеры превратились в кондиционеры. Наконец, поскольку размеры транзисторов постоянно уменьшаются, в какой-то момент времени каждый транзистор будет состоять из нескольких атомов, поэтому основной проблемой могут стать законы квантовой механики (например, гейзенберговский принцип неопределенности).

Чтобы решать более сложные задачи, разработчики обращаются к компьютерам параллельного действия. Невозможно построить компьютер с одним процессором и временем цикла в 0,001 не, но зато можно построить компьютер с 1000 процессорами, время цикла каждого из которых составляет 1 не. И хотя во втором случае мы используем процессоры, которые работают с более низкой скоростью, общая производительность теоретически должна быть такой же.

Параллелизм можно вводить на разных уровнях. На уровне команд, например, можно использовать конвейеры и суперскалярную архитектуру, что позволяет увеличивать производительность примерно в 10 раз. Чтобы увеличить производительность в 100, 1000 или 1 000 000 раз, нужно продублировать процессор или, по крайней мере, какие-либо его части и заставить все эти процессоры работать вместе.

В этой главе мы изложим основные принципы разработки компьютеров параллельного действия и рассмотрим различные примеры. Все эти машины состоят из элементов процессора и элементов памяти. Отличаются они друг от друга количеством элементов, их типом и способом взаимодействия между элементами. В одних

разработках используется небольшое число очень мощных элементов, а в других — огромное число элементов со слабой мощностью. Существуют промежуточные типы компьютеров. В области параллельной архитектуры проведена огромная работа. Мы кратко расскажем об этом в данной главе. Дополнительную информацию можно найти в книгах [86,115, 131, 159].

Вопросы разработки компьютеров параллельного действия

Когда мы сталкиваемся с новой компьютерной системой параллельного действия, возникает три вопроса:

1. Каков тип, размер и количество процессорных элементов?
2. Каков тип, размер и количество модулей памяти?
3. Как взаимодействуют элементы памяти и процессорные элементы?

Рассмотрим каждый из этих пунктов. Процессорные элементы могут быть самых различных типов — от минимальных АЛУ до полных центральных процессоров, а по размеру один элемент может быть от небольшой части микросхемы до кубического метра электроники. Очевидно, что если процессорный элемент представляет собой часть микросхемы, то можно поместить в компьютер огромное число таких элементов (например, миллион). Если процессорный элемент представляет собой целый компьютер со своей памятью и устройствами ввода-вывода, цифры будут меньше, хотя были сконструированы такие системы даже с 10 000 процессорами. Сейчас компьютеры параллельного действия конструируются из серийно выпускаемых частей. Разработка компьютеров параллельного действия часто зависит от того, какие функции выполняют эти части и каковы ограничения.

Системы памяти часто разделены на модули, которые работают независимо друг от друга, чтобы несколько процессоров одновременно могли осуществлять доступ к памяти. Эти модули могут быть маленького размера (несколько килобайтов) или большого размера (несколько мегабайтов). Они могут находиться или рядом с процессорами, или на другой плате. Динамическая память (динамическое ОЗУ) работает гораздо медленнее центральных процессоров, поэтому для повышения скорости доступа к памяти обычно используются различные схемы кэш-памяти. Может быть два, три и даже четыре уровня кэш-памяти.

Хотя существуют самые разнообразные процессоры и системы памяти, системы параллельного действия различаются в основном тем, как соединены разные части. Схемы взаимодействия можно разделить на две категории: статические и динамические. В статических схемах компоненты просто связываются друг с другом определенным образом. В качестве примеров статических схем можно привести звезду, кольцо и решетку. В динамических схемах все компоненты подсоединены к переключательной схеме, которая может трассировать сообщения между компонентами. У каждой из этих схем есть свои достоинства и недостатки.

Компьютеры параллельного действия можно рассматривать как набор микросхем, которые соединены друг с другом определенным образом. Это один подход. При другом подходе возникает вопрос, какие именно процессы выполняются

параллельно. Здесь существует несколько вариантов. Некоторые компьютеры параллельного действия одновременно выполняют несколько независимых задач. Эти задачи никак не связаны друг с другом и не взаимодействуют. Типичный пример — компьютер, содержащий от 8 до 64 процессоров, представляющий собой большую систему UNIX с разделением времени, с которой могут работать тысячи пользователей. В эту категорию попадают системы обработки транзакций, которые используются в банках (например, банковские автоматы), на авиалиниях (например, системы резервирования) и в больших web-серверах. Сюда же относятся независимые прогоны моделирующих программ, при которых используется несколько наборов параметров.

Другие компьютеры параллельного действия выполняют одну задачу, состоящую из нескольких параллельных процессов. В качестве примера рассмотрим программу игры в шахматы, которая анализирует данные позиции на доске, порождает список возможных из этих позиций ходов, а затем порождает параллельные процессы, чтобы проанализировать каждую новую ситуацию параллельно. Здесь параллелизм нужен не для того, чтобы обслуживать большое количество пользователей, а чтобы ускорить решение одной задачи.

Далее идут машины с высокой степенью конвейеризации или с большим количеством АЛУ, которые обрабатывают одновременно один поток команд. В эту категорию попадают суперкомпьютеры со специальным аппаратным обеспечением для обработки векторных данных. Здесь решается одна главная задача, и при этом все части компьютера работают вместе над одним аспектом этой задачи (например, разные элементы двух векторов суммируются параллельно).

Эти три примера различаются по так называемой **степени детализации.** В многопроцессорных системах с разделением времени блок параллелизма достаточно велик — целая пользовательская программа. Параллельная работа больших частей программного обеспечения практически без взаимодействия между этими частями называется **параллелизмом на уровне крупных** структурных единиц. Диаметрально противоположный случай (при обработке векторных данных) называется **параллелизмом на уровне мелких структурных единиц.**

Термин «степень детализации» применяется по отношению к алгоритмам и программному обеспечению, но у него есть прямой аналог в аппаратном обеспечении. Системы с небольшим числом больших процессоров, которые взаимодействуют по схемам с низкой скоростью передачи данных, называются **системами с косвенной (слабой) связью.** Им противопоставляются **системы с непосредственной (тесной) связью,** в которых компоненты обычно меньше по размеру, расположены ближе друг к другу и взаимодействуют через специальные коммуникационные сети с высокой пропускной способностью. В большинстве случаев задачи с параллелизмом на уровне крупных структурных единиц лучше всего решаются в системах со слабой связью, а задачи с параллелизмом на уровне мелких структурных единиц лучше всего решаются в системах с непосредственной связью. Однако существует множество различных алгоритмов и множество разнообразного программного и аппаратного обеспечения. Разнообразие степени детализации и возможности различной степени связности систем привели к многообразию архитектур, которые мы будем изучать в этой главе.

В следующих разделах мы рассмотрим некоторые вопросы разработки компьютеров параллельного действия. Мы начнем с информационных моделей и сетей межсоединений, затем рассмотрим вопросы, связанные с производительностью и программным обеспечением, и, наконец, перейдем к классификации архитектур компьютеров параллельного действия.

Информационные модели

В любой системе параллельной обработки процессоры, выполняющие разные части одной задачи, должны как-то взаимодействовать друг с другом, чтобы обмениваться информацией. Как именно должен происходить этот обмен? Было предложено и реализовано две разработки: мультипроцессоры и мультикомпьютеры. Мы рассмотрим их ниже.

Мультипроцессоры

В первой разработке все процессоры разделяют общую физическую память, как показано на рис. 8.1, а. Такая система называется **мультипроцессором** или **системой с совместно используемой памятью.**

Мультипроцессорная модель распространяется на программное обеспечение. Все процессы, работающие вместе на мультипроцессоре, могут разделять одно виртуальное адресное пространство, отображенное в общую память. Любой процесс может считывать слово из памяти или записывать слово в память с помощью команд LOAD и STORE. Больше ничего не требуется. Два процесса могут обмениваться информацией, если один из них будет просто записывать данные в память, а другой будет считывать эти данные.

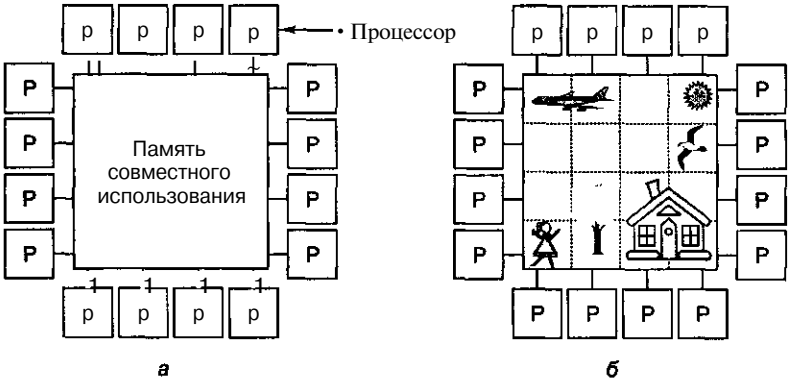


Рис. 8.1. Мультипроцессор, содержащий 16 процессоров, которые разделяют общую память (а); изображение, разбитое на 16 секций, каждую из которых анализирует отдельный процессор (б)

Благодаря такой возможности взаимодействия двух и более процессов мультипроцессоры весьма популярны. Данная модель понятна программистам и приложима к широкому кругу задач. Рассмотрим программу, которая изучает битовое отображение и составляет список всех его объектов. Одна копия отображения хранится в памяти, как показано на рис. 8.1, б. Каждый из 16 процессоров запускает

один процесс, которому приписана для анализа одна из 16 секций. Если процесс обнаруживает, что один из его объектов переходит через границу секции, этот процесс просто переходит вслед за объектом в следующую секцию, считывая слова этой секции. В нашем примере некоторые объекты обрабатываются несколькими процессами, поэтому в конце потребуется некоторая координация, чтобы определить количество домов, деревьев и самолетов.

В качестве примеров мультипроцессоров можно назвать Sun Enterprise 10000, Sequent NUMA-Q, SGI Origin 2000 и HP/Convex Exemplar.

Мультикомпьютеры

Во втором типе параллельной архитектуры каждый процессор имеет свою собственную память, доступную только этому процессору. Такая разработка называется **мультикомпьютером** или **системой с распределенной памятью.** Она изображена на рис. 8.2, *а.* Мультикомпьютеры обычно (хотя не всегда) являются системами со слабой связью. Ключевое отличие мультикомпьютера от мультипроцессора состоит в том, что каждый процессор в мультикомпьютере имеет свою собственную локальную память, к которой этот процессор может обращаться, выполняя команды LOAD и STORE, но никакой другой процессор не может получить доступ к этой памяти с помощью тех же команд LOAD и STORE. Таким образом, мультипроцессоры имеют одно физическое адресное пространство, разделяемое всеми процессорами, а мультикомпьютеры содержат отдельное физическое адресное пространство для каждого центрального процессора.

Поскольку процессоры в мультикомпьютере не могут взаимодействовать друг с другом просто путем чтения из общей памяти и записи в общую память, здесь необходим другой механизм взаимодействия. Они посылают друг другу сообщения, используя сеть межсоединений. В качестве примеров мультикомпьютеров можно назвать IBM SP/2, Intel/Sandia Option Red и Wisconsin COW.

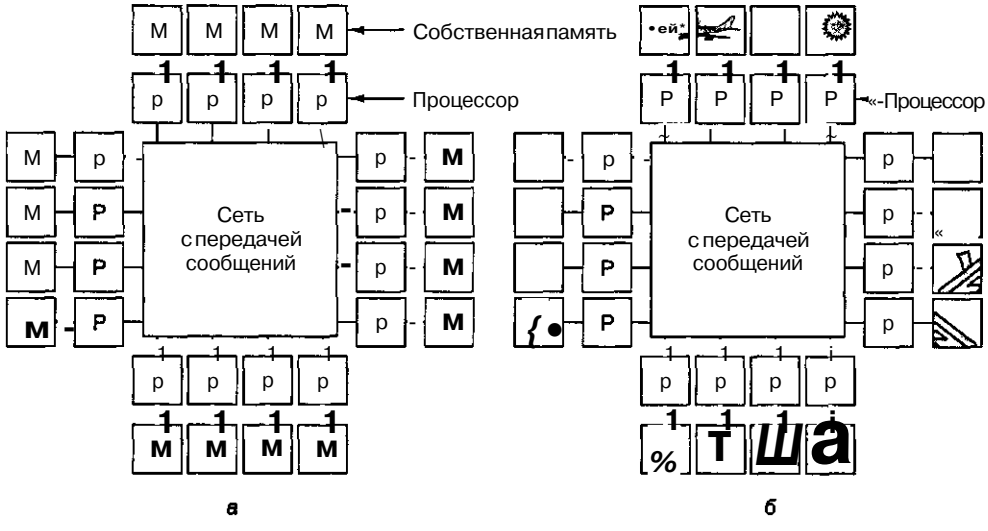


Рис. 8.2. Мультикомпьютер, содержащий 16 процессоров, каждый из которых имеет свою собственную память (а); битовое отображение рис. 8.1, разделенное между 16 участками памяти (б)

При отсутствии памяти совместного использования в аппаратном обеспечении предполагается определенная структура программного обеспечения. В мультикомпьютере невозможно иметь одно виртуальное адресное пространство, из которого все процессы могут считывать информацию и в которое все процессы могут записывать информацию просто путем выполнения команд LOAD и STORE. Например, если процессор 0 (в верхнем левом углу) на рис. 8.1, ^обнаруживает, что часть его объекта попадает в другую секцию, относящуюся к процессору 1, он может продолжать считывать информацию из памяти, чтобы получить хвост самолета. Однако если процессор 0 на рис. 8.2, # обнаруживает это, он не может просто считать информацию из памяти процессора 1. Для получения необходимых данных ему нужно сделать что-то другое.

В частности, ему нужно как-то определить, какой процессор содержит необходимые ему данные, и послать этому процессору сообщение с запросом копии данных. Затем процессор 0 блокируется до получения ответа. Когда процессор 1 получает сообщение, программное обеспечение должно проанализировать его и отправить назад необходимые данные. Когда процессор 0 получает ответное сообщение, программное обеспечение разблокируется и продолжает работу.

В мультикомпьютере для взаимодействия между процессорами часто используются примитивы send и receive. Поэтому программное обеспечение мультикомпьютера имеет более сложную структуру, чем программное обеспечение мультипроцессора. При этом основной проблемой становится правильное разделение данных и разумное их размещение. В мультипроцессоре размещение частей не влияет на правильность выполнения задачи, хотя может повлиять на производительность. Таким образом, мультикомпьютер программировать гораздо сложнее, чем мультипроцессор.

Возникает вопрос: зачем вообще создавать мультикомпьютеры, если мультипроцессоры гораздо проще запрограммировать? Ответ прост: гораздо проще и дешевле построить большой мультикомпьютер, чем мультипроцессор с таким же количеством процессоров. Реализация общей памяти, разделяемой несколькими сотнями процессоров, — это весьма сложная задача, а построить мультикомпьютер, содержащий 10 000 процессоров и более, довольно легко.

Таким образом, мы сталкиваемся с дилеммой: мультипроцессоры сложно строить, но легко программировать, а мультикомпьютеры легко строить, но трудно программировать. Поэтому стали предприниматься попытки создания гибридных систем, которые относительно легко конструировать и относительно легко программировать. Это привело к осознанию того, что совместную память можно реализовывать по-разному, и в каждом случае будут какие-то преимущества и недостатки. Практически все исследования в области архитектур с параллельной обработкой направлены на создание гибридных форм, которые сочетают в себе преимущества обеих архитектур. Здесь важно получить такую систему, которая расширяема, то есть которая будет продолжать исправно работать при добавлении все новых и новых процессоров.

Один из подходов основан на том, что современные компьютерные системы не монолитны, а состоят из ряда уровней. Это дает возможность реализовать общую

память на любом из нескольких уровней, как показано на рис. 8.3. На рис. 8.3, *а* мы видим память совместного использования, реализованную в аппаратном обеспечении в виде реального мультипроцессора. В данной разработке имеется одна копия операционной системы с одним набором таблиц, в частности таблицей распределения памяти. Если процессу требуется больше памяти, он прерывает работу операционной системы, которая после этого начинает искать в таблице свободную страницу и отображает эту страницу в адресное пространство вызывающей программы. Что касается операционной системы, имеется единая память, и операционная система следит, какая страница в программном обеспечении принадлежит тому или иному процессу. Существует множество способов реализации совместной памяти в аппаратном обеспечении.

Второй подход — использовать аппаратное обеспечение мультикомпьютера и операционную систему, которая моделирует разделенную память, обеспечивая единое виртуальное адресное пространство, разбитое на страницы. При таком подходе, который называется **DSM (Distributed Shared Memory — распределенная совместно используемая память)** [82,83, 84], каждая страница расположена в одном из блоков памяти (см. рис. 8.2, с). Каждая машина содержит свою собственную виртуальную память и собственные таблицы страниц. Если процессор совершает команду LOAD или STORE над страницей, которой у него нет, происходит прерывание операционной системы. Затем операционная система находит нужную страницу и требует, чтобы процессор, который обладает нужной страницей, преобразовал ее в исходную форму и послал по сети межсоединений. Когда страница достигает пункта назначения, она отображается в память, и выполнение прерванной команды возобновляется. По существу, операционная система просто вызывает недостающие страницы не с диска, а из памяти. Но у пользователя создается впечатление, что машина содержит общую разделенную память. DSM мы рассмотрим ниже в этой главе.

Третий подход — реализовать общую разделенную память на уровне программного обеспечения. При таком подходе абстракцию разделенной памяти создает язык программирования, и эта абстракция реализуется компилятором. Например, модель Linda основана на абстракции разделенного пространства кортежей (записей данных, содержащих наборы полей). Процессы любой машины могут взять кортеж из общего пространства или отправить его в общее пространство. Поскольку доступ к этому пространству полностью контролируется программным обеспечением (системой Linda), никакого специального аппаратного обеспечения или специальной операционной системы не требуется.

Другой пример памяти совместного использования, реализованной в программном обеспечении, — модель общих объектов в системе Orca. В модели Orca процессы разделяют объекты, а не кортежи, и могут выполнять над ними те или иные процедуры. Если процедура изменяет внутреннее состояние объекта, операционная система должна проследить, чтобы все копии этого объекта на всех машинах одновременно были изменены. И опять, поскольку объекты — это чисто программное понятие, их можно реализовать с помощью программного обеспечения без вмешательства операционной системы или аппаратного обеспечения. Модели Linda и Orca мы рассмотрим ниже в этой главе.

Рис. 8.3. Уровни, на которых можно реализовать память совместного использования: аппаратное обеспечение (а); операционная система (б); программное обеспечение (в)

Сети межсоединений

На рис. 8.2 мы показали, что мультикомпьютеры связываются через сети межсоединений. Рассмотрим их подробнее. Интересно отметить, что мультикомпьютеры и мультипроцессоры очень сходны в этом отношении, поскольку мультипроцессоры часто содержат несколько модулей памяти, которые также должны быть связаны друг с другом и с процессорами. Следовательно, многое из того, о чем мы будем говорить в этом разделе, применимо к обоим типам систем.

Основная причина сходства коммуникационных связей в мультипроцессоре и мультикомпьютере заключается в том, что в обоих случаях применяется передача сообщений. Даже в однопроцессорной машине, когда процессору нужно считать или записать слово, он устанавливает определенные линии на шине и ждет ответа. Это действие представляет собой то же самое, что и передача сообщений: инициатор посылает запрос и ждет ответа. В больших мультипроцессорах взаимодействие между процессорами и удаленной памятью почти всегда состоит в том, что процессор посылает в память сообщение, так называемый пакет, который запрашивает определенные данные, а память посылает процессору ответный пакет.

Сети межсоединений могут состоять максимум из пяти компонентов:

1. Центральные процессоры.
2. Модули памяти.
3. Интерфейсы.
4. Каналы связи.
5. Коммутаторы.

Процессоры и модули памяти мы уже рассматривали в этой книге и больше не будем к этому возвращаться. Интерфейсы — это устройства, которые вводят и выводят сообщения из центральных процессоров и модулей памяти. Во многих разработках интерфейс представляет собой микросхему или плату, к которой подсоединяется локальная шина каждого процессора и которая может передавать сигналы процессору и локальной памяти (если таковая есть). Часто внутри интерфейса содержится программируемый процессор со своим собственным ПЗУ, которое принадлежит только этому процессору. Обычно интерфейс способен считывать и записывать информацию в различные мсдули памяти, что позволяет ему перемещать блоки данных.

Каналы связи — это каналы, по которым перемещаются биты. Каналы могут быть электрическими или оптико-волоконными, последовательными (шириной 1 бит) или параллельными (шириной более 1 бита). Каждый канал связи характеризуется максимальной пропускной способностью (это максимальное число битов, которое он способен передавать в секунду). Каналы могут быть симплексными (передавать биты только в одном направлении), полудуплексными (передавать информацию в обоих направлениях, но не одновременно) и дуплексными (передавать биты в обоих направлениях одновременно).

Коммутаторы — это устройства с несколькими входными и несколькими выходными портами. Когда на входной порт приходит пакет, некоторые биты в этом пакете используются для выбора выходного порта, в который посылается пакет. Размер пакета может составлять 2 или 4 байта, но может быть и значительно больше (например, 8 Кбайт).

Сети межсоединений можно сравнить с улицами города. Улицы похожи на каналы связи. Каждая улица может быть с односторонним и двусторонним движением, она характеризуется определенной «скоростью передачи данных» (имеется в виду ограничение скорости движения) и имеет определенную ширину (число рядов). Перекрестки похожи на коммутаторы. На каждом перекрестке прибывающий пакет (пешеход или машина) выбирает, в какой выходной порт (улицу) поступить дальше в зависимости от того, каков конечный пункт назначения.

При разработке и анализе сети межсоединений важно учитывать несколько ключевых моментов. Во-первых, это топология (то есть способ расположения компонентов). Во-вторых, это то, как работает система переключения и как осуществляется связь между ресурсами. В-третьих, какой алгоритм выбора маршрута используется для доставки сообщений в пункт назначения. Ниже мы рассмотрим каждый из этих пунктов.

Топология

Топология сети межсоединений определяет, как расположены каналы связи и коммутаторы (это, например, может быть кольцо или решетка). Топологии можно изображать в виде графов, в которых дуги соответствуют каналам связи, а узлы — коммутаторам (рис. 8.4). С каждым узлом в сети (или в соответствующем графе) связан определенный ряд каналов связи. Математики называют число каналов **степенью** узла, инженеры — **коэффициентом разветвления.** Чем больше степень, тем больше вариантов маршрута и тем выше отказоустойчивость. Если каждый узел содержит **к** дуг и соединение сделано правильно, то можно построить сеть межсоединений так, чтобы она оставалась полносвязной, даже если к-1 каналов повреждены.

Следующее свойство сети межсоединений — это ее **диаметр.** Если расстоянием между двумя узлами мы будем считать число дуг, которые нужно пройти, чтобы попасть из одного узла в другой, то диаметром графа будет расстояние между двумя узлами, которые расположены дальше всех друг от друга. Диаметр сети определяет самую большую задержку при передаче пакетов от одного процессора к другому или от процессора к памяти, поскольку каждая пересылка через канал связи занимает определенное количество времени. Чем меньше диаметр, тем выше производительность. Также имеет большое значение среднее расстояние между двумя узлами, поскольку от него зависит среднее время передачи пакета.

Еще одно важное свойство сети межсоединений — это ее пропускная способность, то есть количество данных, которое она способна передавать в секунду. Очень важная характеристика — **бисекционная пропускная способность.** Чтобы вычислить это число, нужно мысленно разделить сеть межсоединений на две равные (с точки зрения числа узлов) несвязанные части путем удаления ряда дуг из графа. Затем нужно вычислить общую пропускную способность дуг, которые мы удалили. Существует множество способов разделения сети межсоединений на две равные части. Бисекционная пропускная способность — минимальная из всех возможных. Предположим, что бисекционная пропускная способность составляет 800 бит/с. Тогда если между двумя частями много взаимодействий, то общую пропускную способность в худшем случае можно сократить до 800 бит/с. По мнению многих разработчиков, бисекционная пропускная способность — это самая важ-

ная характеристика сети межсоединений. Часто основная цель при разработке сети межсоединений — сделать бисекционную пропускную способность максимальной.

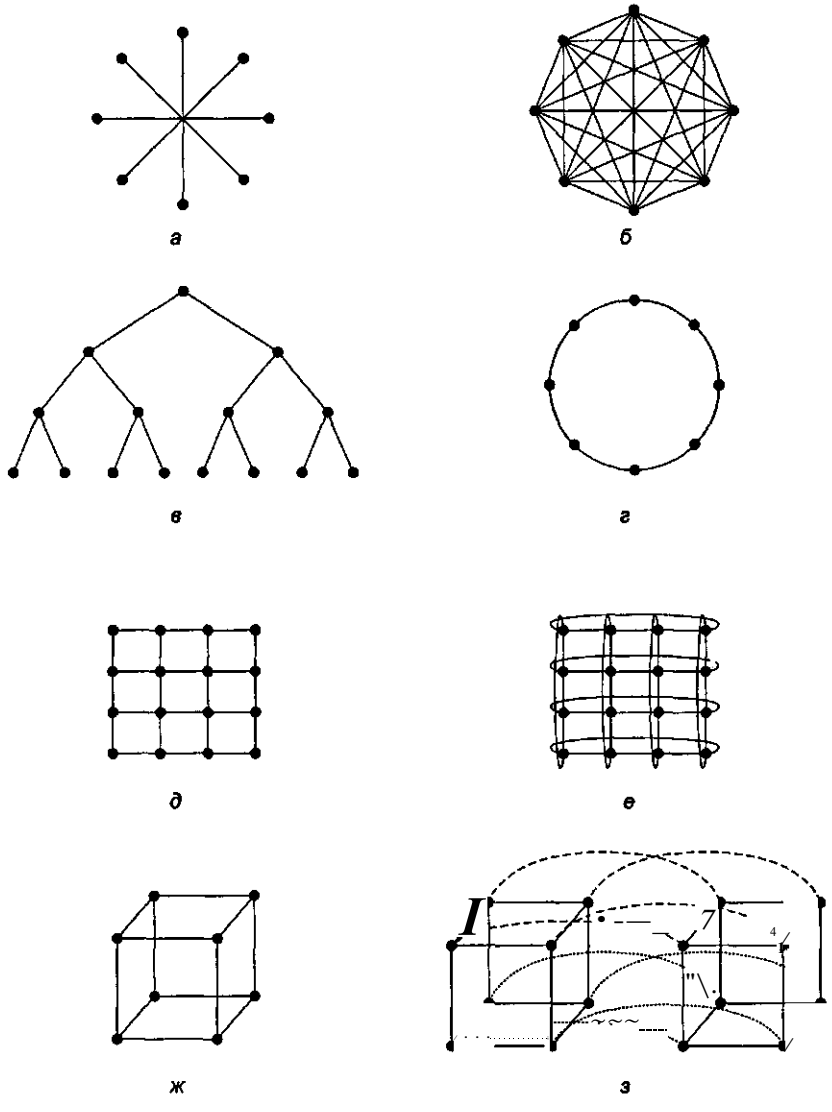


Рис. 8.4. Различные топологии. Жирные точки соответствуют коммутаторам. Процессоры и модули памяти не показаны: звезда (а); полное межсоединение (full interconnect) (б); дерево (в); кольцо (г); решетка (д); двойной тор (е); куб (ж); гиперкуб (з)

Сети межсоединений можно характеризовать по их **размерности.** Размерность определяется по числу возможных вариантов перехода из исходного пункта в пункт назначения. Если выбора нет (то есть существует только один путь из каждого исходного пункта в каждый конечный пункт), то сеть нульмерная. Если есть два возможных варианта (например, если можно пойти либо направо, либо налево),

то сеть одномерна. Если есть две оси и пакет может направиться направо или налево либо вверх или вниз, то такая сеть двумерна и т. д.

На рис. 8.4 показано несколько топологий. Здесь изображены только каналы связи (это линии) и коммутаторы (это точки). Модули памяти и процессоры (они на рисунке не показаны) подсоединяются к коммутаторам через интерфейсы. На рис. 8.4, *а* изображена нульмерная конфигурация звезда, где процессоры и модули памяти прикрепляются к внешним узлам, а переключение совершает центральный узел. Такая схема очень проста, но в большой системе центральный коммутатор будет главным критическим параметром, который ограничивает производительность системы. И с точки зрения отказоустойчивости это очень неудачная разработка, поскольку одна ошибка в центральном коммутаторе может разрушить всю систему.

На рис. 8.4, *б* изображена другая нульмерная топология — **полное межсоединение** (full interconnect). Здесь каждый узел непосредственно связан с каждым имеющимся узлом. В такой разработке пропускная способность между двумя секциями максимальна, диаметр минимален, а отказоустойчивость очень высока (даже при утрате шести каналов связи система все равно будет полностью взаимосвязана). Однако для **к** узлов требуется к(к-1)/2 каналов, а это совершенно неприемлемо для больших значений к.

На рис. 8.4, *в* изображена третья нульмерная топология — **дерево.** Здесь основная проблема состоит в том, что пропускная способность между секциями равна пропускной способности каналов. Обычно у верхушки дерева наблюдается очень большой поток обмена информации, поэтому верхние узлы становятся препятствием для повышения производительности. Можно разрешить эту проблему, увеличив пропускную способность верхних каналов. Например, самые нижние каналы будут иметь пропускную способность Ь, следующий уровень — пропускную способность 2Ь, а каждый канал верхнего уровня — пропускную способность 4b. Такая схема называется **толстым деревом (fat tree).** Она применялась в коммерческих мультикомпьютерах Thinking Machines' CM-5.

Кольцо (рис. 8.4, *г)* — это одномерная топология, поскольку каждый отправленный пакет может пойти направо или налево. **Решетка** или **сетка** (рис. 8.4, *д)* — это двумерная топология, которая применяется во многих коммерческих системах. Она отличается регулярностью и применима к системам большого размера, а диаметр составляет квадратный корень от числа узлов (то есть при расширении системы диаметр увеличивается незначительно). **Двойной тор** (рис. 8.4, *е)* является разновидностью решетки. Это решетка, у которой соединены края. Она характеризуется большей отказоустойчивостью и меньшим диаметром, чем обычная решетка, поскольку теперь между двумя противоположными узлами всего два транзитных участка.

Куб (рис. 8.4, *ж)* — это правильная трехмерная топология. На рисунке изображен куб 2x2x2, но в общем случае он может быть kxkxk. На рис. 8.4, *з* показан четырехмерный куб, полученный из двух трехмерных кубов, которые связаны между собой. Можно сделать пятимерный куб, соединив вместе 4 четырехмерных куба. Чтобы получить 6 измерений, нужно продублировать блок из 4 кубов и соединить соответствующие узлы и т. д.; n-мерный куб называется **гиперкубом.** Эта топология используется во многих компьютерах параллельного действия, поскольку ее

диаметр находится в линейной зависимости от размерности. Другими словами, диаметр — это логарифм по основанию 2 от числа узлов, поэтому 10-мерный гиперкуб имеет 1024 узла, но диаметр равен всего 10, что дает очень незначительные задержки при передаче данных. Отметим, что решетка 32x32, которая также содержит 1024 узла, имеет диаметр 62, что более чем в шесть раз превышает диаметр гиперкуба. Однако чем меньше диаметр гиперкуба, тем больше разветвление и число каналов (и следовательно, тем выше стоимость). Тем не менее в системах с высокой производительностью чаще всего используется именно гиперкуб.

Коммутация

Сеть межсоединений состоит из коммутаторов и проводов, соединяющих их. На рисунке 8.5 изображена небольшая сеть межсоединений с четырьмя коммутаторами. В данном случае каждый коммутатор имеет 4 входных порта и 4 выходных порта. Кроме того, каждый коммутатор содержит несколько центральных процессоров и схемы соединения (на рисунке они показано не полностью). Задача коммутатора — принимать пакеты, которые приходят на любой входной порт, и отправлять пакеты из соответствующих выходных портов.

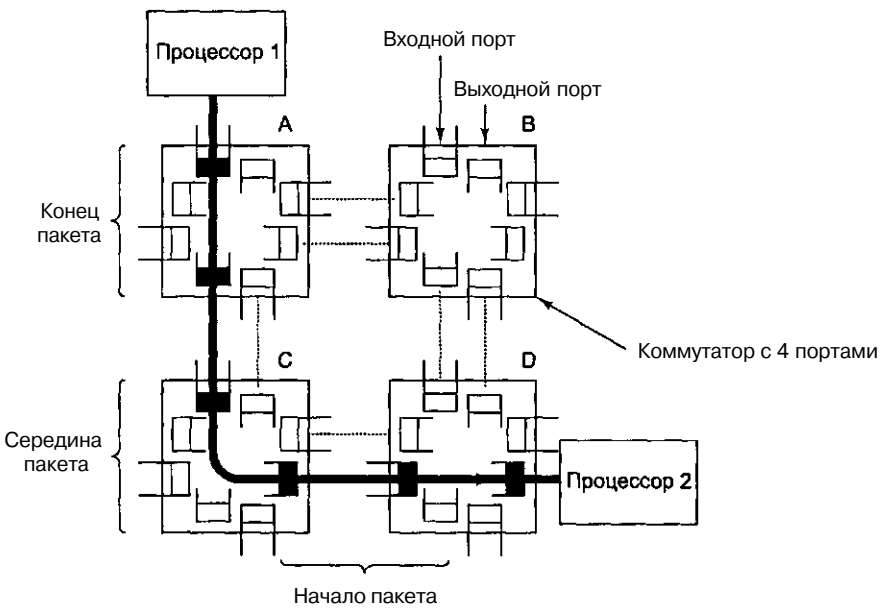


Рис. 8.5. Сеть межсоединений в форме квадратной решетки с четырьмя коммутаторами. Здесь показаны только два процессора

Каждый выходной порт связан с входным портом другого коммутатора через последовательный или параллельный канал (на рис. 8.5. это пунктирная линия). Последовательные каналы передают один бит единовременно. Параллельные каналы могут передавать несколько битов сразу. Существуют специальные сигналы для управления каналом. Параллельные каналы характеризуются более высокой производительностью, чем последовательные каналы с такой же тактовой часто-

той, но в них возникает проблема **расфазировки данных** (нужно быть уверенным, что все биты прибывают одновременно), и они стоят гораздо дороже.

Существует несколько стратегий переключения Первая из них — **коммутация каналов.** Перед тем как послать пакет, весь путь от начального до конечного пункта резервируется заранее. Все порты и буферы затребованы заранее, поэтому когда начинается процесс передачи, все необходимые ресурсы гарантированно доступны, и биты могут на полной скорости перемещаться от исходного пункта через все коммутаторы к пункту назначения. На рис. 8.5 показана коммутация каналов, где резервируются канал от процессора 1 к процессору 2 (черная жирная стрелка). Здесь резервируются три входных и три выходных порта.

Коммутацию каналов можно сравнить с перекрытием движения транспорта во время парада, когда блокируются все прилегающие улицы. При этом требуется предварительное планирование, но после блокирования прилегающих улиц парад может продвигаться на полной скорости, поскольку никакой транспорт препятствовать этому не будет Недостаток такого метода состоит в том, что требуется предварительное планирование и любое движение транспорта запрещено, даже если парад (или пакеты) еще не приближается.

Вторая стратегия — **коммутация с промежуточным хранением.** Здесь не требуется предварительного резервирования. Из исходного пункта посылается целый пакет к первому коммутатору, где он хранится целиком. На рис. 8 6, а исходным пунктом является процессор 1, а весь пакет, который направляется в процессор 2, сначала сохраняется внутри коммутатора A. Затем этот пакет перемещается в коммутатор C, как показано на рис. 8.6, *б.* Затем весь пакет целиком перемещается в коммутатор D (рис. 8.6, *в).* Наконец, пакет доходит до пункта назначения — до процессора 2. Отметим, что никакого предварительного резервирования ресурсов не требуется.

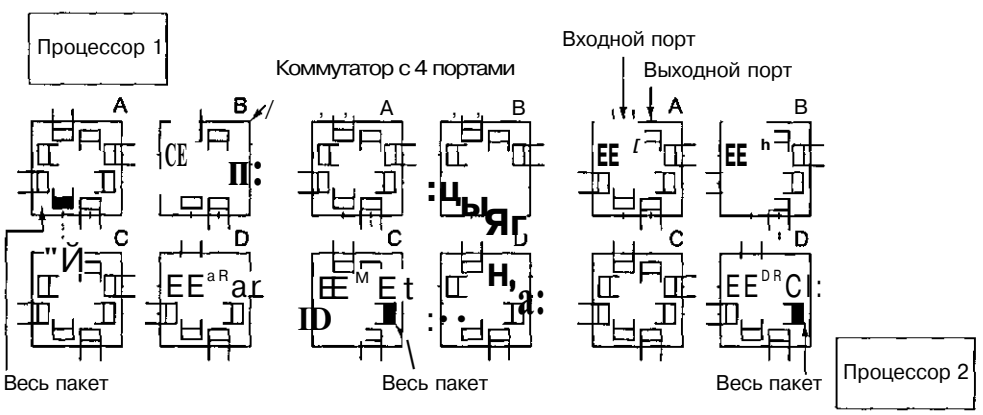


Рис. 8.6. Коммутация с промежуточным хранением

Коммутаторы с промежуточным хранением должны отправлять пакеты в буфер, поскольку когда исходный пункт (например, процессор, память или коммутатор) выдает пакет, требующийся выходной порт может быть в данный момент занят передачей другого пакета. Если бы не было буферизации, входящие пакеты, кото-

рым нужен занятый в данный момент выходной порт, пропадали бы. Применяется три метода буферизации. При **буферизации** на **входе** один или несколько буферов связываются с каждым входным портом в форме очереди типа FIFO («первым вошел, первым вышел»). Если пакет в начале очереди нельзя передать по причине занятости нужного выходного порта, этот пакет просто ждет своей очереди.

Однако если пакет ожидает, когда освободится выходной порт, то пакет, идущий за ним, тоже не может передаваться, даже если нужный ему порт свободен. Ситуация называется **блокировкой начала** очереди. Проиллюстрируем ситуацию на примере. Представим дорогу из двух рядов. Вереница машин в одном из рядов не может двигаться дальше, поскольку первая машина в этом ряду хочет повернуть налево, но не может из-за движения машин другого ряда. Даже если второй и всем следующим за ней машинам нужно ехать прямо, первая машина в ряду препятствует их движению.

Проблему можно устранить с помощью **буферизации на выходе.** В этой системе буферы связаны с выходными портами. Биты пакета по мере пребывания сохраняются в буфере, который связан с нужным выходным портом. Поэтому пакеты, направленные в порт т, не могут блокировать пакеты, направленные в порт п.

И при буферизации на входе, и при буферизации на выходе с каждым портом связано определенное количество буферов. Если места недостаточно для хранения всех пакетов, то какие-то пакеты придется выбрасывать. Чтобы разрешить эту проблему, можно использовать **общую буферизацию,** при которой один буферный пул динамически распределяется по портам по мере необходимости. Однако такая схема требует более сложного управления, чтобы следить за буферами, и позволяет одному занятому соединению захватить все буферы, оставив другие соединения ни с чем. Кроме того, каждый коммутатор должен вмещать самый большой пакет и даже несколько пакетов максимального размера, а для этого потребуется ужесточить требования к памяти и снизить максимальный размер пакета.

Хотя метод коммутации с промежуточным хранением гибок и эффективен, здесь возникает проблема возрастающей задержки при передаче данных по сети межсоединений. Предположим, что время, необходимое для перемещения пакета по одному транзитному участку на рис. 8.6, занимает Т не. Чтобы переместить пакет из процессора 1 в процессор 2, нужно скопировать его 4 раза (в А, в С, в D и в процессор 2), и следующее копирование не может начаться, пока не закончится предыдущее, поэтому задержка по сети составляет 4Т. Чтобы выйти из этой ситуации, нужно разработать гибридную сеть межсоединений, объединяющую в себе коммутацию каналов и коммутацию пакетов. Например, каждый пакет можно разделить на части. Как только первая часть поступила в коммутатор, ее можно сразу направить в следующий коммутатор, даже если оставшиеся части пакета еще не прибыли в этот коммутатор.

Такой подход отличается от коммутации каналов тем, что ресурсы не резервируются заранее. Следовательно, возможна конфликтная ситуация в соревновании за право обладания ресурсами (портами и буферами). При **коммутации без буферизации пакетов,** если первый блок пакета не может двигаться дальше, оставшаяся часть пакета продолжает поступать в коммутатор. В худшем случае эта схема

превратится в коммутацию с промежуточным хранением. При другом типе маршрутизации, так называемой **«wormhole routing» (червоточина),** если первый блок не может двигаться дальше, в исходный пункт передается сигнал остановить передачу, и пакет может оборваться, будучи растянутым на два и более коммутаторов. Когда необходимые ресурсы становятся доступными, пакет может двигаться дальше.

Следует отметить, что оба подхода аналогичны конвейерному выполнению команд в центральном процессоре. В любой момент времени каждый коммутатор выполняет небольшую часть работы, и в результате получается более высокая производительность, чем если бы эту же работу выполнял один из коммутаторов.

Алгоритмы выбора маршрута

В любой сети межсоединений с размерностью один и выше можно выбирать, по какому пути передавать пакеты от одного узла к другому. Часто существует множество возможных маршрутов. Правило, определяющее, какую последовательность узлов должен пройти пакет при движении от исходного пункта к пункту назначения, называется **алгоритмом выбора маршрута.**

Хорошие алгоритмы выбора маршрута необходимы, поскольку часто свободными оказываются несколько путей. Хороший алгоритм поможет равномерно распределить нагрузку по каналам связи, чтобы полностью использовать имеющуюся в наличии пропускную способность. Кроме того, алгоритм выбора маршрута помогает избегать взаимоблокировки в сети межсоединений. Взаимоблокировка возникает в том случае, ее та при одновременной передаче нескольких пакетов ресурсы затребованы таким образом, что ни один из пакетов не может продвигаться дальше и все они блокируются навечно.

Пример тупиковой ситуации в сети с коммутацией каналов приведен на рис. 8.7. Тупиковая ситуация может возникать и в сети с пакетной коммутацией, но ее легче представить графически в сети с коммутацией каналов. Здесь каждый процессор пытается послать пакет процессору, находящемуся напротив него по диагонали. Каждый из них смог зарезервировать входной и выходной порты своего локального коммутатора, а также один входной порт следующего коммутатора, но он уже не может получить необходимый выходной порт на втором коммутаторе, поэтому он просто ждет, пока не освободится этот порт. Если все четыре процессора начинают этот процесс одновременно, то все они блокируются и сеть зависает.

Алгоритмы выбора маршрута можно разделить на две категории: маршрутизация от источника и распределенная маршрутизация. При **маршрутизации от источника** источник определяет весь путь по сети заранее. Этот путь выражается списком из номеров портов, которые нужно будет использовать в каждом коммутаторе по пути к пункту назначения. Если путь проходит через **к** коммутаторов, то первые к байтов в каждом пакете будут содержать к номеров выходных портов, 1 байт на каждый порт. Когда пакет доходит до коммутатора, первый байт отсекается и используется для определения выходного порта. Оставшаяся часть пакета затем направляется в соответствующий порт. После каждого транзитного участка пакет становится на 1 байт короче, показывая новый номер порта, который нужно выбрать в следующий раз.

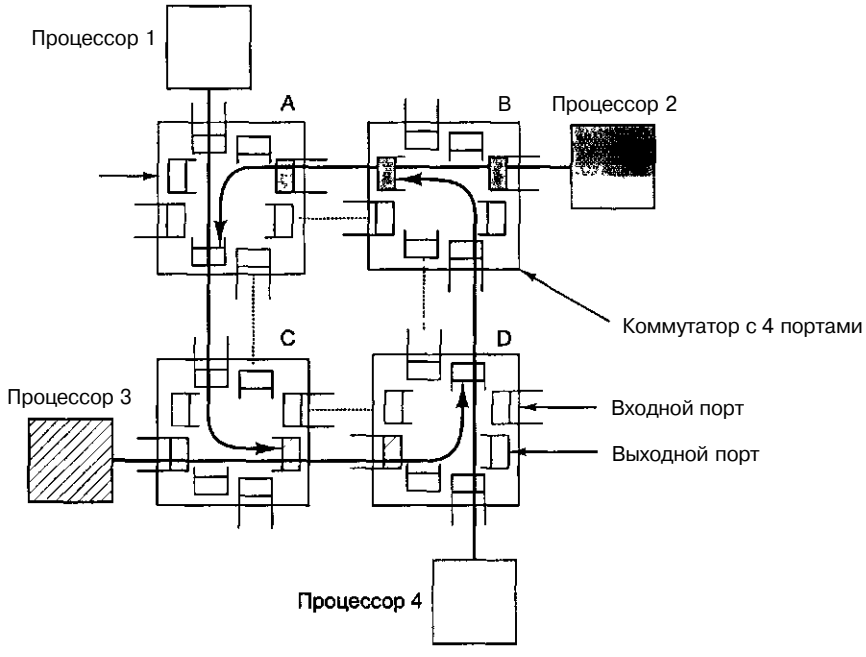


Рис. 8.7. Тупиковая ситуация в сети с коммутацией каналов

При распределенной маршрутизации каждый коммутатор сам решает, в какой порт отправить каждый приходящий пакет. Если выбор одинаков для каждого пакета, направленного к одному и тому же конечному пункту, то маршрутизация является **статической.** Если коммутатор при выборе принимает во внимание текущий трафик, то маршрутизация является **адаптивной.**

Популярным алгоритмом маршрутизации, который применяется для прямоугольных решеток с любым числом измерений и в котором никогда не возникает тупиковых ситуаций, является **пространственная маршрутизация.** В соответствии с этим алгоритмом пакет сначала перемещается вдоль оси *x* до нужной координаты, а затем вдоль оси *y* до нужной координаты и т. д. (в зависимости от количества измерений). Например, чтобы перейти из (3,7, 5) в (6,9, 8), пакет сначала должен переместиться из точки x=3 в точку x=6 через (4, 7, 5), (5, 7, 5) и (6,7, 5). Затем он должен переместиться по оси *y* через (6, 8, 5) и (6, 9, 5). Наконец, он должен переместиться по оси *z* в (6, 9, 6), (6, 9, 7) и (6, 9, 8). Такой алгоритм предотвращает тупиковые ситуации.

Производительность

Цель создания компьютера параллельного действия — сделать так, чтобы он работал быстрее, чем однопроцессорная машина. Если эта цель не достигнута, то никакого смысла в построении компьютера параллельного действия нет. Более того, эта цель должна быть достигнута при наименьших затратах. Машина, которая работает в два раза быстрее, чем однопроцессорная, но стоит в 50 раз дороже послед-

ней, не будет пользоваться особым спросом. В этом разделе мы рассмотрим некоторые вопросы производительности, связанные с созданием архитектур параллельных компьютеров.

Метрика аппаратного обеспечения

В аппаратном обеспечении наибольший интерес представляет скорость работы процессоров, устройств ввода-вывода и сети. Скорость работы процессоров и устройств ввода-вывода такая же, как и в однопроцессорной машине, поэтому ключевыми параметрами в параллельной системе являются те, которые связаны с межсоединением. Здесь есть два ключевых момента: время ожидания и пропускная способность. Мы рассмотрим их по очереди.

Полное время ожидания — это время, которое требуется на то, чтобы процессор отправил пакет и получил ответ. Если пакет посылается в память, то время ожидания — это время, которое требуется на чтение и запись слова или блока слов. Если пакет посылается другому процессору, то время ожидания — это время, которое требуется на межпроцессорную связь для пакетов данного размера. Обычно интерес представляет время ожидания для пакетов минимального размера (как правило, для одного слова или небольшой строки кэш-памяти).

Время ожидания строится из нескольких факторов. Для сетей с коммутацией каналов, сетей с промежуточным хранением и сетей без буферизации пакетов характерно разное время ожидания. Для коммутации каналов время ожидания составляет сумму времени установки и времени передачи. Для установки схемы нужно выслать пробный пакет, чтобы зарезервировать необходимые ресурсы, а затем передать назад сообщение об этом. После этого можно ассемблировать пакет данных. Когда пакет готов, биты можно передавать на полной скорости, поэтому если общее время установки составляет T_s, размер пакета равен p бит, а пропускная способность b битов в секунду, то время ожидания в одну сторону составит T_s+p/b. Если схема дуплексная и никакого времени установки на ответ не требуется, то минимальное время ожидания для передачи пакета размером в p бит и получения ответа размером в p битов составляет T_s+2p/b секунд.

При пакетной коммутации не нужно посылать пробный пакет в пункт назначения заранее, но все равно требуется некоторое время установки, T_a, на компоновку пакета. Здесь время передачи в одну сторону составляет $T_a+p/Ь$, но за этот период пакет доходит только до первого коммутатора. При прохождении через сам коммутатор получается некоторая задержка, $T<ь$ а затем происходит переход к следующему коммутатору и т. д. Время T_d состоит из времени обработки и задержки в очереди (когда нужно ждать, пока не освободится выходной порт). Если имеется n коммутаторов, то общее время ожидания в одну сторону составляет $T_a+n(p/b+T_d)+p/b$, где последнее слагаемое отражает копирование пакета из последнего коммутатора в пункт назначения.

Время ожидания в одну сторону для коммутации без буферизации пакетов и «червоточины» в лучшем случае будет приближаться к $T_a+p/Ь$, поскольку здесь нет пробных пакетов для установки схемы и нет задержки, обусловленной промежуточным хранением. По существу, это время начальной установки для компоновки пакета плюс время на передачу битов. Следовало бы еще прибавить задержку на распространение сигнала, но она обычно незначительна.

Следующая характеристика аппаратного обеспечения — пропускная способность. Многие программы параллельной обработки, особенно в естественных науках, перемещают огромное количество данных, поэтому число байтов, которое система способна перемещать в секунду, имеет очень большое значение для производительности. Существует несколько показателей пропускной способности. Один из них — пропускная способность между двумя секциями — мы уже рассмотрели. Другой показатель — **суммарная пропускная способность** — вычисляется путем суммирования пропускной способности всех каналов связи. Это число показывает максимальное число битов, которое можно передать сразу. Еще один важный показатель — средняя пропускная способность каждого процессора. Если каждый процессор способен выдавать только 1 Мбайт/с, то от сети с пропускной способностью между секциями в 100 Гбайт/с не будет толку. Скорость взаимодействия будет ограничена тем, сколько данных может выдавать каждый процессор.

На практике приблизиться к теоретически возможной пропускной способности очень трудно. Пропускная способность сокращается по многим причинам. Например, каждый пакет всегда содержит какие-то служебные сигналы и данные: это компоновка, построение заголовка, отправка. При отправке 1024 пакетов по 4 байта каждый мы никогда не достигнем той же пропускной способности, что и при отправке 1 пакета на 4096 байтов. К сожалению, для достижения маленького времени ожидания лучше использовать маленькие пакеты, поскольку большие надолго блокируют линии и коммутаторы. В результате возникает конфликт между достижением низкого времени ожидания и высокой пропускной способности. Для одних прикладных задач первое важнее, чем второе, для других — наоборот. Важно знать, что всегда можно купить более высокую пропускную способность (добавив больше проводов или поставив более широкие провода), но нельзя купить низкое время ожидания. Поэтому лучше сначала сделать время ожидания как можно меньше, а уже потом заботиться о пропускной способности.

Метрика программного обеспечения

Метрика аппаратного обеспечения показывает, на что способно аппаратное обеспечение. Но пользователей интересует совсем другое. Они хотят знать, насколько быстрее будут работать их программы на компьютере параллельного действия по сравнению с однопроцессорным компьютером. Для них ключевым показателем является коэффициент ускорения: насколько быстрее работает программа в n-процессорной системе по сравнению с 1-процессорной системой. Результаты обычно иллюстрируются графиком (рис. 8.8.). Здесь мы видим несколько разных параллельных программ, которые работают на мультикомпьютере, состоящем из 64 процессоров Pentium Pro. Каждая кривая показывает повышение скорости работы одной программы с k процессорами как функцию от k. Идеальное повышение скорости показано пунктирной линией, где использование **k** процессоров заставляет программу работать в k раз быстрее для любого k. Лишь немногие программы достигают совершенного повышения скорости, но есть достаточное число программ, которые приближаются к идеалу. Скорость работы N-объектной задачи с добавле-

нием новых процессоров увеличивается очень стремительно; авари (африканская игра) ускоряется вполне сносно; но инвертирование матрицы нельзя ускорить более чем в пять раз, сколько бы процессоров мы не использовали. Программы и результаты обсуждаются в книге [14].

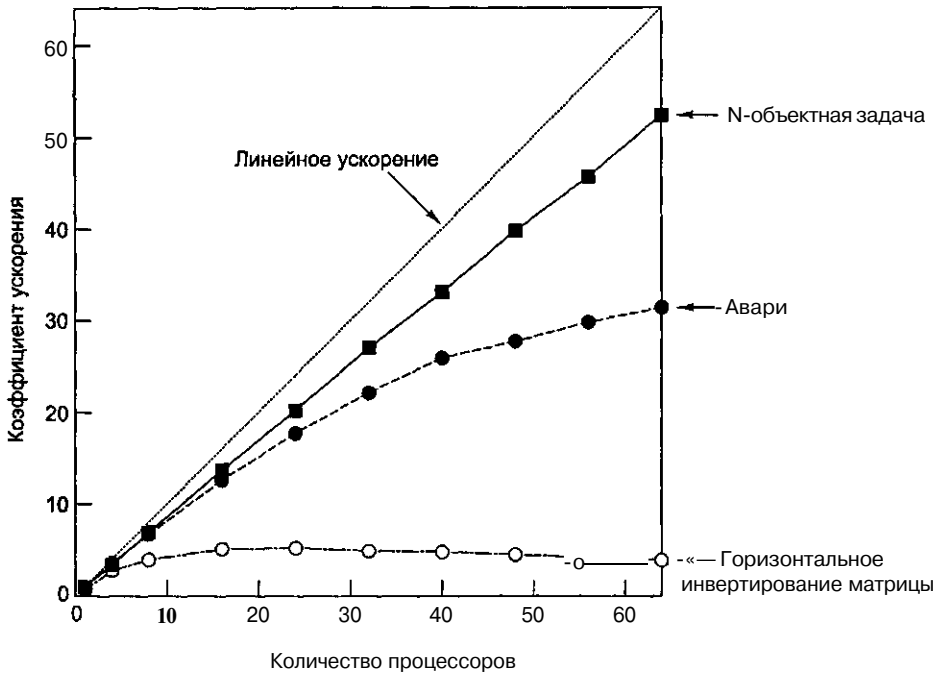


Рис. 8.8. На практике программы не могут достичь идеального повышения скорости. Идеальный коэффициент ускорения показан пунктирной линией

Есть ряд причин, по которым практически невозможно достичь идеального повышения скорости: все программы содержат последовательную часть, они часто имеют фазу инициализации, они обычно должны считывать данные и собирать результаты. Большое количество процессоров здесь не поможет. Предположим, что на однопроцессорном компьютере программа работает T секунд, причем доля (f) от этого времени является последовательным кодом, а доля (1-f) потенциально параллелизуется, как показано на рис. 8.9, а. Если второй код можно запустить на n процессорах без издержек, то время выполнения программы в лучшем случае сократится с (1-f)T до (1-f)T/n, как показано на рис. 8.9, б. В результате общее время выполнения программы (и последовательной и параллельной частей) будет fT+(I-f)T/n. Коэффициент ускорения — это время выполнения изначальной программы (T), разделенное на это новое время выполнения:

Speedup» n /d+(n-l)f)

Для f=0 мы можем получить линейное повышение скорости, но для f>0 идеальное повышение скорости невозможно, поскольку в программе имеется последовательная часть. Это явление носит название **закона Амдала.**

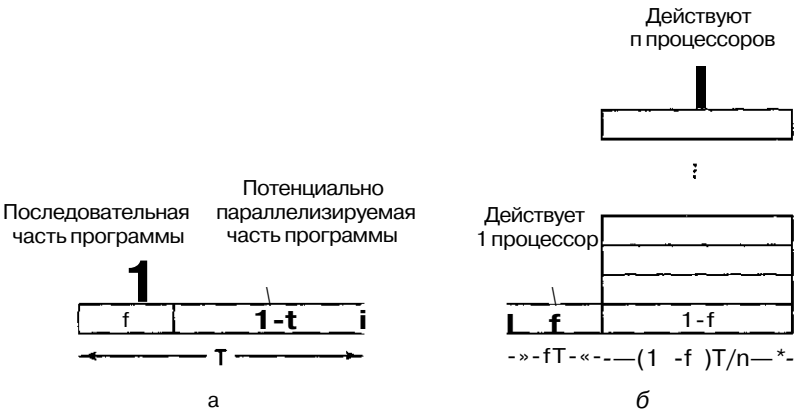


Рис. 8.9. Программа содержит последовательную часть и параллелизуемую часть (а); результат параллельной обработки части программы (б)

Закон Амдала — это только одна причина, по которой невозможно идеальное повышение скорости. Определенную роль в этом играет и время ожидания в коммуникациях, и ограниченная пропускная способность, и недостатки алгоритмов. Даже если мы имели бы в наличии 1000 процессоров, не все программы можно написать так, чтобы использовать такое большое число процессоров, а непроизводительные издержки для запуска их всех могут быть очень значительными. Кроме того, многие известные алгоритмы трудно подвергнуть параллельной обработке, поэтому в данном случае приходится использовать субоптимальный алгоритм. Для многих прикладных задач желательно заставить программу работать в п раз быстрее, даже если для этого потребуется 2п процессоров. В конце концов, процессоры не такие уж и дорогие.

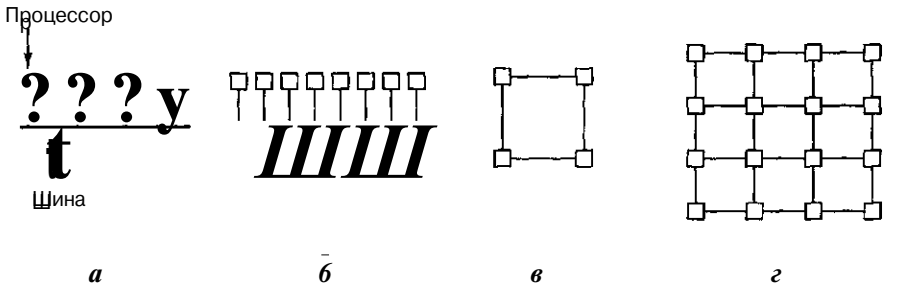
Как достичь высокой производительности

Самый простой способ — включить в систему дополнительные процессоры. Однако добавлять процессоры нужно таким образом, чтобы при этом не ограничивать повышение производительности системы. Система, к которой можно добавлять процессоры и получать соответственно этому большую производительность, называется **расширяемой.**

Рассмотрим 4 процессора, которые соединены шиной (рис. 8.10, а). А теперь представим, что мы расширили систему до 16 процессоров, добавив еще 12 (рис. 8.10, *б).* Если пропускная способность шины составляет b Мбайт/с, то увеличив в 4 раза число процессоров, мы сократим имеющуюся пропускную способность каждого процессора с Ь/4 Мбайт/с до Ь/16 Мбайт/с. Такая система не является расширяемой.

А теперь проделаем то же действие с сеткой (решеткой) межсоединений (рис. 8.10, *в, г).* В такой топологии при добавлении новых процессоров мы добавляем новые каналы, поэтому при расширении системы суммарная пропускная способность на каждый процессор не снизится, как это было в случае с шиной. Отношение числа каналов к числу процессоров увеличивается от 1,0 при наличии 4 процессоров (4 процессора, 4 канала) до 1,5 при наличии 16 процессоров

(16 процессоров, 24 канала), поэтому с добавлением новых процессоров суммарная пропускная способность на каждый процессор увеличивается.



а *б* *в* *г*

Рис. 8.10. Система из 4 процессоров, соединенных шиной (а); система из 16 процессоров, соединенных шиной (б); сетка межсоединений из 4 процессоров (в); сетка межсоединений из 16 процессоров (г)

Естественно, пропускная способность — это не единственный параметр. Добавление процессоров к шине не увеличивает диаметр сети или время ожидания при отсутствии трафика, а добавление процессоров к решетке, напротив, увеличивает. Диаметр решетки пхп равен 2(п—1), поэтому в худшем случае время ожидания растет примерно как квадратный корень от числа процессоров. Для 400 процессоров диаметр равен 38, а для 1600 процессоров — 78, поэтому если увеличить число процессоров в 4 раза, то диаметр, а следовательно, и среднее время ожидания вырастут приблизительно вдвое.

В идеале расширяемая система при добавлении новых процессоров должна сохранять одну и ту же среднюю пропускную способность на каждый процессор и постоянное среднее время ожидания. На практике сохранение достаточной пропускной способности на каждый процессор осуществимо, но время ожидания растет с увеличением размера. Лучше всего было бы сделать так, чтобы она росла логарифмически, как в гиперкубе.

Дело в том, что время ожидания часто является фатальным для производительности в мелкомодульных и среднемодульных приложениях. Если программе требуются данные, которых нет в ее локальной памяти, на их получение требуется существенное количество времени, и чем больше система, тем больше получается задержка. Эта проблема существует и в мультипроцессорах, и в мультикомпьютерах, поскольку в обоих случаях физическая память разделена на неизменяемые широко раскинувшиеся модули.

Системные разработчики применяют несколько различных технологий, которые позволяют сократить или, по крайней мере, скрыть время ожидания. Первая технология — это копирование данных. Если копии блока данных можно хранить в нескольких местах, то можно увеличить скорость доступа к этим данным. Один из возможных вариантов — использование кэш-памяти, когда одна или несколько копий блоков данных хранятся близко к тому месту, где они могут понадобиться. Другой вариант — сохранять несколько равноправных копий — копий с равным статусом (в противоположность асимметричным отношениям первичности/вторичности, которые наблюдаются при использовании кэш-памяти). Когда сохраняется несколько копий, главные вопросы — это кем, когда и куда они помещены.

Здесь возможны самые разные варианты — от динамического размещения по требованию аппаратного обеспечения до намеренного размещения во время загрузки директив компилятора. Во всех случаях главным вопросом является согласованность управления.

Вторая технология — так называемая **упреждающая выборка.** Элемент данных можно вызвать еще до того, как он понадобится. Это позволяет перекрыть процесс вызова и процесс выполнения, и когда потребуется этот элемент данных, он уже будет доступен. Упреждающая выборка может быть автоматической, а может контролироваться программой. В кэш-память загружается не только нужное слово, а вся строка кэш-памяти целиком, и другие слова из этой строки тоже могут пригодиться в будущем.

Процессом упреждающей выборки можно управлять и явным образом. Когда компилятор узнает, что ему потребуются какие-либо данные, он может выдать явную команду, чтобы получить эти данные, и выдает он эту команду заранее с таким расчетом, чтобы получить нужные данные вовремя. Такая стратегия требует, чтобы компилятор обладал полными знаниями о машине и ее синхронизации, а также контролировал, куда помещаются все данные. Спекулятивные команды LOAD работают лучше всего, когда абсолютно точно известно, что эти данные потребуются. Ошибка из-за отсутствия страницы при выполнении команды LOAD для ветви, которая в конечном итоге не используется, очень невыгодна.

Третья технология — это **многопоточная обработка.** В большинстве современных систем поддерживается мультипрограммирование, при котором несколько процессов могут работать одновременно (либо создавать иллюзию параллельной работы на основе разделения времени). Если переключение между процессами можно совершать достаточно быстро, например, предоставляя каждому из них его собственную схему распределения памяти и аппаратные регистры, то когда один процесс блокируется и ожидает прибытия данных, аппаратное обеспечение может быстро переключиться на другой процесс. В предельном случае процессор выполняет первую команду из потока 1, вторую команду из потока 2 и т. д. Таким образом, процессор всегда будет занят, даже при длительном времени ожидания в отдельных потоках.

Некоторые машины автоматически переключаются от процесса к процессу после каждой команды, чтобы скрыть длительное время ожидания. Эта идея была реализована в одном из первых суперкомпьютеров CDC 6600. Было объявлено, что он содержит 10 периферийных процессоров, которые работают параллельно. А на самом деле он содержал только один периферийный процессор, который моделировал 10 процессоров. Он выполнял по порядку по одной команде из каждого процессора, сначала одну команду из процессора 1, затем одну команду из процессора 2 и т. д.

Четвертая технология — использование неблокирующих записей. Обычно при выполнении команды STORE, процессор ждет, пока она не закончится, и только после этого продолжает работу. При наличии неблокирующих записей начинается операция памяти, но программа все равно продолжает работу. Продолжать работу программы при выполнении команды LOAD сложнее, но даже это возможно, если применять исполнение с изменением последовательности.

Программное обеспечение

Эта глава в первую очередь посвящена архитектуре параллельных компьютеров, но все же стоит сказать несколько слов о программном обеспечении. Без программного обеспечения с параллельной обработкой параллельное аппаратное обеспечение не принесет никакой пользы, поэтому хорошие разработчики аппаратного обеспечения должны учитывать особенности программного обеспечения. Подробнее о программном обеспечении для параллельных компьютеров см. [160].

Существует 4 подхода к разработке программного обеспечения для параллельных компьютеров. Первый подход — добавление специальных библиотек численного анализа к обычным последовательным языкам. Например, библиотечная процедура для инвертирования большой матрицы или для решения ряда дифференциальных уравнений с частными производными может быть вызвана из последовательной программы, после чего она будет выполняться на параллельном процессоре, а программист даже не будет знать о существовании параллелизма. Недостаток этого подхода состоит в том, что параллелизм может применяться только в нескольких процедурах, а основная часть программы останется последовательной.

Второй подход — добавление специальных библиотек, содержащих примитивы коммуникации и управления. Здесь программист сам создает процесс параллелизма и управляет им, используя дополнительные примитивы.

^_ Следующий шаг — добавление нескольких специальных конструкций к существующим языкам программирования, позволяющих, например, легко порождать новые параллельные процессы, выполнять повторения цикла параллельно или выполнять арифметические действия над всеми элементами вектора одновременно. Этот подход широко используется, и очень во многие языки программирования были включены элементы параллелизма.

Четвертый подход — ввести совершенно новый язык специально для параллельной обработки. Очевидное преимущество такого языка — он очень хорошо подходит для параллельного программирования, но недостаток его в том, что программисты должны изучать новый язык. Большинство новых параллельных языков императивные (их команды изменяют переменные состояния), но некоторые из них функциональные, логические или объектно-ориентированные.

Существует очень много библиотек, расширений языков и новых языков, изобретенных специально для параллельного программирования, и они дают широчайший спектр возможностей, поэтому их очень трудно классифицировать. Мы сосредоточим наше внимание на пяти ключевых вопросах, которые формируют основу программного обеспечения для компьютеров параллельного действия:

1. Модели управления.
2. Степень распараллеливания процессов.
3. Вычислительные парадигмы.
4. Методы коммуникации.
5. Базисные элементы синхронизации.

Ниже мы обсудим каждый из этих вопросов в отдельности.

Модели управления

Самый фундаментальный вопрос в работе программного обеспечения — сколько будет потоков управления, один или несколько. В первой модели существует одна программа и один счетчик команд, но несколько наборов данных. Каждая команда выполняется над всеми наборами данных одновременно разными обрабатывающими элементами.

В качестве примера рассмотрим программу, которая получает ежечасные измерения температур от тысяч датчиков и должна вычислить среднюю температуру для каждого датчика. Когда программа вызывает команду

```
LOAD THE TEMPERATURE FOR 1 A.M. INTO REGISTER R1
```

(загрузить температуру в 1 час ночи в регистр R1), каждый процессор выполняет эту команду, используя свои собственные данные и свой регистр R1. Затем, когда программа вызывает команду

```
ADD THE TEMPERATURE FOR 2 A.M. TO REGISTER R1
```

(добавить температуру в 2 часа ночи в регистр R1), каждый процессор выполняет эту команду, используя свои собственные данные. В конце вычислений каждый процессор должен будет сосчитать среднюю температуру для каждого отдельного датчика.

Такая модель программирования имеет очень большое значение для аппаратного обеспечения. По существу, это значит, что каждый обрабатывающий элемент — это АЛУ и память, без схемы декодирования команд. Вместо этого один центральный блок вызывает команды и сообщает всем АЛУ, что делать дальше.

Альтернативная модель предполагает несколько потоков управления, каждый из которых содержит собственный счетчик команд, регистры и локальные переменные. Каждый поток управления выполняет свою собственную программу над своими данными, при этом он время от времени может взаимодействовать с другими потоками управления. Существует множество вариаций этой идеи, и в совокупности они формируют основную модель для параллельной обработки. По этой причине мы сосредоточимся на параллельной обработке с несколькими потоками контроля.

Степень распараллеливания процессов

Параллелизм управления можно вводить на разных уровнях. На самом низком уровне элементы параллелизма могут содержаться в отдельных машинных командах (например, в архитектуре IA-64). Программисты обычно не знают о существовании такого параллелизма — он управляется компилятором или аппаратным обеспечением.

На более высоком уровне мы приходим к **параллелизму на уровне блоков** (block-level parallelism), который позволяет программистам самим контролировать, какие высказывания будут выполняться последовательно, а какие — параллельно. Например, в языке Algol-68 выражение

```
begin Statement-1; Statement-2: Statement-3 end
```

использовалось для создания блока трех (в данном случае трех) произвольных высказываний, которые должны выполняться последовательно, а выражение

```
begin Statement-1. Statement-2. Statement-3 end
```

использовалось для параллельного выполнения тех же трех высказываний. С помощью правильного размещения точек с запятой, запятых, скобок и разграничителей **begin/end** можно было записать произвольную комбинацию команд для последовательного или параллельного выполнения.

Присутствует параллелизм на уровне более крупных структурных единиц, когда можно вызвать процедуру и не заставлять вызывающую программу ждать завершения этой процедуры. Это значит, что вызывающая программа и вызванная процедура будут работать параллельно. Если вызывающая программа находится в цикле, который вызывает процедуру при каждом прохождении и не ждет завершения этих процедур, то значит, большое число параллельных процедур запускается одновременно.

Другая форма параллелизма — создание или порождение для каждого процесса нескольких **потоков,** каждый из которых работает в пределах адресного пространства этого процесса. Каждый поток имеет свой счетчик команд, свои регистры и стек, но разделяет все остальное адресное пространство (а также все глобальные переменные) со всеми другими потоками. (В отличие от потоков разные процессы не разделяют общего адресного пространства.) Потоки работают независимо друг от друга, иногда на разных процессорах. В одних системах операционная система располагает информацией обо всех потоках и осуществляет планирование потоков; в других системах каждый пользовательский процесс сам выполняет планирование потоков и управляет потоками, а операционной системе об этом неизвестно.

Наконец, параллелизм на уровне еще более крупных структурных единиц — несколько независимых процессов, которые вместе работают над решением одной задачи. В отличие от потоков независимые процессы не разделяют общее адресное пространство, поэтому задача должна быть разделена на довольно большие куски, по одному на каждый процесс.

Вычислительные парадигмы

В большинстве параллельных программ, особенно в тех, которые содержат большое число потоков или независимых процессов, используется некоторая парадигма для структуризации их работы. Существует множество таких парадигм. В этом разделе мы упомянем только несколько самых популярных.

Первая парадигма — **SPMD (Single Program Multiple Data — одна программа, несколько потоков данных).** Хотя система состоит из нескольких независимых процессов, они все выполняют одну и ту же программу, но над разными наборами данных. Только сейчас, в отличие от примера с температурой, все процессы выполняют одни и те же вычисления, но каждый в своем пространстве.

Вторая парадигма — **конвейер** с тремя процессами (рис. 8.11, а). Данные поступают в первый процесс, который трансформирует их и передает второму процессу для чтения и т. д. Если поток данных длинный (например, если это видеоизображение), все процессоры могут быть заняты одновременно. Так работают конвейеры в системе UNIX. Они могут работать как отдельные процессы параллельно в мультикомпьютере или мультипроцессоре.

Следующая парадигма — фазированное вычисление (рис. 8.11, б), когда работа разделяется на фазы, например, повторения цикла. Во время каждой фазы несколько процессов работают параллельно, но если один из процессов закончит

свою работу, он должен ждать до тех пор, пока все остальные процессы не завершат свою работу, и только после этого начинается следующая фаза. Четвертая парадигма — **«разделяй и властвуй»,** изображенная на рис. 8.11, б, в которой один процесс запускается, а затем порождает другие процессы, которым он может передать часть работы. Можно провести аналогию с генеральным подрядчиком, который получает приказ, затем поручает значительную часть работы каменщикам, электрикам, водопроводчикам, малярам и другим подчиненным. Каждый из них, в свою очередь, может поручить часть своей работы другим рабочим.

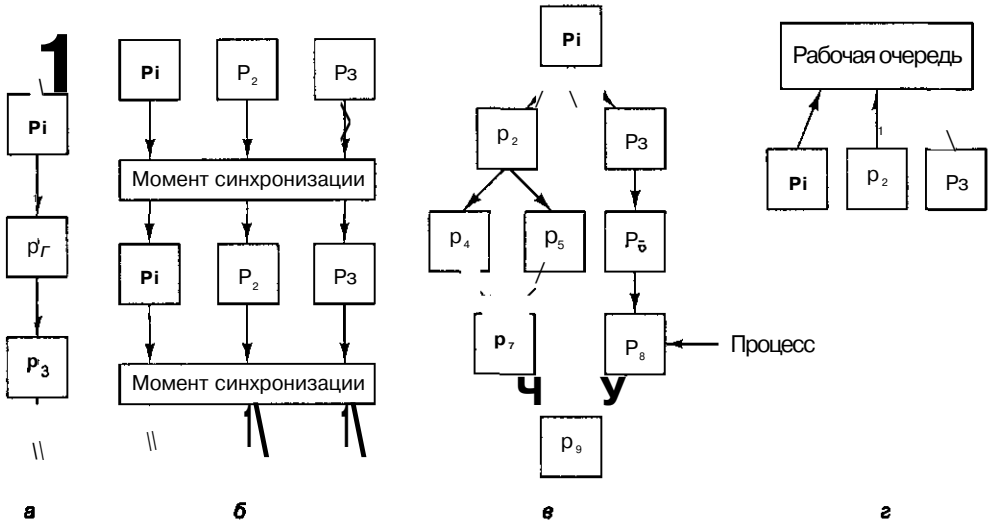


Рис. 8.11. Вычислительные парадигмы: конвейер (а); фазированное вычисление (б); «разделяй и властвуй» (в); replicated worker (г)

Последний пример — парадигма replicated worker (см. рис. 8.11, *г).* Здесь существует центральная очередь, и рабочие процессы получают задачи из этой очереди и выполняют их. Если задача порождает новые задачи, они добавляются к центральной очереди. Каждый раз, когда рабочий процесс завершает выполнение текущей задачи, он получает из очереди следующую задачу.

Методы коммуникации

Если программа разделена на части, скажем, на процессы, которые работают параллельно, эти части (процессы) должны каким-то образом взаимодействовать друг с другом. Такое взаимодействие можно осуществить одним из двух способов: с помощью общих переменных и с помощью передачи сообщений. В первом случае все процессы имеют доступ к общей логической памяти и взаимодействуют, считывая и записывая информацию в эту память. Например, один процесс может установить переменную, а другой процесс может прочитать ее.

В мультипроцессоре переменные могут разделяться между несколькими процессами с помощью отображения одной и той же страницы в адресное пространство каждого процесса. Затем общие переменные можно считывать и записывать с помощью обычных машинных команд LOAD и STORE. Даже в мультикомпьютере

без разделенной физической памяти возможно логическое разделение переменных, но сделать это несколько сложнее. Как мы видели выше, в системе Linda существует общее пространство кортежей, даже на мультикомпьютере, а в системе Orca поддерживаются разделенные объекты, общие с другими машинами. Существует и возможность разделения одного адресного пространства на мультикомпьютере, а также разбиение на страницы в сети. Процессы могут взаимодействовать между собой через общие переменные как в мультипроцессорах, так и в мультикомпьютерах.

Альтернативный подход — взаимодействие через передачу сообщений. В данной модели используются примитивы send и receive. Один процесс выполняет примитив send, называя другой процесс в качестве пункта назначения. Как только второй процесс совершает receive, сообщение копируется в адресное пространство получателя.

Существует множество вариантов передачи сообщений, но все они сводятся к применению двух примитивов send и receive. Они обычно реализуются как системные вызовы. Более подробно этот процесс мы рассмотрим ниже в этой главе.

Следующий вопрос при передаче сообщений — количество получателей. Самый простой случай — один отправитель и один получатель **(двухточечная передача сообщений).** Однако в некоторых случаях требуется отправить сообщение всем процессам **(широковещание)** или определенному набору процессов **(мультивещание).**

Отметим, что в мультипроцессоре передачу сообщений легко реализовать путем простого копирования от отправителя к получателю. Таким образом, возможности физической памяти совместного использования (мультипроцессор/мультикомпьютер) и логической памяти совместного использования (взаимодействие через общие переменные/передача сообщений) никак не связаны между собой. Все четыре комбинации имеют смысл и могут быть реализованы. Они приведены в табл. 8.1.

Таблица 8.1. Комбинации совместного использования физической и логической памяти

Физическая память (аппаратное обеспечение)	Логическая память (программное обеспечение)	Примеры
Мультипроцессор	Разделяемые переменные	Обработка изображения (см. рис. 8.1).
Мультипроцессор	Передача сообщений	Передача сообщений с использованием буферов памяти
Мультикомпьютер	Разделяемые переменные	DSM, Linda, Orca и т. д. на SP/2 или сети персонального компьютера
Мультикомпьютер	Передача сообщений	PVM или MPI на SP/2 или сети персонального компьютера

Базисные элементы синхронизации

Параллельные процессы должны не только взаимодействовать, но и синхронизировать свои действия. Если процессы используют общие переменные, нужно быть уверенным, что пока один процесс записывает что-либо в общую структуру данных, никакой другой процесс не считывает эту структуру. Другими словами, требуется некоторая форма **взаимного исключения,** чтобы несколько процессов не могли использовать одни и те же данные одновременно.

Существуют различные базисные элементы, которые можно использовать для взаимного исключения. Это семафоры, блокировки, мьютексы и критические секции. Все они позволяют процессу использовать какой-то ресурс (общую переменную, устройство ввода-вывода и т. п.), и при этом никакие другие процессы доступа к этому ресурсу не имеют. Если получено разрешение на доступ, процесс может использовать этот ресурс. Если второй процесс запрашивает разрешение, а первый все еще использует этот ресурс, доступ будет запрещен до тех пор, пока первый процесс не освободит ресурс.

Во многих параллельных программах существует и такой вид примитивов (базисных элементов), которые блокируют все процессы до тех пор, пока определенная фаза работы не завершится (см. рис. 8.11, *б)*. Наиболее распространенным примитивом подобного рода является барьер. Когда процесс встречает барьер, он блокируется до тех пор, пока все процессы не наткнутся на барьер. Когда последний процесс встречает барьер, все процессы одновременно освобождаются и продолжают работу.

Классификация компьютеров параллельного действия

Многое можно сказать о программном обеспечении для параллельных компьютеров, но сейчас мы должны вернуться к основной теме данной главы — архитектуре компьютеров параллельного действия. Было предложено и построено множество различных видов параллельных компьютеров, поэтому хотелось бы узнать, можно ли их как-либо категоризировать. Это пытались сделать многие исследователи [39, 43, 148]. К сожалению, хорошей классификации компьютеров параллельного действия до сих пор не существует. Чаще всего используется классификация Флинна (Flynn), но даже она является очень грубым приближением. Классификация приведена в табл. 8.2.

Таблица 8.2. Классификация компьютеров параллельного действия, разработанная Флинном

Потоки команд	Потоки данных	Названия	Примеры
1	1	SISD	Классическая машина фон Неймана
1	Много	S1MD	Векторный суперкомпьютер, массивно-параллельный процессор
Много	1	MISD	Не существует
Много	Много	MIMD	Мультипроцессор, мультикомпьютер

В основе классификации лежат два понятия: потоки команд и потоки данных. Поток команд соответствует счетчику команд. Система с n процессорами имеет n счетчиков команд и, следовательно, n потоков команд.

Поток данных состоит из набора операндов. В примере с вычислением температуры, приведенном выше, было несколько потоков данных, один для каждого датчика.

Потоки команд и данных в какой-то степени независимы, поэтому существует 4 комбинации (см. табл. 8.2). SISD (Single Instruction stream Single Data stream —

один поток команд, один поток данных) — это классический последовательный компьютер фон Неймана. Он содержит один поток команд и один поток данных и может выполнять только одно действие одномоментно. Машины SIMD (Single Instruction stream Multiple Data stream — один поток команд, несколько потоков данных) содержат один блок управления, выдающий по одной команде, но при этом есть несколько АЛУ, которые могут обрабатывать несколько наборов данных одновременно. ILLIAC IV (см. рис. 2.6) — прототип машин SIMD. Существуют и современные машины SIMD. Они применяются для научных вычислений.

Машины MISD (Multiple Instruction stream Single Data stream — несколько потоков команд, один поток данных) — несколько странная категория. Здесь несколько команд оперируют одним набором данных. Трудно сказать, существуют ли такие машины. Однако некоторые считают машинами MISD машины с конвейерами.

Последняя категория — машины MIMD (Multiple Instruction stream Multiple Data stream — несколько потоков команд, несколько потоков данных). Здесь несколько независимых процессоров работают как часть большой системы. В эту категорию попадает большинство параллельных процессоров. И мультипроцессоры, и мультикомпьютеры — это машины MIMD.

Мы расширили классификацию Флинна (схема на рис. 8.12). Машины SIMD распались на две подгруппы. В первую подгруппу попадают многочисленные суперкомпьютеры и другие машины, которые оперируют векторами, выполняя одну и ту же операцию над каждым элементом вектора. Во вторую подгруппу попадают машины типа ILLIAC IV, в которых главный блок управления посылает команды нескольким независимым АЛУ.

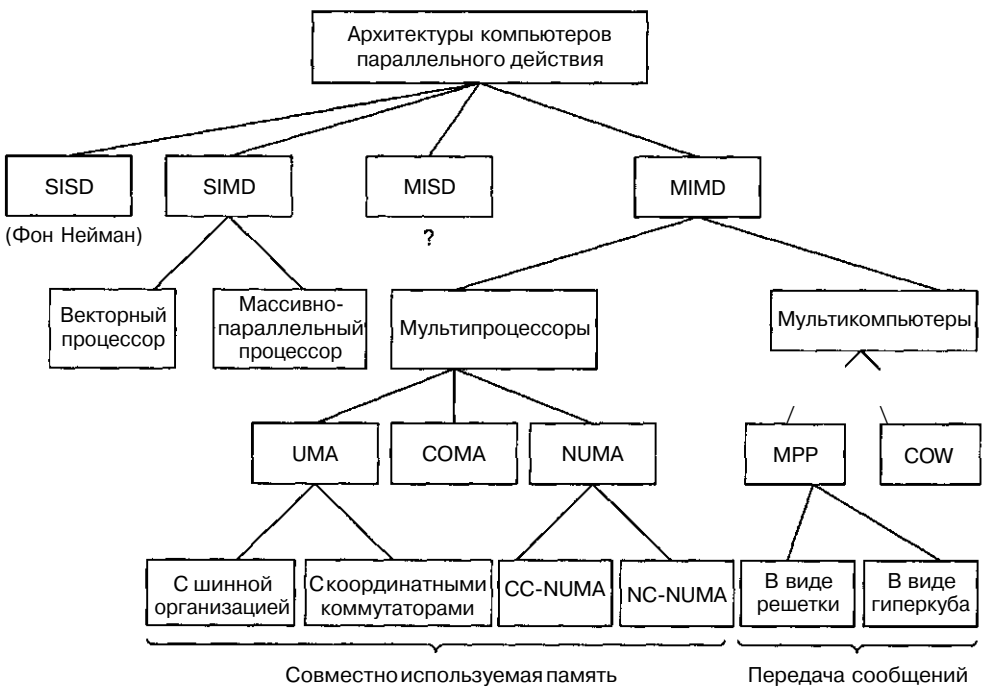


Рис. 8.12. Классификация компьютеров параллельного действия

В нашей классификации категория MIMD распалась на мультипроцессоры (машины с памятью совместного использования) и мультикомпьютеры (машины с передачей сообщений). Существует три типа мультипроцессоров. Они отличаются друг от друга по способу реализации памяти совместного использования. Они называются **UMA (Uniform Memory Access — архитектура с однородным доступом к памяти), NUMA (NonUniform Memory Access — архитектура с неоднородным доступом к памяти)** и **COMA (Cache Only Memory Access — архитектура с доступом только к кэш-памяти).** В машинах UMA каждый процессор имеет одно и то же время доступа к любому модулю памяти. Иными словами, каждое слово памяти можно считать с той же скоростью, что и любое другое слово памяти. Если это технически невозможно, самые быстрые обращения замедляются, чтобы соответствовать самым медленным, поэтому программисты не увидят никакой разницы. Это и значит «однородный». Такая однородность делает производительность предсказуемой, а этот фактор очень важен для написания эффективной программы.

Мультипроцессор NUMA, напротив, не обладает этим свойством. Обычно есть такой модуль памяти, который расположен близко к каждому процессору, и доступ к этому модулю памяти происходит гораздо быстрее, чем к другим. С точки зрения производительности очень важно, куда помещаются программа и данные. Машины COMA тоже с неоднородным доступом, но по другой причине. Подробнее каждый из этих трех типов мы рассмотрим позднее, когда будем изучать соответствующие подкатегории.

Во вторую подкатегорию машин MIMD попадают мультикомпьютеры, которые в отличие от мультипроцессоров не имеют памяти совместного использования на архитектурном уровне. Другими словами, операционная система в процессоре мультикомпьютера не может получить доступ к памяти, относящейся к другому процессору, просто путем выполнения команды LOAD. Ему приходится отправлять сообщение и ждать ответа. Именно способность операционной системы считывать слово из отдаленного модуля памяти с помощью команды LOAD отличает мультипроцессоры от мультикомпьютеров. Как мы уже говорили, даже в мультикомпьютере пользовательские программы могут обращаться к другим модулям памяти с помощью команд LOAD и STORE, но эту иллюзию создает операционная система, а не аппаратное обеспечение. Разница незначительна, но очень важна. Так как мультикомпьютеры не имеют прямого доступа к отдаленным модулям памяти, они иногда называются машинами **NORMA (NO Remote Memory Access — без доступа к отдаленным модулям памяти).**

Мультикомпьютеры можно разделить на две категории. Первая категория содержит процессоры **MPP (Massively Parallel Processors — процессоры с массовым параллелизмом)** — дорогостоящие суперкомпьютеры, которые состоят из большого количества процессоров, связанных высокоскоростной коммуникационной сетью. В качестве примеров можно назвать Cray T3E и IBM SP/2.

Вторая категория мультикомпьютеров включает рабочие станции, которые связываются с помощью уже имеющейся технологии соединения. Эти примитивные машины называются **NOW (Network of Workstations — сеть рабочих станций)** и **COW (Cluster of Workstattions — кластер рабочих станций).**

В следующих разделах мы подробно рассмотрим машины SIMD, мультипроцессоры MIMD и мультикомпьютеры MIMD. Цель — показать, что собой представляет каждый из этих типов, что собой представляют подкатегории и каковы ключевые принципы разработки. В качестве иллюстраций мы рассмотрим несколько конкретных примеров.

Компьютеры SIMD

Компьютеры SIMD (Single Instruction Stream Multiple Data Stream — один поток команд, несколько потоков данных) используются для решения научных и технических задач с векторами и массивами. Такая машина содержит один блок управления, который выполняет команды по одной, но каждая команда оперирует несколькими элементами данных. Два основных типа компьютеров SIMD — это массивно-параллельные процессоры (array processors) и векторные процессоры (vector processors). Рассмотрим каждый из этих типов по отдельности.

Массивно-параллельные процессоры

Идея массивно-параллельных процессоров была впервые предложена более 40 лет назад [151]. Однако прошло еще около 10 лет, прежде чем такой процессор (ILLIAC IV) был построен для NASA. С тех пор другие компании создали несколько коммерческих массивно-параллельных процессоров, в том числе CM-2 и Maspar MP-2, но ни один из них не пользовался популярностью на компьютерном рынке.

В **массивно-параллельном процессоре** содержится один блок управления, который передает сигналы, чтобы запустить несколько обрабатывающих элементов, как показано на рис. 2.6. Каждый обрабатывающий элемент состоит из процессора или усовершенствованного АЛУ и, как правило, локальной памяти.

Хотя все массивно-параллельные процессоры соответствуют этой общей модели, они могут отличаться друг от друга в некоторых моментах. Первый вопрос — это структура обрабатывающего элемента. Она может быть различной — от чрезвычайно простой до чрезвычайно сложной. Самые простые обрабатывающие элементы — 1-битные АЛУ (как в CM-2). В такой машине каждый АЛУ получает два 1-битных операнда из своей локальной памяти плюс бит из слова состояния программы (например, бит переноса). Результат операции — 1 бит данных и несколько флаговых битов. Чтобы совершить сложение двух целых 32-битных чисел, блоку управления нужно транслировать команду 1-битного сложения 32 раза. Если на одну команду затрачивается 600 не, то для сложения целых чисел потребуется 19,2 мкс, то есть получается медленнее, чем в первоначальной IBM PC. Но при наличии 65 536 обрабатывающих элементов можно получить более трех миллиардов сложений в секунду при времени сложения 300 пикосекунд.

Обрабатывающим элементом может быть 8-битное АЛУ, 32-битное АЛУ или более мощное устройство, способное выполнять операции с плавающей точкой. В какой-то степени выбор типа обрабатывающего элемента зависит от типа целей машины. Операции с плавающей точкой могут потребоваться для сложных мате-

матических расчетов (хотя при этом существенно сократится число обрабатывающих элементов), но для информационного поиска они не нужны.

Второй вопрос — как связываются обрабатывающие элементы друг с другом. Здесь применимы практически все топологии, приведенные на рис. 8.4. Чаще всего используются прямоугольные решетки, поскольку они подходят для задач с матрицами и отображениями и хорошо применимы к большим размерам, так как с добавлением новых процессоров автоматически увеличивается пропускная способность.

Третий вопрос — какую локальную автономию имеют обрабатывающие элементы. Блок управления сообщает, какую команду нужно выполнить, но во многих массивно-параллельных процессорах каждый обрабатывающий элемент может выбирать на основе некоторых локальных данных (например, на основе битов кода условия), выполнять ему эту команду или нет. Эта особенность придает процессору значительную гибкость.

Векторные процессоры

Второй тип машины SIMD — **векторный процессор.** Он более популярен на рынке. Машины, разработанные Сеймуром Креем (Seymour Cray) для Cray Research (сейчас это часть Silicon Graphics), — Cray-1 в 1976 году, а затем C90 и T90 доминировали в научной сфере на протяжении десятилетий. В этом разделе мы рассмотрим основные принципы, которые используются при создании таких высокопроизводительных компьютеров.

Типичное приложение для быстрой переработки больших объемов цифровых данных полно таких выражений, как:

```
for(i=0: i<n: i++) a[i]-b[i]+c[i];
```

где a, b и c — это **векторы**[1] (массивы чисел), обычно с плавающей точкой. Цикл приказывает компьютеру сложить i-е элементы векторов b и c, и сохранить результат в i-м элементе массива a. Программа определяет, что элементы должны складываться последовательно, но обычно порядок не играет никакой роли.

На рис. 8.13 изображено векторное АЛУ. Такая машина на входе получает два n-элементных вектора и обрабатывает соответствующие элементы параллельно, используя векторное АЛУ, которое может оперировать n элементами одновременно. В результате получается вектор. Входные и выходные векторы могут сохраняться в памяти или в специальных векторных регистрах.

Векторные компьютеры применяются и для скалярных (невекторных) операций, а также для смешанных векторно-скалярных операций. Основные типы векторных операций приведены в табл. 8.3. Первая из них, *U,* выполняет ту или иную операцию (например, квадратный корень или косинус) над каждым элементом одного вектора. Вторая, f_2 на входе получает вектор, а на выходе выдает скалярное значение. Типичный пример — суммирование всех элементов. Третья, f3, выполняет бинарную операцию над двумя векторами, например сложение соответствую-

[1] Строго говоря, использование термина «вектор» в данном контексте неправомочно, хотя уже много лет говорят именно «вектор». Дело в том, что вектор, в отличие от одномерной матрицы, имеет метрику, тогда как одномерный массив представляет собой просто определенным образом упорядоченный набор значений, характеризующих некоторый объект. — *Примеч. научн. ред.*

щих элементов. Наконец, четвертая, *U,* соединяет скалярный операнд с векторным. Типичный пример — умножение каждого элемента вектора на константу. Иногда быстрее переделать скалярный операнд в вектор, каждое значение которого равно скалярному операнду, а затем выполнить операцию над двумя векторами.

Все обычные операции с векторами могут производиться с использованием этих четырех форм. Например, чтобы получить скалярное произведение двух векторов, нужно сначала перемножить соответствующие элементы векторов (f_3), а затем сложить полученные результаты (f_2).

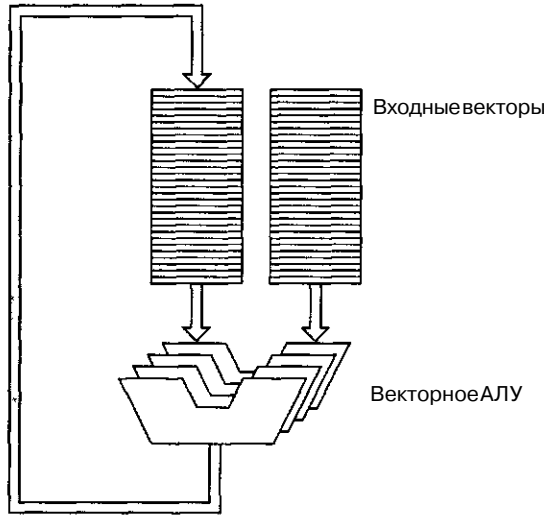


Рис. 8.13. Векторное АЛУ

Таблица 8.3. Комбинации векторных и скалярных операций

Операция	Примеры
A=f,(B,)	f1 _ косинус, квадратный корень
Скаляр=f2(A)	f2 — сумма, минимум
A,=f_3(B,, C)	f3 — сложение, вычитание
A^$_4$(скаляр, B,)	f_4_ умножение B, на константу

На практике суперкомпьютеры редко строятся по схеме, изображенной на рис. 8.13. Причина не техническая — такую машину вполне можно построить, а экономическая. Создание компьютера с 64 высокоскоростными АЛУ слишком дорого обойдется.

Обычно векторные процессоры сочетаются с конвейерами. Операции с плавающей точкой достаточно сложны. Они требуют выполнения нескольких шагов, а для выполнения любой многошаговой операции лучше использовать конвейер. Если вы не знакомы с арифметикой с плавающей точкой, обратитесь к приложению Б.

Рассмотрим табл. 8.4. В данном примере нормализованное число имеет мантиссу больше 1, но меньше 10 или равную 1. Здесь требуется вычесть 9,212x10" из 1,082xЮ^{12}.

Таблица 8.4. Вычитание чисел с плавающей точкой

Номер шага	Название шага	Значения
1	Вызов операндов	$1{,}082 \times 10^{12}$-$9{,}212 \times 10$"
2	Выравнивание экспоненты	$1{,}082 \times Ю^{12}$-$0{,}9212 \times Ю^{12}$
3	Вычитание	$0{,}1608 \times 10^{12}$
4	Нормализация результата	$1{,}608 \times 10$"

Чтобы из одного числа с плавающей точкой вычесть другое число с плавающей точкой, сначала нужно подогнать их таким образом, чтобы их экспоненты имели одно и то же значение. В нашем примере мы можем либо преобразовать уменьшаемое (число, из которого производится вычитание) в 10,82x10", либо преобразовать вычитаемое (число, которое вычитается) в $0{,}9212 \times 10^{12}$. При любом преобразовании мы рискуем. Увеличение экспоненты может привести к антипереполнению (исчезновению значащих разрядов) мантиссы, а уменьшение экспоненты может вызвать переполнение мантиссы. Антипереполнение менее опасно, поскольку число с антипереполнением можно округлить нулем. Поэтому мы выбираем первый путь. Приведя обе экспоненты к 12, мы получаем значения, которые показаны в табл. 8.4 на шаге 2. Затем мы выполняем вычитание, а потом нормализуем результат.

Конвейеризацию можно применять к циклу for, приведенному в начале раздела. В табл. 8.5 показана работа конвейеризированного сумматора с плавающей точкой. В каждом цикле на первой стадии вызывается пара операндов. На второй стадии меньшая экспонента подгоняется таким образом, чтобы соответствовать большей. На третьей стадии выполняется операция, а на четвертой стадии нормализуется результат. Таким образом, в каждом цикле из конвейера выходит один результат.

Таблица 8.5. Работа конвейеризированного сумматора с плавающей точкой

	Цикл						
Шаг	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Вызов операндов	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6	B_7, C_7
Выравнивание экспоненты		B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_6	B_6, C_6
Вычитание			Bi+Ci	B_2+C_2	B_3+C_3	B_4+C_4	B_6+C_5
Нормализация результата				B_1+Ci	B_2+C_2	B_3+C_3	B_4+C_4

Существенное различие между использованием конвейера для операций над векторами и использованием его для выполнения обычных команд — отсутствие скачков при работе с векторами. Каждый цикл используется полностью, и никаких пустых циклов нет.

Векторный суперкомпьютер Cray-1

Суперкомпьютеры обычно содержат несколько АЛУ, каждое из которых предназначено для выполнения определенной операции, и все они могут работать параллельно. В качестве примера рассмотрим суперкомпьютер Cray-1. Он больше не используется, но зато имеет простую архитектуру типа RISC, поэтому его очень удобно применять в учебных целях, и к тому же его архитектуру можно встретить во многих современных векторных суперкомпьютерах.

Машина Cray-1 регистровая (что типично для машин типа RISC), большинство команд 16-битные, состоят из 7-битного кода операции и трех 3-битных номеров регистров для трех операндов. Имеется пять типов регистров (рис. 8.14). Восемь 24-битных регистров A используются для обращения к памяти. 64 24-битных регистра B используются для хранения регистров A, когда они не нужны, чтобы не записывать их обратно в память. Восемь 64-битных регистров S предназначены для хранения скалярных величин (целых чисел и чисел с плавающей точкой). Значения этих регистров можно использовать в качестве операндов как для операций с целыми числами, так и для операций над числами с плавающей точкой. 64 64-битных регистра T — это регистры для хранения регистров S, опять-таки для сокращения количество операций LOAD и STORE.

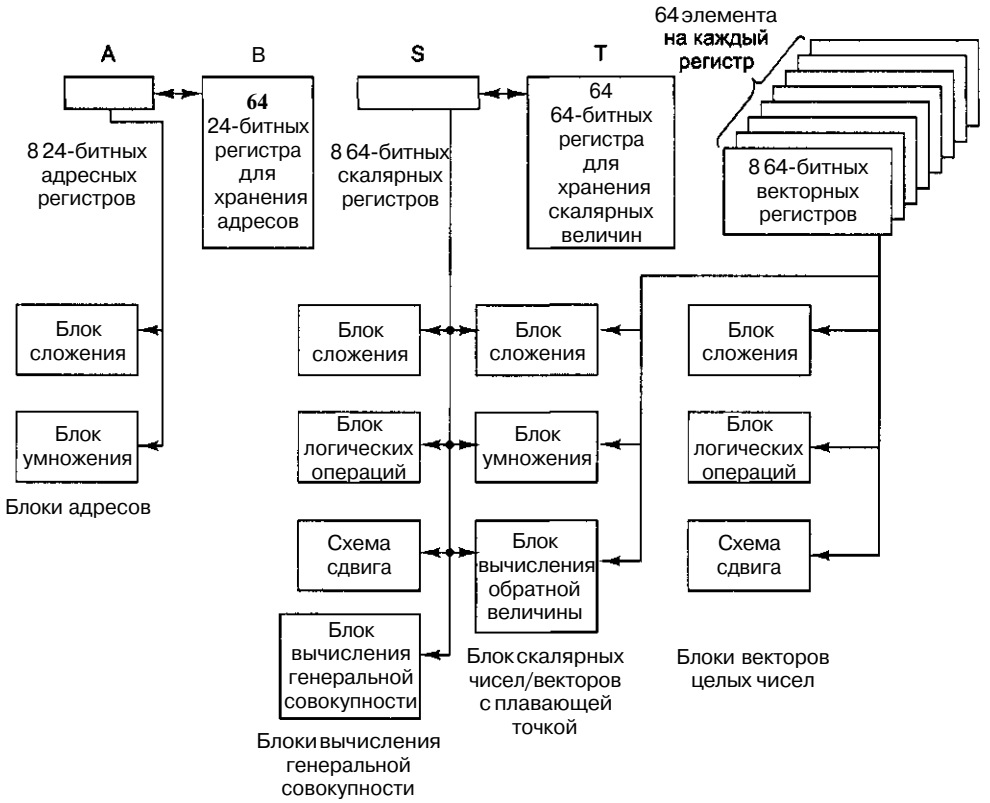


Рис. 8.14. Регистры и функциональные блоки машины Cray-1

Самый интересный набор регистров — это группа из восьми векторных регистров. Каждый такой регистр может содержать 64-элементный вектор с плавающей точкой. В одной 16-битной команде можно сложить, вычесть или умножить два вектора. Операция деления невозможна, но можно вычислить обратную величину. Векторные регистры могут загружаться из памяти, сохраняться в память, но такие перемещения выполнять очень невыгодно, поэтому их следует свести к минимуму. Во всех векторных операциях используются регистровые операнды.

Не всегда в суперкомпьютерах операнды должны находиться в регистрах. Например, машина CDC Cyber 205 выполняла операции над векторами в памяти. Такой подход позволял работать с векторами произвольной длины, но это сильно снижало скорость работы машины.

Cray-1 содержит 12 различных функциональных блоков (см. рис. 8.14). Два из них предназначены для арифметических действий с 24-битными адресами. Четыре нужны для операций с 64-битными целыми числами. Cray-1 не имеет блока для целочисленного умножения (хотя есть блок для перемножения чисел с плавающей точкой). Оставшиеся шесть блоков работают над векторами. Все они конвейеризированы. Блоки сложения, умножения и вычисления обратной величины работают как над скалярными числами с плавающей точкой, так и над векторами.

Как и многие другие векторные компьютеры, Cray-1 допускает операции сцепления. Например, чтобы вычислить выражение

```
R1=R1*R2+R3
```

где R1, R2 и R3 — векторные регистры, машина выполнит векторное умножение элемент за элементом, сохранит результат где-нибудь, а затем выполнит векторное сложение. При сцеплении, как только первые элементы перемножены, произведение сразу направляется в сумматор вместе с первым элементом регистра R3. Сохранения промежуточного результата не требуется. Такая технология значительно улучшает производительность.

Интересно рассмотреть абсолютную производительность Cray-1. Тактовый генератор работает с частотой 80 МГц, а объем основной памяти составляет 8 Мбайт. В то время (в середине — конце 70-х) это был самый мощный компьютер в мире. Сегодня вряд ли кто-нибудь сможет купить компьютер с таким медленным тактовым генератором и такой маленькой памятью — их уже никто не производит. Это наблюдение показывает, как быстро развивается компьютерная промышленность.

Мультипроцессоры с памятью совместного использования

Как показано на рис. 8.12, системы MIMD можно разделить на мультипроцессоры и мультикомпьютеры. В этом разделе мы рассмотрим мультипроцессоры, а в следующем — мультикомпьютеры. Мультипроцессор — это компьютерная система, которая содержит несколько процессоров и одно адресное пространство, видимое для всех процессоров. Он запускает одну копию операционной системы с одним набором таблиц, в том числе таблицами, которые следят, какие страницы памяти заняты, а какие свободны. Когда процесс блокируется, его процессор сохраняет свое состояние в таблицах операционной системы, а затем просматривает эти таблицы для нахождения другого процесса, который нужно запустить. Именно наличие одного отображения и отличает мультипроцессор от мультикомпьютера.

Мультипроцессор, как и все компьютеры, должен содержать устройства ввода-вывода (диски, сетевые адаптеры и т. п.). В одних мультипроцессорных системах только определенные процессоры имеют доступ к устройствам ввода-вывода и, следовательно, имеют специальную функцию ввода-вывода. В других мультипро-

цессорных системах каждый процессор имеет доступ к любому устройству ввода-вывода. Если все процессоры имеют равный доступ ко всем модулям памяти и всем устройствам ввода-вывода и каждый процессор взаимозаменим с другими процессорами, то такая система называется **SMP (Symmetric Multiprocessor — симметричный мультипроцессор).** Ниже мы будем говорить именно о таком типе систем.

Семантика памяти

Несмотря на то, что во всех мультипроцессорах процессорам предоставляется отображение общего разделенного адресного пространства, часто наряду с этим имеется множество модулей памяти, каждый из которых содержит какую-либо часть физической памяти. Процессоры и модули памяти соединяются сложной коммуникационной сетью (мы это обсуждали в разделе «Сети межсоединений»). Несколько процессоров могут пытаться считать слово из памяти, а в это же время несколько других процессоров пытаются записать то же самое слово, и некоторые сообщения могут быть доставлены не в том порядке, в каком они были отправлены. Добавим к этой проблеме существование многочисленных копий некоторых блоков памяти (например, в кэш-памяти), и в результате мы придем к хаосу, если не принять определенные меры. В этом разделе мы увидим, что в действительности представляет собой память совместного использования, и посмотрим, как блоки памяти могут правильно реагировать при таких обстоятельствах.

Семантику памяти можно рассматривать как контракт между программным обеспечением и аппаратным обеспечением памяти [3]. Если программное обеспечение соглашается следовать определенным правилам, то память соглашается выдавать определенные результаты. Основная проблема здесь — каковы должны быть правила. Эти правила называются **моделями согласованности.** Было предложено и разработано множество таких правил.

Предположим, что процессор 0 записывает значение 1 в какое-то слово памяти, а немного позже процессор 1 записывает значение 2 в то же самое слово. Процессор 2 считывает это слово и получает значение 1. Должен ли владелец компьютера обратиться после этого в мастерскую? Это зависит от того, что обещано в контракте.

Строгая согласованность

Самая простая модель — **строгая согласованность.** В такой модели при любом считывании из адреса x всегда возвращается значение самой последней записи в x. Программистам очень нравится эта модель, но ее можно реализовать на практике только следующим образом: должен быть один модуль памяти, который просто обслуживает все запросы по мере поступления (первым поступил — первым обработан), без кэш-памяти и без дублирования данных. К несчастью, это очень сильно замедляет работу памяти.

Согласованность по последовательности

Следующая модель называется **согласованностью по последовательности [79].** Здесь при наличии нескольких запросов на чтение и запись аппаратное обеспечение определяет порядок всех запросов, но все процессоры наблюдают одну и ту же последовательность запросов.

Рассмотрим один пример. Предположим, что процессор 1 записывает значение 100 в слово x, а через 1 не процессор 2 записывает значение 200 в слово x. А теперь предположим, что через 1 не после начала второй записи (процесс записи может быть еще не закончен) два других процессора, 3 и 4, считывают слово x по два раза (рис. 8.15). Возможный порядок шести событий представлен в листингах, относящихся к этому рисунку. В первом из них процессор 3 получает значения (20Q, 200), и процессор 4 получает значения (200, 200). Во втором они получают (100, 200) и (200,200) соответственно. В третьем они получают (100,100) и (200,100) соответственно. Все эти варианты допустимы, как и некоторые другие, которые мы здесь не показали.

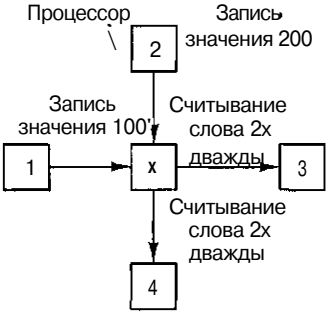


Рис. 8.15. Два процессора записывают, а другие два процессора считывают одно и то же слово из общей памяти

В листингах 8.1—8.3 представлены возможные варианты двух записей и четырех чтений.

Листинг 8.1. К рис. 8.15 (а)

```
W100
W200
R3=200
R3=200
R4=200
R4=200
```

Листинг 8.2. К рис. 8.15 (б)

```
W100
R3=100
W200
R4=200
R3=200
R4=200
```

Листинг 8.3. К рис. 8.15 (в)

```
W200
R4=200
W100
R3=100
R4=100
R3=100
```

Согласованная по последовательности память никогда не позволит процессору 3 получить значение (100, 200), если процессор 4 получил (200, 100). Если бы это произошло, с точки зрения процессора 3 это бы означало, что запись значения 100 процессором 1 завершилась раньше записи значения 200, которую осуществляет процессор 2. Это вполне возможно. Но с точки зрения процессора 4 это также значит, что запись процессором 2 числа 200 завершилась до записи процессором 1 числа 100. Сам по себе такой результат тоже возможен, но он противоречит первому результату. Согласованность по последовательности гарантирует единый глобальный порядок записей, который виден всем процессорам. Если процессор 3 видит, что первым было записано значение 100, то процессор 4 должен видеть тот же порядок.

Согласованность по последовательности очень полезна. Если несколько событий совершаются одновременно, существует определенный порядок, в котором эти события происходят (порядок может определяться случайно), но все процессоры наблюдают тот же самый порядок. Ниже мы рассмотрим модели согласованности, которые не гарантируют такого порядка.

Процессорная согласованность

Процессорная согласованность [48] — более проигрышная модель, но зато ее легче реализовать на больших мультипроцессорах. Она имеет два свойства:

1. Все процессоры воспринимают записи любого процессора в том порядке, в котором они начинаются.
2. Все процессоры видят записи в любое слово памяти в том же порядке, в котором они происходят.

Эти два пункта очень важны. В первом пункте говорится, что если процессор 1 начинает запись значений 1A, 1B и 1C в какое-либо место в памяти именно в таком порядке, то все другие процессоры видят эти записи в том же порядке. Иными словами, никогда не произойдет такого, чтобы какой-либо процессор сначала увидел значение 1B, а затем значение 1A. Второй пункт нужен, чтобы каждое слово в памяти имело определенное недвусмысленное значение после того, как процессор совершил несколько записей в это слово, а затем остановился. Все должны воспринимать последнее значение.

Даже при таких ограничениях у разработчика есть много возможностей. Посмотрим, что произойдет, если процессор 2 начинает записи 2A, 2B и 2C одновременно с тремя записями процессора 1. Другие процессоры, которые заняты считыванием слов из памяти, увидят какую-либо последовательность из шести записей, например, 1A, 1B, 2A, 2B, 1C, 2C или 2A, 1A, 2B, 2C, 1B, 1C и т. п. При процессорной согласованности не гарантируется, что каждый процессор видит один и тот же порядок (в отличие от согласованности по последовательности). Вполне может быть так, что одни процессоры воспринимают первый порядок из указанных выше, другие — второй порядок, а третьи — иной третий порядок. Единственное, что гарантируется абсолютно точно, — ни один процессор не увидит последовательность, в которой сначала идет 1B, а затем 1A.

Слабая согласованность

В следующей модели, слабой согласованности, записи, произведенные одним процессором, воспринимаются по порядку [33]. Один процессор может увидеть 1A до 1B, а другой — 1A после 1B. Чтобы внести порядок в этот хаос, в памяти содержатся переменные синхронизации либо операция синхронизации. Когда выполняется синхронизация, все незаконченные записи завершаются и ни одна новая запись не может начаться, пока не будут завершены все старые записи и не будет произведена синхронизация. Синхронизация приводит память в стабильное состояние, когда не остается никаких незавершенных операций. Сами операции синхронизации согласованы по последовательности, то есть если они вызываются несколькими процессорами, выбирается какой-то определенный порядок, причем все процессоры воспринимают один и тот же порядок.

При слабой согласованности время разделяется на последовательные периоды, разграниченные моментами синхронизации (рис. 8.16). Никакого определенного порядка для 1A и 1B не гарантируется, и разные процессоры могут воспринимать эти две записи в разном порядке, то есть один процессор может сначала видеть 1A, а затем 1B, а другой процессор может сначала видеть 1B, а затем 1A. Такая ситуация допустима. Однако все процессоры видят сначала 1B, а затем 1C, поскольку первая операция синхронизации требует, чтобы сначала завершились записи 1A, 1B и 2A, и только после этого начались записи 1C, 2B, 3A или 3B. Таким образом, с помощью операций синхронизации программное обеспечение может вносить порядок в последовательность событий, хотя это занимает некоторое время.

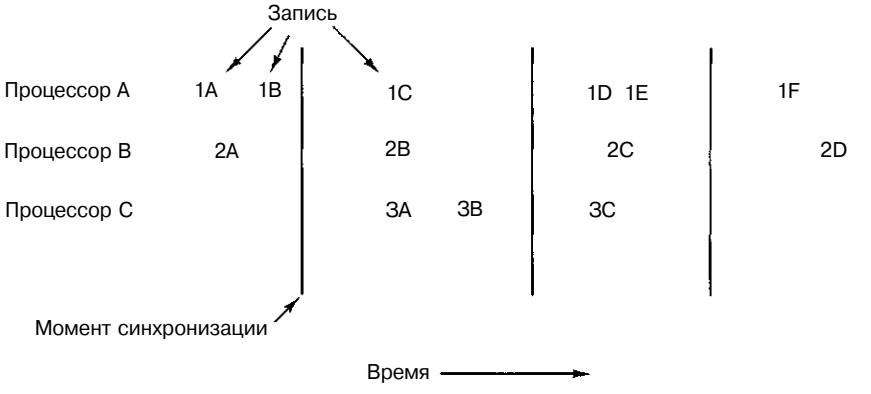


Рис. 8.16. Слабо согласованная память использует операции синхронизации, которые делят время на последовательные периоды

Свободная согласованность

Слабая согласованность — не очень эффективный метод, поскольку он требует завершения всех операций памяти и задерживает выполнение новых операций до тех пор, пока старые не будут завершены. При свободной согласованности дела обстоят гораздо лучше, поскольку здесь используется нечто похожее на критические секции программы [46]. Идея состоит в следующем. Если процесс выходит за пределы критической области, это не значит, что все записи должны немедленно

завершиться. Требуется только, чтобы все записи были завершены до того, как любой процесс снова войдет в эту критическую область.

В этой модели операция синхронизации разделяется на две разные операции. Чтобы считать или записать общую переменную, процессор (то есть его программное обеспечение) сначала должен выполнить операцию acquire над переменной синхронизации, чтобы получить монопольный доступ к общим разделяемым данным. Затем процессор может использовать эти данные по своему усмотрению (считывать или записывать их). Потом процессор выполняет операцию release над переменной синхронизации, чтобы показать, что он завершил работу. Операция release не требует завершения незаконченных записей, но сама она не может быть завершена, пока не закончатся все ранее начатые записи. Более того, новые операции памяти могут начинаться сразу же.

Когда начинается следующая операция acqui re, производится проверка, все ли предыдущие операции rel ease завершены. Если нет, то операция acqui re задерживается до тех пор, пока они все не будут сделаны (а все записи должны быть завершены перед тем, как завершатся все операции release). Таким образом, если следующая операция acquire появляется через достаточно длительный промежуток времени после последней операции rel ease, ей не нужно ждать, и она может войти в критическую область без задержки. Если операция acquire появляется через небольшой промежуток времени после операции release, эта операция acquire (и все команды, которые следуют за ней) будет задержана до завершения всех операций release. Это гарантирует, что все переменные в критической области будут обновлены. Такая схема немного сложнее, чем слабая согласованность, но она имеет существенное преимущество: здесь не нужно задерживать выполнение команд так часто, как в слабой согласованности.

Архитектуры UMA SMP с шинной организацией

В основе самых простых мультипроцессоров лежит одна шина, как показано на рис. 8.17, *а*. Два или более процессоров и один или несколько модулей памяти используют для взаимодействия одну и ту же шину. Если процессору нужно считать слово из памяти, он сначала проверяет, свободна ли шина. Если шина свободна, процессор помещает адрес нужного слова на шину, устанавливает несколько сигналов управления и ждет, когда память поместит на шину нужное слово.

Если шина занята, процессор просто ждет, когда она освободится. С этой разработкой связана одна проблема. При наличии двух или трех процессоров доступ к шине вполне управляем; при наличии 32 и 64 процессоров возникают трудности. Производительность системы будет полностью ограничиваться пропускной способностью шины, а большинство процессоров будут простаивать большую часть времени.

Чтобы разрешить эту проблему, нужно добавить кэш-память к каждому процессору, как показано на рис. 8.17, *б*. Кэш-память может находиться внутри микросхемы процессора, рядом с микросхемой процессора, на плате процессора. Допустимы комбинации этих вариантов. Поскольку теперь считывать слова можно из кэш-памяти, движения в шине будет меньше, и система сможет поддерживать большее количество процессоров.

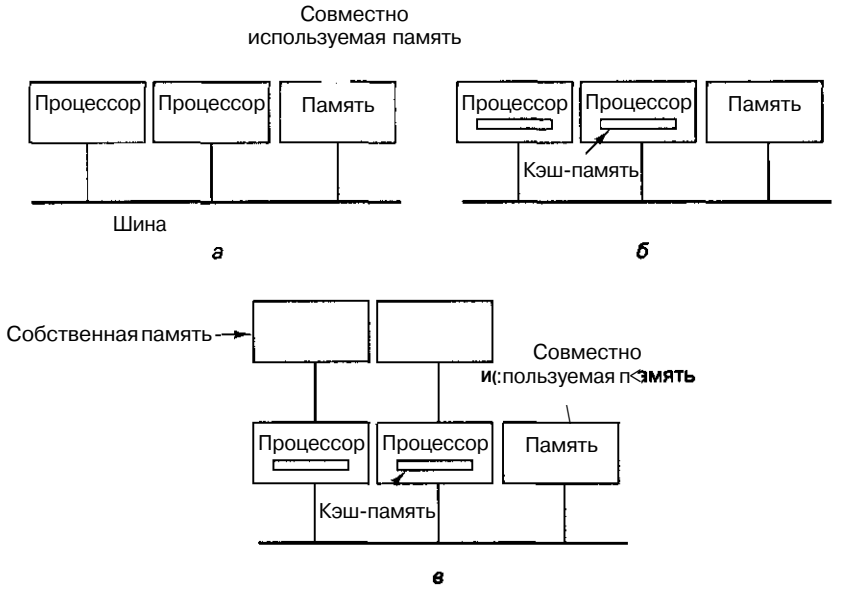


Рис. 8.17. Три мультипроцессора на одной шине: без кэш-памяти (а); с кэш-памятью (б); с кэш-памятью и отдельными блоками памяти (в)

Еще одна возможность — разработка, в которой каждый процессор имеет не только кэш-память, но и свою локальную память, к которой он получает доступ через назначенную локальную шину (рис. 8.17, *в*). Чтобы оптимально использовать такую конфигурацию, компилятор должен поместить в локальные модули памяти весь текст программы, цепочки, константы, другие данные, предназначенные только для чтения, стеки и локальные переменные. Общая разделенная память используется только для общих переменных. В большинстве случаев такое разумное размещение сильно сокращает количество данных, передаваемых по шине, и не требует активного вмешательства со стороны компилятора.

Отслеживание изменений данных в кэш-памяти

Предположим, что память согласована по последовательности. Что произойдет, если процессор 1 содержит в своей кэш-памяти строку, а процессор 2 пытается считать слово из той же строки кэш-памяти? При отсутствии специальных правил процессор 2 получит копию этой строки в свою кэш-память. В принципе помещение одной и той же строки в кэш-память дважды вполне приемлемо. А теперь предположим, что процессор 1 изменяет строку, и сразу после этого процессор 2 считывает копию этой строки из своей кэш-памяти. Он получит **устаревшие данные,** нарушая контракт между программным обеспечением и памятью. Ни к чему хорошему это не приведет.

Эта проблема, которую называют **непротиворечивостью кэшей,** очень важна. Если ее не разрешить, нельзя будет использовать кэш-память, и число мультипроцессоров, подсоединенных к одной шине, придется сократить до двух-трех. Специалистами было предложено множество различных решений (например, [47,109]). Хотя все эти алгоритмы, называемые **протоколами когерентности кэширования,**

различаются в некоторых деталях, все они не допускают одновременного появления разных вариантов одной и той же строки в разных блоках кэш-памяти.

Во всех решениях контроллер кэш-памяти разрабатывается так, чтобы кэш-память могла перехватывать запросы на шине, контролируя все запросы шины от других процессоров и других блоков кэш-памяти и предпринимая те или иные действия в определенных случаях. Эти устройства называются **кэш-памятью с отслеживанием (snooping caches** или **snoopy caches),** поскольку они отслеживают шину. Набор правил, которые выполняются кэш-памятью, процессорами и основной памятью, чтобы предотвратить появление различных вариантов данных в нескольких блоках кэш-памяти, формируют протокол когерентности кэширования. Единица передачи и хранения кэш-памяти называется строкой кэш-памяти. Обычно строка кэш-памяти равна 32 или 64 байтам.

Самый простой протокол когерентности кэширования называется **сквозным кэшированием.** Чтобы лучше понять его, рассмотрим 4 случая, приведенные в табл. 8.6. Если процессор пытается считать слово, которого нет в кэш-памяти, контроллер кэш-памяти загружает в кэш-память строку, содержащую это слово. Строку предоставляет основная память, которая в это: > протоколе всегда обновлена. В дальнейшем информация может считываться из кэш-памяти.

Таблица 8.6. Сквозное кэширование. Пустые графы означают, что никакого действия не происходит

Действие	Локальный запрос	Удаленный запрос
Промах при чтении	Вызов данных из памяти	
Попадание при чтении	Использование данных из локальной кэш-памяти	
Промах при записи	Обновление данных в памяти	
Попадание при записи	Обновление кэш-памяти и основной памяти	Объявление элемента кэш-памяти недействительным

В случае промаха кэш-памяти при записи слово, которое было изменено, записывается в основную память. Строка, содержащая нужное слово, не загружается в кэш-память. В случае результативного обращения к кэш-памяти при записи кэш обновляется, а слово плюс ко всему записывается в основную память. Суть протокола состоит в том, что в результате всех операций записи записываемое слово обязательно проходит через основную память, чтобы информация в основной памяти всегда обновлялась.

Рассмотрим все эти действия снова, но теперь с точки зрения кэш-памяти с отслеживанием (крайняя правая колонка в табл. 8.6). Назовем кэш-память, которая выполняет действия, кэш-1, а кэш с отслеживанием — кэш-2. Если при считывании произошел промах кэша-1, он запрашивает шину, чтобы получить нужную строку из основной памяти. Кэш-2 видит это, но ничего не делает. Если нужная строка уже содержится в кэш-1, запроса шины не происходит, поэтому кэш-2 не знает о результативных считываниях из кэша-1.

Процесс записи более интересен. Если процессор 1 записывает слово, кэш-1 запрашивает шину как в случае промаха кэша, так и в случае попадания. Всегда при записи кэш-2 проверяет наличие у себя записываемого слова. Если данное слово

отсутствует, кэш-2 рассматривает это как промах отдаленной памяти и ничего не делает. (Отметим, что в табл. 8.6 промах отдаленной памяти означает, что слово не присутствует в кэш-памяти отслеживателя; не имеет значения, было ли это слово в кэш-памяти инициатора или нет. Таким образом, один и тот же запрос может быть результативным логически и промахом для отслеживателя и наоборот.)

А теперь предположим, что кэш-1 записывает слово, которое присутствует в кэш-2. Если кэш-2 не произведет никаких действий, он будет содержать устаревшие данные, поэтому элемент кэш-памяти, содержащий измененное слово, помечается как недействительный. Соответствующая единица просто удаляется из кэш-памяти. Все кэши отслеживают все запросы шины, и всякий раз, когда записывается слово, нужно обновить его в кэш-памяти инициатора запроса, обновить его в основной памяти и удалять его из всех других кэшей. Таким образом, неправильные варианты слова исключаются.

Процессор кэш-памяти-2 вправе прочитать то же самое слово на следующем цикле. В этом случае кэш-2 считает слово из основной памяти, которая уже обновилась. В этот момент кэш-1, кэш-2 и основная память содержат идентичные копии этого слова. Если какой-нибудь процессор произведет запись, то другие кэши будут очищены, а основная память опять обновится.

Возможны различные вариации этого основного протокола. Например, при успешной записи отслеживающий кэш обычно объявляет недействительным элемент, содержащий данное слово. С другой стороны, вместо того чтобы объявлять слово недействительным, можно принять новое значение и обновить кэш-память. По существу, обновить кэш-память — это то же самое, что признать слово недействительным, а затем считать нужное слово из основной памяти. Во всех кэш-протоколах нужно сделать выбор между **стратегией обновления** и **стратегией с признанием данных недействительными.** Эти протоколы работают по-разному. Сообщения об обновлении несут полезную нагрузку, и следовательно, они больше по размеру, чем сообщения о недействительности, но зато они могут предотвратить дальнейшие промахи кэш-памяти.

Другой вариант — загрузка отслеживающей кэш-памяти при промахах. Такая загрузка никак не влияет на правильность выполнения алгоритма. Она влияет только на производительность. Возникает вопрос: какова вероятность, что только что записанное слово вскоре будет записано снова? Если вероятность высока, то можно говорить в пользу загрузки кэш-памяти при промахах записи **(политика заполнения по записи).** Если вероятность мала, лучше не обновлять кэш-память в случае промаха при записи. Если данное слово скоро будет считываться, оно все равно будет загружено после промаха при считывании, и нет смысла загружать его в случае промаха при записи.

Как и большинство простых решений, это решение не очень эффективно. Каждая операция записи должна передаваться в основную память по шине, а при большом количестве процессоров это затруднительно. Поэтому были разработаны другие протоколы. Все они характеризуются одним общим свойством: не все записи проходят непосредственно через основную память. Вместо этого при изменении строки кэш-памяти внутри кэш-памяти устанавливается бит, который указывает, что строка в кэш-памяти правильная, а в основной памяти — нет. В конечном итоге эту строку нужно будет записать в основную память, но перед этим в память

можно произвести много записей. Такой тип протокола называется **протоколом с обратной записью.**

Протокол MESI

Один из популярных протоколов с обратной записью называется **MESI** (по первым буквам названий четырех состояний, M, E, S и I) [109]. В его основе лежит **протокол однократной записи** [47]. Протокол MESI используется в Pentium II и других процессорах для отслеживания шины. Каждый элемент кэш-памяти может находиться в одном из следующих четырех состояний:

1. Invalid — элемент кэш-памяти содержит недействительные данные.
2. Shared — несколько кэшей могут содержать данную строку; основная память обновлена.
3. Exclusive — никакой другой кэш не содержит эту строку; основная память обновлена.
4. Modified — элемент действителен; основная память недействительна; копий элемента не существует.

При загрузке процессора все элементы кэш-памяти помечаются как недействительные. При первом считывании из основной памяти нужная строка вызывается в кэш-память данного процессора и помечается как E (Exclusive), поскольку это единственная копия в кэш-памяти (рис. 8.18, *а).* При последующих считываниях процессор использует эту строку и не использует шину. Другой процессор может вызвать ту же строку и поместить ее в кэш-память, но при отслеживании исходный держатель строки (процессор 1) узнает, что он уже не единственный, и объявляет, что у него есть копия. Обе копии помечаются состоянием S (Shared) (см. рис. 8.18, *б).* При последующих чтениях кэшированных строк в состоянии S процессор не использует шину и не меняет состояние элемента.

Посмотрим, что произойдет, если процессор 2 произведет запись в строку кэш-памяти, находящуюся в состоянии S. Тогда процессор помещает сигнал о недействительности на шину, который сообщает всем другим процессорам, что нужно отбросить свои копии. Соответствующая строка переходит в состояние M (Modified) (см. рис. 8.18, в). Эта строка не записывается в основную память. Отметим, что если записываемая строка находится в состоянии E, никакого сигнала о недействительности на шину передавать не следует, поскольку известно, что других копий нет.

А теперь рассмотрим, что произойдет, если процессор 3 считывает эту строку. Процессор 2, который в данный момент содержит строку, знает, что копия в основной памяти недействительна, поэтому он передает на шину сигнал, чтобы процессор 3 подождал, пока он запишет строку обратно в память. Как только строка записана в основную память, процессор 3 вызывает из памяти копию этой строки, и в обоих кэшах строка помечается как S (см. рис. 8.18, *г).* Затем процессор 2 записывает эту строку снова, что делает недействительной копию в кэш-памяти процессора 3 (см. рис. 8.18, *в).*

Наконец, процессор 1 производит запись в слово в этой строке. Процессор 2 видит это и передает на шину сигнал, который сообщает процессору 1, что нужно подождать, пока строка не будет записана в основную память. Когда это действие закончится, процессор помечает собственную копию строки как недействительную, поскольку он знает, что другой процессор собирается изменить ее. Возникает

ситуация, в которой процессор записывает что-либо в некэшированную строку. Если применяется политика write-allocate, строка будет загружаться в кэш-память и помечаться как M (рис. 8.18, *е)*. Если политика write-allocate не применяется, запись будет производиться непосредственно в основную память, а строка в кэш-памяти сохранена не будет.

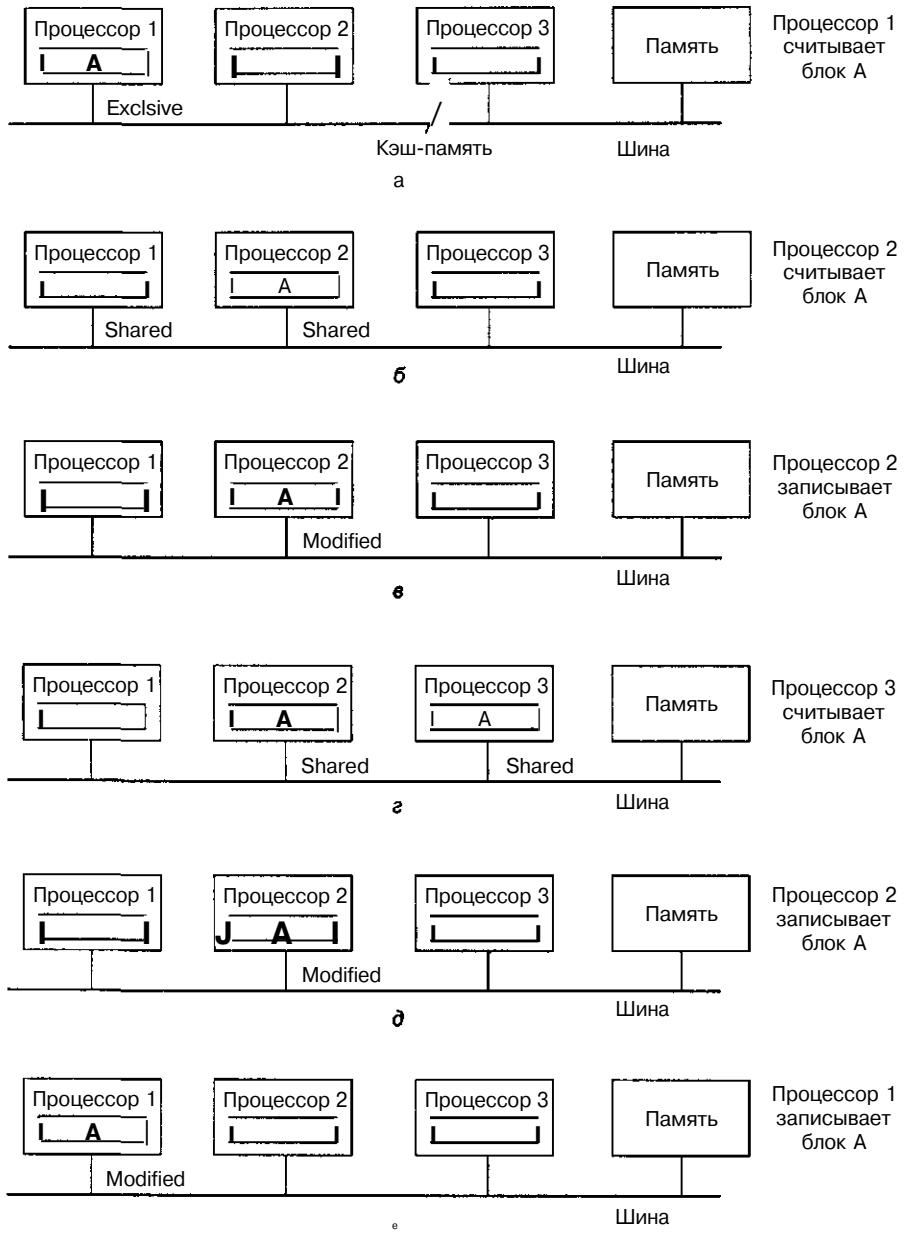


Рис. 8.18. Протокол MESI

Мультипроцессоры UMA с координатными коммутаторами

Даже при всех возможных оптимизациях использование только одной шины ограничивает размер мультипроцессора UMA до 16 или 32 процессоров. Чтобы получить больший размер, требуется другой тип коммуникационной сети. Самая простая схема соединения n процессоров с **к** блоками памяти — **координатный коммутатор** (рис. 8.19). Координатные коммутаторы используются на протяжении многих десятилетий для соединения группы входящих линий с рядом выходящих линий произвольным образом.

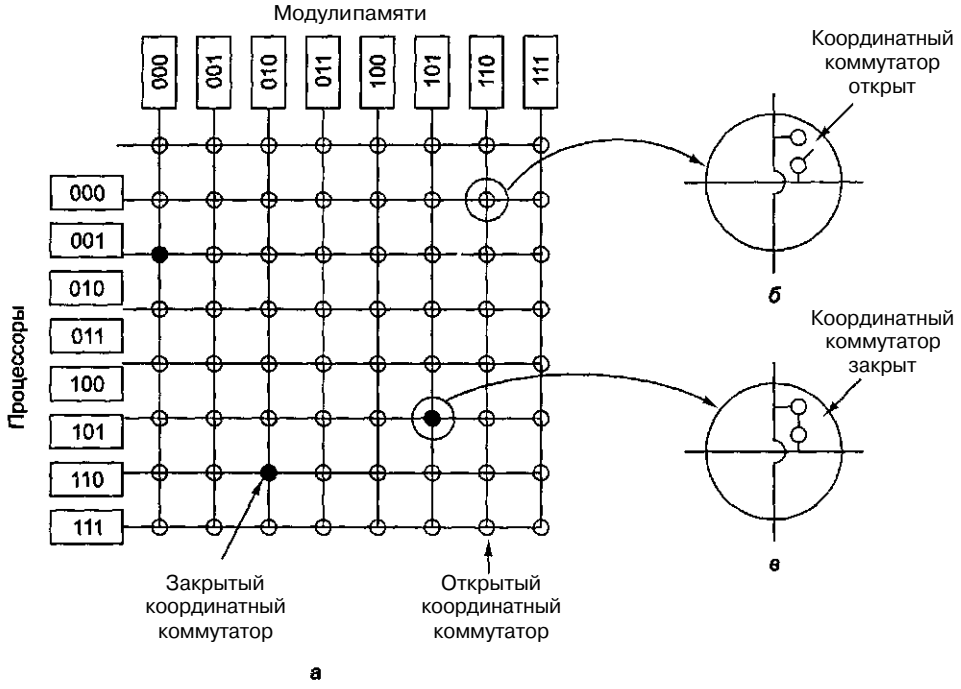


Рис. 8.19. Координатный коммутатор 8x8 (а); открытый узел (б); закрытый узел (в)

В каждом пересечении горизонтальной (входящей) и вертикальной (исходящей) линии находится **соединение** (crosspoint), которое можно открыть или закрыть в зависимости от того, нужно соединять горизонтальную и вертикальную линии или нет. На рис. 8.19, а мы видим, что три узла закрыты, благодаря чему устанавливается связь между парами (процессор, память) (001, 000), (101, 101) и (110, 010) одновременно. Возможны другие комбинации. Число комбинаций равно числу способов, которыми можно расставить 8 ладей на шахматной доске.

Координатный коммутатор представляет собой **неблокируемую сеть.** Это значит, что процессор всегда будет связан с нужным блоком памяти, даже если какая-то линия или узел уже заняты. Более того, никакого предварительного планирования не требуется. Даже если уже установлено семь произвольных связей, всегда

можно связать оставшийся процессор с оставшимся блоком памяти. Ниже мы рассмотрим схемы, которые не обладают такими свойствами.

Не лучшим свойством координатного коммутатора является то, что число узлов растет как n^2. При наличии 1000 процессоров и 1000 блоков памяти нам понадобится миллион узлов. Это неприемлемо. Тем не менее координатные коммутаторы вполне применимы для систем средних размеров.

Sun Enterprise 10000

В качестве примера мультипроцессора UMA, основанного на координатном коммутаторе, рассмотрим систему Sun Enterprise 10000 [23, 24]. Эта система состоит из одного корпуса с 64 процессорами. Координатный коммутатор **Gigaplahe-XB** запакован в плату, содержащую 8 гнезд на каждой стороне. Каждое гнездо вмещает огромную плату процессора (40x50 см), содержащую 4 процессора UltraSPARC на 333 МГц и ОЗУ на 4 Гбайт. Благодаря жестким требованиям к синхронизации и малому времени ожидания доступ к памяти вне платы занимает столько же времени, сколько доступ к памяти на плате.

Иметь только одну шину для взаимодействия всех процессоров и всех блоков памяти неудобно, поэтому в системе Enterprise 10000 применяется другая стратегия. Здесь есть координатный коммутатор 16x16 для перемещения данных между основной памятью и блоками кэш-памяти. Длина строки кэш-памяти составляет 64 байта, а ширина канала связи составляет 16 байтов, поэтому для перемещения строки кэш-памяти требуется 4 цикла. Координатный коммутатор работает от точки к точке, поэтому его нельзя использовать для сохранения совместимости по кэш-памяти.

По этой причине помимо координатного коммутатора имеются 4 адресные шины, которые используются для отслеживания строк в кэш-памяти (рис. 8.20). Каждая шина используется для 1/4 физического адресного пространства. Для выбора шины используется два адресных бита. В случае промаха кэш-памяти при считывании процессор должен считывать нужную ему информацию из основной памяти, и тогда он обращается к соответствующей адресной шине, чтобы узнать, нет ли нужной строки в других блоках кэш-памяти. Все 16 плат отслеживают все адресные шины одновременно, поэтому если ответа нет, это значит, что требуемая строка отсутствует в кэш-памяти и ее нужно вызывать из основной памяти.

Вызов из памяти происходит от точки к точке по координатному коммутатору по 16 байтов. Цикл шины составляет 12 не (83,3 МГц), и каждая адресная шина может отслеживаться в каждом цикле любой другой шины, то есть всего возможно 167 млн отслеживаний/с. Каждое отслеживание может потребовать передачи строки кэш-памяти в 64 байта, поэтому узел должен быть способен передавать 9,93 Гбайт/с (напомним, что 1 Гбайт= $1{,}0737 \times 10^9$ байт/с, а не 10^9 байт/с). Строку кэш-памяти в 64 байта можно передать через узел за 4 цикла шины (48 не) при пропускной способности 1,24 Гбайт/с за одну передачу. Поскольку узел может обрабатывать 16 передач одновременно, его максимальная пропускная способность составляет 19,87 Гбайт/с, а этого достаточно для поддержания скорости отслеживания, даже если принять во внимание конфликтную ситуацию, которая сокращает практическую пропускную способность примерно до 60% от теоретической.

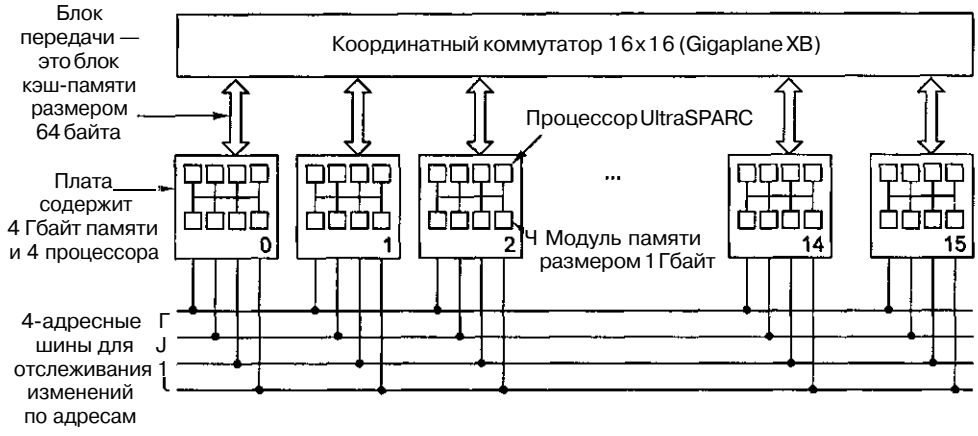


Рис. 8.20. Мультипроцессор Sun Enterprise 10000

Enterprise 10000 использует 4 отслеживающие шины параллельно, плюс очень широкий координатный коммутатор для передачи данных. Ясно, что такая система преодолевает предел в 64 процессора. Но чтобы существенно увеличить количество процессоров, требуется совсем другой подход.

Мультипроцессоры UMA с многоступенчатыми сетями

В основе «совсем другого подхода» лежит небольшой коммутатор 2x2 (рис. 8.21, *а)*. Этот коммутатор содержит два входа и два выхода. Сообщения, приходящие на любую из входных линий, могут переключаться на любую выходную линию. В нашем примере сообщения будут содержать до четырех частей (рис. 8.21, *б)*. Поле *Модуль* сообщает, какую память использовать. Поле *Адрес* определяет адрес в этом модуле памяти. В поле *Код операции* содержится операция, например READ или WRITE. Наконец, дополнительное поле *Значение* может содержать операнд, например 32-битное слово, которое нужно записать при выполнении операции WRITE. Коммутатор исследует поле *Модуль* и использует его для определения, через какую выходную линию нужно отправить сообщение: через X или через Y.

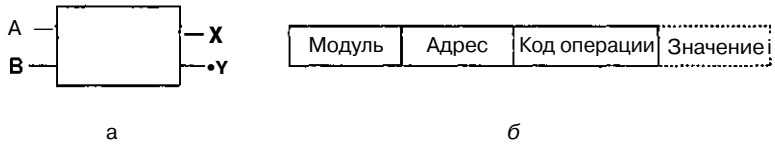


Рис. 8.21. Коммутатор 2x2 (а); формат сообщения (б)

Наши коммутаторы 2x2 можно компоновать различными способами и получать **многоступенчатые сети** [1, 15, 78]. Один из возможных вариантов — **сеть omega** (рис. 8.22). Здесь мы соединили 8 процессоров с 8 модулями памяти, используя 12 коммутаторов. Для n процессоров и n модулей памяти нам понадобится $\log_2 n$ ступеней, n/2 коммутаторов на каждую ступень, то есть всего (n/2)log2n

коммутаторов, что намного лучше, чем n^2 узлов (точек пересечения), особенно для больших n.

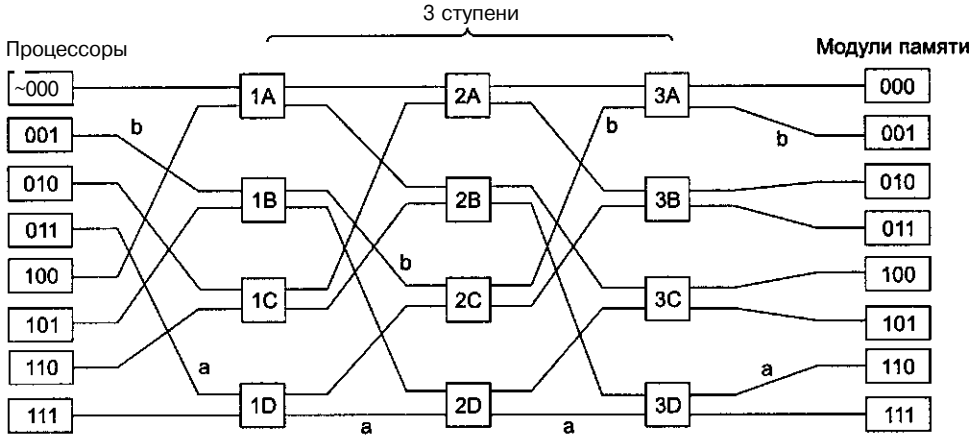


Рис. 8.22. Сеть omega

Рисунок разводки сети omega часто называют **полным тасованием,** поскольку смешение сигналов на каждой ступени напоминает колоду карт, которую разделили пополам, а затем снова соединили, чередуя карты. Чтобы понять, как работает сеть omega, предположим, что процессору 011 нужно считать слово из модуля памяти 110. Процессор посылает сообщение READ, чтобы переключить коммутатор ID, который содержит 110 в поле *Модуль.* Коммутатор берет первый (то есть крайний левый) бит от 110 и по нему узнает направление. 0 указывает на верхний выход, а 1 — на нижний. Поскольку в данном случае этот бит равен 1, сообщение отправляется через нижний выход в 2D.

Все коммутаторы второй ступени, включая 2D, для определения направления используют второй бит. В данном случае он равен 1, поэтому сообщение отправляется через нижний выход в 3D. Затем проверяется третий бит. Он равен 0. Следовательно, сообщение переходит в верхний выход и прибывает в память 110, чего мы и добивались. Путь, пройденный данным сообщением, обозначен на рис. 8.22 буквой a.

Как только сообщение пройдет через сеть, крайние левые биты номера модуля больше не требуются. Их можно использовать, записав туда номер входной линии, чтобы было известно, по какому пути посылать ответ. Для пути a входные линии — это 0 (верхний вход в ID), 1 (нижний вход в 2D) и 1 (нижний вход в 3D) соответственно. При отправке ответа тоже используется 011, только теперь число читается справа налево.

В то время как все это происходит, процессору 001 нужно записать слово в модуль памяти 001. Здесь происходит аналогичный процесс. Сообщение отправляется через верхний, верхний и нижний выходы соответственно. На рис. 8.22 этот путь отмечен буквой Ь. Когда сообщение пребывает в пункт назначения, в поле *Модуль* содержится 001. Это число показывает путь, который прошло сообщение. Поскольку эти два запроса используют совершенно разные коммутаторы, линии и модули памяти, они могут протекать параллельно.

А теперь рассмотрим, что произойдет, если процессору 000 одновременно с этим понадобился доступ к модулю памяти 000. Его запрос вступит в конфликт с запросом процессора 001 на коммутаторе 3А. Одному из них придется подождать. В отличие от координатного коммутатора, сеть omega — это **блокируемая сеть.** Не всякий набор запросов может передаваться одновременно. Конфликты могут возникать при использовании одного и того же провода или одного и того же коммутатора, а также между запросами, направленными к памяти, и ответами, исходящими из памяти.

Желательно равномерно распределить обращения к памяти по модулям. Один из возможных способов — использовать младшие биты в качестве номера модуля памяти. Рассмотрим адресное пространство с побайтовой адресацией для компьютера, который в основном получает доступ к 32-битным словам. Два младших бита обычно будут 00, но следующие три бита будут равномерно распределены. Если использовать эти три бита в качестве номера модуля памяти, последовательно адресуемые слова будут находиться в последовательных модулях. Система памяти, в которой последовательные слова находятся в разных модулях памяти, называется **расслоенной.** Расслоенная система памяти доводит параллелизм до максимума, поскольку большая часть обращений к памяти — это обращения к последовательным адресам. Можно разработать неблокируемые сети, в которых существует несколько путей от каждого процессора к каждому модулю памяти.

Мультипроцессоры NUMA

Размер мультипроцессоров UMA с одной шиной обычно ограничивается до нескольких десятков процессоров, а для координатных мультипроцессоров или мультипроцессоров с коммутаторами требуется дорогое аппаратное обеспечение, и они ненамного больше по размеру. Чтобы получить более 100 процессоров, нужно что-то предпринять. Отметим, что все модули памяти имеют одинаковое время доступа. Это наблюдение приводит к разработке мультипроцессоров **NUMA (NonUniform Memory Access — с неоднородным доступом к памяти).** Как и мультипроцессоры UMA, они обеспечивают единое адресное пространство для всех процессоров, но, в отличие от машин UMA, доступ к локальным модулям памяти происходит быстрее, чем к удаленным. Следовательно, все программы UMA будут работать без изменений на машинах NUMA, но производительность будет хуже, чем на машине UMA с той же тактовой частотой.

Машины NUMA имеют три ключевые характеристики, которыми все они обладают и которые в совокупности отличают их от других мультипроцессоров:

1. Существует одно адресное пространство, видимое для всех процессоров.
2. Доступ к удаленной памяти производится с использованием команд LOAD и STORE.
3. Доступ к удаленной памяти происходит медленнее, чем доступ к локальной памяти.

Если время доступа к удаленной памяти не скрыто (поскольку кэш-память отсутствует), то такая система называется **NC-NUMA (No Caching NUMA — NUMA без кэширования).** Если присутствуют согласованные кэши, то система называется **CC-NUMA (Coherent Cache NUMA — NUMA с согласованной кэш-памятью).**

Программисты часто называют ее **аппаратной DSM (Distributed Shared Memory — распределенная совместно используемая память),** поскольку она по сути сходна с программной DSM, но реализуется в аппаратном обеспечении с использованием страниц маленького размера.

Одной из первых машин NC-NUMA была Carnegie-Mellon Cm*. Она проиллюстрирована в упрощенной форме рис. 8.23 [143]. Машина состояла из набора процессоров LSI-11, каждый с собственной памятью, обращение к которой производится по локальной шине. (LSI-11 — это один из видов процессора DEC PDP-11 на одной микросхеме; этот мини-компьютер был очень популярен в 70-е годы.) Кроме того, системы LSI-11 были связаны друг с другом системной шиной. Когда запрос памяти приходил в блок управления памятью, производилась проверка и определялось, находится ли нужное слово в локальной памяти. Если да, то запрос отправлялся по локальной шине. Если нет, то запрос направлялся по системной шине к системе, которая содержала данное слово. Естественно, вторая операция занимала гораздо больше времени, чем первая. Выполнение программы из удаленной памяти занимало в 10 раз больше времени, чем выполнение той же программы из локальной памяти.

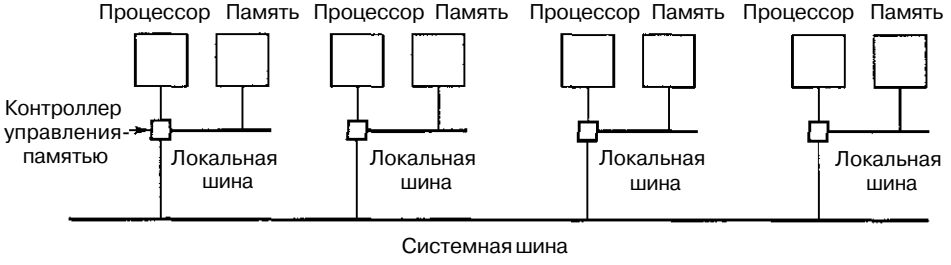


Рис. 8.23. Машина NUMA с двумя уровнями шин. Cm* — первый мультипроцессор, в котором использовалась данная разработка

Согласованность памяти гарантирована в машине NC-NUMA, поскольку там отсутствует кэш-память. Каждое слово памяти находится только в одном месте, поэтому нет никакой опасности появления копии с устаревшими данными: здесь вообще нет копий. Имеет большое значение, в какой именно памяти находится та или иная страница, поскольку от этого зависит производительность. Машины NC-NUMA используют сложное программное обеспечение для перемещения страниц, чтобы максимально увеличить производительность.

Обычно существует «сторожевой» процесс (демон), так называемый страничный сканер, который запускается каждые несколько секунд. Он должен следить за статистикой использования страниц и перемещать их таким образом, чтобы улучшить производительность. Если страница окажется в неправильном месте, страничный сканер преобразует ее таким образом, чтобы следующее обращение к ней вызвало ошибку из-за отсутствия страницы. Когда происходит такая ошибка, принимается решение о том, куда поместить эту страницу, возможно, в другую память, из которой она была взята раньше. Для предотвращения пробуксовки существует правило, которое гласит, что если страница была помещена в то или иное место, она должна оставаться в этом месте на время AT. Было рассмотрено

множество алгоритмов, но ни один из них не работает лучше других при любых обстоятельствах [80].

Мультипроцессоры CC-NUMA

Мультипроцессоры, подобные тому, который изображен на рис. 8.23, плохо расширяются, поскольку в них нет кэш-памяти. Каждый раз переходить к удаленной памяти, чтобы получить доступ к слову, которого нет в локальной памяти, очень невыгодно: это сильно снижает производительность. Однако с добавлением кэш-памяти нужно будет добавить и способ совместимости кэшей. Один из способов — отслеживать системную шину. Технически это сделать несложно, но мы уже видели (когда рассматривали Enterprise 10000), что даже с четырьмя отслеживающими шинами и высокоскоростным координатным коммутатором шириной 16 байтов для передачи данных 64 процессора — это верхний предел. Для создания мультипроцессоров действительно большого размера нужен совершенно другой подход.

Самый популярный подход для построения больших мультипроцессоров **CC-NUMA (Cache Coherent NUMA — NUMA с согласованной кэш-памятью) — мультипроцессор на основе каталога.** Основная идея состоит в сохранении базы данных, которая сообщает, где именно находится каждая строка кэш-памяти и каково ее состояние. При обращении к строке кэш-памяти из базы данных выявляется информация о том, где находится эта строка и изменялась она или нет. Поскольку обращение к базе данных происходит на каждой команде, которая обращается к памяти, база данных должна находиться в высокоскоростном специализированном аппаратном обеспечении, которое способно выдавать ответ на запрос за долю цикла шины.

Чтобы лучше понять, что собой представляет мультипроцессор на основе каталога, рассмотрим в качестве примера систему из 256 узлов, в которой каждый узел состоит из одного процессора и 16 Мбайт ОЗУ, связанного с процессором через локальную шину. Общий объем памяти составляет 2^{32} байтов. Она разделена на 2^{26} строк кэш-памяти по 64 байта каждая. Память статически распределена по узлам: 0-16 М в узле 0, 16 М-32 М — в узле 1 и т. д. Узлы связаны через сеть (рис. 8.24, *а).* Сеть может быть в виде решетки, гиперкуба или другой топологии. Каждый узел содержит элементы каталога для 2^{18} 64-байтных строк кэш-памяти, составляя свою 2^{24}-байтную память. На данный момент мы предполагаем, что строка может содержаться максимум в одной кэш-памяти.

Чтобы понять, как работает каталог, проследим путь команды LOAD из процессора 20, который обращается к кэшированной строке. Сначала процессор, выдавший команду, передает ее в блок управления памятью, который переводит ее в физический адрес, например 0x24000108. Блок управления памятью разделяет этот адрес на три части, как показано на рис. 8.24, *б.* В десятичной системе счисления эти три части — узел 36, строка 4 и смещение 8. Блок управления памятью видит, что слово памяти, к которому производится обращение, находится в узле 36, а не в узле 20, поэтому он посылает запрос через сеть в узел 36, где находится нужная строка, узнает, есть ли строка 4 в кэш-памяти, и если да, то где именно.

Когда запрос прибывает в узел 36, он направляется в аппаратное обеспечение каталога. Аппаратное обеспечение индексирует таблицу их 2^{18} элементов (один

элемент на каждую строку кэш-памяти) и извлекает элемент 4. Из рис. 8.24, *в* видно, что эта строка отсутствует в кэш-памяти, поэтому аппаратное обеспечение вызывает строку 4 из локального ОЗУ, отправляет ее в узел 20 и обновляет элемент каталога 4, чтобы показать, что эта строка находится в кэш-памяти в узле 20.

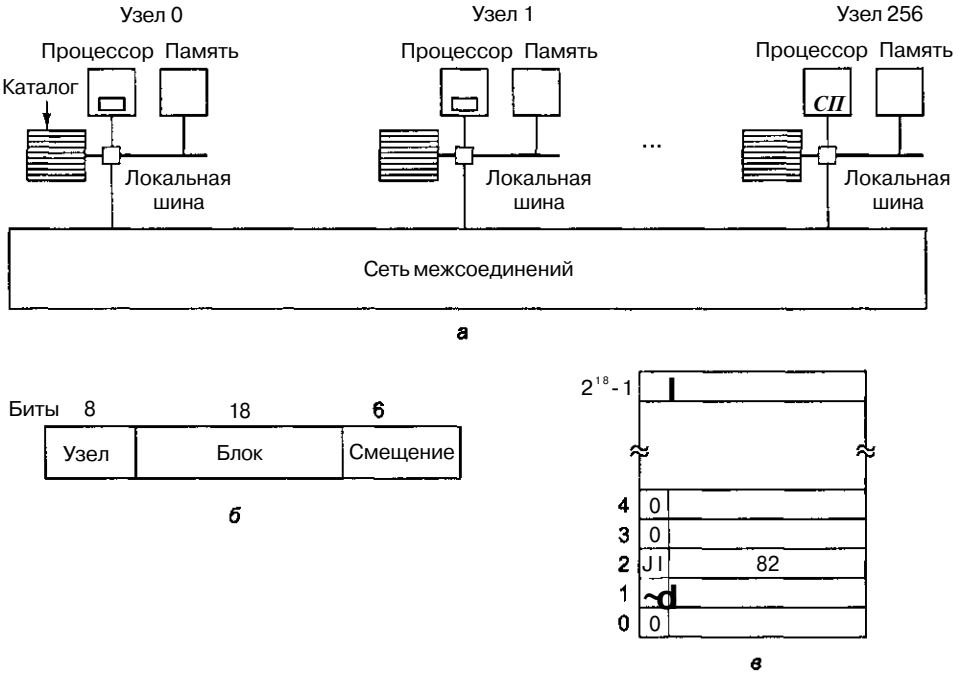


Рис. 8.24. Мультипроцессор на основе каталога, содержащий 256 узлов (а); разбиение 32-битного адреса памяти на поля (б); каталог в узле 36 (в)

А теперь рассмотрим второй запрос, на этот раз о строке 2 из узла 36. Из рис. 8.24, *в* видно, что эта строка находится в кэш-памяти в узле 82. В этот момент аппаратное обеспечение может обновить элемент каталога 2, чтобы сообщить, что строка находится теперь в узле 20, а затем может послать сообщение в узел 82, чтобы строка из него была передана в узел 20, и объявить недействительной его кэш-память. Отметим, что даже в так называемом мультипроцессоре с памятью совместного использования перемещение многих сообщений проходит скрыто.

Давайте вычислим, сколько памяти занимают каталоги. Каждый узел содержит 16 Мбайт ОЗУ и 2^{18} 9-битных элементов для слежения за этим ОЗУ. Таким образом, непроизводительные затраты каталога составляют примерно $9x2^{18}$ битов от 16 Мбайт или около 1,76%, что вполне допустимо. Даже если длина строки кэш-памяти составляет 32 байта, непроизводительные затраты составят всего 4%. Если длина строки кэш-памяти равна 128 байтов, непроизводительные затраты будут ниже 1%.

Очевидным недостатком этой разработки является то, что строка может быть кэширована только в одном узле. Чтобы строки можно было кэшировать в нескольких узлах, потребуется какой-то способ их нахождения (например, чтобы объявлять недействительными или обновлять их при записи). Возможны различные варианты.

Одна из возможностей — предоставить каждому элементу каталога k полей для определения других узлов, что позволит сохранять каждую строку в нескольких блоках кэш-памяти (допустимо до k различных узлов). Вторая возможность — заменить номер узла битовым отображением, один бит на узел. Здесь нет ограничений на количество копий, но существенно растут непроизводительные затраты. Каталог, содержащий 256 битов для каждой 64-байтной (512-битной) строки кэш-памяти, подразумевает непроизводительные затраты выше 50%. Третья возможность — хранить в каждом элементе каталога 8-битное поле и использовать это поле как заголовок связанного списка, который связывает все копии строки кэш-памяти вместе. При такой стратегии требуется дополнительное пространство в каждом узле для указателей связанного списка. Кроме того, требуется просматривать связанный список, чтобы в случае необходимости найти все копии. Каждая из трех стратегий имеет свои преимущества и недостатки. На практике используются все три стратегии.

Еще одна проблема данной разработки — как следить за тем, обновлена ли исходная память или нет. Если нужно считать строку кэш-памяти, которая не изменялась, запрос может быть удовлетворен из основной памяти, и при этом не нужно направлять запрос в кэш-память. Если нужно считать строку кэш-памяти, которая была изменена, то этот запрос должен быть направлен в тот узел, в котором находится нужная строка кэш-памяти, поскольку только здесь имеется действительная копия. Если разрешается иметь только одну копию строки кэш-памяти, как на рис. 8.24, то нет никакого смысла в отслеживании изменений в строках кэш-памяти, поскольку любой новый запрос должен пересылаться к существующей копии, чтобы объявить ее недействительной.

Когда строка кэш-памяти меняется, нужно сообщить в исходный узел, даже если существует только одна копия строки кэш-памяти. Если существует несколько копий, изменение одной из них требует объявления всех остальных недействительными. Поэтому нужен какой-либо протокол, чтобы устранить ситуацию состояния гонок. Например, чтобы изменить общую строку кэш-памяти, один из держателей этой строки должен запросить монопольный доступ к ней перед тем, как изменить ее. В результате все другие копии будут объявлены недействительными. Другие возможные оптимизации CC-NUMA обсуждаются в книге [140].

Мультипроцессор Stanford DASH

Первый мультипроцессор CC-NUMA на основе каталога — **DASH (Directory Architecture for SHared memory — архитектура на основе каталога для памяти совместного использования)** — был создан в Стенфордском университете как исследовательский проект [81]. Данная разработка проста для понимания. Она повлияла на ряд промышленных изделий, например SGI Origin 2000. Мы рассмотрим 64-процессорный прототип данной разработки, который был реально сконструирован. Он подходит и для машин большего размера.

Схема машины DASH в немного упрощенном виде представлена на рис. 8.25, *а*. Она состоит из 16 кластеров, каждый из которых содержит шину, 4 процессора MIPS R3000, 16 Мбайт глобальной памяти, а также некоторые устройства ввода-вывода (диски и т. д.), которые на схеме не показаны. Каждый процессор отслеживает только свою локальную шину. Локальная совместимость поддерживается

с помощью отслеживания; для глобальной согласованности нужен другой механизм, поскольку глобального отслеживания не существует.

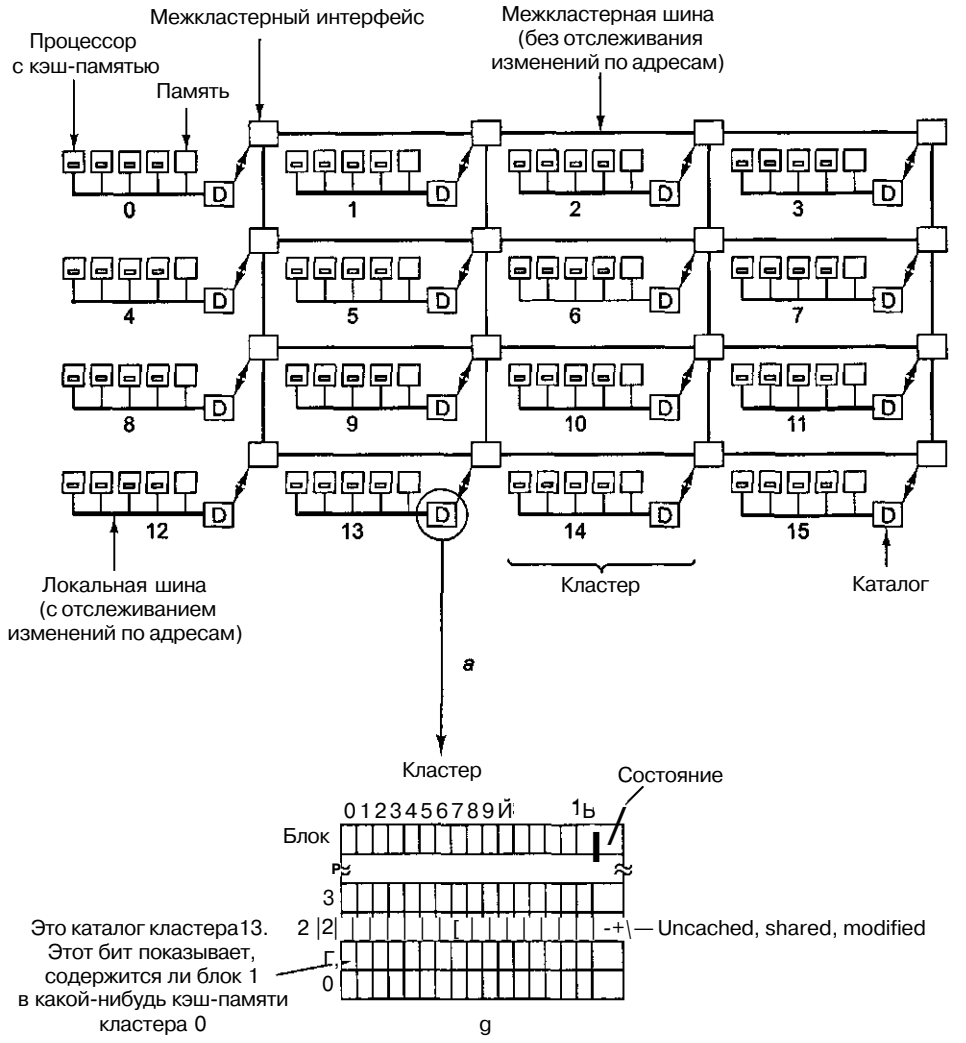


Рис. 8.25. Архитектура DASH (а); каталог DASH (б)

Полный объем адресного пространства в данной системе равен 256 Мбайт. Адресное пространство разделено на 16 областей по 16 Мбайт каждая. Глобальная память кластера 0 включает адреса с 0 по 16 М. Глобальная память кластера 1 включает адреса с 16 М по 32 М и т. д. Размер строки кэш-памяти составляет 16 байтов. Передача данных также осуществляется по строкам в 16 байтов. Каждый кластер содержит 1 М строк.

Каждый кластер содержит каталог, который следит за тем, какие кластеры в настоящий момент имеют копии своих строк. Поскольку каждый кластер содержит

1 М строк, в каталоге содержится 1 М элементов, по одному элементу на каждую строку. Каждый элемент содержит битовое отображение по одному биту на кластер. Этот бит показывает, имеется ли в данный момент строка данного кластера в кэш-памяти. Кроме того, элемент содержит 2-битное поле, которое сообщает о состоянии строки.

1 М элементов по 18 битов каждый означает, что общий размер каждого каталога превышает 2 Мбайт. При наличии 16 кластеров вся память каталога будет немного превышать 36 Мбайт, что составляет около 14% от 256 Мбайт. Если число процессоров на кластер возрастает, объем памяти каталога не меняется. Большое число процессоров на каждый кластер позволяет погашать стоимость памяти каталога, а также контроллера шины при наличии большого числа процессоров, сокращая стоимость на каждый процессор. Именно поэтому каждый кластер имеет несколько процессоров.

Каждый кластер в DASH связан с интерфейсом, который дает возможность кластеру обмениваться информацией с другими кластерами. Интерфейсы связаны через межкластерные каналы в прямоугольную решетку, как показано на рис. 8.25, а. Чем больше добавляется кластеров в систему, тем больше нужно добавлять межкластерных каналов, поэтому пропускная способность системы возрастает. В системе используется маршрутизация «червоточина», поэтому первая часть пакета может быть направлена дальше еще до того, как получен весь пакет, что сокращает задержку в каждом транзитном участке. Существует два набора межкластерных каналов: один — для запрашивающих пакетов, а другой — для ответных пакетов (на рисунке это не показано). Межкластерные каналы нельзя отслеживать.

Каждая строка кэш-памяти может находиться в одном из трех следующих состояний:

1. UNCACHED (некэшированная) — строка находится только в памяти.
2. SHARED (совместно используемая) — память содержит новейшие данные; строка может находиться в нескольких блоках кэш-памяти.
3. MODIFIED (измененная) — строка, содержащаяся в памяти, неправильная; данная строка находится только в одной кэш-памяти.

Состояние каждой строки кэш-памяти содержится в поле *Состояние* в соответствующем элементе каталога, как показано на рис. 8.25, *б*.

Протоколы DASH основаны на обладании и признании недействительности. В каждый момент у каждой строки имеется уникальный владелец. Для строк в состоянии UNCHANGED или SHARED владельцем является собственный кластер данной строки. Для строк в состоянии MODIFIED владельцем является тот кластер, в котором содержится единственная копия этой строки. Прежде чем записать что-либо в строку в состоянии SHARED, нужно найти и объявить недействительными все существующие копии.

Чтобы понять, как работает этот механизм, рассмотрим, как процессор считывает слово из памяти. Сначала он проверяет свою кэш-память. Если там слова нет, на локальную шину кластера передается запрос, чтобы узнать, содержит ли какой-нибудь другой процессор того же кластера строку, в которой присутствует нужное слово. Если да, то происходит передача строки из одной кэш-памяти в другую. Если строка находится в состоянии SHARED, то создается ее копия. Если строка нахо-

дится в состоянии MODIFIED, нужно проинформировать исходный каталог, что строка теперь SHARED. В любом случае слово берется из какой-то кэш-памяти, но это не влияет на битовое отображение каталогов (поскольку каталог содержит 1 бит на кластер, а не 1 бит на каждый процессор).

Если нужная строка не присутствует ни в одной кэш-памяти данного кластера, то пакет с запросом отправляется в исходный кластер, содержащий данную строку. Этот кластер определяется по 4 старшим битам адреса памяти, и им вполне может оказаться кластер запрашивающей стороны. В этом случае сообщение физически не посылается. Аппаратное обеспечение в исходном кластере проверяет свои таблицы и выясняет, в каком состоянии находится строка. Если она UNCACHED или SHARED, аппаратное обеспечение, которое управляет каталогом, вызывает эту строку из глобальной памяти и посылает ее обратно в запрашивающий кластер. Затем аппаратное обеспечение обновляет свой каталог, помечая данную строку как сохраненную в кэш-памяти кластера запрашивающей стороны.

Если нужная строка находится в состоянии MODIFIED, аппаратное обеспечение находит кластер, который содержит эту строку, и посылает запрос туда. Затем кластер, который содержит данную строку, посылает ее в запрашивающий кластер и помечает ее копию как SHARED, поскольку теперь эта строка находится более чем в одной кэш-памяти. Он также посылает копию обратно в исходный кластер, чтобы обновить память и изменить состояние строки на SHARED.

Запись происходит по-другому. Перед тем как осуществить запись, процессор должен убедиться, что он является единственным обладателем данной строки кэш-памяти в системе. Если в кэш-памяти данного процессора уже есть эта строка и она находится в состоянии MODIFIED, то запись можно осуществить сразу же. Если строка в кэш-памяти есть, но она находится в состоянии SHARED, то сначала в исходный кластер посылается пакет, чтобы объявить все остальные копии недействительными.

Если нужной строки нет в кэш-памяти данного процессора, этот процессор посылает запрос на локальную шину, чтобы узнать, нет ли этой строки в соседних процессорах. Если данная строка там есть, то она передается из одной кэш-памяти в другую. Если эта строка SHARED, то все остальные копии должны быть объявлены недействительными.

Если строка находится где-либо еще, пакет посылается в исходный кластер. Здесь может быть три варианта. Если строка находится в состоянии UNCACHED, она помечается как MODIFIED и отправляется к запрашивающему процессору. Если строка находится в состоянии SHARED, все копии объявляются недействительными, и после этого над строкой совершается та же процедура, что и над UNCASHED строкой. Если строка находится в состоянии MODIFIED (изменена), то запрос направляется в тот кластер, в котором строка содержится в данный момент. Этот кластер удовлетворяет запрос, а затем объявляет недействительной свою собственную копию.

Сохранить согласованность памяти в системе DASH довольно трудно, и происходит это очень медленно. Для одного обращения к памяти порой нужно отправлять большое количество сообщений. Более того, чтобы память была согласованной, доступ нельзя завершить, пока прием всех пакетов не будет подтвержден, а это плохо влияет на производительность. Для разрешения этих проблем в систе-

ме DASH используется ряд специальных приемов (это могут быть два набора межкластерных каналов, конвейеризированные записи, а также использование свободной согласованности вместо согласованности по последовательности).

Мультипроцессор Sequent NUMA-Q

Машина DASH никогда не была коммерческим продуктом. В этом разделе мы рассмотрим одно из коммерческих изделий — машину Sequent NUMA-Q 2000. В ней используется очень интересный протокол когерентности кэширования — **SCI (Scalable Coherent Interface — масштабируемый когерентный интерфейс).** Этот протокол стандартный (стандарт IEEE 1569), поэтому он используется и в ряде других машин CC-NUMA.

В основе машины NUMA-Q лежит стандартная плата **quard board,** которая произведена компанией Intel. Плата содержит 4 процессора Pentium Pro и до 4 Гбайт ОЗУ. Каждый процессор содержит кэш-память первого уровня и кэш-память второго уровня. Непротиворечивость кэшей сохраняется благодаря отслеживанию локальной шины платы quard board с использованием протокола MESI. Скорость передачи данных в локальной шине составляет 534 Мбайт/с. Размер строки кэш-памяти равен 64 байтам. Схема мультипроцессора NUMA-Q изображена на рис. 8.26.

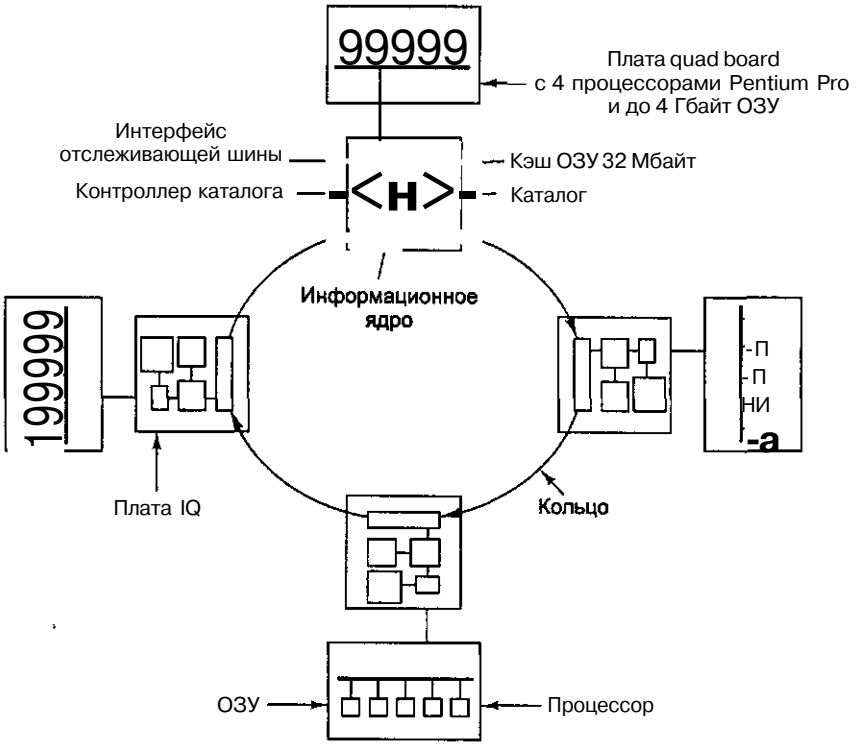


Рис. 8.26. Мультипроцессор NUMA-Q

Чтобы расширить систему, нужно вставить плату сетевого контроллера в гнездо платы quad board, предназначенное для контроллеров сети. Сетевой контроллер,

плата IQ-Link, соединяет все платы quad board в один мультипроцессор. Ее главная задача — реализовать протокол SCI. Каждая плата IQ-Link содержит 32 Мбайт кэш-памяти, каталог, который следит за тем, что находится в кэш-памяти, интерфейс с локальной шиной платы quad board и микросхему, называемую **информационным ядром,** соединяющую плату IQ-Link с другими платами IQ-Link. Эта микросхема подкачивает данные от входа к выходу, сохраняя те данные, которые направляются в данный узел, и передавая все прочие данные далее без изменений.

Все платы IQ-Link в совокупности формируют кольцо, как показано на рис. 8.26. В данной разработке присутствует два уровня протокола когерентности кэширования. Протокол SCI поддерживает непротиворечивость всех кэшей платы IQ-Link, используя это кольцо. Протокол MESI используется для сохранения непротиворечивости между четырьмя процессорами и кэш-памятью на 32 Мбайт в каждом узле.

В качестве связи между платами quad board используется интерфейс SCI. Этот интерфейс был разработан для того, чтобы заменить шину в больших мультипроцессорах и мультикомпьютерах (например, NUMA-Q). SCI поддерживает непротиворечивость кэшей, которая необходима в мультипроцессорах, а также позволяет быстро передавать блоки, что необходимо в мультикомпьютерах. SCI выдерживает нагрузку до 64 К узлов, адресное пространство каждого из которых может быть до 2^{48} байтов. Самая большая система NUMA-Q состоит из 63 плат quad board, которые содержат 252 процессора и почти 2^{38} байтов физической памяти. Как видим, возможности SCI гораздо выше.

Кольцо, которое соединяет платы IQ-Link, соответствует протоколу SCI. В действительности это вообще не кольцо, а отдельные двухточечные кабели. Ширина кабеля составляет 18 битов: 1 бит синхронизации, 1 флаговый бит и 16 битов данных. Все они передаются параллельно. Каналы синхронизируются с тактовой частотой 500 МГц, при этом скорость передачи данных составляет 1 Гбайт/с. По каналам передаются пакеты. Каждый пакет содержит заголовок из 14 байтов, 0, 16, 64 или 256 байтов данных и контрольную сумму на 2 байта. Трафик состоит из запросов и ответов.

Физическая память в машине NUMA-Q 2000 распределена по узлам, так что каждая страница памяти имеет свою собственную машину. Каждая плата quad board может вмещать до 4 Гбайт ОЗУ. Размер строки кэш-памяти равен 64 байтам, поэтому каждая плата quad board содержит 2^{26} строк кэш-памяти. Когда строка не используется, она находится только в одном месте — в собственной памяти.

Однако строки могут находиться в нескольких разных кэшах, поэтому для каждого узла должна существовать **таблица локальной памяти** из 2^{26} элементов, по которой можно находить местоположение строк. Один из возможных вариантов — иметь на каждый элемент таблицы битовое отображение, которое показывает, какие платы IQ-Link содержат эту строку. Но в SCI не используется такое битовое отображение, поскольку оно плохо расширяется. (Напомним, что SCI может выдерживать нагрузку до 64 К узлов, и иметь 2^{26} элементов по 64 К битов каждый было бы слишком накладно.)

Вместо этого все копии строки кэш-памяти собираются в дважды связанный список. Элемент в таблице локальной памяти исходного узла показывает, в каком узле содержится головная часть списка. В машине NUMA-Q 2000 достаточно 6-битного номера, поскольку здесь может быть максимум 63 узла. Для системы SCI максимального размера достаточно будет 16-битного номера. Такая схема подходит

для больших систем гораздо лучше, чем битовое отображение. Именно это свойство делает SCI более расширяемой по сравнению с системой DASH.

Кроме таблицы локальной памяти, каждая плата IQ-Link содержит каталог с одним элементом для каждой строки кэш-памяти, которую плата в данный момент содержит. Поскольку размер кэш-памяти составляет 32 Мбайт, а строка кэш-памяти включает 64 байта, каждая плата IQ-Link может содержать до 2^{19} строк кэш-памяти. Поэтому каждый каталог содержит 2^{19} элементов, по одному элементу на каждую строку кэш-памяти.

Если строка находится только в одной кэш-памяти, то тот узел, в котором находится строка, указывается в таблице локальной памяти исходного узла. Если после этого данная строка появится в кэш-памяти другого узла, то в соответствии с новым протоколом исходный каталог будет указывать на новый элемент, который, в свою очередь, указывает на старый элемент. Таким образом формируется двухэлементный список. Все новые узлы, содержащие ту же строку, прибавляются к началу списка. Следовательно, все узлы, которые содержат эту строку, связываются в сколь угодно длинный список. На рис. 8.27 проиллюстрирован этот процесс (в данном случае строка кэш-памяти содержится в узлах 4, 9 и 22).

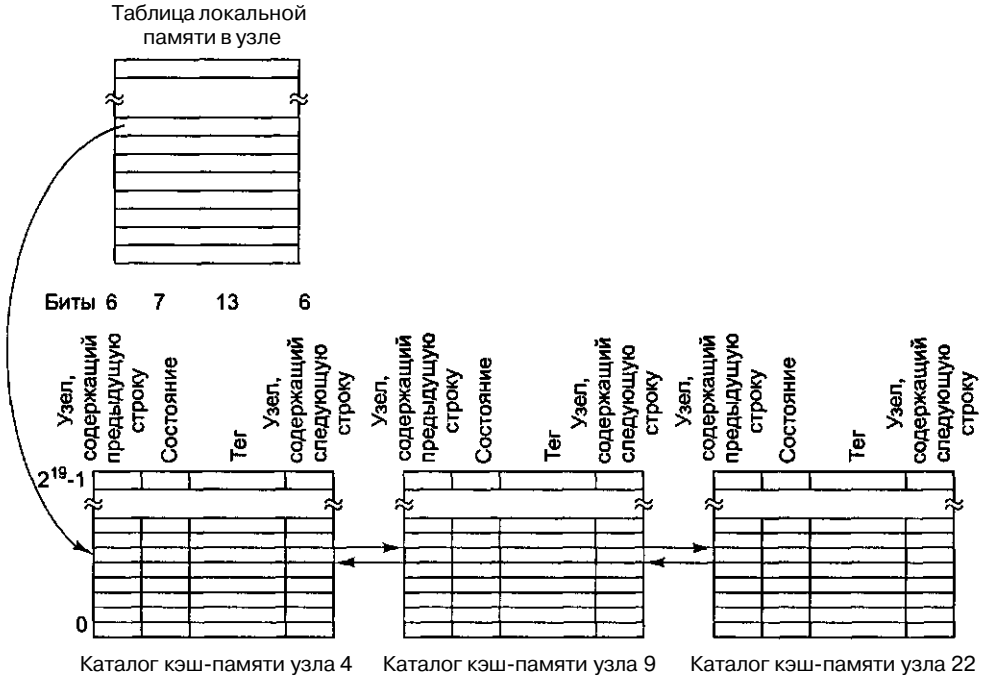


Рис. 8.27. Протокол SCI соединяет всех держателей данной строки в дважды связанный список. В данном примере строка находится одновременно в трех узлах

Каждый элемент каталога состоит из 36 битов. Шесть битов указывают на узел, который содержит предыдущую строку цепочки. Следующие шесть битов указывают на узел, содержащий следующую строку цепочки. Ноль указывает на конец цепочки, и именно поэтому максимальный размер системы составляет 63 узла, а не 64. Следующие 7 битов предназначены для записи состояния строки.

Последние 13 битов — это тег (нужен для идентификации строки). Напомним, что самая большая система NUMA-Q2000 содержит $63x2^{32}=2^{38}$ байтов ОЗУ, поэтому имеется почти 2^{32} строк кэш-памяти. Все кэши прямого отображения, поэтому 2^{32} строк отображаются на 2^{19} элементов кэш-памяти, существует 2^{13} строк, которые отображаются на каждый элемент. Следовательно, 13-битный тег нужен для того, чтобы показывать, какая именно строка находится там в данный момент.

Каждая строка имеет фиксированную позицию только в одном блоке памяти (исходном). Эти строки могут находиться в одном из трех состояний: UNCACHED (некэшированном), SHARED (совместного использования) и MODIFIED (измененном). В протоколе SCI эти состояния называются HOME, FRESH и GONE соответственно, но мы во избежание путаницы будем использовать прежнюю терминологию. Состояние UNCACHED означает, что строка не содержится ни в одном из кэшей на плате IQ-Link, хотя она может находиться в локальной кэш-памяти на той же самой плате quad board. Состояние SHARED означает, что строка находится по крайней мере в одной кэш-памяти платы IQ-Link, а память содержит обновленные данные. Состояние MODIFIED означает, что строка находится в кэш-памяти на какой-то плате IQ-Link, но, возможно, эта строка была изменена, поэтому память может содержать устаревшие данные.

Блоки кэш-памяти могут находиться в одном из 29 устойчивых состояний или в одном из переходных состояний (их более 29). Для записи состояния каждой строки необходимо 7 битов (рис. 8.27). Каждое устойчивое состояние показывается двумя полями. Первое поле указывает, где находится строка — в начале списка, в конце, в середине или вообще является единственным элементом списка. Второе поле содержит информацию о том, изменялась ли строка, находится ли она только в этой кэш-памяти и т. п.

В протоколе SCI определены 3 операции со списком: добавление узла к списку, удаление узла из списка и очистка всех узлов кроме одного. Последняя операция нужна в том случае, если разделяемая строка изменена и становится единственной.

Протокол SCI имеет три варианта сложности. Протокол минимальной степени сложности разрешает иметь только одну копию каждой строки в кэш-памяти (см. рис. 8.24). В соответствии с протоколом средней степени сложности каждая строка может кэшироваться в неограниченном количестве узлов. Полный протокол включает различные особенности для увеличения производительности. В системе NUMA-Q используется протокол средней степени сложности, поэтому ниже мы рассмотрим именно его.

Разберемся, как обрабатывается команда READ. Если процессор, выполняющий эту команду, не может найти нужную строку на своей плате, то плата IQ-Link посылает пакет исходной плате IQ-Link, которая затем смотрит на состояние этой строки. Если состояние UNCACHED, оно превращается в SHARED, и нужная строка берется из основной памяти. Затем таблица локальной памяти в исходном узле обновляется. После этого таблица содержит одноэлементный список, указывающий на узел, в кэш-памяти которого в данный момент находится строка.

Предположим, что состояние строки SHARED. Эта строка также берется из памяти, и ее номер сохраняется в элементе каталога в исходном узле. Элемент каталога в запрашивающем узле устанавливается на другое значение — чтобы указывать на старый узел. Эта процедура увеличивает список на один элемент. Новая кэш-память становится первым элементом.

Представим теперь, что требуемая строка находится в состоянии MODIFIED. Исходный каталог не может вызвать эту строку из памяти, поскольку в памяти содержится недействительная копия. Вместо этого исходный каталог сообщает запрашивающей плате IQ-Link, в какой кэш-памяти находится нужная строка, и отдает приказ вызвать строку оттуда. Каталог также изменяет элемент таблицы локальной памяти, поскольку теперь он должен указывать на новое местоположение строки.

Обработка команды WRITE происходит немного по-другому. Если запрашивающий узел уже находится в списке, он должен удалять все остальные элементы, чтобы остаться в списке единственным. Если его нет в списке, он должен удалить все элементы, а затем войти в список в качестве единственного элемента. В любом случае в конце концов этот элемент остается единственным в списке, а исходный каталог указывает на этот узел. В действительности обработка команд READ и WRITE довольно сложна, поскольку протокол должен работать правильно, даже если несколько машин одновременно выполняют несовместимые операции на одной линии. Все переходные состояния были введены именно для того, чтобы протокол работал правильно даже во время одновременно выполняемых операций. В этой книге мы не можем изложить полное описание протокола. Если вам это необходимо, обратитесь к стандарту IEEE 1596.

Мультипроцессоры COMA

Машины NUMA и CC-NUMA имеют один большой недостаток: обращения к удаленной памяти происходят гораздо медленнее, чем обращения к локальной памяти. В машине CC-NUMA эта разница в производительности в какой-то степени нейтрализуется благодаря использованию кэш-памяти. Однако если количество требуемых удаленных данных сильно превышает вместимость кэш-памяти, промахи будут происходить постоянно и производительность станет очень низкой.

Мы видим, что машины UMA, например Sun Enterprise 10000, имеют очень высокую производительность, но ограничены в размерах и довольно дорого стоят. Машины NUMA могут расширяться до больших размеров, но в них требуется ручное или полуавтоматическое размещение страниц, а оно не всегда проходит удачно. Дело в том, что очень трудно предсказать, где какие страницы могут понадобиться, и кроме того, страницы трудно перемещать из-за большого размера. Машины CC-NUMA, например Sequent NUMA-Q, могут работать с очень низкой производительностью, если большому числу процессоров требуется много удаленных данных. Так или иначе, каждая из этих разработок имеет существенные недостатки.

Однако существует процессор, в котором все эти проблемы разрешаются за счет того, что основная память каждого процессора используется как кэш-память. Такая разработка называется **COMA (Cache Only Memory Access).** В ней страницы не имеют собственных фиксированных машин, как в системах NUMA и CC-NUMA.

Вместо этого физическое адресное пространство делится на строки, которые перемещаются по системе в случае необходимости. Блоки памяти не имеют собственных машин. Память, которая привлекает строки по мере необходимости, называется **attraction memory.** Использование основной памяти в качестве большой кэш-памяти увеличивает частоту успешных обращений в кэш-память, а следовательно, и производительность.

К сожалению, ничего идеального не бывает. В системе COMA появляется две новых проблемы:

1. Как размещаются строки кэш-памяти?
2. Если строка удаляется из памяти, что произойдет, если это последняя копия?

Первая проблема связана со следующим фактом. Если блок управления памятью транслировал виртуальный адрес в физический и если строки нет в аппаратной кэш-памяти, то очень трудно определить, есть ли вообще эта строка в основной памяти. Аппаратное обеспечение здесь не поможет, поскольку каждая страница состоит из большого количества отдельных строк кэш-памяти, которые перемещаются в системе независимо друг от друга. Даже если известно, что строка отсутствует в основной памяти, как определить, где она находится? В данном случае нельзя спросить об этом собственную машину, поскольку таковой машины в системе нет.

Было предложено несколько решений этой проблемы. Можно ввести новое аппаратное обеспечение, которое будет следить за тегом каждой строки кэш-памяти. Тогда блок управления памятью может сравнивать тег нужной строки с тегами всех строк кэш-памяти, пока не обнаружит совпадение.

Другое решение — отображать страницы полностью, но при этом не требовать присутствия всех строк кэш-памяти. В этом случае аппаратному обеспечению понадобится битовое отображение для каждой страницы, где один бит для каждой строки указывает на присутствие или отсутствие этой строки. Если строка присутствует, она должна находиться в правильной позиции на этой странице. Если она отсутствует, то любая попытка использовать ее вызовет прерывание, что позволит программному обеспечению найти нужную строку и ввести ее.

Таким образом, система будет искать только те строки, которые действительно находятся в удаленной памяти. Одно из решений — предоставить каждой странице собственную машину (ту, которая содержит элемент каталога данной страницы, а не ту, в которой находятся данные). Затем можно отправить сообщение в собственную машину, чтобы найти местоположение данной строки. Другое решение — организовать память в виде дерева и осуществлять поиск по направлению вверх, пока не будет обнаружена требующаяся строка.

Вторая проблема связана с удалением последней копии. Как и в машине CC-NUMA, строка кэш-памяти может находиться одновременно в нескольких узлах. Если происходит промах кэша, строку нужно откуда-то вызвать, а это значит, что ее нужно отбросить. А что произойдет, если выбранная строка окажется последней копией? В этом случае ее нельзя отбрасывать.

Одно из возможных решений — вернуться к каталогу и проверить, существуют ли другие копии. Если да, то строку можно смело выбрасывать. Если нет, то ее нужно переместить куда-либо еще. Другое решение — пометить одну из копий каждой строки кэш-памяти как главную копию и никогда ее не выбрасывать. При таком подходе не требуется проверка каталога. Машина COMA обещает обеспечить лучшую производительность, чем CC-NUMA, но дело в том, что было построено очень мало машин COMA, и нужно больше опыта. До настоящего момента создано всего две машины COMA: KSR-1 [20] и Data Diffusion Machine [53]. Дополнительную информацию о машинах COMA можно найти в книгах [36, 67,98, 123].

Мультикомпьютеры с передачей сообщений

Как видно из схемы на рис. 8.12, существует два типа параллельных процессоров MIMD: мультипроцессоры и мультикомпьютеры. В предыдущем разделе мы рассматривали мультипроцессоры. Мы увидели, что мультипроцессоры могут иметь разделенную память, доступ к которой можно получить с помощью обычных команд LOAD и STORE. Такая память реализуется разными способами, включая отслеживающие шины, многоступенчатые сети, а также различные схемы на основе каталога. Программы, написанные для мультипроцессора, могут получать доступ к любому месту в памяти, не имея никакой информации о внутренней топологии или схеме реализации. Именно благодаря такой иллюзии мультипроцессоры весьма популярны.

Однако мультипроцессоры имеют и некоторые недостатки, поэтому мультикомпьютеры тоже очень важны. Во-первых, мультипроцессоры нельзя расширить до больших размеров. Чтобы расширить машину Interprise 10000 до 64 процессоров, пришлось добавить огромное количество аппаратного обеспечения.

В Sequent NUMA-Q дошли до 256 процессоров, но ценой неодинакового времени доступа к памяти. Ниже мы рассмотрим два мультикомпьютера, которые содержат 2048 и 9152 процессора соответственно. Через много лет кто-нибудь сконструирует мультипроцессор, содержащий 9000 узлов, но к тому времени мультикомпьютеры будут содержать уже 100 000 узлов.

Кроме того, конфликтная ситуация при обращении к памяти в мультипроцессоре может сильно повлиять на производительность. Если 100 процессоров постоянно пытаются считывать и записывать одни и те же переменные, конфликтная ситуация для различных модулей памяти, шин и каталогов может сильно ударить по производительности.

Вследствие этих и других факторов разработчики проявляют огромный интерес к параллельным компьютерам, в которых каждый процессор имеет свою собственную память, к которой другие процессоры не могут получить прямой доступ. Это мультикомпьютеры. Программы на разных процессорах в мультикомпьютере взаимодействуют друг с другом с помощью примитивов send и receive, которые используются для передачи сообщений (поскольку они не могут получить доступ к памяти других процессоров с помощью команд LOAD и STORE). Это различие полностью меняет модель программирования.

Каждый узел в мультикомпьютере состоит из одного или нескольких процессоров, ОЗУ (общее для процессоров только данного узла), диска и(или) других устройств ввода-вывода, а также процессора передачи данных. Процессоры передачи данных связаны между собой по высокоскоростной коммуникационной сети (см. раздел «Сети межсоединений»). Используется множество различных топологий, схем коммутации и алгоритмов выбора маршрута. Все мультикомпьютеры сходны в одном: когда программа выполняет примитив send, процессор передачи данных получает уведомление и передает блок данных в целевую машину (возможно, после предварительного запроса и получения разрешения). Схема мультикомпьютера показана на рис. 8.28.

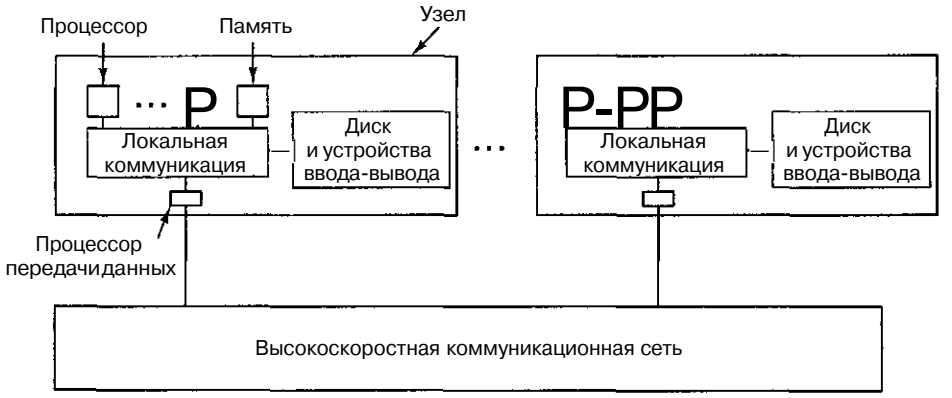


Рис. 8.28. Схема мультикомпьютера

Мультикомпьютеры бывают разных типов и размеров, поэтому очень трудно привести хорошую классификацию. Тем не менее можно назвать два общих типа: MPP и COW. Ниже мы рассмотрим каждый из этих типов.

MPP — процессоры с массовым параллелизмом

MPP (Massively Parallel Processors — процессоры с массовым параллелизмом) — это огромные суперкомпьютеры стоимостью несколько миллионов долларов. Они используются в различных науках и промышленности для выполнения сложных вычислений, для обработки большого числа транзакций в секунду или для хранения больших баз данных и управления ими.

В большинстве таких машин используются стандартные процессоры. Это могут быть процессоры Intel Pentium, Sun UltraSPARC, IBM RS/6000 и DEC Alpha. Отличает мультикомпьютеры то, что в них используется сеть, по которой можно передавать сообщения, с низким временем ожидания и высокой пропускной способностью. Обе характеристики очень важны, поскольку большинство сообщений малы по размеру (менее 256 байтов), но при этом суммарная нагрузка в большей степени зависит от длинных сообщений (более 8 Кбайт).

Еще одна характеристика MPP — огромная производительность процесса ввода-вывода. Часто приходится обрабатывать огромные массивы данных, иногда терабайты. Эти данные должны быть распределены по дискам, и их нужно перемещать в машине с большой скоростью.

Следующая специфическая черта MPP — отказоустойчивость. При наличии не одной тысячи процессоров несколько неисправностей в неделю неизбежны. Прекращать работу системы из-за неполадок в одном из процессоров было бы неприемлемо, особенно если такая неисправность ожидается каждую неделю. Поэтому большие MPP всегда содержат специальное аппаратное и программное обеспечение для контроля системы, обнаружения неполадок и их исправления.

Было бы неплохо теперь рассмотреть основные принципы разработки MPP, но, по правде говоря, их не так много. MPP представляет собой набор более или

менее стандартных узлов, которые связаны друг с другом высокоскоростной сетью. Поэтому ниже мы просто рассмотрим несколько конкретных примеров систем MPP: Cray T3E и Intel/Sandia Option Red.

CrayT3E

В семейство T3E (последователя T3D) входят самые последние суперкомпьютеры, восходящие к компьютеру 6600. Различные модели — T3E, T3E-900 и T3E-1200 — идентичны с точки зрения архитектуры и различаются только ценой и производительностью (например, 600, 900 или 1200 мегафлопов на процессор). Мегафлоп — это 1 млн операций с плавающей точкой/с. (FLOP — FLoating-point OPerations — операции с плавающей точкой). В отличие от 6600 и Cray-1, в которых очень мало параллелизма, эти машины могут содержать до 2048 процессоров. Мы используем термин T3E для обозначения всего семейства, но величины производительности будут приведены для машины T3E-1200. Эти машины продает компания Cray Research, филиал Silicon Graphics. Они применяются для разработки лекарственных препаратов, поиска нефти и многих других задач.

В системе T3E используются процессоры DEC Alpha 21164. Это суперскалярный процессор RISC, способный выдавать 4 команды за цикл. Он работает с частотой 300, 450 и 600 МГц в зависимости от модели. Тактовая частота — основное различие между разными моделями T3E. Alpha — это 64-битная машина с 64-битными регистрами. Размер виртуальных адресов ограничен до 43 битов, а физических — до 40 битов. Таким образом, возможен доступ к 1 Тбайт физической памяти.

Каждый процессор Alpha имеет двухуровневую кэш-память, встроенную в микросхему процессора. Кэш-память первого уровня содержит 8 Кбайт для команд и 8 Кбайт для данных. Кэш-память второго уровня — это смежная трехвходовая ассоциативная кэш-память на 96 Кбайт, содержащая и команды и данные вместе. Кэш-память обоих уровней содержит команды и данные только из локального ОЗУ, а это может быть до 2 Гбайт на процессор. Поскольку максимальное число процессоров равно 2048, общий объем памяти может составлять 4 Тбайт.

Каждый процессор Alpha заключен в особую схему, которая называется оболочкой (shell) (рис. 8.29). Оболочка содержит память, процессор передачи данных и 512 специальных регистров (так называемых E-регистров). Эти регистры могут загружаться адресами удаленной памяти с целью чтения или записи слов из удаленной памяти (или блоков из 8 слов). Это значит, что в машине T3E есть доступ к удаленной памяти, но осуществляется он не с помощью обычных команд LOAD и STORE. Эта машина представляет собой гибрид между NC-NUMA и MPP, но все-таки больше похожа на MPP. Непротиворечивость памяти гарантируется, поскольку слова, считываемые из удаленной памяти, не попадают в кэш-память.

Узлы в машине T3E связаны двумя разными способами (см. рис. 8.29). Основная топология — дуплексный 3-мерный тор. Например, система, содержащая 512 узлов, может быть реализована в виде куба 8x8x8. Каждый узел в 3-мерном торе имеет 6 каналов связи с соседними узлами (по направлению вперед, назад, вправо, влево, вверх и вниз). Скорость передачи данных в этих каналах связи равна 480 Мбайт/с в любом направлении.

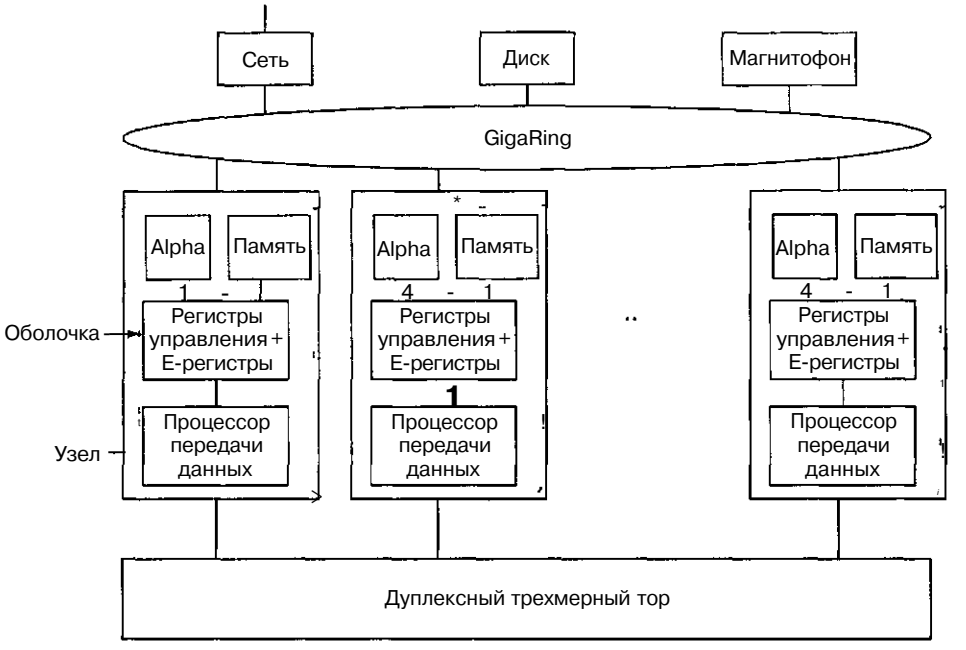


Рис. 8.29. Cray Research T3E

Узлы также связаны одним или несколькими **GigaRings** — подсистемами ввода-вывода с коммутацией пакетов, обладающими высокой пропускной способностью Узлы используют эту подсистему для взаимодействия друг с другом, а также с сетями, дисками и другими периферическими устройствами. Они по ней посылают пакеты размером до 256 байтов. Каждый GigaRing состоит из пары колец шириной в 32 бита, которые соединяют узлы процессоров со специальными узлами ввода-вывода. Узлы ввода-вывода содержат гнезда для сетевых карт (например, HIPPI, Ethernet, ATM, FDDI), дисков и других устройств.

В системе T3E может быть до 2048 узлов, поэтому неисправности будут происходить регулярно. По этой причине в системе на каждые 128 пользовательских узлов содержится один запасной узел. Испорченные узлы могут быть замещены запасными во время работы системы без перезагрузки Кроме пользовательских и запасных узлов есть узлы, которые предназначены для запуска серверов операционной системы, поскольку пользовательские узлы запускают не всю систему, а только ядро. В данном случае используется операционная система UNIX.

Intel/Sandia Option Red

Компьютеры с высокой производительностью и вооруженные силы идут в США рука об руку с 1943 года, начиная с машины ENIAC, первого электронного компьютера. Связь между американскими вооруженными силами и высокоскоростными вычислениями до сих пор продолжается. В середине 90-х годов департаменты обороны и энергетики приступили к выполнению программы разработки 5 систем MPP, которые будут работать со скоростью 1, 3, 10, 30 и 100 терафлопов/с соот-

ветственно. Для сравнения: 100 терафлопов (10^{14} операций с плавающей точкой в секунду) — это в 500000 раз больше, чем мощность процессора Pentium Pro, работающего с частотой 200 МГц.

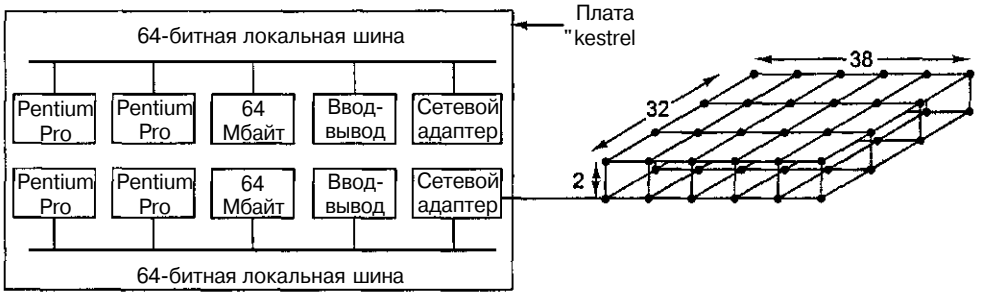
В отличие от машины T3E, которую можно купить в магазине (правда, за большие деньги), машины, работающие со скоростью 10^{14} операций с плавающей точкой, — это уникальные системы, распределяемые в конкурентных торгах Департаментом энергетики, который руководит национальными лабораториями. Компания Intel выиграла первый контракт; IBM выиграла следующие два. Если вы планируете вступить в соревнование в будущем, вам понадобится 80 млн долларов. Эти машины предназначены для военных целей. Какой-то сообразительный работник Пентагона придумал патриотические названия для первых трех машин: red, white и blue (красный, белый и синий — цвета флага США). Первая машина, выполнявшая 10^{14} операций с плавающей точкой, называлась **Option Red** (Sandia National Laborotary, декабрь 1996), вторая — **Option Blue** (1999), а третья — **Option White** (2000). Ниже мы будем рассматривать первую из этих машин, Option Red.

Машина Option Red состоит из 4608 узлов, которые организованы в трехмерную сетку. Процессоры запакованы на платах двух разных типов. Платы **kestrel** используются в качестве вычислительных узлов, а платы **eagle** используются для сервисных, дисковых, сетевых узлов и узлов загрузки. Машина содержит 4536 вычислительных узлов, 32 сервисных узла, 32 дисковых узла, 6 сетевых узлов и 2 узла загрузки.

Плата kestrel (рис. 8.30, *а)* содержит 2 логических узла, каждый из которых включает 2 процессора Pentium Pro на 200 МГц и разделенное ОЗУ на 64 Мбайт. Каждый узел kestrel содержит собственную 64-битную локальную шину и собственную микросхему **NIC (Network Interface Chip — сетевой адаптер).** Две микросхемы NIC связаны вместе, поэтому только одна из них подсоединена к сети, что делает систему более компактной. Платы eagle также содержат процессоры Pentium Pro, но всего два на каждую плату. Кроме того, они отличаются высокой производительностью процесса ввода-вывода.

Платы связаны в виде решетки 32x38x2 в виде двух взаимосвязанных плоскостей 32x38 (размер решетки продиктован целями компоновки, поэтому не во всех узлах решетки находятся платы). В каждом узле находится маршрутизатор с шестью каналами связи: вперед, назад, вправо, влево, с другой плоскостью и с платой kerstel или eagle. Каждый канал связи может передавать информацию одновременно в обоих направлениях со скоростью 400 Мбайт/с. Применяется маршрутизация «червоточина», чтобы сократить время ожидания.

Применяется пространственная маршрутизация, когда пакеты сначала потенциально перемещаются в другую плоскость, затем вправо-влево, затем вперед-назад и, наконец, в нужную плоскость, если они еще не оказались в нужной плоскости. Два перемещения между плоскостями нужны для повышения отказоустойчивости. Предположим, что пакет нужно переместить в соседний узел, находящийся впереди исходного, но канал связи между ними поврежден. Тогда сообщение можно отправить в другую плоскость, затем на один узел вперед, а затем обратно в первую плоскость, минуя таким образом поврежденный канал связи.



а *б*

Рис. 8.30. Система Intel/Sandia Option Red: плата kestrel (а); сеть (б)

Систему можно логически разделить на 4 части: сервис, вычисление, ввод-вывод и система. Сервисные узлы — это машины UNIX общего назначения с разделением времени, которые позволяют программистам писать и отлаживать свои программы. Вычислительные узлы запускают большие приложения. Они запускают не всю систему UNIX, а только микроядро, которое называется кугуаром (coguar)[1]. Узлы ввода-вывода управляют 640 дисками, содержащими более 1 Тбайт данных. Есть два независимых набора узлов ввода-вывода. Узлы первого типа предназначены для секретной военной работы, а узлы второго типа — для несекретной работы. Эти два набора вводятся и удаляются из системы вручную, поэтому в каждый момент времени подсоединен только один набор узлов, чтобы предотвратить утечку информации с секретных дисков на несекретные диски. Наконец, системные узлы используются для загрузки системы.

COW — Clusters of Workstations (кластеры рабочих станций)

Второй тип мультикомпьютеров — это системы **COW (Cluster of Workstations — кластер рабочих станций)** или **NOW (Network of Workstations — сеть рабочих станций)** [8,90]. Обычно он состоит из нескольких сотен персональных компьютеров или рабочих станций, соединенных посредством сетевых плат. Различие между MPP и COW аналогично разнице между большой вычислительной машиной и персональным компьютером. У обоих есть процессор, ОЗУ, диски, операционная система и т. д. Но в большой вычислительной машине все это работает гораздо быстрее (за исключением, может быть, операционной системы). Однако они применяются и управляются по-разному. Это же различие справедливо для MPP и COW.

Процессоры в MPP — это обычные процессоры, которые любой человек может купить. В системе T3E используются процессоры Alpha; в системе Option Red — процессоры Pentium Pro. Ничего специфического. Используются обычные динамические ОЗУ и система UNIX.

[1] Кугуар (или пума), как известно, очень быстрое животное. Видимо, этот термин был использован с целью отметить высокое быстродействие микроядра. — *Примеч. научи, ред.*

Исторически система MPP отличалась высокоскоростной сетью. Но с появлением коммерческих высокоскоростных сетей это отличие начало сглаживаться. Например, исследовательская группа автора данной книги собрала систему COW, которая называется **DAS (Distributed ASCII Supercomputer).** Она состоит из 128 узлов, каждый из которых содержит процессор Pentium Pro на 200 МГц и ОЗУ на 128 Мбайт (см. http://www.cs.vu.nl/~baL/das.htmt). Узлы организованы в 2-мерный тор. Каналы связи могут передавать информацию со скоростью 160 Мбайт/с в обоих направлениях одновременно. Эти характеристики практически не отличаются от характеристик машины Option Red: скорость передачи информации по каналам связи в два раза ниже, но размер ОЗУ каждого узла в два раза больше. Единственное существенное различие состоит в том, что бюджет Sandia был значительно больше. Технически эти две системы практически не различаются.

Преимущество системы COW над MPP в том, что COW полностью состоит из доступных компонентов, которые можно купить. Эти части выпускаются большими партиями. Эти части, кроме того, существуют на рынке с жесткой конкуренцией, из-за которой производительность растет, а цены падают. Вероятно, системы COW постепенно вытеснят MMP, подобно тому как персональные компьютеры вытеснили большие вычислительные машины, которые применяются теперь только в специализированных областях.

Существует множество различных видов COW, но доминируют два из них: централизованные и децентрализованные. Централизованные системы COW представляют собой кластер рабочих станций или персональных компьютеров, смонтированных в большой блок в одной комнате. Иногда они компонуются более компактно, чем обычно, чтобы сократить физические размеры и длину кабеля. Как правило, эти машины гомогенны и не имеют никаких периферических устройств, кроме сетевых карт и, возможно, дисков. Гордон Белл (Gordon Bell), разработчик PDP-11 и VAX, назвал такие машины **«автономными рабочими станциями»** (поскольку у них не было владельцев).

Децентрализованная система COW состоит из рабочих станций или персональных компьютеров, которые раскиданы по зданию или по территории учреждения. Большинство из них простаивают много часов в день, особенно ночью. Обычно они связаны через локальную сеть. Они гетерогенны и имеют полный набор периферийных устройств. Самое важное, что многие компьютеры имеют своих владельцев.

Планирование

Возникает вопрос: чем отличается децентрализованная система COW от локальной сети, соединяющей пользовательские машины? Отличие связано с программным обеспечением и не имеет никакого отношения к аппаратному обеспечению. В локальной сети пользователи работают с персональными машинами и используют только их для своей работы. Децентрализованная система COW, напротив, является общим ресурсом, которому пользователи могут поручить работу, требующую для выполнения нескольких процессоров. Чтобы система COW могла обрабатывать запросы от нескольких пользователей, каждому из которых нужно несколько процессоров, этой системе необходим планировщик заданий.

Рассмотрим самую простую модель планирования. Должно быть известно, сколько процессоров нужно для каждой работы (задачи). Тогда задачи выстраиваются в порядке FIFO («первым вошел — первым вышел») (рис. 8.31, *а)*. Когда первая задача начала выполняться, происходит проверка, есть ли достаточное количество процессоров для выполнения задачи, следующей по очереди. Если да, то она тоже начинает выполняться и т. д. Если нет, то система ждет, пока не появится достаточное количество процессоров. В нашем примере система COW содержит 8 процессоров, но она вполне могла бы содержать 128 процессоров, расположенных в блоках по 16 процессоров (получилось бы 8 групп процессоров) или в какой-нибудь другой комбинации.

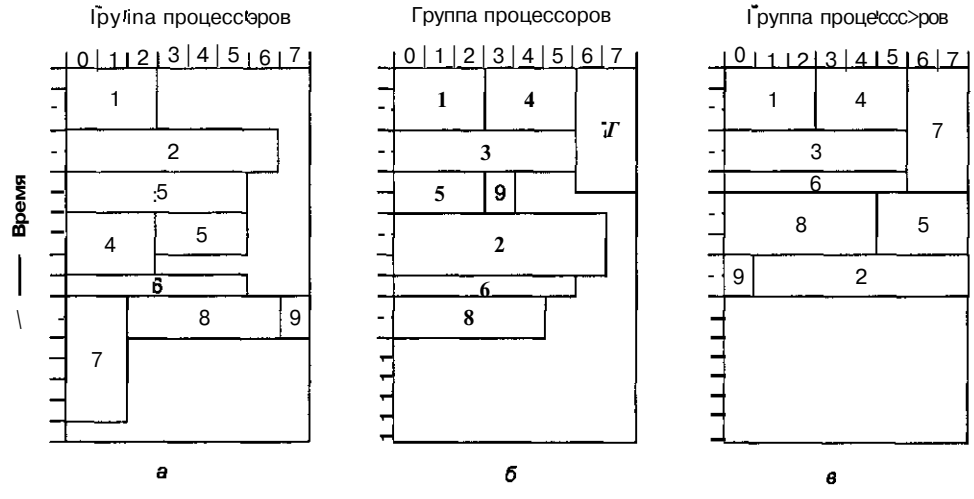


Рис. 8.31. Планирование работы в системе COW: FIFO («первым вошел — первым вышел») (а); без блокировки начала очереди (б); заполнение прямоугольника «процессоры-время» (в). Серым цветом показаны свободные процессоры

В глубже разработанном алгоритме задачи, которые не соответствуют количеству имеющихся в наличии процессоров, пропускаются и берется первая задача, для которой процессоров достаточно. Всякий раз, когда завершается выполнение задачи, очередь из оставшихся задач проверяется в порядке «первым вошел — первым вышел». Результат применения этого алгоритма изображен на рис. 8.31, *б.*

Еще более сложный алгоритм требует, чтобы было известно, сколько процессоров нужно для каждой задачи и сколько минут займет ее выполнение. Располагая такой информацией, планировщик заданий может попытаться заполнить прямоугольник «процессоры—время». Это особенно эффективно, когда задачи представлены на рассмотрение днем, а выполняться будут ночью. В этом случае планировщик заданий получает всю информацию о задачах заранее и может выполнять их в оптимальном порядке, как показано на рис. 8.31, *в.*

Коммерческие сети межсоединений

В этом разделе мы рассмотрим некоторые технологии связи. Наш первый пример — система Ethernet. Существует три версии этой системы: classic Ethernet, fast

Ethernet и gigabit Ethernet. **Они** работают со скоростью 10,100 и 1000 Мбит/с (1,25, 12,5 и 125 Мбайт/с)[1] соответственно. Все они совместимы относительно среды, формата пакетов и протоколов[2]. Отличие только в производительности.

Каждый компьютер в сети Ethernet содержит микросхему Ethernet, обычно на съемной плате. Изначально провод из платы вводился в середину толстого медного кабеля, это называлось **«зуб вампира».** Позднее появились более тонкие кабели и Т-образные коннекторы. В любом случае платы Ethernet на всех машинах соединены электрически, как будто они соединены пайкой. Схема подсоединения трех машин к сети Ethernet изображена на рис. 8.32, *а.*

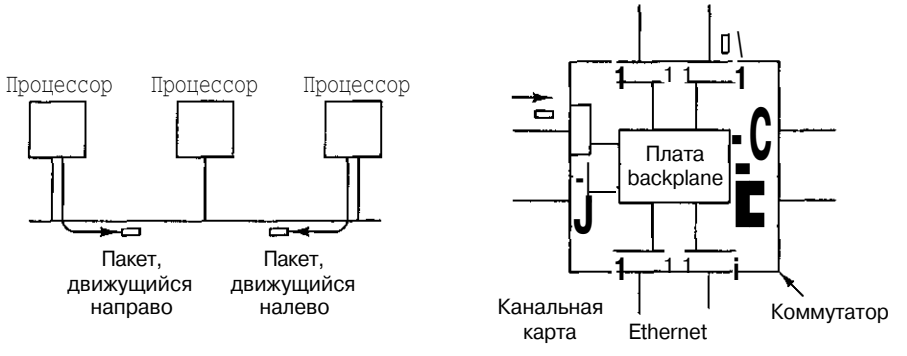


Рис. 8.32. Три компьютера в сети Ethernet (а); коммутатор Ethernet (б)

В соответствии с протоколом Ethernet, если машине нужно послать пакет, сначала она должна проверить, не совершает ли передачу в данный момент какая-либо другая машина. Если кабель свободен, то машина просто посылает пакет. Если кабель занят, то машина ждет окончания передачи и только после этого посылает пакет. Если две машины начинают передачу пакета одновременно, происходит конфликтная ситуация. Обе машины определяют, что произошла конфликтная ситуация, останавливают передачу, затем останавливаются на произвольный период времени и пробуют снова. Если конфликтная ситуация случается во второй раз, они снова останавливаются и снова начинают передачу пакетов, удваивая среднее время ожидания с каждой последующей конфликтной ситуацией.

Дело в том, что «зубы вампира» легко ломаются, а определить неполадку в кабеле очень трудно. По этой причине появилась новая разработка, в которой кабель из каждой машины подсоединяется к **сетевому концентратору (хабу).** По суще-

[1] Соотнесение автором скоростных показателей упоминаемых технологий, выраженных отношением скорости передачи бит/с, с отношениями Мбайт/с неправомочно. Ни одна их этих технологий не позволяет передать по сети соответствующее количество байтов за секунду. Даже теоретически возможная скорость для стандарта Ethernet лежит в интервале 800-850 Кбайт/с. Дело в том, что для передачи информации необходимо использовать служебные коды, к тому же передача осуществляется кадрами (фреймами) относительно небольшой длины (1500 байтов), после чего сетевой адаптер обязательно должен освободить среду передачи на фиксированный промежуток времени (с тем, чтобы другие сетевые адаптеры тоже могли воспользоваться средой передачи). — *Примеч. научи, ред.*

[2] Это высказывание является чересчур смелым. В действительности совместимость имеет очень много ограничений, и общим у них следует считать метод доступа к среде передачи. — *Примеч. научн. ред.*

ству, это то же самое, что и в первой разработке, но производить ремонт здесь проще, поскольку кабели можно отсоединять от сетевого концентратора по очереди, пока поврежденный кабель не будет изолирован.

Третья разработка — **Ethernet с использованием коммутаторов** — показана на рис. 8.32, *б*. Здесь сетевой концентратор заменен устройством, содержащим высокоскоростную плату backplane, к которой можно подсоединять **канальные карты.** Каждая канальная карта принимает одну или несколько сетей Ethernet, и разные карты могут воспринимать разные скорости, поэтому classic, fast и gigabit Ethernet могут быть связаны вместе.

Когда пакет поступает в канальную карту, он временно сохраняется там в буфере, пока канальная карта не отправит запрос и не получит доступ к плате backplane, которая функционирует почти как шина. Если пакет был перемещен в канальную карту, к которой подсоединена целевая машина, он может направляться к этой машине. Если каждая канальная карта содержит только один Ethernet и этот Ethernet имеет только одну машину, конфликтных ситуаций больше не возникнет, хотя пакет может быть потерян из-за переполнения буфера в канальной карте. Gigabit Ethernet с использованием коммутаторов с одной машиной на Ethernet и высокоскоростной платой backplane имеет потенциальную производительность (по крайней мере, это касается пропускной способности) в 4 раза меньше, чем каналы связи в машине T3E, но стоит значительно дешевле.

Но при большом количестве канальных карт обычная плата backplane не сможет справляться с такой нагрузкой, поэтому необходимо подсоединить несколько машин к каждой сети Ethernet, вследствие чего опять возникнут конфликтные ситуации. Однако с точки зрения соотношения цены и производительности сеть на основе gigabit Ethernet с использованием коммутаторов — серьезный конкурент на компьютерном рынке.

Следующая технология связи, которую мы рассмотрим, — это **ATM (Asynchronous Transfer Mode — асинхронный режим передачи).** Технология ATM была разработана международным консорциумом телефонных компаний в качестве замены существующей телефонной системы на новую, полностью цифровую. Основная идея проекта состояла в том, чтобы каждый телефон и каждый компьютер в мире связать с помощью безошибочного цифрового битового канала со скоростью передачи данных 155 Мбит/с (позднее 622 Мбит/с). Но осуществить это на практике оказалось не так просто. Тем не менее многие компании сейчас выпускают съемные платы для персональных компьютеров со скоростью передачи данных 155 Мбит/с или 622 Мбит/с. Вторая скорость, **OC-12,** хорошо подходит для мультикомпьютеров.

Провод или стекловолокно, отходящее от платы ATM, переходит в переключатель ATM — устройство, похожее на коммутатор Ethernet. В него тоже поступают пакеты и сохраняются в буфере в канальных картах, а затем поступают в исходящую канальную карту для передачи в пункт назначения. Однако у Ethernet и ATM есть существенные различия.

Во-первых, поскольку ATM была разработана для замещения телефонной системы, она представляет собой сеть с маршрутизацией информации. Перед отправкой пакета в пункт назначения исходная машина должна установить виртуальную цепь от исходного пункта через один или несколько коммутаторов ATM в конеч-

ный пункт. На рис. 8.33. показаны две виртуальные цепи. В сети Ethernet, напротив, нет никаких виртуальных цепей. Поскольку установка виртуальной цепи занимает некоторое количество времени, каждая машина в мультикомпьютере должна устанавливать виртуальную цепь со всеми другими машинами при запуске и использовать их при работе. Пакеты, отправленные по виртуальной цепи, всегда будут доставлены в правильном порядке, но буферы канальных карт могут переполняться, как и в сети Ethernet с коммутаторами, поэтому доставка не гарантируется.

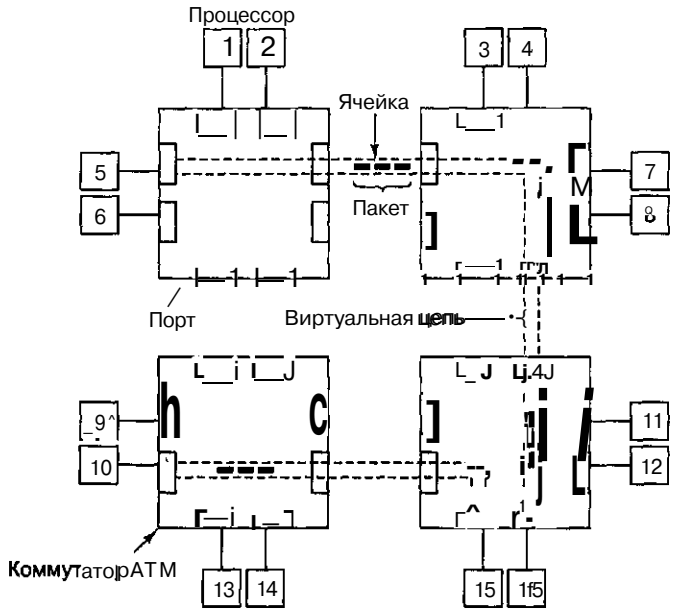


Рис. 8.33. 16 процессоров, связанных четырьмя переключателями АТМ. Пунктиром показаны две виртуальные цепи (канала)

Во-вторых, Ethernet может передавать целые пакеты (до 1500 байтов данных) одним блоком. В АТМ все пакеты разбиваются на ячейки по 53 байта. Пять из этих байтов — это поля заголовка, которые сообщают, какой виртуальной цепи принадлежит ячейка, что это за ячейка, каков ее приоритет, а также некоторые другие сведения. Полезная нагрузка составляет 48 байтов. Разбиение пакетов на ячейки и их компоновку в конце пути совершает аппаратное обеспечение.

Наш третий пример — сеть Myrinet — съемная плата, которая производится одной калифорнийской компанией и пользуется популярностью у разработчиков систем COW [18]. Здесь используется та же модель, что и в Ethernet и АТМ, где каждая съемная плата подсоединяется к коммутатору, а коммутаторы могут соединяться в любой топологии. Каналы связи сети Myrinet дуплексные, они передают информацию со скоростью 1,28 Гбит/с в обоих направлениях. Размер пакетов неограничен, а каждый коммутатор представляет собой полное пересечение, что дает малое время ожидания и высокую пропускную способность.

Myrinet пользуется популярностью у разработчиков систем COW, поскольку платы в этой сети содержат программируемый процессор и большое ОЗУ. Хотя

Myrinet появилась со своей стандартной операционной системой, многие исследовательские группы уже разработали свои собственные операционные системы. У них появились дополнительные функции и повысилась производительность (см., например, [17,107,155]). Из типичных особенностей можно назвать защиту, управление потоком, надежное широковещание и мультивещание, а также возможность запускать часть кода прикладной программы на плате.

Связное программное обеспечение для мультикомпьютеров

Для программирования мультикомпьютера требуется специальное программное обеспечение (обычно это библиотеки), чтобы обеспечить связь между процессами и синхронизацию. В этом разделе мы расскажем о таком программном обеспечении. Отметим, что большинство этих программных пакетов работают в системах MPP и COW.

В системах с передачей сообщений два и более процессов работают независимо друг от друга. Например, один из процессов может производить какие-либо данные, а другой или несколько других процессов могут потреблять их. Если у отправителя есть еще данные, нет никакой гарантии, что получатель (получатели) готов принять эти данные, поскольку каждый процесс запускает свою программу.

В большинстве систем с передачей сообщений имеется два примитива send и receive, но возможны и другие типы семантики. Ниже даны три основных варианта:

1. Синхронная передача сообщений.
2. Буферная передача сообщений.
3. Неблокируемая передача сообщений.

Синхронная передача сообщений. Если отправитель выполняет операцию send, а получатель еще не выполнил операцию receive, то отправитель блокируется до тех пор, пока получатель не выполнит операцию receive, а в это время сообщение копируется. Когда к отправителю возвращается управление, он уже знает, что сообщение было отправлено и получено. Этот метод имеет простую семантику и не требует буферизации. Но у него есть большой недостаток: отправитель блокируется до тех пор, пока получатель не примет и не подтвердит прием сообщения.

Буферная передача сообщений. Если сообщение отправляется до того, как получатель готов его принять, это сообщение временно сохраняется где-либо, например в почтовом ящике, и хранится там, пока получатель не возьмет его оттуда. При таком подходе отправитель может продолжать работу после операции send, даже если получатель в этот момент занят. Поскольку сообщение уже отправлено, отправитель может снова использовать буфер сообщений сразу же. Такая схема сокращает время ожидания. Вообще говоря, как только система отправила сообщение, отправитель может продолжать работу. Однако нет никаких гарантий, что сообщение было получено. Даже при надежной системе коммуникаций получатель мог сломаться еще до получения сообщения.

Неблокируемая передача сообщений. Отправитель может продолжать работу сразу после вызова. Библиотека только сообщает операционной системе, что

она сделает вызов позднее, когда у нее будет время. В результате отправитель вообще не блокируется. Недостаток этого метода состоит в том, что когда отправитель продолжает работу после совершения операции send, он не может снова использовать буфер сообщений, так как есть вероятность, что сообщение еще не отправлено. Отправитель каким-то образом должен определять, когда он может снова использовать буфер. Например, можно опрашивать систему или совершать прерывание, когда буфер имеется в наличии. В обоих случаях программное обеспечение очень сложное.

В следующих двух разделах мы рассмотрим две популярные системы с передачей сообщений, которые применяются во многих мультикомпьютерах: PVM и MPI. Существуют и другие системы, но эти две наиболее распространенные.

PVM — виртуальная машина параллельного действия

PVM (Parallel Virtual Machine — виртуальная машина параллельного действия) — это система с передачей сообщений, изначально разработанная для машин COW с операционной системой UNIX [45, 142]. Позднее она стала применяться в других машинах, в том числе в системах MPP. Это самодостаточная система с управлением процессами и системой ввода-вывода.

PVM состоит из двух частей: библиотеки, вызываемой пользователем, и «сторожевого» процесса, который работает постоянно на каждой машине в мультикомпьютере. Когда PVM начинает работу, она определяет, какие машины должны быть частью ее виртуального мультикомпьютера. Для этого она читает конфигурационный файл. «Сторожевой» процесс запускается на каждой из этих машин. Машины можно добавлять и убирать, вводя команды на консоли PVM.

Можно запустить n параллельных процессов с помощью команды

```
spawn -count n prog
```

Каждый процесс запустит *prog*. PVM решает, куда поместить процессы, но пользователь может сам подменять их с помощью аргументов команды spawn. Процессы могут запускаться из работающего процесса — для этого нужно вызвать процедуру *Pvm_spawn*. Процессы могут быть организованы в группы, причем состав групп может меняться во время выполнения программы.

Взаимодействие в машине PVM осуществляется с помощью примитивов для передачи сообщений таким образом, чтобы взаимодействовать могли машины с разными системами счисления. Каждый процесс PVM в каждый момент времени имеет один активный пересылочный буфер и один активный приемный буфер. Отправляя сообщение, процесс вызывает библиотечные процедуры, запаковывающие значения с самоописанием в активный пересылочный буфер, чтобы получатель мог узнать их и преобразовать в исходный формат.

Когда сообщение скомпоновано, отправитель вызывает библиотечную процедуру *pvm_send,* которая представляет собой блокирующий сигнал send. Получатель может поступить по-разному. Во-первых, он может вызвать процедуру *pvm_recv,* которая блокирует получателя до тех пор, пока не придет подходящее сообщение. Когда вызов возвратится, сообщение будет в активном приемном буфере. Оно может быть распаковано и преобразовано в подходящий для данной машины формат с помощью набора распаковывающих процедур. Во-вторых, получатель может вызвать процедуру *pvmjtrecv,* которая блокирует получателя на

определенный промежуток времени, и если подходящего сообщения за это время не пришло, он разблокируется. Процедура нужна для того, чтобы не заблокировать процесс навсегда. Третий вариант — процедура *pvmjirecv,* которая сразу же возвращает значение — это может быть либо сообщение, либо указание на отсутствие сообщений. Вызов можно повторять, чтобы опрашивать входящие сообщения.

Помимо всех этих примитивов PVM поддерживает широковещание (процедура*pvm_bcast)* и мультивещание (процедура*pvm_mcasi).* Первая процедура отправляет сообщение всем процессам в группе, вторая посылает сообщение только некоторым процессам, входящим в определенный список.

Синхронизация между процессами осуществляется с помощью процедуры *pvmjbarrier.* Когда процесс вызывает эту процедуру, он блокируется до тех пор, пока определенное число других процессов не достигнет барьера и они не вызовут эту же процедуру. Существуют другие процедуры для управления главной вычислительной машиной, группами, буферами, для передачи сигналов, проверки состояния и т. д. PVM — это простой, легкий в применении пакет, имеющийся в наличии в большинстве компьютеров параллельного действия, что и объясняет его популярность.

MPI — интерфейс с передачей сообщений

Следующий пакет для программирования мультикомпьютеров — **MPI (Message-Passing Interface — интерфейс с передачей сообщений).** MPI гораздо сложнее, чем PVM. Он содержит намного больше библиотечных вызовов и намного больше параметров на каждый вызов. Первая версия MPI, которая сейчас называется MPI-1, была дополнена второй версией, MPI-2, в 1997 году. Ниже мы в двух словах расскажем о MPI-1, а затем посмотрим, что нового появилось в MPI-2. Более подробную информацию об MPI см. в книгах [52, 134].

MPI-1, в отличие от PVM, никак не связана с созданием процессов и управлением процессами. Создавать процессы должен сам пользователь с помощью локальных системных вызовов. После создания процессы организуются в группы, которые уже не изменяются. Именно с этими группами и работает MPI.

В основе MPI лежат 4 понятия: коммуникаторы, типы передаваемых данных, операции коммуникации и виртуальные топологии. Коммуникатор — это группа процессов плюс контекст. Контекст — это метка, которая идентифицирует что-либо (например, фазу выполнения). В процессе отправки и получения сообщений контекст может использоваться для того, чтобы несвязанные сообщения не мешали друг другу.

Сообщения могут быть разных типов: символьные, целочисленные (short, regular и long integers), с обычной и удвоенной точностью, с плавающей точкой и т. д. Можно образовать новые типы сообщений из уже существующих.

MPI поддерживает множество операций коммуникации. Ниже приведена операция, которая используется для отправки сообщений:

```
MPI_Send(buffer. count. data_type. destination, tag. communicator)
```

Этот вызов отправляет содержимое буфера *(buffer)* с элементами определенного типа *(datatype)* в пункт назначения. *Count* — это число элементов буфера. Поле *tag* помечает сообщение; получатель может сказать, что он будет принимать

сообщение только с данным тегом. Последнее поле показывает, к какой группе процессов относится целевой процесс (поле *destination* — это просто индекс списка процессов из определенной группы). Соответствующий вызов для получения сообщения таков:

```
MPI_Recv(&buffer, count, datajtype, source, tag. communicator, Sstatus)
```

В нем сообщается, что получатель ищет сообщение определенного типа из определенного источника с определенным тегом.

MPI поддерживает 4 основных типа коммуникации. Первый тип синхронный. В нем отправитель не может начать передачу данных, пока получатель не вызовет процедуру MPI_Recv. Второй тип — коммуникация с использованием буфера. Ограничение для первого типа здесь недействительно. Третий тип стандартный. Он зависит от реализации и может быть либо синхронным, либо с буфером. Четвертый тип сходен с первым. Здесь отправитель требует, чтобы получатель был доступен, но без проверки. Каждый из этих примитивов бывает двух видов: блокирующим и неблокирующим, что в сумме дает 8 примитивов. Получение может быть только в двух вариантах: блокирующим и неблокирующим.

MPI поддерживает коллективную коммуникацию — широковещание, распределение и сбор данных, обмен данными, агрегацию и барьер. При любых формах коллективной коммуникации все процессы в группе должны делать вызов, причем с совместимыми параметрами. Если этого сделать не удается, возникает ошибка. Например, процессы могут быть организованы в виде дерева, в котором значения передаются от листьев к корню, подчиняясь определенной обработке на каждом шаге (это может быть сложение значений или взятие максимума). Это типичная форма коллективной коммуникации.

Четвертое основное понятие в MPI — **виртуальная топология,** когда процессы могут быть организованы в дерево, кольцо, решетку, тор и т. д. Такая организация процессов обеспечивает способ наименования каналов связи и облегчает коммуникацию.

В MPI-2 были добавлены динамические процессы, доступ к удаленной памяти, неблокирующая коллективная коммуникация, расширяемая поддержка процессов ввода-вывода, обработка в режиме реального времени и многие другие особенности. В научном сообществе идет война между лагерями MPI и PVM. Те, кто поддерживают PVM, утверждают, что эту систему проще изучать и легче использовать. Те, кто на стороне MPI, утверждают, что эта система выполняет больше функций и, кроме того, она стандартизована, что подтверждается официальным документом.

Совместно используемая память на прикладном уровне

Из наших примеров видно, что мультикомпьютеры можно расширить до гораздо больших размеров, чем мультипроцессоры. Sun Enterprise 10000, где максимальное число процессоров — 64, и NUMA-Q, где максимальное число процессоров — 256, являются представителями больших мультипроцессоров UMA и NUMA соответственно. А машины T3E и Option Red содержат 2048 и 9416 процессоров соответственно.

Однако мультикомпьютеры не имеют совместно используемой памяти на архитектурном уровне. Это и привело к появлению таких систем с передачей сообщений, как PVM и MPI. Большинство программистов предпочитают иллюзию совместно используемой памяти, даже если ее на самом деле не существует. Если удастся достичь этой цели, мы сразу получим дешевое аппаратное обеспечение больших размеров плюс обеспечим легкость программирования.

Многие исследователи пришли к выводу, что общая память на архитектурном уровне может быть нерасширяемой, но зато существуют другие способы достижения той же цели. Из рисунка 8.3 видно, что есть и другие уровни, на которых можно ввести совместно используемую память. В следующих разделах мы рассмотрим, как в мультикомпьютере можно ввести совместно используемую память в модель программирования при отсутствии ее на уровне аппаратного обеспечения.

Распределенная совместно используемая память

Один из классов систем с общей памятью на прикладном уровне — это системы со страничной организацией памяти. Такая система называется **DSM (Distributed Shared Memory — распределенная совместно используемая память).** Идея проста: ряд процессоров в мультикомпьютере разделяет общее виртуальное адресное пространство со страничной организацией. Самый простой вариант — когда каждая страница содержится в ОЗУ ровно для одного процессора. На рис. 8.34, *а* мы видим общее виртуальное адресное пространство, которое состоит из 16 страниц и распространяется на 4 процессора.

Когда процессор обращается к странице в своем локальном ОЗУ, чтение и запись происходят без задержки. Однако если процессор обращается к странице из другого ОЗУ, происходит ошибка из-за отсутствия страницы. Только в этом случае отсутствующая страница берется не с диска. Вместо этого операционная система посылает сообщение в узел, в котором находится данная страница, чтобы преобразовать ее и отправить к процессору. После получения страницы она преобразуется в исходное состояние, а приостановленная команда выполняется заново, как и при обычной ошибке из-за отсутствия страницы. На рис. 8.34, *б* мы видим ситуацию после того, как процессор 0 получил ошибку из-за отсутствия страницы 10, которая была передана из процессора 1 в процессор 0.

Впервые идея была реализована в машине IVY [83, 84]. В результате в мультикомпьютере появилась память совместного использования, согласованная по последовательности. В целях улучшения производительности возможны оптимизации. Первая оптимизация, появившаяся в IVY, — страницы, предназначенные только для чтения, могли присутствовать в нескольких узлах одновременно. В случае ошибки из-за отсутствия страницы в запрашивающую машину посылается копия этой страницы, но оригинал остается на месте, поскольку нет никакой опасности конфликтов. На рисунке 8.34, *в* показана ситуация, когда два процессора делят общую страницу, предназначенную только для чтения (это страница 10).

Но даже при такой оптимизации трудно достичь высокой производительности, особенно когда один процесс записывает несколько слов вверху какой-либо страницы, а другой процесс в другом процессоре в это же время записывает несколько слов внизу той же страницы. Поскольку существует только одна копия этой стра-

ницы, страница постоянно должна передаваться туда и обратно. Эта ситуация называется **ложным совместным использованием.**

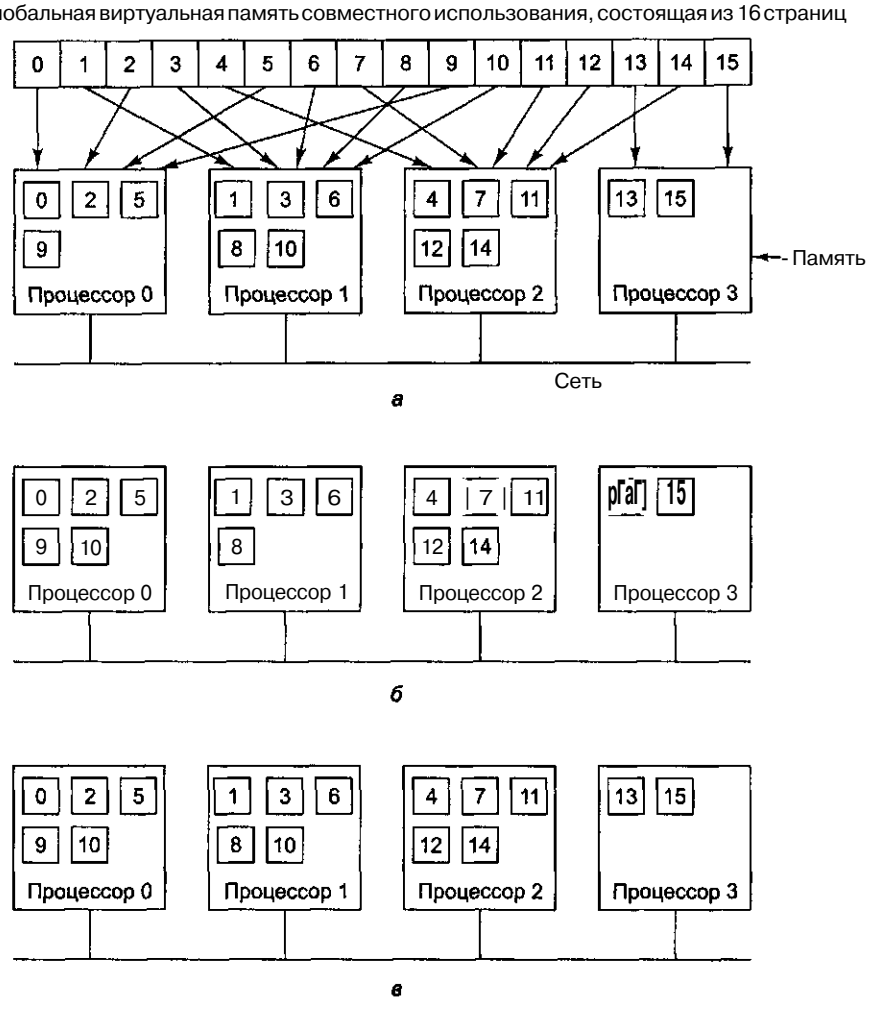


Рис. 8.34. Виртуальное адресное пространство, состоящее из 16 страниц, которые распределены между четырьмя узлами в мультикомпьютере: исходное положение (а); положение после обращения процессора 0 к странице 10 (б); положение после обращения процессора 0 к странице 10 (в); в данном случае эта страница предназначена только для чтения

Проблему ложного совместного использования можно разрешить по-разному. Например, можно отказаться от согласованности по последовательности в пользу свободной согласованности [6]. Потенциально записываемые страницы могут присутствовать в нескольких узлах одновременно, но перед записью процесс должен совершить операцию acquire, чтобы сообщить о своем намерении. В этот момент все копии, кроме последней, объявляются недействительными. Нельзя делать

никаких копий до тех пор, пока не будет совершена операция release и после этого страница снова станет общей.

Второй способ оптимизации — изначально преобразовывать каждую записываемую страницу в режим «только для чтения». Когда в страницу запись производится впервые, система создает копию страницы, называемую **двойником.** Затем страница преобразуется в формат «для чтения и записи», и последующие записи могут производиться на полной скорости. Если позже произойдет ошибка из-за отсутствия страницы и страницу придется туда доставлять, между текущей страницей и двойником производится пословное сравнение. Пересылаются только те слова, которые были изменены, что сокращает размер сообщений.

Если возникает ошибка из-за отсутствия страницы, нужно определить, где находится отсутствующая страница. Возможны разные способы, в том числе каталоги, которые используются в машинах NUMA и COMA. Многие средства, которые используются в DSM, применимы к машинам NUMA и COMA, поскольку DSM — это программная реализация машины NUMA или COMA, в которой каждая страница трактуется как строка кэш-памяти.

DSM — это обширная область для исследования. Большой интерес представляют системы CASHMERE [76, 141], CRL [68], Shata [124] и Treadmarks [6, 87].

Linda

Системы DSM со страничной организацией памяти (например, IVY и Treadmarks) используют аппаратный блок управления памятью, чтобы закрывать доступ к отсутствующим страницам. Хотя пересылка только отдельных слов вместо всей страницы и влияет на производительность, страницы неудобны для совместного использования, поэтому применяются и другие подходы.

Один из таких подходов — система Linda, в которой процессы на нескольких машинах снабжены структурированной распределенной памятью совместного использования [21, 22]. Доступ к такой памяти осуществляется с помощью небольшого набора примитивных операций, которые можно добавить к существующим языкам (например, C или FORTRAN), в результате чего получатся параллельные языки, в данном случае C-Linda и FORTRAN-Linda.

В основе системы Linda лежит понятие абстрактного **пространства кортежей,** которое глобально по отношению ко всей системе и доступно всем процессам этой системы. Пространство кортежей похоже на глобальную память совместного использования, только с определенной встроенной структурой. Пространство содержит ряд **кортежей,** каждый из которых состоит из одного или нескольких полей. Для C-Linda поля могут содержать целые числа, числа типа long integers, числа с плавающей точкой, а также массивы (в том числе цепочки) и структуры (но не другие кортежи). В листинге 8.4. приведено 3 примера кортежей.

Над кортежами можно совершать 4 операции. Первая из них, out, помещает кортеж в пространство кортежей. Например, операция

```
outC'abc", 2, 5);
```

помещает кортеж («abc», 2,5) в пространство кортежей. Поля операции out обычно содержат константы, переменные и выражения, например

```
out("mathx-l". i. j, 3.14);
```

Это выражение помещает в пространство кортеж с четырьмя полями, причем второе и третье поля определяются по текущим значениям переменных i и j.

Вторая операция, in, извлекает кортеж из пространства. Обращение к кортежам ирогаъодатся. wo содержанию, л we wo жлтан vura адресу. X\оля операнда Vw шэтут содержать выражения или формальные параметры. Рассмотрим операцию

```
mCabc". 2, ? i);
```

Эта операция отыскивает кортеж, состоящий из цепочки «abc», целого числа 2 и третьего поля, которое может содержать любое целое число (предполагается, что i — это целое число). Найденный кортеж удаляется из пространства кортежей, а переменной i приписывается значение третьего поля. Когда два процесса выполняют одну и ту же операцию in одновременно, успешно выполнит эту операцию только один из них, если нет двух и более подходящих кортежей. Пространство кортежей может содержать много копий одного кортежа.

Листинг 8.4. Три кортежа системы Linda

```
("abc".2.5)
Cmatrix-1".1.6.3.14)
("family", "is sister". Carolyn, Elinor)
```

Алгоритм соответствия, который используется в операции in, довольно прост. Поля **шаблона** операции in сравниваются с соответствующими полями каждого кортежа в пространстве кортежей. Совпадение происходит, если удовлетворены следующие три условия:

1. Шаблон и кортеж имеют одинаковое количество полей.
2. Типы соответствующих полей совпадают.
3. Каждая константа или переменная в шаблоне совпадает с полем кортежа.

Формальные параметры, указываемые знаком вопроса, за которым следует имя переменной или типа, не участвуют в сравнении (за исключением проверки типа), хотя тем параметрам, которые содержат имя переменной, присваиваются значения после совпадения.

Если соответствующий кортеж отсутствует, процесс приостанавливается, пока другой процесс не вставит нужный кортеж, и в этот момент вызывающий процесс автоматически возобновит работу и найдет этот новый кортеж. Процессы блокируются и разблокируются автоматически, поэтому если один процесс собирается выводить кортеж, а другой — вводить его, то не имеет никакого значения, какой из процессов будет первым.

Третья операция, read, сходна с in, только она не удаляет кортеж из пространства. Четвертая операция, eval, параллельно оценивает свои параметры, а полученный в результате кортеж копируется в пространство кортежей. Этот механизм можно использовать для выполнения любых вычислений. Именно так создаются параллельные процессы в системе Linda.

Основной принцип программирования в системе Linda — это модель **replicated worker.** В основе этой модели лежит понятие **пакета задач** (task bag), которые нужно выполнить. Основной процесс начинается с выполнения цикла, содержащего операцию

```
out("task-bag",job):
```

При каждом прохождении цикла одна из задач выдается в пространство кортежей. Каждый рабочий процесс начинает работу с получения кортежа с описанием задачи, используя операцию

```
in("Task-bag". ?job):
```

Затем он выполняет эту задачу. По выполнении задачи он получает следующую. Новая задача тоже может быть помещена в пакет задач во время выполнения. Таким образом, работа динамически распределяется среди рабочих процессов, и каждый рабочий процесс занят постоянно.

Существуют различные реализации Linda в мультикомпьютерных системах. Во всех реализациях ключевым является вопрос о том, как распределить кортежи по машинам и как их находить. Возможные варианты — широковещание и каталоги. Дублирование — это тоже важный вопрос. Подробнее об этом см. [16].

Orca

Несколько другой подход к совместно используемой памяти на прикладном уровне в мультикомпьютере — в качестве общей совместно используемой единицы использовать полные объекты, а не просто кортежи. Объекты состоят из внутреннего (скрытого) состояния и процедур для оперирования этим состоянием. Поскольку программист не имеет прямого доступа к состоянию, появляется возможность совместного использования ресурсов машинами, которые не имеют общей физической памяти.

Одна из таких систем называется Orca [11, 13, 14]. Orca — это традиционный язык программирования (основанный на Modula 2), к которому добавлены две особенности — объекты и возможность создавать новые процессы. Объект Orca — это стандартный тип данных, аналогичный объекту в языке Java или пакету в языке Ada. Объект заключает в себе внутренние структуры данных и написанные пользователем процедуры, которые называются операциями. Объекты пассивны, то есть они не содержат потоков, которым можно посылать сообщения. Процессы получают доступ к внутренним данным объекта путем вызова его процедур.

Каждая процедура в Orca состоит из списка пар (предохранитель (guard), блок операторов). Предохранитель — это логическое выражение, которое может принимать значение «истина» *(true)* или «ложь *(false)*. Когда вызывается операция, все ее предохранители оцениваются в произвольном порядке. Если все они ложны *(false),* вызывающий процесс приостанавливается до тех пор, пока один из них не примет значение *true*. При нахождении предохранителя со значением *true* выполняется следующий за ним блок выражений. В листинге 8.5 показан объект *stack* с двумя операциями *push* и *pop*.

Листинг 8.5. Упрощенный объект stack в системе Orca с внутренними данными и двумя операциями

```
Object implementation stack;
   top:integer;                          #хранилище для стека
   stack:array[integer 0..N-l]of integer;

   operation pushOtem:integer);          #функция. которая ничего не
   begin                                 #возвращает
      stack[top]:=item;                  #помещаем элемент в стек
      top:=top+l:                        #увеличения указателя стека
   end:
```

```
    operation pop0 integer.              #функция, которая возвращает
    begin                                #целое число
        guard top>0 do                   Приостанавливает работу, если стек пуст
            top =top-1.                  #уменьшает указатель стека
            return stack[top]:           # возвращает вершину стека
        od.
    end

begin
    top =0,                              Инициализация
end
```

Как только определен объект *stack,* нужно объявить переменные этого типа:

```
s, t  stack.
```

Такая запись создает два стековых объекта и устанавливает переменную *top* в каждом объекте на 0. Целочисленную переменную к можно поместить в стек s с помощью выражения

```
s$push(k)
```

и т. д. Операция *pop* содержит предохранитель, поэтому попытка вытолкнуть переменную из пустого стека вызовет блокировку вызывающего процесса до тех пор, пока другой процесс не положит что-либо в стек.

Occa содержит **оператор ветвления** (fork statement) для создания нового процесса в процессоре, определяемом пользователем. Новый процесс запускает процедуру, названную в команде ветвления. Параметры, в том числе объект, могут передаваться новому процессу. Именно так объекты распределяются среди машин. Например, выражение

```
for I in I  n do fork foobar(s) on I; od;
```

порождает новый процесс на каждой из машин с 1 по n, запуская программу *foobar* в каждой из них. Поскольку эти n новых процессов (а также исходный процесс) работают параллельно, они все могут помещать элементы в общий стек s и выталкивать элементы из общего стека s, как будто все они работают на мультипроцессоре с памятью совместного использования.

Операции в объектах совместного использования атомарны и согласованы по последовательности. Если несколько процессов выполняют операции над одним общим объектом практически одновременно, система выбирает определенный порядок выполнения, и все процессы «видят» этот же порядок.

В системе Occa данные совместного использования совмещаются с синхронизацией не так, как в системах со страничной организацией памяти. В программах с параллельной обработкой нужны два вида синхронизации. Первый вид — взаимное исключение. Этот метод не позволяет двум процессам одновременно выполнять одну и ту же критическую область. В системе Occa каждая операция над общим объектом похожа на критическую область, поскольку система гарантирует, что конечный результат будет таким же, как если бы все критические области выполнялись последовательно одна за другой. В этом отношении объект Occa похож на распределенное контролирующее устройство [61].

Второй вид синхронизации — условная синхронизация, при которой процесс блокируется и ждет выполнения определенного условия. В системе Occa условная синхронизация осуществляется при помощи предохранителей. В примере, приве-

денном в листинге 8.5, процесс, который пытается вытолкнуть элемент из пустого стека, блокируется до появления в стеке элементов.

В системе Orca допускается копирование объектов, миграция и вызов операций. Каждый объект может находиться в одном из двух состояний: он может быть единственным, а может быть продублирован. В первом случае объект существует только на одной машине, поэтому все запросы отправляются туда. Продублированный объект присутствует на всех машинах, которые содержат процесс, использующий этот объект. Это упрощает операцию чтения (поскольку ее можно производить локально), но усложняет процесс обновления. При выполнении операции, которая изменяет продублированный объект, сначала нужно получить от центрального процесса порядковый номер. Затем в каждую машину, содержащую копию объекта, отправляется сообщение о необходимости выполнить эту операцию. Поскольку все такие обновления обладают порядковыми номерами, все машины просто выполняют операции в порядке номеров, что гарантирует согласованность по последовательности.

Globe

Большинство систем DSM, Linda и Orca работают в пределах одного здания или предприятия. Однако можно построить систему с совместно используемой памятью на прикладном уровне, которая может распространяться на весь мир. В системе Globe объект может находиться в адресном пространстве нескольких процессов одновременно, возможно, даже на разных континентах [72,154]. Чтобы получить доступ к данным общего объекта, пользовательские процессы должны пройти через его процедуры, поэтому для разных объектов возможны разные способы реализации. Например, можно иметь один экземпляр данных, который запрашивается по мере необходимости (это удобно для данных, часто обновляемых одним владельцем). Другой вариант — когда все данные находятся в каждой копии объекта, а сигналы об обновлении посылаются каждой копии в соответствии с надежным протоколом широковещания.

Цель системы Globe — работать на миллиард пользователей и содержать триллион объектов — делает эту систему амбициозной. Ключевыми моментами являются размещение объектов, управление ими, а также расширение системы. Система Globe содержит общую сеть, в которой каждый объект может иметь собственную стратегию дублирования, стратегию защиты и т. д.

Среди других широкомасштабных систем можно назвать Globus [40,41] и Legion [50,51], но они, в отличие от Globe, не создают иллюзию совместного использования памяти.

Краткое содержание главы

Компьютеры параллельной обработки можно разделить на две основные категории: SIMD и MIMD. Машины SIMD выполняют одну команду одновременно над несколькими наборами данных. Это массивно-параллельные процессоры и векторные компьютеры. Машины MIMD выполняют разные программы на разных машинах. Машины MIMD можно подразделить на мультипроцессоры, которые

совместно используют общую основную память, и мультикомпьютеры, которые не используют общую основную память. Системы обоих типов состоят из процессоров и модулей памяти, связанных друг с другом различными высокоскоростными сетями, по которым между процессорами и модулями памяти передаются пакеты запросов и ответов. Применяются различные топологии, в том числе решетки, торы, кольца и гиперкубы.

Для всех мультипроцессоров ключевым вопросом является модель согласованности памяти. Из наиболее распространенных моделей можно назвать согласованность по последовательности, процессорную согласованность, слабую согласованность и свободную согласованность. Мультипроцессоры можно строить с использованием отслеживающей шины, например, в соответствии с протоколом MESI. Кроме того, возможны различные сети, а также машины на основе каталога NUMAHCOMA.

Мультипроцессоры можно разделить на системы MPP и COW, хотя граница между ними произвольна. К системам MPP относятся Cray T3E и Intel/Sandia Option Red. В них используются запатентованные высокоскоростные сети межсоединений. Системы COW, напротив, строятся из таких стандартных деталей, как Ethernet, ATM и Myrinet.

Мультикомпьютеры часто программируются с использованием пакета с передачей сообщений (например, PVM или MPI). Оба пакета поддерживают библиотечные вызовы для отправки и получения сообщений. Оба работают поверх существующих операционных систем.

Альтернативный подход — использование памяти совместного использования на прикладном уровне (например, система DSM со страничной организацией, пространство кортежей в системе Linda, объекты в системах Orca и Globe). Система DSM моделирует совместно используемую память на уровне страниц, и в этом она сходна с машиной NUMA. Системы Linda, Orca и Globe создают иллюзию совместно используемой памяти с помощью кортежей, локальных объектов и глобальных объектов соответственно.

Вопросы и задания

1. Утром пчелиная матка созывает рабочих пчел и сообщает им, что сегодня им нужно собрать нектар ноготков. Рабочие пчелы вылетают из улья и летят в разных направлениях в поисках ноготков. Что это за система — SIMD или MIMD?
2. Вычислите диаметр сети для каждой топологии, изображенной на рис. 8.4.
3. Для каждой топологии, изображенной на рис. 8.4, определите степень отказоустойчивости. Отказоустойчивость — максимальное число каналов связи, после утраты которых сеть не будет разделена на две части.
4. Рассмотрим топологию двойной тор (см. рис. 8.4, е), расширенную до размера kxk. Каков диаметр такой сети? Подсказка: четное и нечетное к нужно рассматривать отдельно.

5. Представим сеть в форме куба 8x8x8. Каждый канал связи имеет дуплексную пропускную способность 1 Гбайт/с. Какова бисекционная пропускная способность этой сети?
6. Рассмотрим сеть в форме прямоугольной решетки размером 4 коммутатора в ширину и 3 коммутатора в высоту. В ней пакеты, исходящие из левого верхнего угла и направляющиеся в правый нижний угол, могут следовать по любому из нескольких возможных путей. Пронумеруйте верхний ряд коммутаторов с 1 по 4, следующий ряд — с 5 по 8, а нижний — с 9 по 12. Выпишете все пути, по которым пакеты перемещаются только вправо или вниз, начиная с коммутатора 1 и заканчивая коммутатором 12.
7. Закон Амдала ограничивает потенциальный коэффициент ускорения, достижимый в компьютере параллельного действия. Вычислите как функцию от f максимально возможный коэффициент ускорения, если число процессоров стремится к бесконечности. Каково значение этого предела для f=0,l?
8. На рисунке 8.10 показано, что расширение невозможно с шиной, но возможно с решеткой. Каждая шина или канал связи имеет пропускную способность Ь. Вычислите среднюю пропускную способность на каждый процессор для каждого из четырех случаев. Затем расширьте каждую систему до 64 процессоров и выполните те же вычисления. Чему равен предел, если число процессоров стремится к бесконечности?
9. Компьютерная компания выпускает системы, состоящие из n компьютеров с совместно используемой памятью, организованных в квадратную решетку. Однажды вице-президенту компании приходит в голову идея выпустить новый продукт: трехмерную решетку, в которой n компьютеров организованы в правильный куб (это возможно, например, для n=4096).
    1. Как это изменение повлияет на максимальное время ожидания?
    2. Как это изменение повлияет на общую пропускную способность?
10. Когда мы говорили о согласованности памяти, мы сказали, что модель согласованности — это вид контракта между программным обеспечением и памятью. Зачем нужен такой контракт?
11. Векторный процессор (например, Cray-1) содержит арифметические устройства с конвейерами из четырех стадий. Прохождение каждой стадии занимает 1 не. Сколько времени понадобится для сложения двух векторов из 1024 элементов?
12. Рассмотрим мультипроцессор с общей шиной. Что произойдет, если два процессора попытаются получить доступ к глобальной памяти в один и тот же момент?
13. Предположим, что по техническим причинам отслеживающий кэш может отслеживать только адресные линии и не может отслеживать информационные. Повлияет ли это изменение на протокол сквозной записи?
14. Рассмотрим простую модель мультипроцессорной системы с шиной без кэш-памяти. Предположим, что одна из каждых четырех команд обращается к памяти и что при каждом обращении к памяти шина занимается на все

время выполнения команды. Если шина занята, то запрашивающий процессор становится в очередь «первым вошел — первым вышел». Насколько быстрее будет работать система с 64 процессорами по сравнению с однопроцессорной системой?

15. Протокол MESI содержит 4 состояния. Каким из 4 состояний можно пожертвовать и каковы будут последствия каждого из четырех вариантов? Если бы вам пришлось выбрать только три состояния, какие бы вы выбрали?
16. Бывают ли в протоколе MESI такие ситуации, когда строка кэш-памяти присутствует в локальной кэш-памяти, но при этом все равно требуется транзакция шины? Если да, то опишите такую ситуацию.
17. Предположим, что к общей шине подсоединено n процессоров. Вероятность, что любой процессор пытается использовать шину в данном цикле, равна p. Какова вероятность, что:
    1. Шина свободна (0 запросов).
    2. Совершается ровно один запрос.
    3. Совершается более одного запроса.
18. Процессоры Sun Enterprise 10000 работают с частотой 333 МГц, а отслеживающие шины — с частотой 83,3 МГц. Если имеется 64 процессора, ясно, что шины не справятся с такой нагрузкой. Тем не менее машина работает. Объясните, почему.
19. В этой книге мы вычислили, что производительности координатного коммутатора было достаточно для обработки 167 млн отслеживаний/с, когда к нему подсоединено 16 плат, причем мы даже принимаем во внимание тот факт, что из-за конфликтных ситуаций на практике пропускная способность составляет 60% от теоретической. Небольшая машина Enterprise 10000 может содержать всего 4 платы (16 процессоров). Будет ли такая машина работать с полной скоростью?
20. Предположим, что провод между коммутатором 2A и коммутатором 3B в сети Omega поврежден. Какие именно элементы будут отрезаны друг от друга?
21. Области памяти, к которым часто происходят обращения, представляют большую проблему в многоступенчатых сетях. А являются ли они проблемой в системах с шинной организацией?
22. Сеть Omega соединяет 4096 процессоров RISC, время цикла каждого из которых составляет 60 не, с 4096 бесконечно быстрыми модулями памяти. Каждый переключательный элемент дает задержку 5 не. Сколько пустых циклов требуется для команды LOAD?
23. Рассмотрим машину с сетью Omega (см. рис. 8.22). Предположим, что программа и стек i хранятся в модуле памяти i. Какое незначительное изменение топологии может сильно повлиять на производительность? (IBM RP3 и BBN Butterfly используют эту измененную топологию.) Какой недостаток имеет новая топология по сравнению со старой?

24. В мультипроцессоре NUMA обращение к локальной памяти занимает 20 не, а к памяти другого процессора — 120 не. Программа во время выполнения совершает N обращений к памяти, 1% из которых — обращения к странице P. Изначально эта страница находится в удаленной памяти, а на копирование ее из локальной памяти требуется C не. При каких обстоятельствах эту страницу следует копировать локально, если ее не используют другие процессоры?
25. Система DASH содержит b байтов памяти, которые распределены между n кластерами. В каждом кластере содержится p процессоров. Размер строки кэш-памяти составляет c байтов. Напишите формулу для общего количества памяти, предоставленного каталогам (исключая два бита состояния на каждый элемент каталога).
26. Рассмотрим мультипроцессор CC-NUMA (см. рис. 8.24), содержащий 512 узлов по 8 Мбайт каждый. Если длина строки кэш-памяти составляет 64 байта, каков процент непроизводительных затрат для каталогов? Как повлияет увеличение числа узлов на непроизводительные затраты (они увеличатся, уменьшатся или останутся без изменений)?
27. На какую операцию в SCI требуется больше всего времени?
28. Мультипроцессор на базе SCI содержит 63 узла. Длина строки кэш-памяти составляет 32 байта, а общее адресное пространство составляет 2^{32} байтов. Размер кэш-памяти в каждом узле — 1 Мбайт. Сколько байтов нужно иметь в каждом кэш-каталоге?
29. Машина NUMA-Q 2000 содержит 63 узла. Предложите ввести 64 узла вместо 63. Почему компания Sequent в качестве максимума выбрала именно 63, а не 64?
30. В этой книге мы обсуждали 3 варианта примитива send — синхронный, блокирующий и неблокирующий. Предложите четвертый метод, который схож с блокирующей операцией send, но немного отличается по свойствам. Какое преимущество и какой недостаток имеет новый метод по сравнению с обычной блокирующей операцией send?
31. Рассмотрим компьютер, который работает в сети с аппаратным широковещанием (например, Ethernet). Почему важно соотношение операций чтения (которые не изменяют внутреннее состояние переменных) и операций записи (которые изменяют внутреннее состояние переменных)?
32. Многие вопросы, возникающие при разработке мультипроцессоров, возникают также при разработке совместно используемой памяти на прикладном уровне. Один из таких вопросов — выбор одной из двух политик: обновление или объявление недействительным. Какая политика используется в системе Orca?

Глава 9
Библиография

В предыдущих главах мы с разной степенью подробности обсуждали достаточно широкий ряд вопросов. Эта глава предназначена для читателей, которые заинтересованы в более глубоком изучении строения компьютеров. В разделе «Литература для дальнейшего чтения» содержится список литературы к каждой главе. В разделе «Алфавитный список литературы» приводится алфавитный список всех книг и статей, на которые мы ссылались в этой книге.

Литература для дальнейшего чтения

Вводная и неспециальная литература

1. Hamacher et al., *Computer Organization,* 4th ed.

   Традиционный учебник об организации компьютеров (процессоры, память, ввод-вывод, арифметика, периферийные устройства). Основные примеры — 68000 и Power PC.

2. Hayes, *Computer Architecture and Organization,* 3rd ed.

   Еще одна традиционная книга о компьютерной организации с уклоном к аппаратному обеспечению. В книге рассматриваются процессоры, тракт данных, микропрограммирование, конвейеры, организация памяти и процесс ввода-вывода.

3. Patterson and Hennessy, *Computer Organization and Design*.

   Эта книга почти на 1000 страниц содержит обширный материал о компьютерной архитектуре, в частности о разработке процессоров RISC. Упор делается на то, как достичь высокой производительности с помощью конвейеризации и других технологий.

4. Price, *A History of Calculating Machines*.

   Современные компьютеры восходят к машине Бэббиджа, созданной в XIX веке, но люди производили различные вычисления с самого начала цивилизации. В этой иллюстрированной статье прослеживается вся история счета, математики, календарей и вычислений с 3000 г. до н. э. до начала XX века.

5. Slater, *Portraits in Silicon*.

   Почему Деннис Ритчи защитил докторскую диссертацию в Гарварде? Почему Стив Джобе стал вегетарианцем? Ответы на эти и другие вопросы вы можете найти в этой увлекательной книге. Книга содержит 34 короткие биографии людей, которые сформировали компьютерную промышленность (от Чарльза Бэббиджа до Дональда Кнута).

6. Stallings, *Computer Organization and Archirecture,* 4th ed.

   Книга по компьютерной архитектуре. В ней рассматриваются и те вопросы, которые мы обсуждали в нашей книге.

7. Wilkes, *Computers Then and Now*.

   Автор книги Морис Уилкс, один из первых компьютерных разработчиков и изобретатель микропрограммирования, излагает историю компьютеров с 1946 по 1968 год. Он рассказывает о войне между приверженцами автоматического программирования («space cadets») и традиционалистами, которые предпочитали программировать в восьмеричной системе.

Организация компьютерных систем

1. Ng, *Advances in Disk Technology: Performance Issues*.

   В течение последних 20 лет специалисты предсказывают устаревание магнитных дисков. Однако они все еще в ходу. В этой работе говорится о том, что технологии магнитных дисков стремительно развиваются и что они будут служить нам еще долгие годы.

2. Messmer, *The Indispensable PC Hardware Book,* 3rd ed.

   Толстая книга на 1384 страницы (36 глав и 13 приложений). Здесь очень подробно описываются процессоры 80x86, память, шины, вспомогательные микросхемы и периферийные устройства. Если вы уже прочитали книгу Нортона и Гудмэна (см. ниже) и хотите получить более подробную информацию, обратитесь к этой работе.

3. Norton and Goodman, *Inside the PC,* 7th ed.

   Большинство книг по аппаратному обеспечению написаны для инженеров-электриков, и их сложно читать тем, кто занимается программным обеспечением. Эта книга не такая. В ней просто и доступно рассказывается об аппаратном обеспечении персональных компьютеров. Речь идет о процессоре, памяти, шинах, дисках, мониторах, устройствах ввода-вывода, переносных персональных компьютерах, сетях и т. п. Очень редкая и ценная книга.

4. Pilgrim, *Build Your Own Pentium II PC*.

   Если вы умеете обращаться с отверткой и паяльником и хотите сконструировать свой собственный компьютер из отдельных деталей, эта книга может вам пригодиться. Даже если вы намерены купить компьютер в магазине, в этой книге вы можете найти полезную информацию о компонентах персонального компьютера, о том, как они работают и какие у них особенности и дополнительные возможности.

Цифровой логический уровень

1. Floyd, *Digital Fundamentals,* 6th ed.

   Эта огромная иллюстрированная книга вполне подойдет для тех, кто хочет более подробно изучить цифровой логический уровень. В ней рассказывается о комбинационных логических схемах, программируемых логических устройствах, триггерах, сдвиговых регистрах, о памяти, интерфейсах и многом другом.

2. Katayama, *Trends in Semiconductor Memory.*

   Хотя память работает гораздо медленнее процессоров, полупроводниковая память все же развивается. В этой статье рассказывается о некоторых тенденциях развития динамического ОЗУ.

3. Mano and Kime, *Logic and Computer Design Fundamentals.*

   Эта книга не обладает такой проработанностью и ясностью, как книга Флойда (Floyd), но тоже является неплохим пособием по цифровому логическому уровню. В ней содержится информация о комбинационных и последовательных схемах, регистрах, памяти, процессорах и устройствах ввода-вывода.

4. Mazidi and Mazidi, *The 80x86 IBM PC and Compatible Computers,* 2nd ed.

   Книга предназначена для читателей, которые интересуются устройством всех микросхем персонального компьютера. В книге содержатся целые главы об основных микросхемах, а также масса прочей информации об аппаратном обеспечении и программировании на языке ассемблера.

5. McKee et al., *Smarter Memory: Improving Bandwidth for Streamed References.*

   По сравнению с процессорами память с течением десятилетий работает все медленнее и медленнее. В этой работе рассматриваются различные вопросы, связанные с производительностью памяти, а также возможности решения этой проблемы.

6. Nelson et al., *Digital Logic and Circuit Analysis and Design.*

   Еще одна всеобъемлющая книга по цифровой логике. В ней подробно рассказывается о последовательных и комбинационных схемах.

7. Triebel, *The 80386, 80486 and Pentium Processor.*

   Эта книга имеет отношение и к аппаратному, и к программному обеспечению, а также к интерфейсам. В ней рассказывается все о процессорах, памяти, устройствах ввода-вывода и о сопряжении микросхем компьютера 80x86, а также о том, как их программировать на языке ассемблера. В ней всего 915 страниц, но она содержит столько же материала, как и книга Мессмера, поскольку страницы здесь больше по размеру.

Микроархитектурный уровень

1. Handy, *The Cache Memory Book,* 2nd ed.

   Вопрос разработки кэш-памяти очень важен, поэтому существуют целые книги, посвященные этому вопросу. В этой книге обсуждаются логическая и физическая кэш-память, размер строки, сквозная и обратная запись,

объединенная и разделенная кэш-память, а также некоторые вопросы программного обеспечения. Целая глава посвящена когерентности кэш-памяти в мультипроцессоре.

2. Johnson, *Superscalar Microprocessor Design*.

Если вы интересуетесь подробностями разработки суперскалярных процессоров, вам нужно начать именно с этой книги. В ней рассказывается о вызове и декодировании команд, о выдаче команд с изменением последовательности, переименовании регистров, резервациях, прогнозировании переходов и о многом другом.

3. Normoyle et al. *UltraSPARC Hi: Expending the Boundaries of a System on a Chip*.

UltraSPARC Iii — это версия UltraSPARC II с шиной PCI. В этом труде разработчики рассказывают о том, как работает эта система.

4. McChan and O' Connor, *picojava: A Direct Execution Engine for Java ByteCode*.

Эта статья представляет собой краткое введение в микроархитектуру picojava (и следовательно, микросхемы microjava 701). В ней дана блок-схема, обсуждаются вопросы конвейеризации и рассказывается о различных способах оптимизации.

5. Shriver and Smith, *The Anatomy of a High-Performance Microprocessor*.

Эта книга хорошо подходит для детального изучения современного процессора на микроархитектурном уровне. Подробно описывается микросхема AMD Кб (клон Pentium). Рассказывается о конвейерах, планировании выполнения команд и о способах повышения производительности.

6. Sima, *Superscalar Instruction Issue*.

Суперскалярная подача команд чрезвычайно важна в современных процессорах. В этой книге мы затронули некоторые вопросы, связанные с этим (в частности, переименование и спекулятивное выполнение). В статье рассматриваются эти и многие другие вопросы.

7. Wilson, *Challenges and Trends in Processor Design*.

Неужели разработка процессоров не продвигается? Шесть ведущих разработчиков процессоров из компаний Sun, Cyrix, Motorola, Mips, Intel и Digital рассказывают о перспективах развития процессоров в следующие несколько лет. В 2008 году читать это будет смешно, но в настоящее время ее стоит прочитать.

Уровень команд

1. Antonakos, *The Pentium Microprocessor*.

Первые девять глав этой книги посвящены тому, как программировать Pentium на языке ассемблера. В последних двух рассказывается об аппаратном обеспечении машины Pentium. Приводятся многочисленные фрагменты программ. Рассматривается базовая система ввода-вывода.

2. Paul, *SPARC Architecture, Assembly Language, Programming, and C*

   Удивительно, но эта книга по программированию на языке ассемблера посвящена вовсе не серии Intel 80x86. Здесь рассказывается о компьютере SPARC и о том, как программировать на нем.

3. Weaver and Germond, *The SPARC Architecture Manual.*

   В связи с интернационализацией компьютерной промышленности стандарты приобретают особую важность. В этой книге дается определение Version 9 SPARC, а также подробно рассказывается о том, что представляет собой стандарт. В книге содержится много информации о том, как работают 64-битные процессоры SPARC.

Уровень операционной системы

1. Hart, *Win32 System Programming.*

   В отличие от большинства книг по Windows, эта посвящена вовсе не графическому пользовательскому интерфейсу. В ней основное внимание уделяется системным вызовам Windows и тому, как они используются для доступа к файлам, управления памятью и процессами, осуществления взаимодействия между процессами, управления потоками, процессами ввода-вывода и т. д.

2. Jacob and Mudge, *Virtual Memory: Issues of Implementation.*

   Хорошая книга о современной виртуальной памяти. В ней рассказывается о таблицах страниц и TLB на примере MIPS, Power PC и процессоров Pentium.

3. Korn, *Porting UNIX to Windows NT.*

   На первый взгляд может показаться, что переносить программы UNIX на NT легко, поскольку система NT содержит очень много системных вызовов. Однако практика показывает, что сделать это не так-то просто. Автор статьи рассказывает, почему возникают трудности.

4. McKusick et al., *Design and Implementation of the 4.4 BSD Operating System.*

   В отличие от большинства книг по UNIX, эта начинается с фотографии четырех авторов на конференции USENIX Conference. Трое из них много написали о пакете BSD версии 4.4 и высококвалифицированы в этом вопросе. В книге говорится о системных вызовах, процессах, о процессе ввода-вывода. Целый раздел посвящен сетям.

5. Ritchie and Thompson, *The UNIX Time-Sharing System.*

   Это самая первая работа, посвященная системе UNIX. И тем не менее ее стоит прочитать. Из этого маленького зернышка выросла большая операционная система.

6. Solomon, *Inside Windows NT,* 2nd ed.

   Если вы хотите знать, как работает система NT, эта книга для вас. В ней обсуждаются архитектура и механизмы системы, процессы, потоки, управление памятью, защита, процесс ввода-вывода, кэш-память, файловые системы и многие другие вопросы.

7. Tanenbaum and Woodhull, *Operating Systems: Design and Implementation,* 2nd ed.
Большинство книг об операционных системах касаются только теоретических вопросов. В этой книге теория проиллюстрирована на примере реального программного кода операционной системы MINIX, сходной с UNIX, которая работает на IBM PC и других компьютерах. Исходный код с подробными комментариями приводится в приложении.

Уровень языка ассемблера

1. Irvine, *Assembly Language for Intel-Based Computers,* 3rd ed.

Тема этой книги — программирование процессоров Intel на языке ассемблера. В ней также рассказывается о программировании ввода-вывода, макросах, файлах, связывании, прерываниях и т. д.

2. Saloman, *Assemblers and Loaders.*

Все, что вы хотели знать об однопроходных и двупроходных ассемблерах, а также о том, как работают компоновщики и загрузчики, макросы и условная компоновка программы.

Архитектуры компьютеров параллельного действия

1. Adve and Gharachorloo, *Shared Memory Consistency Models: A Tutorial.*

Во многих современных компьютерах, особенно мультипроцессорах, поддерживается более слабая модель памяти, чем согласованность по последовательности. В этом учебном пособии обсуждаются различные модели и объясняется, как они работают. Здесь также приводятся и опровергаются многочисленные мифы о слабо согласованной памяти.

2. Almasi and Gottlieb, *Highly Parallel Computing,* 2nd ed.

В этой книге рассказывается о параллельной вычислительной обработке, в том числе о сетях, архитектуре, компиляторах, моделях и приложениях. В ней представлены проблемы аппаратного и программного обеспечения, а также прикладные вопросы.

3. Hill, Multiprocessors Should Support Simple Memory-Consistency Models.

Слабые модели памяти — важная и спорная проблема в разработке памяти мультипроцессора. Слабые модели допускают определенные оптимизации аппаратного обеспечения (например, совершение обращений к памяти в другом порядке), но усложняют программирование. В этой статье автор обсуждает различные вопросы, связанные с согласованностью памяти, и приходит к выводу, что слабо согласованная память создает больше проблем, чем преимуществ.

4. Hwang and Xu, *Scalable Parallel Computing.*

Авторы рассматривают и программное, и аппаратное обеспечение, поэтому им удалось дать всестороннее, но доступное описание параллельной вы-

числительной обработки. В книге говорится о мультипроцессорах UMA и NUMA, системах MPP и COW, о передаче сообщений и параллельном программировании.

5. Pfister, *In Search of Clusters,* 2nd ed.

   Хотя определение кластера появляется только на 72-й странице (группа компьютеров, работающих вместе), он, очевидно, включает в себя все обычные мультикомпьютерные и мультипроцессорные системы. Подробно рассматриваются их аппаратное и программное обеспечение, производительность и доступность. Предупредим читателя: хотя стиль автора кажется поначалу увлекательным, к 500-й странице вся увлекательность исчезает.

6. Snir et al., *MPI: The Complete reference Manual.*

   Название книги говорит само за себя. Если вы хотите научиться программировать на MPI, обратитесь к ней. В книге рассказывается о двухточечной и коллективной коммуникации, коммуникаторах, об управлении средой и о многом другом.

7. Stenstrom et al., Trends in Shared Memory Multiprocessing.

   Мультипроцессоры с памятью совместного использования часто считают суперкомпьютерами для сложных научных вычислений. В действительности это только крошечная часть их рынка. В статье обсуждается, какие сферы охватывает рынок таких машин и каково значение их архитектуры.

Двоичные числа и числа с плавающей точкой

1. Cody, *Analysis of Proposals for the Floating-Point Standard.*

   Несколько лет назад Институт инженеров по электротехнике и электронике (IEEE) разработал архитектуру с плавающей точкой, которая стала стандартом *de facto* для всех современных процессоров. Автор обсуждает различные вопросы, предложения и возражения, которые возникали во время процесса стандартизации.

2. Garner, *Nubmer Systems and Arithmetic.*

   Учебное пособие о понятиях двоичной арифметики (в том числе о распространении переноса, системах избыточных чисел, системах остаточных классов и о нестандартном умножении и делении). Особенно рекомендуется для тех, что считает, что узнал все об арифметике в шестом классе.

3. IEEE, *Pmc. of the n-th Symposium on Computer Arithmetic.*

   Вопреки общепринятому мнению арифметика является активной областью исследования. Специалистами написано много научных трудов. На симпозиуме обсуждаются проблемы прогрессий, развитие высокоскоростного сложения и умножения, арифметическое аппаратное обеспечение СБИС, сопроцессоры, отказоустойчивость, округление и многие другие вопросы.

4. Knuth, *Seminumerical Algorithms,* 3rd ed.

   Обширный материал о позиционных системах счисления, арифметике с плавающей точкой, арифметике с многократно увеличенной точностью и о случайных числах. Книга требует и заслуживает внимательного изучения.

5. Wilson, *Floating-Point Survival Kit.*

   Хорошая книга для начинающих о числах с плавающей точкой и о стандартах. Обсуждаются некоторые популярные задачи с плавающей точкой (например, *Unpack).*

Алфавитный список литературы

1. *Adams, G. B. HI, Agrawal, D. P., and Siegel, H.J.* «A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks», IEEE Computer Magazine, vol. 20, p. 14-27, June 1987.

2. *Adve, S. V., and Charachorloo, K.* «Shared Memory Consistency Models: A Tutorial», IEEE Computer Magazine, vol. 29, p. 66-76, Dec. 1996.

3. *Adve, S V., and Hill, M.* «Weak Ordering: A New Definition», Proc. 17th Ann. Int'l. Symp. on Computer Arch., ACM, p. 2-14, 1990.

4. *Agerwala, T., and Cocke, J.* «High Performance Reduced Instruction Set Processors», IBM TJ. Watson Research Center Technical Report RC12434, 1987.

5. *Almasi, G. S., and Gottlieb, A.* Highly Parallel Computing, 2nd ed. Redwood City, CA: Benjamin/Cummings, 1994.

6. *Amza, C, COX, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.* «TreadMarks: Shared Memory Computing on a Network of Workstations», IEEE Computer Magazine, vol. 29, p. 18-28, Feb. 1996.

7. *Anderson, D.* Universal Serial Bus System Architecture, Reading, MA: Addison-Wesley, 1997.

8. *Anderson, T. E., Culler, D. E., Patterson, D.A., and the NOW team* «A Case for NOW (Networks of Workstations)», IEEE Micro Magazine, vol. 15, p. 54-64, Feb. 1995.

9. *Antonakos J. L.* The Pentium Microprocessor, Upper Saddle River, NJ: Prentice Hall, 1997.

10. *August, D. I., Connors, D.A., Mshlke, S.A., SIASJ. W., Crozier, K. M., Cheng, B.-C, Eaton, P. R., Olaniran, Q. B., and HWU, W.-M.* «Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 227-237, 1998.

11. *Bal, H. E.* Programming Distributed Sysytems, Hemel Hempstead, England: Prentice Hall Int'l., 1991.

12. *Bal, H. E., Bhoedjang, R., Hofman, R, Jacobs, C, Langendoen, K., Ruhl, T., and Kaashoek, M. F.* «Performance Evaluation of the Orca Shared Object System», ACM Trans, on Computer Systems, vol. 16, p. 1-40, Feb. 1998.

13. *Bal, H. E., Kaashoek, M.F., and Tanenbaum, A. S.* «Orca: A Language for Parallel Programming of Distributed Systems», IEEE trans, on Software Engeneering, vol. 18, p. 190-205, March 1992.

14. *Bal, H. E., and Tanenbaum, A. S.* «Distributed Programming with Shared Data», Proc. 1988 Int'l. Conf. on Computer Languages, IEEE, p. 82-91, 1988.

15. *Bhuyan, L. N., Yang, Q., andAgrawal, D. P.* «Performance of Multiprocessor Interconnection Networks», IEEE Computer Magazine, vol. 22, p. 25-37, Feb. 1989.

16. *Bjornson, R. D.* «Linda on Distributed Memory Multiprocessors», Ph. D. Thesis, Yale Univ., 1993.

17. *Blumrich, M.A., Dubnicki, C, Felten, E. W., Li, K., and Mesarina, M. R.* «Virtual-Memory Mapped Network Interfaces», IEEE Micro Magazine, vol. 15, p. 21-28, Feb. 1995.

18. *Boden, N.J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C L., SeizovicJ. N., and Su, W. -K.* «Myrinet: A Gigabit per second Local Area Network», IEEE Micro Magazine, vol. 15, p. 29-36, Feb. 1995.

19. *Bouknight, W.J., Denenberg, S.A., Mcintyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L.* «The Illiac IV System», Proc. IEEE, p. 369-388, April 1972.

20. *Burkhardt, H., Frank, S., Knobe, B., and RothnieJ.* «Overview of the KSR-1 Computer Sysytem», Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.

21. *Carriero, N., and Gelernter, D.* «The S/Net's Linda Kernel», ACM Trans, on Coputer Sysytems, vol. 4, p. 110-129, May 1986.

22. *Carriero, N., andGelernter, D.* «Linda and Context», Commun. of the ACM, vol. 32, p. 444-458, April 1989.

23. *Charlesworth, A.* «Starfire: Extending the SMP Envelope», IEEE Micro Magazine, vol. 18,39-49, Jan./Feb. 1998.

24. *Charlesworth, A., Phelps, A., Williams, R., andGilbert, G.* «Gigaplane-XB: Extending the Ultra Enterprise Family», Proc. Hot Interconnects V, IEEE, 1988.

25. *Cody, W.J.* «Analysis of Proposals for the Floating-Point Standard», IEEE Computer Magazine, vol. 14, p. 63-68, Mar. 1981.

26. *Cohen, D.* «On Holy Wars and a Plea for Peace», IEEE Computer Magazine, vol. 14, p. 48-54, Oct. 1981.

27. *Corbaty, F.J.* «PL/1 as a Tool for System Programming», Datamation, vol. 15, p. 68-76, May 1969.

28. *Corbaty, F.J., and Vyssotsky, V.A.* «Introduction and Overview of the MULTICS System», Proc. FJCC, p. 185-196, 1965.

29. *Denning, P.J.* «The Working Set Model for Program Behavior», Commun. of the ACM, vol. 11, p. 323-333, May 1968.

30. *Dijkstra, E. W.* «GOTO Statement Considered Harmful», Commun. of the ACM, vol. 11, p. 147-148, Mar. 1968a.

31. *Dijkstra, E. W.* «Co-operating Sequential Processes», in Programming Languages, F. Genuys (ed.), New York: Academic Press, 1968b.

32. *Driesen, K., and Holzie, URS* «Accurate Indirect Branch Prediction», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 167-177, 1998.

33. *Dubois, M., Scheurich, C, and Briggs, FA.* «Memory Access Buffering in Multiprocessors», Proc. 13th Ann. Int'l. Symp. on Computer Arch., ACM, p. 434-442, 1986.

34. *Dulong, C* «The IA-64 Architecture at Work», IEEE Computer Magazine, vol. 31, p. 24-32, July 1998.

35. *Faggin, F., Hoff, M. E. Jr., Mazor, S., and Shima, M.* «The History of the 4004», IEEE Micro Magazine, vol. 16, p. 10-20, Dec. 1996.

36. *Falsafi, B., and Wood, DA.* «Reactive NUMA: A Design Unifying S-COMA and CC-NUMA», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 229-240, 1997.

37. *Fisher J. A., and Freudenberger, S. M.* «Predicting Conditional Branch Directions from Previous Runs of a Program», Proc. 5th Conf. on Arch. Support for Prog. Lang, and Operating Syst, ACM, p. 85-95, 1992.

38. *Floyd, T. L.* Digital Fundamentals, 6th ed., Upper Saddle River, NJ: Prentice Hall, **1997.**

39. *Flynn, M.J.* «Some Computer Organizations and Their Effectiveness», IEEE **Trans, on** Computers, vol. C-21, p. 948-960, Sept. 1972.

40. *Foster, I., and Kesselman, C* «Globus: A Metacomputing Infrastructure Toolkit», Int'l. J. of Supercomputer Applications, vol. 11, p. 115-128, 1998a.

41. *Foster, I., and Kesselman, C* «The Globus Project: A Status Report», IPPS/SPDP '98 Heterogeneous Computing Workshop, IEEE, p. 4-18, 1998b.

42. *Fotheringham J'.* «Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store», Commun. of the ACM, vol. 4, p. 435-436, Oct. 1961.

43. *Gajski, D. D., and Pier, K. -K.* «Essential Issues in Multiprocessor Systems», IEEE Computer Magazine, vol. 18, p. 9-27, June 1985.

44. *Garner, H. L.* «Number Sysytems and Arithmetic», in Advances in Computers, vol. 6, F. Alt and M. Rubinoff (eds.), New York: Academic Press, 1965, p. **131—** 194.

45. *Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderram, V.* PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing, Cambridge, MA: M.I.T. Press, 1994.

46. *Gharachorloo, K., Lenoski, D., Laudon J., Gibbons, P., Gupta, A., and Hennessy J.* «Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors», Proc. 17th Ann. Int'l. Symp. on Computer Arch., ACM, p. 15-26, 1990.

47. *Goodman, J. R.* «Using Cache Memory to Reduce Processor Memory Traffic», Proc. 10th Ann. Int'l. Symp. on Computer Arch., ACM, p. 124-131, 1983.

48. *Goodman J. R.* «Cache Consistency and Sequential Consistency», Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

49. *Graham, R.* «Use of High Level Languages for System Programming», Project MAC Report TM-13, Project MAC, MIT, Sept. 1970.

50. *Grimshaw, A. S., and Wulf, W.* «Legion: A View from 50,000 Feet», Proc. Fifth Int'l. Symp. on High-Performance Distributed Computing, IEEE, p. 89-99, Aug. 1996.

51. *Grimshaw, A. S., and Wulf, W.* «The Legion Vision of a Worldwide Virtual Computer», Commun. of the ACM, vol. 40, p. 39-45, Jan. 1997.

52. *Gropp, W., Lusk, E., and Skjellum, A.* «Using MPI: Portable Parallel Programming with the Message Passing Interface», Cambridge, MA: M.I.T. Press, 1994.

53. *Hagersten, E., Landin, A., Haridi, S.* «**DDM** — A Cache-Only Memory Architecture», IEEE Computer Magazine, vol. 25, p. 44-54, **Sept. 1992.**

54. *Hamacher, V. V., Vranesic, Z. G., and Zaky, S. G.* Computer Organization, 4th ed., New York: McGraw-Hill, 1996.

55. *Hamming, R. W.* «Error Detecting and Error Correcting Codes», Bell Syst. Tech. J., vol. 29, p. 147-160, April 1950.

56. *Handy, J.* The Cache Memoru Book, 2nd ed., Orlando, FL: Academic Press, 1998.

57. *Hart, J. M.* Win32 System Programming, Reading, MA: Addison-Wesley, 1997.

58. *Hayes, J. P.* Computer Architecture and Organization, 3rd ed., New York: McGraw-Hill, 1998.

59. *Hennessy, J. L.* «VLSI Processor Architecture», IEEE Trans, on Computers, vol. C-33, p. 1221-1246, Dec. 1984.

60. *Hill, M.* «Multiprocessors Should Support Simple Memory-Consistency Models», IEEE Computer Magazine, vol. 31, p. 28-34, Aug. 1998.

61. *Hoare, C A. R.* «Monitors, An Operating System Structuring Concept», Commun. of the ACM, vol. 17, p. 549-557, Oct. 1974; Erratum in Commun. of the ACM, vol. 18,p.95,Feb.1975.

62. *Hwang, K., and Xu, Z.* Scalable Parallel Computing, New York: McGraw-Hill, 1998.

63. *Hwu, W. -M.* «Introduction to Predicated Execution», IEEE Computer Magazine, vol. 31, p. 49-50, Jan. 1998.

64. *Irvine, K.* Assembly Language for Intel-Based Computers, 3rd ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

65. *Jacob, B., and Mudge, T.* «Virtual Memory: Issues of Implementation», IEEE Computer Magazine, vol. 31, p. 33-43, June 1998a.

66. *Jacob, B., and Mudge, T.* «Virtual Memory in Contemporary Microprocessors», IEEE Micro Magazine, vol. 18, p. 60-75, July/Aug. 1998b.

67. *Joe, T., and Hennessy,J. L.* «Evaluating the Memory Overhead Required for COMA Architectures», Proc. 21th Ann. Int'l. Symp. on Computer Arch., ACM, p. 82-93,1994.

68. *Johnson, K. L., Kaashoek, M. F., and Wallach, D. A.* «CRL: High-Performance All-Software Distributed Shared Memory», Proc. 15th Symp. on Operating Systems Principles, ACM, p. 213-228,1995.

69. *Johnson, M.* Superscalar Microprocessor Design, Englewood Cliffs, NJ: Prentice Hall, 1991.

70. *Juan, T., Sanjeevan, S., and Navarro,J.J.* «Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM. p. 155-166,1998.

71. *Katayama, Y.* «Trends in Semiconductor Memories», IEEE Micro Magazine, p. 10-17, Nov./Dec. 1997.

72. *Kermarrec, A. -M., Kuz, I., Van Steen, M., and Tanenbaum, A. S.* «A Framework for Consistent Replicated Web Objects», Proc. 18th Int'l. Conf. on Distr. Computing Syst, IEEE, p. 276-284, 1998.

73. *Knuth, D. E.* «An Empirical Study of FORTRAN Programs», Software — Practice & Experience, vol. 1, p. 105-133,1971.

74. *Knuth, D. E.* The Art of Computer Programming: Fundamental Algorithms, 3rd ed.. Reading, MA: Addison-Wesley, 1997.

75. *Knuth, D. E.* The Art of Computer Programming: Seminumerical Algorithms, 3rd ed., Reading, MA: Addison-Wesley, 1998.

76. *Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniad, M., Parthasarathy, S., Meira, W., Dwarkadas, S., and Scott, M.* VM-Based Shared Memory on Low Latency Remote Memory Access Networks, Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 157-169,1997.

77. *Kom, D.* «Porting UNIX to Windows NT», Proc. Winter 1997 USENIX Conf., p. 43-57,1997.

78. *Kumar, V. P., and Reddy, S. M.* «Augmented Shuffle-Exchange Multistage Interconnection Networks», IEEE Computer Magazine, vol. 20, p. 30-40, June 1987.

79. *Lamport, L.* «How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs», IEEE Trans, on Computers, vol. C-28, p. 690-691, Sept. 1979.

80. *LaRowe, R. P., and Ellis, C. S.* «Experimental Comparison of Memory Management Policies for NUMA Multiprocessors», ACM Trans, on Computer Systems, vol. 9, p. 319-363, Nov. 1991.

81. *Lenoski, D., Laudon J., Gharachorloo, K, Weber, W.-D., Gupta, A., Hennessy J., Horowitz, M., and Lam, M.* «The Stanford Dash Multiprocessor», IEEE Computer Magazine, vol. 25, p. 63-79, March 1992.

82. *Li, K.* «IVY: A Shared Virtual Memory System for Parallel Computing», Proc. 1988 Int'l. Conf. on Parallel Proc. (Vol. 11), IEEE, p. 94-101, 1988.

83. *Li, K., and Hudak, P.* «Memory Coherence in Shared Virtual Memory Systems», ACM Trans, on Computer Systems, vol. 7, p. 321-359, Nov. 1989.

84. *Li, K., and Hudak, P.* «Memory Coherence in Shared Virtual Memory Systems», Proc. 5th Ann. ACM Symp. on Prin. of Distr. Computing, ACM, p. 229-239, 1986.

85. *Lindholm, T., and Yellin, F.* The Java Virtual Machine Specification, Reading, MA: Addison-Wesley, 1997.

86. *Loshin, D.* High Performance Computing Demystified, Cambridge, MA: AP Prof., 1994.

87. *Lu, H., Cox, A. L., Dwarkadas, S., Rajamony, R., and Zwaenepoel, W.* «Software Distributed Shared Memory Support for Irregular Applications», Proc. 6th Conf. on Prin. and Practice of Parallel Progr., p. 48-56, June 1997.

88. *Lukasiewicz, J.* Aristotle's Syllogistic, 2nd ed., Oxford: Oxford University Press, 1958.

89. *Mano, M. M., andKime, C. R.* Logic and Computer Design Fundamentals, Upper Saddle River, NJ: Prentice Hall, 1997.

90. *Martin, R. P., Vahdat, A. M., Culler, D. E., andAnderson, T. E.* «Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 85-97,1997.

91. *Mazidi, M. A., andMazidiJ. G.* The 80x86 IBM PC and Compatible Computers, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1998.

92. *McGhan, H. and O'connorJ. M.* «picojava: A Direct Execution Engine for Java Bytecode», IEEE Computer Magazine, vol. 31., Oct. 1998.

93. *McKee, S.A., Klenke, R.#., Wright, K.L, Wulf, W.A., Saunas, M.H., AylorJ.H., and Batson, A. P.* «Smarter Memory: Improving Bandwidth for Streamed References», IEEE Computer Magazine, vol. 31, p. 54-63, July 1998.

94. *McKusick, M. K., Bostic, K., Karels, M., and QuartermanJ. S.* «The Design and Implementation of the 4.4 BSD Operating System», Reading, MA: Addison-Wesley, 1996.

95. *McKusick, M. K.Joy, W. N, Leffler, S.J., andFabry, R. S.* «A Fast File System for UNIX», ACM Trans, on Computer Systems, vol. 2, p. 181-197, Aug. 1984.

96. *Messmer, H.-P.* The Indispensible PC Hardware Book, 3rd ed.. Reading, MA: Addison-Wesley, 1997.

97. *Morgan, C* Portraits in Computing, New York: ACM Press, 1997.

98. *Morin, C, Gefflaut, A., Banatre, M., and Kermarrec, A. -M.* «COMA: An Opportunity for Building a Fault-Tolerant Scalable Shared Memory Multiprocessor», Proc. 24th Ann. tnt'l. Symp. on Computer Arch., ACM, p. 65-65,1996.

99. *Moudgill, M., and Vassiliadis, S.* «Precise Interrupts», IEEE Micro Magazine, vol. 16, p. 58-67, Feb. 1996.

100. *Mullender, S.J., and Tanenbaum,A.S.* «Immediate Files», Software— Practice and Experience, vol. 14, p. 365-368,1984.

101. *Nelson, V. P., Nagle, H. T., Carroll, B. D., andlrwinj. D.* Digital Logic and Circuit Analysis and Design, Englewood Cliffs, NJ: Prentice Hall, 1995.

102. *Ng, S. W.* «Advances in Disk Technology: Performance Issues», IEEE Computer Magazine, vol. 31, p. 75-81, May 1998.

103. *Normoyle, K.B., Csoppenszky, M.A., Tzeng, A., Johnson, T.P., Furman, CD., and Mos-Toufi,J.* «UltraSPARC Hi: Expanding the Boundaries of a System on a Chip», IEEE Micro Magazine, vol. 18, p. 14-24, March/April 1998.

104. *Norton, P., and Goodman, J.* Inside the PC, 7th ed., Indianapolis, IN: Sams, 1997.

105. *O'connor,J.M., and Tremblay, M.* «Picojava-I: The Java Virtual Machine in Hardware», IEEE Micro Magazine, vol. 17, p. 45-53, March/April 1997.

106. *Organick, E.* The MULTICS System, Cambridge, MA: M.I.T. Press, 1972.

107. *Pakin, S., Karamcheti, V., and Cfflen.A.A.* «Fast Messages (FM): Efficient, Portable Communication for Workstation Cluster and Massively-Parallel Processors», IEEE Concurrency, vol. 5, p. 60-73, April-June 1997.

108. *Pan,S. -T.,So,K.,andRahmeh,J. T.* «Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation», Proc. 5th Int'l. Conf. on Arch. Support for Prog. Long, and Operating Syst, ACM, p. 76-84, Oct. 1992.

109. *Papamarcos, M., and Patel.J.* «A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories», Proc. 1 lth Ann. Int'l. Symp. on Computer Arch., ACM, p. 348-354, 1984.

110. *Patterson, D. A.* «Reduced Instruction Set Computers», Commun. of the ACM, vol. 28, p. 8-21, Jan. 1985.

111. *Patterson, D. A., Gibson, G., andKatz, R.* «A case for redundant arrays of inexpensive disks (RAID)», Proc. ACM SIGMOD Int'l. Conf. on Management of Data, ACM, p. 109-166,1988.

112. *Patterson, D. A., andHennessyJ. L.* Computer Organization and Design, 2nd ed., San Francisco, CA: Morgan Kaufmann, 1998.

113. *Patterson, D. A., and Suquin, C. H.* «A VLSI RISC», IEEE Computer Magazine, vol. 15, p. 8-22, Sept. 1982.

114. *Paul, R. P.* SPARC Architecture, Assembly Language, Programming, and C, Englewood Cliffs, NJ: Prentice Hall, 1994.

115. *Pfister, G.F.* In Search of Clusters, 2nded., Upper Saddle River, NJ: Prentice Hall, 1998.

116. *Pilgrim, A.* Build Your Own Pentium II PC, New York: McGraw-Hill, 1998.

117. *Pountain, D.* «Pentium: More RISC than CISC», Byte, vol. 18, p. 195-204, Sept. 1993.

118. *Price, D.* «A History of Calculating Machines», IEEE Micro Magazine, vol.4, p.22-52,Feb.l984.

119. *Radin, G.* «The 801 Minicomputer», Computer Arch. News, vol. 10, p. 39-47, March 1982.

120. *Ritchie, D. M., and Thompson, K.* «The UNIX Time-Sharing System», Commun. of the ACM, vol. 17, p. 365-375, July 1974.

121. *Rosenblum, M., and OusterhoutJ. K.* «The Design and Implementation of a Log-Structured File System», Proc. Thirteenth Symp. on Operating System Principles, ACM, p. 1-15, 1991.

122. *Saloman, D.* Assemblers and Loaders, Englewood Cliffs, NJ: Prentice Hall, 1993.

123. *Saulsbury, A., Wilkinson, T., Carger,J., and Landin, A.* «An Argument for Simple COMA», Proc. of First IEEE Symp. on High-Performance Сотр. Arch., IEEE, p. 276-285,1995.

124. *Scales, D.J., Gharachorloo, K., and Thekkath, CA.* «Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory», Proc. 7th Int'l. Conf. on Arch. Support for Prog. Long, and Oper. Syst., ACM, p. 174-185,1996.

125. *Sechrest, S., Lee, C. -C, and Mudge, T.* «Correlation and Aliasing in Dynamic Branch Predictors», Proc. 23th Ann. Int'l. Symp. on Computer Arch., ACM, p. 22-32, 1996.

126. *Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C* «An Implementation of a Log-Structured File System for UNIX», Proc. Winter 1993 USENIX Technical Conf., p. 307-326,1993.

127. *Shanley, T., and Anderson, D.* ISA System Architecture, Reading, MA: Addison-Wesley, 1995a.

128. *Shanley, T., and Anderson, D.* PCI System Architecture, 3rd ed.. Reading, MA: Addison-Wesley, 1995b.

129. *Shriver, B., and Smith, B.* The Anatomy of a High-Performance Microprocessor: A Systems Perspective, Los Alamitos, CA: IEEE Computer Society, 1998.

130. *Sima, D.* «Superscalar Instruction Issue», IEEE Micro Magazine, vol. 17, p. 28-39, Sept./Oct 1997.

131. *Sima, D., Fountain, T., and Kacsuk, P.* Advanced Computer Architectures: A Design Space Approach, Reading, MA: Addison-Wesley, 1997.

132. *Slater, R.* Portraits in Silicon, Cambridge, MA: M.I.T. Press, 1987.

133. *Smith, A.J.* «Cache Memories», Computing Surveys, vol. 14, p. 473-530, Sept. 1982.

134. *Snip, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and DongarraJ.* MPI: The Complete Reference Manual, Cambridge, MA: M.I.T. Press, 1996.

135. *Solari, E.* ISA & EISA Theory and Operation, San Diego, CA: Annabooks, 1993.

136. *Solari, E., and Willse, G.* PCI Hardware and Software Architecture and Design, 4th ed., San Diego, CA: Annabooks, 1998.

137. *Solomon, DA.* Inside Windows NT, 2nd ed., Redmond, WA: Microsoft Press, 1998.

138. *Sprangle, E., Chappell, R. S., Alsup, M., and Patt, Y. N.* «The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 284-291,1997.

139. *Stallings, W.* Computer Organization and Architecture, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1996.

140. *Stenstrom, P., Hagersten, E., Lilja, D.J., Martonosi, M., and Venugopal, M.* «Trends in Shared Memory Multiprocessing», IEEE Computer Magazine, vol. 30, p. 44-50, Dec. 1997.

141. *Stets, K, Dwarkadas, S., Hardave Uas, N, Hunt, G., Kontothanassis, L, Parthasarathy, S., and Scott, M.* «CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks», Proc. 16th Symp. on Operating Systems Principles, ACM, p. 170-183, 1997.

142. *Sunderram, V. B.* «PVM: A Framework for Parallel Distributed Computing», Concurrency: Practice and Experience, vol. 2, p. 315-339, Dec. 1990.

143. *Swan, R.J., Fuller, S. H., and Siewiorek, D. P.* «Cm* —A Modular Multiprocessor», Proc. NCC, p. 645-655,1977.

144. *Tan, W. M.* Developing USB PC Peripherals, San Diego, CA: Annabooks, 1997.

145. *Tanenbaum, A. S.* «Implications of Structured Programming for Machine Architecture», Commun. of the ACM, vol. 21, p. 237-246, Mar. 1978.

146. *Tanenbaum, A. S., and Woodhull, A. W.* Operating Systems: Design and Implementation, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997.

147. *Thompson, K.* «UNIX Implementation», Bell Syst. Tech. J., vol. 57, p. 1931-1946, July-Aug. 1978.

148. *Treleaven, P.* «Control-Driven, Data-Driven, and Demand-Driven Computer Architecture», Parallel Computing, vol. 2, 1985.

149. *Tremblay, M. and O 'connorj. M.* «UltraSPARC I: A Four-Issue Processor Supporting Multimedia», IEEE Micro Magazine, vol. 16, p. 42-50, April 1996.

150. *Triebel, W. A.* The 80386, 80486, and Pentium Processor, Upper Saddle River, NJ: Prentice Hall, 1998.

151. *linger, S. H.* «A Computer Oriented Toward Spatial Problems», Proc. IRE, vol. 46, p. 1744-1750,1958.

152. *Vahalia, U.* UNIX Internals, Upper Saddle River, NJ: Prentice Hall, 1996.

153. *Van DerPoel, W. L.* «The Software Crisis, Some Thoughts and Outlooks», Proc. IFIP Congr. 68, p. 334-339, 1968.

154. *Van Steen, M., Homburg, P. C, and Tanenbaum, A. S.* «The Architectural Design of Globe: A Wide-Area Distributed System», IEEE Concurrency, vol. 7, p. 70-78Jan.-March 1999.

155. *Verstoep, K., Langedoen, K., andBal, H. E.* «Efficient Reliable Mulicast on Myrinet», Proc. 1996 Int'l. Conf. on Parallel Processing, IEEE, p. 156-165,1996.

156. *Weaver, D. L., and Germond, T.* The SPARC Architecture Manual, Version 9, Englewood Cliffs, NJ: Prentice Hall, 1994.

157. *mikes, M. V.* «Computers Then and Now», J. ACM, vol. 15, p. 1-7, Jan. 1968.

158. *Wilkes, M. V.* «The Best Way to Design an Automatic Calculating Machine», Proc. Manchester Univ. Computer Inaugural Conf., 1951.

159. *Wilkinson, B.* Computer Architectural Design and Performance, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1994. Wilinson, 1994.

160. *Wilkinson, B. and Allen, M.* Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Upper Saddle River, NJ: Prentice Hall, 1999.

161. *Wilson, J.* «Challenges and Trends in Processor Design», IEEE Computer Magazine, vol. 31, p. 39-48, Jan. 1998.

162. *Wilson, P.* «Floating-Point Survival Kit», Byte, vol. 13, p. 217-226, March 1988.

163. *Yeh, T.-Y., and Pott, Y. -N.* «Two-Level Adaptive Training Branch Prediction», Proc. 24th Int'l. Symp. on Microarchitectwe, ACM/IEEE, p. 51-61,1991.

Приложение А
Двоичныечисла

Арифметика, применяемая в компьютерах, сильно отличается от арифметики, которая используется людьми. Во-первых, компьютеры выполняют операции над числами, точность которых конечна и фиксирована. Во-вторых, в большинстве компьютеров используется не десятичная, а двоичная система счисления. Этим двум проблемам и посвящено приложение.

Числа конечной точности

Когда люди выполняют какие-либо арифметические действия, их не волнует вопрос, сколько десятичных разрядов занимает то или иное число. Физики, к примеру, могут вычислить, что во Вселенной присутствует 10^{78} электронов, и их не волнует тот факт, что полная запись этого числа потребует 79 десятичных разрядов. Никогда не возникает проблемы нехватки бумаги для записи числа.

С компьютерами дело обстоит иначе. В большинстве компьютеров количество доступной памяти для хранения чисел фиксировано и зависит от того, когда был разработан этот компьютер. Если приложить усилия, программист сможет представлять числа в два, три и более раз большие, чем позволяет размер памяти, но это не меняет природы данной проблемы. Память компьютера ограничена, поэтому мы можем иметь дело только с такими числами, которые можно представить в фиксированном количестве разрядов. Такие числа называются **числами конечной точности.**

Рассмотрим ряд положительных целых чисел, которые можно записать тремя десятичными разрядами без десятичной запятой и без знака. В этот ряд входит ровно 1000 чисел: 000, 001, 002, 003,..., 999. При таком ограничении невозможно выразить определенные типы чисел. Сюда входят:

1. Числа больше 999.
2. Отрицательные числа.
3. Дроби.
4. Иррациональные числа.

5. Комплексные числа.

Одно из свойств набора всех целых чисел — замкнутость по отношению к операциям сложения, вычитания и умножения. Другими словами, для каждой пары целых чисел *i nj* числа *i+j, i-j* и *ixj* — тоже целые числа. Ряд целых чисел не замкнут относительно деления, поскольку существуют такие значения *ż nj,* для которых *i/j* не выражается в виде целого числа (например, 7/2 или 1/0).

Числа конечной точности не замкнуты относительно всех четырех операций. Ниже приведены примеры операций над трехразрядными десятичными числами:

600+600=1200 (слишком большое число);
003-005=-2 (отрицательное число);
050x050=2500 (слишком большое число);
007/002=3,5 (не целое число).

Отклонения можно разделить на два класса: операции, результат которых превышает самое большое число ряда (ошибка переполнения) или меньше, чем самое маленькое число ряда (ошибка из-за потери значимости), и операции, результат которых не является слишком маленьким или слишком большим, а просто не является членом ряда. Из четырех примеров, приведенных выше, первые три относятся к первому классу, а четвертый — ко второму классу.

Поскольку размер памяти компьютера ограничен и он должен выполнять арифметические действия над числами конечной точности, результаты определенных вычислений будут неправильными с точки зрения классической математики. Вычислительное устройство, которое выдает неправильный ответ, может показаться странным на первый взгляд, но ошибка в данном случае — это только следствие его конечной природы. Некоторые компьютеры содержат специальное аппаратное обеспечение, которое обнаруживает ошибки переполнения.

Алгебра чисел конечной точности отличается от обычной алгебры. В качестве примера рассмотрим ассоциативный закон

a+(b-c)=(a+b)-c.

Вычислим обе части выражения для а=700, 6=400 и с=300. В левой части сначала вычислим значение *(b-c).* Оно равно 100. Затем прибавим это число К Й И получим 800. Чтобы вычислить правую часть, сначала вычислим *(a+b).*

Для трехразрядных целых чисел получится переполнение. Результат будет зависеть от компьютера, но он не будет равен 1100. Вычитание 300 из какого-то числа, отличного от 1100, не даст результата 800. Ассоциативный закон не имеет силы. Порядок операций важен.

Другой пример — дистрибутивный закон:

ax (b-c)=axb-axc.

Сосчитаем обе части выражения для *a=5,*6=210 и с= 195. В левой части 5x15=75. В правой части 75 не получается, поскольку *axb* выходит за пределы ряда.

Исходя из этих примеров, кто-то может сделать вывод, что компьютеры совершенно непригодны для выполнения арифметических действий. Вывод, естествен-

но, неверен, но эти примеры наглядно иллюстрируют важность понимания, как работает компьютер и какие ограничения он имеет.

Позиционные системы счисления

Обычное десятичное число состоит из цепочки десятичных разрядов и иногда десятичной запятой. Общая форма записи показана на рис. А. 1. Десятка выбрана в качестве основы возведения в степень (это называется **основанием системы счисления),** поскольку мы используем 10 цифр. В компьютерах удобнее иметь дело с другими основаниями системы счисления. Самые важные из них — 2, 8 и 16. Соответствующие системы счисления называются **двоичной, восьмеричной** и **шестнадцатеричной** соответственно.

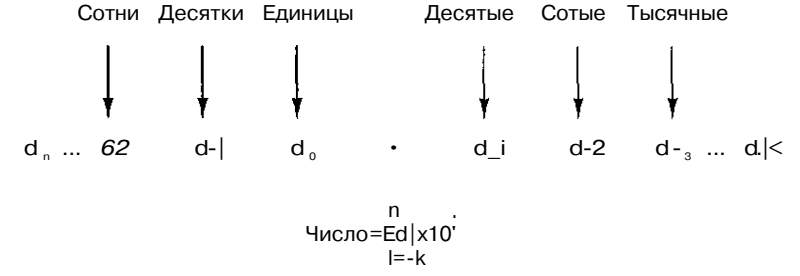


Рис. А. 1. Общая форма десятичного числа

&-ичная система требует *k* различных символов для записи разрядов с 0 по *k*—1. Десятичные числа строятся из 10 десятичных цифр

0 1 2 3 4 5 6 7 8 9

Двоичные числа, напротив, строятся только из двух двоичных цифр

0 1

Восьмеричные числа состоят из восьми цифр

0 1 2 3 4 5 6 7

Для шестнадцатеричных чисел требуется 16 цифр. Это значит, что нам нужно 6 новых символов. Для обозначения цифр, следующих за 9, принято использовать прописные латинские буквы от A до F. Таким образом, шестнадцатеричные числа строятся из следующих цифр.

0 1 2 3 4 5 6 7 8 9 A B C D E F

Двоичный разряд (то есть 1 или 0) обычно называют **битом.** На рис. А.2 десятичное число 2001 представлено в двоичной, восьмеричной и шестнадцатеричной системе. Число 7B9 очевидно шестнадцатеричное, поскольку символ B встречается только в шестнадцатеричных числах. А число 111 может быть в любой из четырех систем счисления. Чтобы избежать двусмысленности, нужно использовать индекс для указания основания системы счисления.

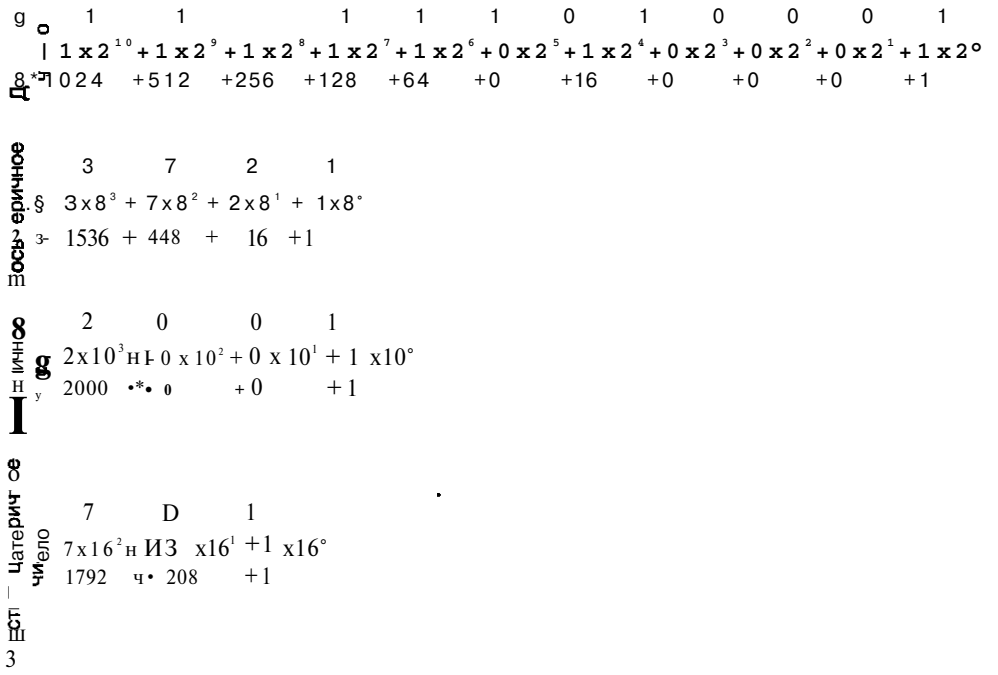


Рис. А.2. Число 2001 в двоичной, восьмеричной и шестнадцатеричнои системе

В таблице АЛ ряд неотрицательных целых чисел представлен в каждой из четырех систем счисления.

Таблица А. 1. Десятичные числа и их двоичные, восьмеричные и шестнадцатеричные эквиваленты

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Преобразование чисел из одной системы счисления в другую

Преобразовывать числа из восьмеричной в шестнадцатеричную или двоичную систему и обратно легко. Чтобы преобразовать двоичное число в восьмеричное, нужно разделить его на группы по три бита, причем три бита непосредственно слева от двоичной запятой формируют одну группу, следующие три бита слева от этой группы формируют вторую группу и т. д. Каждую группу по три бита можно преобразовать в один восьмеричный разряд со значением от 0 до 7 (см. первые строки табл. А.1). Чтобы дополнить группу до трех битов, нужно спереди приписать один или два нуля. Преобразование из восьмеричной системы в двоичную тоже тривиально. Каждый восьмеричный разряд просто заменяется эквивалентным 3-битным числом. Преобразование из 16-ричной в двоичную систему, по сути, сходно с преобразованием из 8-ричной в двоичную систему, только каждый 16-ричный разряд соответствует группе из четырех битов, а не из трех. На рис. А.3 приведены примеры преобразований из одной системы в другую.

Преобразование десятичных чисел в двоичные можно совершать двумя разными способами. Первый способ непосредственно вытекает из определения двоичных чисел. Самая большая степень двойки, меньшая, чем число, вычитается из этого числа. Та же операция проделывается с полученной разностью. Когда число разложено по степеням двойки, двоичное число может быть получено следующим образом. Единички ставятся в тех позициях, которые соответствуют полученным степеням двойки, а нули — во всех остальных позициях.

Второй способ — деление числа на 2. Частное записывается непосредственно под исходным числом, а остаток (0 или 1) записывается рядом с частным. То же

проделывается с полученным частным. Процесс повторяется до тех пор, пока не останется 0. В результате должны получиться две колонки чисел — частных и остатков. Двоичное число можно считать из колонки остатков снизу вверх. На рисунке А.4 показано, как происходит преобразование из десятичной в двоичную систему.

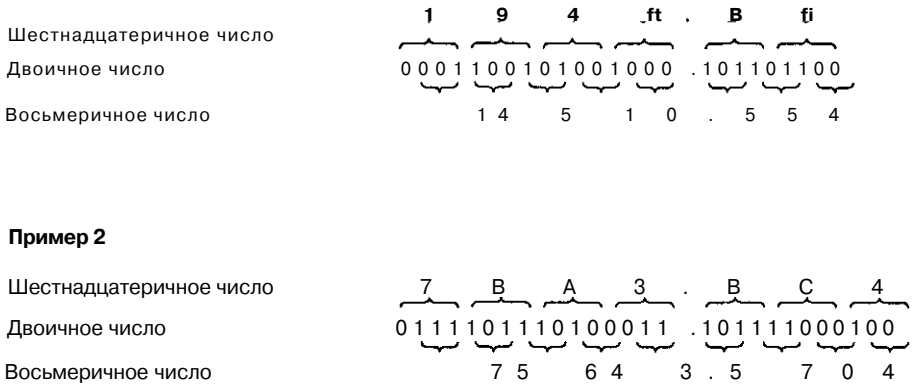


Рис. А.3. Примеры преобразования из 8-ричной системы счисления в двоичную иив 16-ричной в двоичную

Двоичные числа можно преобразовывать в десятичные двумя способами. Первый способ — суммирование степеней двойки, которые соответствуют биту 1 в двоичном числе. Например:

$10110=2^4+2^2+2'=16+4+2=22$

Второй способ. Двоичное число записывается вертикально по одному биту в строке, крайний левый бит находится внизу. Самая нижняя строка — это строка 1, затем идет строка 2 и т. д. Десятичное число строится напротив этой колонки. Сначала обозначим строку 1. Элемент строки *n* состоит из удвоенного элемента строки и-1 плюс бит строки *n* (0 или 1). Элемент, полученный в самой верхней строке, и будет ответом. Метод проиллюстрирован на рис. А.5.

Преобразование из десятичной в восьмеричную или 16-ричную систему можно выполнить либо путем преобразования сначала в двоичную, а затем в нужную нам систему, либо путем вычитания степеней 8 или 16.

Отрицательные двоичные числа

На протяжении всей истории цифровых компьютеров для репрезентации отрицательных чисел использовались 4 различные системы. Первая из них называется системой со знаком. В такой системе крайний левый бит — это знаковый бит (0 — это «+», а 1 — это «-»), а оставшиеся биты показывают абсолютное значение числа.

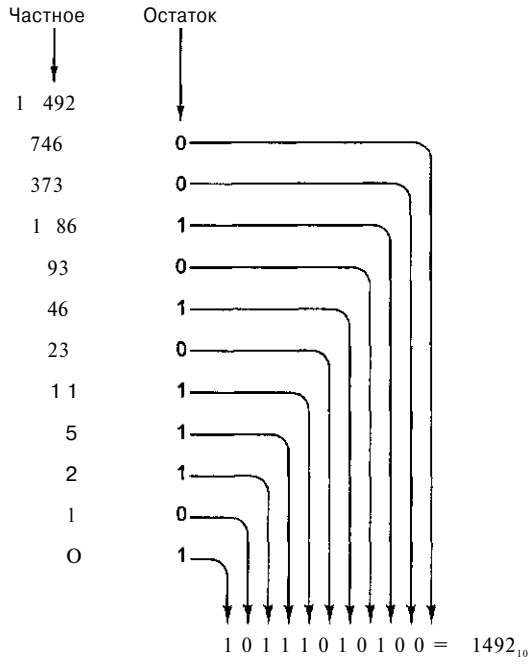


Рис. А.4. Преобразование десятичного числа 1492 в двоичное путем последовательного деления (сверху вниз). Например, 93 делится на 2, получается 46 и остаток 1. Остаток записывается в строку внизу

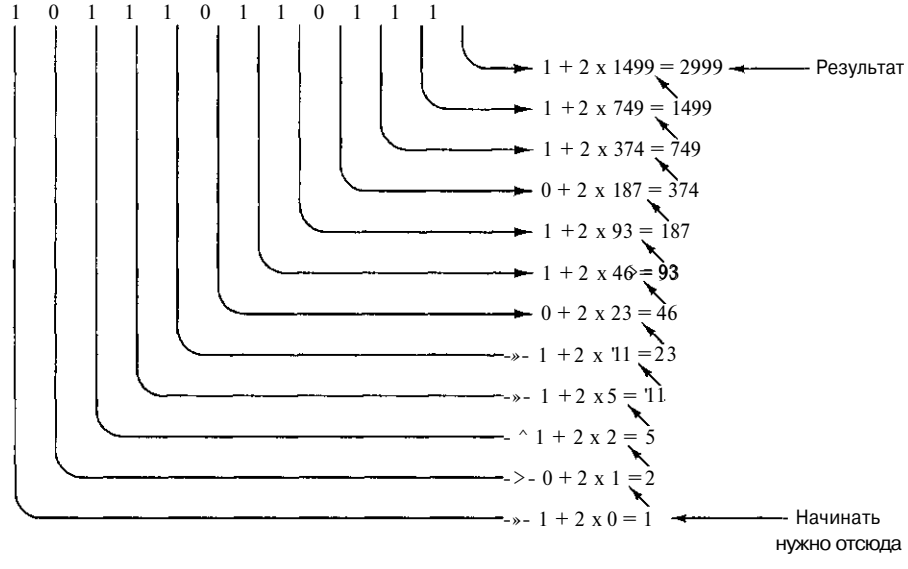


Рис. А.5. Преобразование двоичного числа 101110110111 в десятичное путем последовательного удваивания снизу вверх. В каждой следующей строке удваивается значение предыдущей строки и прибавляется соответствующий бит. Например, 374 умножается на 2 и прибавляется бит соответствующей строки (в данном случае 1). В результате получается 749

Во второй системе, которая называется **дополнением до единицы,** тоже присутствует знаковый бит (0 — это плюс, а 1 — это минус). Чтобы сделать число отрицательным, нужно заменить каждую 1 на 0 и каждый 0 на 1. Это относится и к знаковому биту. Система дополнения до единицы уже устарела.

Третья система, **дополнение до двух,** содержит знаковый бит (0 — это «+», а 1 — это «-»). Отрицание числа происходит в два этапа. Сначала каждая единица меняется на 0, а каждый 0 — на 1 (как и в системе дополнения до единицы). Затем к полученному результату прибавляется 1. Двоичное сложение происходит точно так же, как и десятичное, только перенос совершается в том случае, если сумма больше 1, а не больше 9. Например, рассмотрим преобразование числа 6 в форму с дополнением до двух:

00000110 (+6);
11111001 (-6 в системе с дополнением до единицы);
11111010 (-6 в системе с дополнением до двух).

Если нужно совершить перенос из крайнего левого бита, он просто отбрасывается.

В четвертой системе, которая для ти-битных чисел называется **excess 2^{m}~',** число представляется как сумма этого числа и 2^{m}~K Например, для 8-битного числа (те=8) система называется excess 128, а число сохраняется в виде суммы исходного числа и 128. Следовательно, -3 превращается в -3+128=125, и это число (-3) представляется 8-битным двоичным числом 125 (01111101). Числа от -128 до +127 выражаются числами от 0 до 255 (все их можно записать в виде 8-битного положительного числа). Отметим, что эта система соответствует системе с дополнением до двух с обращенным знаковым битом. В таблице А.2 представлены примеры отрицательных чисел во всех четырех системах.

Таблица А.2. Отрицательные 8-битные числа в четырех различных системах

N десятичное	N двоичное	-N в системе со знаком	-N дополнение до единицы	-N дополнение до двух	-N excess 128
1	00000001	10000001	11111110	11111111	011111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	10101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Не сущест.	Не сущест.	Не сущест.	10000000	00000000

В системах со знаком и с дополнением до единицы есть два представления нуля: +0 и -0. Такая ситуация нежелательна. В системе с дополнением до двух такой проблемы нет, поскольку здесь плюс нуль это всегда плюс нуль. Но зато в этой системе есть другая особенность. Набор битов, состоящий из 1, за которой следуют все нули, является дополнением самого себя. В результате ряд положительных и отрицательных чисел несимметричен — существует одно отрицательное число без соответствующего ему положительного.

Мы считаем это проблемами, поскольку хотим иметь систему кодировки, в которой:

1. Существует только одно представление нуля.
2. Количество положительных чисел равно количеству отрицательных.

Дело в том, что любой ряд чисел с равным количеством положительных и отрицательных чисел и только одним нулем содержит нечетное число членов, тогда как *m* битов предполагают четное число битовых комбинаций. В любом случае будет либо одна лишняя битовая комбинация, либо одной комбинации не будет доставать. Лишнюю битовую комбинацию можно использовать для обозначения числа -0, для большого отрицательного числа или для чего-нибудь еще, но она всегда будет создавать неудобства.

Двоичная арифметика

Ниже приведена таблица сложения для двоичных чисел (рис. А.6).

Первое слогаемое	0	0	1	1
Второе слогаемое	_±P_	_'t l	_±2	_±1
Сумма	0	1	1	0
Перенос	0	0	0	1

Рис. А.6. Таблица сложения для двоичных чисел

Сложение двух двоичных чисел начинается с крайнего правого бита. Суммируются соответствующие биты в первом и втором слагаемом. Перенос совершается на одну позицию влево, как и в десятичной арифметике. В арифметике с дополнением до единицы перенос от сложения крайних левых битов прибавляется к крайнему правому биту. Этот процесс называется циклическим переносом. В арифметике с дополнением до двух перенос, полученный в результате сложения крайних левых битов, просто отбрасывается. Примеры арифметических действий над двоичными числами показаны на рис. А.7.

Если первое и второе слагаемые имеют противоположные знаки, ошибки переполнения не произойдет. Если они имеют одинаковые знаки, а результат — противоположный знак, значит, произошла ошибка переполнения и результат неверен. И в арифметике с дополнением до единицы, и в арифметике с дополнением до двух переполнение происходит тогда и только тогда, когда перенос в знаковый бит

отличается от переноса из знакового бита. В большинстве компьютеров перенос из знакового бита сохраняется, но перенос в знаковый бит не виден из ответа, поэтому обычно вводится специальный бит переполнения.

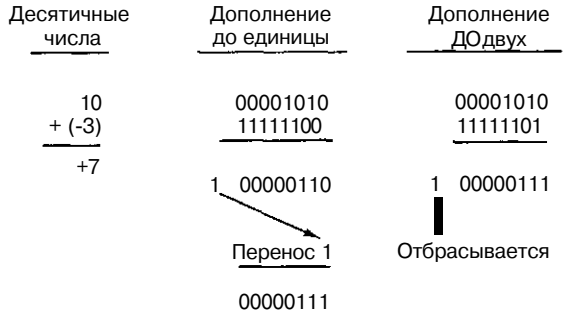


Рис. А.7. Сложение в системах с дополнением до единицы и с дополнением до двух

Вопросы и задания

1. Преобразуйте следующие числа в двоичные: 1984, 4000, 8192.
2. Преобразуйте двоичное число 1001101001 в десятичную, восьмеричную и шестнадцатеричную системы.
3. Какие из следующих цепочек символов являются шестнадцатеричными числами? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.
4. Выразите десятичное число 100 в системах счисления с основаниями от 2 до 9.
5. Сколько различных положительных целых чисел можно выразить в k разрядах, используя числа с основанием системы счисления r?
6. Большинство людей с помощью пальцев на руках могут сосчитать до 10. Однако компьютерщики способны на большее. Представим, что каждый палец соответствует одному двоичному разряду. Пусть вытянутый палец означает 1, а загнутый — 0. До скольки мы можем сосчитать, используя пальцы обеих рук? А если рассматривать пальцы на руках и на ногах? Представим, что большой палец левой ноги — это знаковый бит для чисел с дополнением до двух. Сколько чисел можно выразить таким способом?
7. Выполните следующие вычисления над 8-битными числами с дополнением до двух:

00101101	11111111	00000000	11110111
+01101111	+11111111	-11111111	-11110111

8. Выполните те же вычисления в системе с дополнением до единицы.
9. Ниже приведены задачи на сложение 3-битных двоичных чисел в системе с дополнением до двух. Для каждой суммы установите:
   а. Равен ли знаковый бит результата 1.
   б. Равны ли младшие три бита 0.

в. Не произошло ли переполнения.

000	000	111	100	100
+001	+111	+110	+111	+100

10. Десятичные числа со знаком, состоящие из *n* разрядов, можно представить в га+1 разрядах без знака. Положительные числа содержат 0 в крайнем левом разряде. Отрицательные числа получаются путем вычитания каждого разряда из 9. Например, отрицательным числом от 014725 будет 985274. Такие числа называются числами с дополнением до девяти. Они аналогичны двоичным числам с дополнением до единицы. Выразите следующие числа в виде 3-разрядных чисел в системе с дополнением до девяти: 6, -2, 100, -14, -1, 0.

11. Сформулируйте правило для сложения чисел с дополнением до девяти, а затем выполните следующие вычисления:

0001	0001	9997	9241
+9999	+9998	+9996	+0802

12. Система с дополнением до десяти аналогична системе с дополнением до двух. Отрицательное число в системе с дополнением до десяти получается путем прибавления 1 к соответствующему числу с дополнением до девяти без учета переноса. По какому правилу происходит сложение в системе с дополнением до десяти?

13. Составьте таблицы умножения для чисел системы счисления с основанием 3.

14. Перемножьте двоичные числа 0111 и ООН.

15. Напишите программу, которая на входе получает десятичное число со знаком в виде цепочки ASCII, а на выходе выводит представление этого числа в восьмеричной, шестнадцатеричной и двоичной системе с дополнением до двух.

16. Напишите программу, которая на входе получает 2 цепочки из 32 символов ASCII, содержащие нули и единицы. Каждая цепочка представляет 32-битное двоичное число в системе с дополнением до двух. На выходе программа должна выдавать их сумму в виде цепочки из 32 символов ASCII, содержащей нули и единицы.

Приложение Б
Числа с плавающей точкой

Диапазон чисел, используемых при различных вычислениях, очень велик. Например, в астрономические вычисления может включаться масса электрона ($9x10^{28}$ граммов) и масса Солнца ($2x10^{33}$ граммов). Диапазон чисел здесь превышает 10^{60}. Эти числа можно представить следующим образом:

00000000000000000000000000000000.00000000000000000000000000000009
20000000000000000000000000000000.00000000000000000000000000000000

При всех вычислениях должны сохраняться 34 разряда слева от десятичной запятой и 28 разрядов справа от нее. Это даст 62 значимых разряда в результатах. На бинарном компьютере можно использовать арифметику с многократно увеличенной точностью, чтобы обеспечить достаточную значимость. Однако мы не можем определить массу Солнца с точностью даже до пяти значимых разрядов, не говоря уже о 62. В действительности практически невозможно выполнить какие-либо измерения с точностью до 62 знаков. Можно было бы хранить все промежуточные результаты с точностью до 62 значимых разрядов, а перед выводом окончательных результатов отбрасывать 50 или 60 разрядов, но процессор и память тратили бы на это слишком много времени.

Нам нужна такая система для представления чисел, в которой диапазон выражаемых чисел не зависит от числа значимых разрядов. В этом приложении мы расскажем о такой системе. В ее основе лежит экспоненциальное представление чисел, которое применяется в физике, химии и машиностроении.

Принципы представления с плавающей точкой

Числа можно выражать в следующей общепринятой экспоненциальной форме:

и=/хЮ^е

где / называется **дробью,** или **мантиссой,** а е (это положительное или отрицательное целое число) называется **экспонентой.** Компьютерная версия такого представления называется **представлением с плавающей точкой.** Ниже приведены примеры чисел в такой записи.

3,14 =0,314хЮ^1=3,14x10^0
0,000001 =0,1хЮ-^5=1,0x10-^6
1941 = 0,1941x10^4=1,941x10^3

Область значений определятся по числу разрядов в экспоненте, а точность определяется по числу разрядов в мантиссе. Существует несколько способов представления того или иного числа, поэтому одна форма выбирается в качестве стандартной. Чтобы изучить свойства такого способа представления, рассмотрим представление R с трехразрядной мантиссой со знаком в диапазоне $0,1 \le |f| < 1$ и двухразрядной экспонентой со знаком. Эти числа находятся в диапазоне от $+0,100 \times 10^{-}$ до $+0,999 \times 10^{+}$, то есть простираются почти на 199 значимых разрядов, хотя для записи числа требуется всего 5 разрядов и 2 знака.

Числа с плавающей точкой можно использовать для моделирования системы действительных чисел в математике, хотя здесь есть несколько существенных различий. На рис. Б.1 представлена ось действительных чисел. Она разбита на 7 областей:

1. Отрицательные числа меньше -0,999x10".
2. Отрицательные числа от -0,999x10" до -0,100x10"".
3. Отрицательные числа от -0,1 OOx 10"" до нуля.
4. Нуль.
5. Положительные числа от 0 до 0,1 OOx 10"".
6. Положительные числа от 0,100x10"" до 0,999x10".
7. Положительные числа больше 0,999x10".

Рис. Б. 1. Ось действительных чисел разбита на 7 областей

Первое отличие действительных чисел от чисел с плавающей точкой, которые записываются тремя разрядами в мантиссе и двумя разрядами в экспоненте, состоит в том, что последние нельзя использовать для записи чисел из областей 1,3, 5 и 7. Если в результате арифметической операции получится число из области 1 или 7 (например, $10^{60} \times 10^{60} = 10^{120}$), то произойдет **ошибка переполнения** и результат будет неверным. Причина — ограничение области значений чисел в данном представлении. Точно так же нельзя выразить результат из области 3 или 5. Такая ситуация называется **ошибкой из-за потери значимости.** Эта ошибка менее серьезна, чем ошибка переполнения, поскольку часто нуль является вполне удовлетворительным приближением для чисел из областей 3 или 5. Остаток счета в банке на 10^{-102} не сильно отличается от остатка счета 0.

Второе важное отличие чисел с плавающей запятой от действительных чисел — это их плотность. Между любыми двумя действительными числами *хну* существует другое действительное число независимо от того, насколько близко к *у* расположен *х.* Это свойство вытекает из того, что между любыми различными действи-

тельными числами *хну* существует действительное число *z=(x+y)/2.* Действительные числа формируют континуум.

Числа с плавающей точкой континуума не формируют. В двухзнаковой пятиразрядной системе можно выразить ровно 179100 положительных чисел, 179100 отрицательных чисел и 0 (который можно выразить разными способами), то есть всего 358201 чисел. Из бесконечного числа действительных чисел в диапазоне от -10^{+10^6} до +0,999x10" в этой системе можно выразить только 358201 число. На рис. Б.1 эти числа показаны точками. Результат вычислений может быть и другим числом, даже если он находится в области 2 или 6. Например, результат деления числа $+0{,}100x10^3$ на 3 нельзя выразить *точно* в нашей системе представления. Если полученное число нельзя выразить в используемой системе представления, нужно брать ближайшее число, которое представимо в этой системе. Такой процесс называется **округлением.**

Промежутки между смежными числами, которые можно выразить в представлении с плавающей запятой, во второй и шестой областях не постоянны. Промежуток между числами +0,998x10" и +0,999x10" гораздо больше промежутка между числами +0,998x10° и +0,999x10°. Однако если промежутки между числом и его соседом выразить как процентное отношение от этого числа, большой разницы в промежутках не будет. Другими словами, **относительная погрешность,** полученная при округлении, приблизительно равна и для малых, и для больших чисел.

Выводы, сделанные для системы представления с трехразрядной мантиссой и двухразрядной экспонентой, справедливы и для других систем представления чисел. При изменении числа разрядов в мантиссе или экспоненте просто сдвигаются границы второй и шестой областей и меняется число представляемых единиц в этих областях. С увеличением числа разрядов в мантиссе увеличивается плотность элементов и, следовательно, точность приближений. С увеличением количества разрядов в экспоненте размер областей 2 и 6 увеличивается за счет уменьшения областей 1, 3, 5 и 7. В табл. Б.1 показаны приблизительные границы области 6 для десятичных чисел с плавающей точкой с различным количеством разрядов в мантиссе и экспоненте.

Таблица Б.1. Приблизительные верхняя и нижняя границы чисел с плавающей точкой

Количество разрядов в мантиссе	Количество разрядов в экспоненте	Нижняя граница	Верхняя граница
3	1	Ю-^{12}	10^9
3	2	Ю-102	10"
3	3	Ю-1002	1Q999
3	4	10^{-10002}	10^{9999}
4	1	Ю-^{13}	10^9
4	2	10-ю3	10"
4	3	1 Q-1003	1Q999
4	4	Ю-'0003	Ю9999
5	1	Ю-^{14}	10^9
5	2	Ю-104	10"
5	3	10	1Q999
5	4	10	1 П9999 1 уаааа
10	3	1 Q-1009	Ю^{9}"
20	3	Ю-1019	1Q999

Вариант такого представления применяется в компьютерах. Основа возведения в степень — 2,4,8 или 16, но не 10. В этом случае мантисса состоит из цепочки двоичных, четверичных, восьмеричных и шестнадцатеричных разрядов. Если крайний левый разряд равен 0, все разряды можно сместить на один влево, а экспоненту уменьшить на 1, не меняя при этом значения числа (исключение составляет ситуация потери значимости). Мантисса с ненулевым крайним левым разрядом называется нормализованной.

Нормализованные числа обычно предпочитаются ненормализованным, поскольку существует только одна нормализованная форма, а ненормализованных форм может быть много. Примеры нормализованных чисел с плавающей точкой даны на рис. Б.2. для двух основ возведения в степень. В этих примерах показана 16-битная мантисса (включая знаковый бит) и 7-битная экспонента. Запятая находится слева от крайнего левого бита мантиссы и справа от экспоненты.

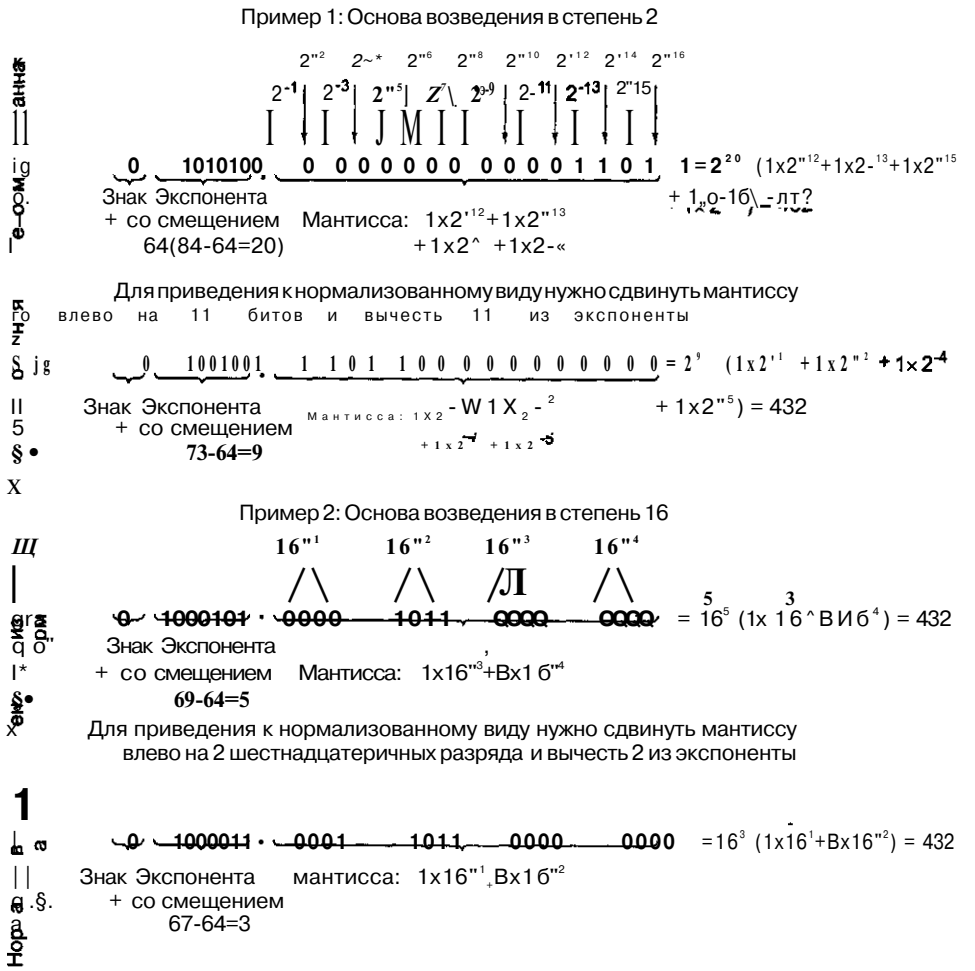


Рис. Б.2. Примеры нормализованных чисел с плавающей точкой

Стандарт IEEE 754

До 80-х годов каждый производитель имел свой собственный формат чисел с плавающей точкой. Все они отличались друг от друга. Более того, в некоторых из них арифметические действия выполнялись неправильно, поскольку арифметика с плавающей точкой имеет некоторые тонкости, которые не очевидны для обычного разработчика аппаратного обеспечения.

Чтобы изменить эту ситуацию, в конце 70-х годов IEEE учредил комиссию для стандартизации арифметики с плавающей точкой. Целью было не только дать возможность переносить данные с одного компьютера на другой, но и обеспечить разработчиков аппаратного обеспечения заведомо правильной моделью. В результате получился стандарт IEEE 754 (IEEE, 1985). В настоящее время большинство процессоров (в том числе Intel, SPARC и JVM) содержат команды с плавающей точкой, которые соответствуют этому стандарту. В отличие от многих стандартов, которые представляли собой неудачные компромиссы и мало кого устраивали, этот стандарт неплох, в большей степени благодаря тому, что его изначально разрабатывал один человек, профессор математики университета Беркли Вильям Каган (William Kahan). Этот стандарт будет описан ниже.

Стандарт определяет три формата: с одинарной точностью (32 бита), с удвоенной точностью (64 бита) и с повышенной точностью (80 битов). Формат с повышенной точностью предназначен для сокращения ошибок округления. Он применяется главным образом в арифметических устройствах с плавающей точкой, поэтому мы не будем о нем говорить. В форматах с одинарной и удвоенной точностью применяется основание возведения в степень 2 для мантисс и смещенная экспонента. Форматы представлены на рис. Б.3.

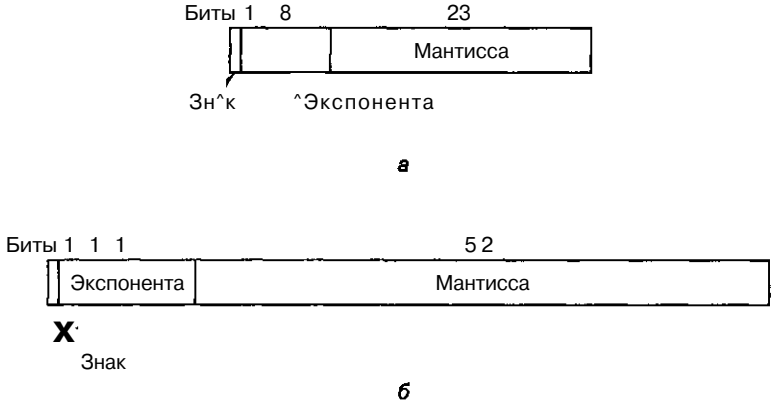


Рис. Б.3. Форматы для стандарта IEEE с плавающей точкой: одинарная точность (а); удвоенная точность (б)

Оба формата начинаются со знакового бита для всего числа; 0 указывает на положительное число, а 1 — на отрицательное. Затем следует смещенная экспонента. Для формата одинарной точности смещение (excess) 127, а для формата удвоенной точности смещение 1023. Минимальная (0) и максимальная (255 и 2047)

экспоненты не используются для нормализованных чисел. У них есть специальное предназначение, о котором мы поговорим ниже. В конце идут мантиссы по 23 и 52 бита соответственно.

Нормализованная мантисса начинается с двоичной запятой, за которой следует 1 бит, а затем остаток мантиссы. Следуя практике, начатой с компьютера PDP-11, компьютерщики осознали, что 1 бит перед мантиссой сохранять не нужно, поскольку можно просто предполагать, что он есть. Следовательно, стандарт определяет мантиссу следующим образом. Она состоит из неявного бита, который всегда равен 1, неявной двоичной запятой, за которыми идут 23 или 52 произвольных бита. Если все 23 или 52 бита мантиссы равны 0, то мантисса имеет значение 1,0. Если все биты мантиссы равны 1, то числовое значение мантиссы немного меньше, чем 2,0. Во избежание путаницы в английском языке для обозначения комбинации из неявного бита, неявной двоичной запятой и 23 или 52 явных битов вместо термина «мантисса» (mantissa) используется термин significand. Все нормализованные числа имеют significand *s* в диапазоне $1 \leq s < 2$.

Числовые характеристики стандарта IEEE для чисел с плавающей точкой даны в табл. Б.2. В качестве примеров рассмотрим числа 0,5,1 и 1,5 в нормализованном формате с одинарной точностью. Они представлены шестнадцатеричными числами 3F000000, 3F800000 и 3FC00000 соответственно.

Таблица Б.2. Характеристики чисел с плавающей точкой стандарта IEEE

Параметр	Одинарная точность	Удвоенная точность
Количество битов в знаке	1	1
Количество битов в экспоненте	8	11
Количество битов в мантиссе	23	52
Общее число битов	32	64
Смещение экспоненты	Смещение (excess) 127	Смещение (excess) 1023
Область значений экспоненты	От -126 до +127	от -1022 до +1023
Самое маленькое нормализованное число	2-126	2-Ю22
Самое большое нормализованное число	$=2^{128}$	$=2^{1024}$
Диапазон десятичных дробей	$= 10^{-м}$ до 10^{38}	И 0^{-308} до 10^{308}
Самое маленькое ненормализованное число	$=Ю^{-45}$	«$Ю^{-324}$

Традиционные проблемы, связанные с числами с плавающей точкой, — что делать с переполнением, потерей значимости и неинициализированными числами. Подход, используемый в стандарте IEEE, отчасти заимствован от машины CDC 6600. Помимо нормализованных чисел в стандарте предусмотрено еще 4 типа чисел (рис. Б.4).

Проблема возникает в том случае, если абсолютное значение (модуль) результата меньше самого маленького нормализованного числа с плавающей точкой, которое можно представить в этой системе. Раньше аппаратное обеспечение действовало одним из двух способов: либо устанавливало результат на 0, либо вызывало ошибку из-за потери значимости. Ни один из этих двух способов не является удовлетворительным, поэтому в стандарт IEEE введены **ненормализованные числа.** Эти числа имеют экспоненту 0 и мантиссу, представленную следующими 23 или

52 битами. Неявный бит 1 слева от двоичной запятой превращается в 0. Ненормализованные числа можно легко отличить от нормализованных, поскольку у последних не может быть экспоненты 0.

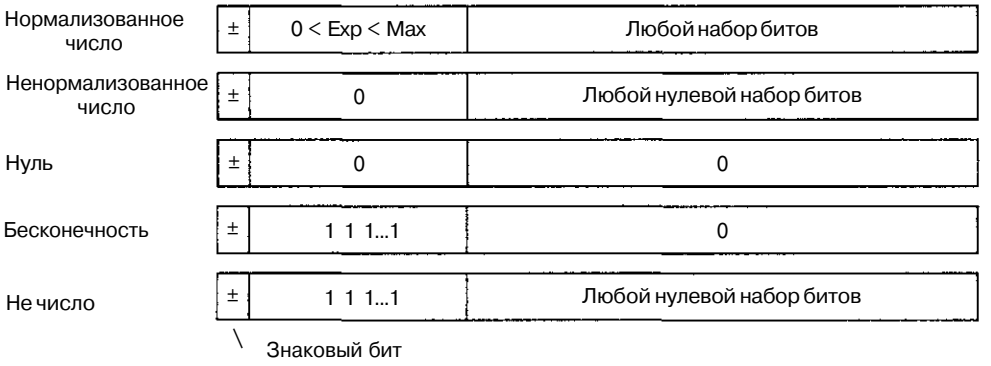


Рис. Б.4. Числовые типы стандарта IEEE

Самое маленькое нормализованное число с одинарной точностью содержит 1 в экспоненте и 0 в мантиссе и представляет $1{,}0 \times 2^{-126}$. Самое большое ненормализованное число содержит 0 в экспоненте и все единицы в мантиссе и представляет примерно $0{,}9999999 \times 2^{-127}$, то есть почти то же самое число. Следует отметить, что это число содержит только 23 бита значимости, а все нормализованные числа — 24 бита.

По мере уменьшения результата при дальнейших вычислениях экспонента попрежнему остается равной 0, а первые несколько битов мантиссы превращаются в нули, что сокращает и значение, и число значимых битов мантиссы. Самое маленькое ненулевое ненормализованное содержит 1 в крайнем правом бите, а все остальные биты равны 0. Экспонента представляет 2^{-127}, а мантисса — 2^{-23}, поэтому значение равно 2^{-150}. Такая схема предусматривает постепенное исчезновение значимых разрядов, а не перескакивает на 0, когда результат нельзя выразить в виде нормализованного числа.

В этой схеме присутствуют 2 нуля, положительный и отрицательный, определяемые по знаковому биту. Оба имеют экспоненту 0 и мантиссу 0. Здесь тоже бит слева от двоичной запятой по умолчанию 0, а не 1.

С переполнением нельзя справиться постепенно. Вместо этого существует специальное представление бесконечности: с экспонентой, содержащей все единицы, и мантиссой, равной 0. Это число можно использовать в качестве операнда. Оно подчиняется обычным математическим правилам для бесконечности. Например, бесконечность и любое число в сумме дают бесконечность. Конечное число разделить на бесконечность равно 0. Любое конечное число, разделенное на 0, стремится к бесконечности.

А что получится, если бесконечность разделить на бесконечность? Результат не определен. Для такого случая существует другой специальный формат, NaN (Not a Number — не число). Его тоже можно использовать в качестве операнда.

Вопросы и задания

1. Преобразуйте следующие числа в формат стандарта IEEE с одинарной точностью. Результаты представьте в восьми шестнадцатеричных разрядах.
   а. 9
   б. 5/32
   в. -5/32
   г. 6.125
2. Преобразуйте следующие числа с плавающей точкой одинарной точности из шестнадцатеричной в десятичную систему счисления:
   а. 42E28000H
   б. 3F880000H
   в. 00800000H
   г. C7F00000H
3. Число с плавающей точкой в формате одинарной точности в IBM/370 состоит из 7-битной смещенной экспоненты (смещение 64), 24-битной мантиссы и знакового бита. Двоичная запятая находится слева от мантиссы. Основание возведения в степень — 16. Порядок полей — знаковый бит, экспонента, мантисса. Выразите число 7/64 в виде нормализованного шестнадцатеричного числа в этой системе.
4. Следующие двоичные числа с плавающей точкой состоят из знакового бита, смещенной экспоненты (смещение 64) с основанием 2 и 16-битной мантиссы. Нормализуйте их.
   а. 0 1000000 0001010100000001
   б. 00111111000001111111111
   в. 0 10000111000000000000000
5. Чтобы сложить два числа с плавающей точкой, нужно уровнять экспоненты (сдвинув мантиссу). Затем можно сложить мантиссы и нормализовать результат, если в этом есть необходимость. Сложите числа одинарной точности 3EE00000H и 3D800000H и выразите нормализованный результат в шестнадцатеричной системе счисления.
6. Компьютерная компания решила выпустить машину с 16-битными числами с плавающей точкой. В модели 0.001 формат состоит из знакового бита, 7-битной смещенной экспоненты (смещение 64) и 8-битной мантиссы. В модели 0.002 формат состоит из знакового бита, 5-битной смещенной экспоненты (смещение 16) и 10-битной мантиссы. В обеих моделях основание возведения в степень равно 2. Каково самое маленькое и самое большое положительное нормализованное число в этих моделях? Сколько десятичных разрядов точности содержится в каждой модели? А вы купили бы какую-нибудь из этих двух моделей?

7. Существует одна ситуация, при которой операция над двумя числами с плавающей точкой может вызвать сильное сокращение количества значимых битов в результате. Что это за ситуация?
8. Некоторые микросхемы с плавающей точкой имеют встроенную команду квадратного корня. Возможно применение итерационного алгоритма (например, метода Ньютона—Рафсона). Итерационные алгоритмы дают последовательные приближения решения. Как можно быстро получить приближенный квадратный корень от числа с плавающей точкой?
9. Напишите процедуру сложения двух чисел одинарной точности с плавающей точкой. Каждое число представлено 32-элементным логическим массивом.
10. Напишите процедуру сложения двух чисел с плавающей точкой одинарной точности, в которых для экспоненты используется основание системы счисления 16, а для мантиссы — основание системы счисления 2 и которые не содержат неявного бита 1 слева от двоичной точки. В нормализованном числе крайние левые 4 бита мантиссы могут быть 0001, 0010,..., 1111, но не 0000. Число нормализуется путем сдвига мантиссы влево на 4 бита и прибавления 1 к экспоненте.

Алфавитный указатель

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

И

V

W

X

Z

А

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я