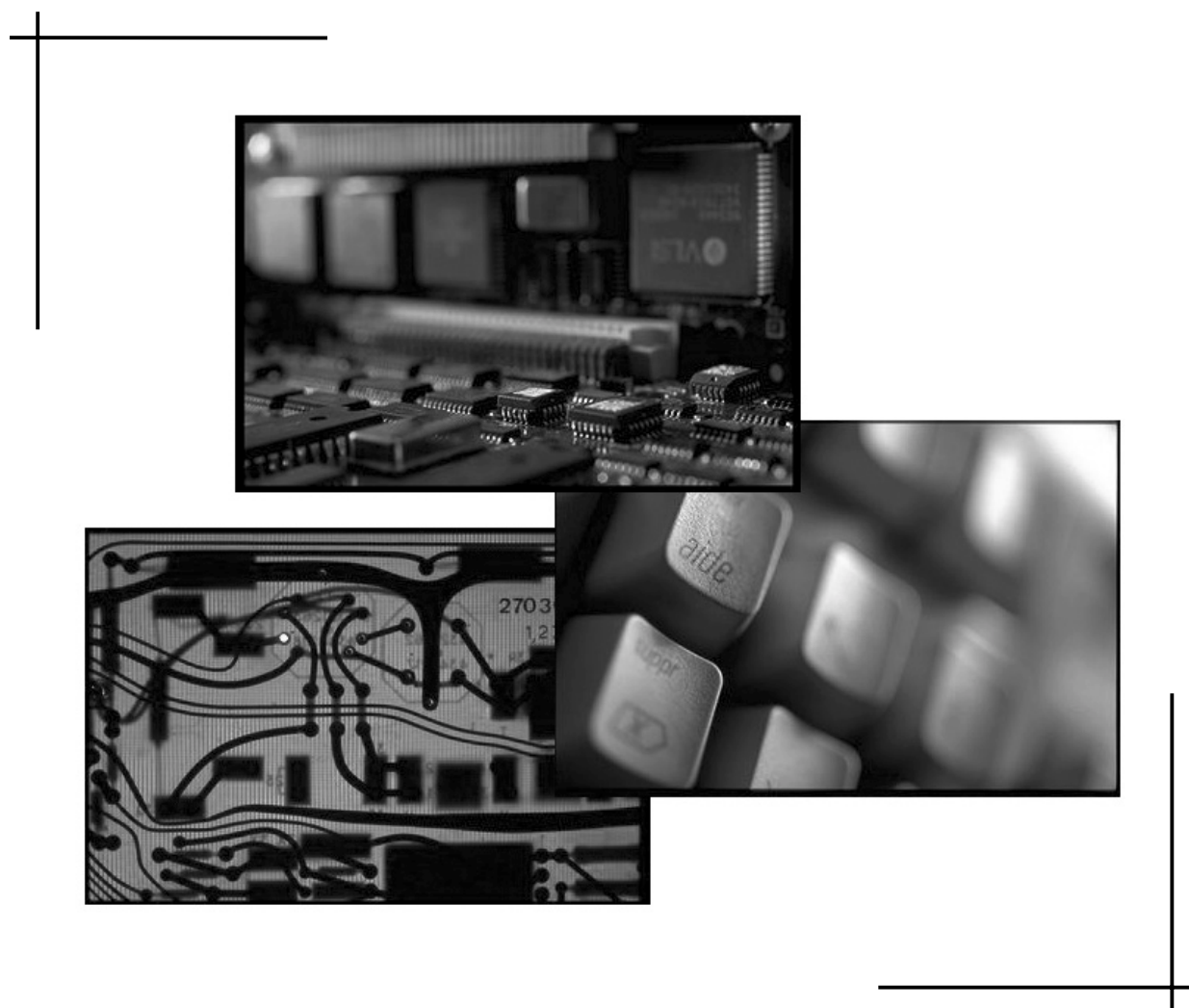


Mihai ROMANCA

Gyula SZEKELY

Calculatoare Arhitectură și organizare



**Universitatea „Transilvania” Braşov
2004**

Mihai ROMANCA

Gyula SZEKELY

CALCULATOARE

- Arhitectură și organizare -

2004

©2004 EDITURA UNIVERSITĂȚII TRANSILVANIA BRAȘOV

Adresa: 500030 Brașov,
B-dul Eroilor, Nr. 9
Tel/Fax: 0268 – 47 53 48
E-mail: editura@unitbv.ro

Tipărit la:
Tipografia Universității “transilvania” din Brașov
B-dul Eroilor 9
tel/fax: 0268 47 53 48

Toate drepturile rezervate

Editură acreditată de CNCSIS
Adresa nr. 1615 din 29 mai 2002

Referenți științifici: Prof. dr. ing. Gheorghe TOACȘE
Prof. dr. ing. Petre OGRUȚAN

Tehnoredactare: Autorul
Corectură: Autorul

Descriere CIP a Bibliotecii Naționale a României
ROMANCA, MIHAI

Calculatoare: arhitectură și organizare/ Romanca Mihai,
Szekely Gyula. – Brașov: Editura Universității “Transilvania”, 2004
Bibliogr.
ISBN 973-635-309-5

004

CUVÂNT ÎNAINTE

Cartea de față prezintă principalele aspecte ale arhitecturii și organizării calculatoarelor numerice. Materialul este organizat astfel încât cunoștințele să poată fi câștigate gradual pornind de la aspectele elementare ale organizării până la extensiile actuale ale arhitecturii calculatorului. Materialul este dedicat în special studenților din învățământul ingineresc de profil electric, constituind suportul pentru cursul de Calculatoare predat studenților de la specializările Electrotehnică și Electromecanică, din cadrul Facultății de Inginerie Electrică și Știința Calculatoarelor de la Universitatea *TRANSILVANIA* din Brașov. Obiectivele generale pe care ni le-am propus în realizarea materialului sunt:

- Prezentarea principalelor noțiuni privind arhitectura setului de instrucțiuni, principii de funcționare și de organizare ale diferitelor variante de implementare a calculatoarelor numerice.
- Clasificarea și exemplificarea arhitecturilor moderne actuale, descrierea și analiza organizării calculatoarelor.
- În urma asimilării cunoștințelor predate la această disciplină, viitorul inginer electrotehnician își va putea asuma responsabilitatea punerii în funcțiune a diferitelor sisteme de comandă cu calculatorul numeric, a întreținerii sau a proiectării și construcției diferitelor interfețe (acestea fiind realizabile și prin studiul suplimentar al documentațiilor de firmă).

În cadrul cursului se prezintă *conceptele generale* privind arhitectura mașinilor electronice de calcul numeric, concepte aplicabile la orice mașină particulară. Se fac, de asemenea, dese referiri, cu titlu de exemplu, la *arhitecturi particulare*, de calculatoare existente pe piață. S-a ales această metodă de abordare, pentru că domeniul este extrem de dinamic. Arhitecturile specifice diferitelor procesoare, necesare eventual în practica viitorului inginer, sunt tratate în documentațiile de firmă, și chiar dacă în viitorul apropiat vor apărea noi implementări sau chiar noi arhitecturi, conceptele de bază, relativ "stabile" vor fi cele care vor ușura înțelegerea.

Materialul cursului de față este destinată *sistemelor uni-procesor*, ca o cale către studierea ulterioară de către studenți a principiilor de organizare și de utilizare a sistemelor de procesare paralelă.

Am organizat cartea 7 capitole, cuprinzând materia ce se predă în cele 14 săptămâni ale semestrului 7 studenților electrotehnicieni. Cartea cuprinde numeroase note de subsol, mai ales cu explicații ale unor denumiri provenite din limba engleză. Unele din noțiunile englezești din domeniul calculatoarelor sunt deja clasice, dar altele încă nu au căpătat o traducere exactă și apar adesea ca "barbarisme" în limba română. Chiar dacă se face explicarea în limba română pentru termenii folosiți peste tot se indică și termenul "original" din limba engleză. Ne-am ferit astfel de a călca în păcatul unei traduceri care să se îndepărteze de la sensul inițial, pentru că așa cum spun italienii "Traduttore - traditore!".

Așteptăm cu interes observațiile studenților și colegilor la materialul de față prin adresele noastre de e-mail: romanca@unitbv.ro și szekelyg@vega.unitbv.ro.

Autorii

CUPRINS

1. INTRODUCERE ÎN ARHITECTURA CALCULATOARELOR	1
1.1. Calculatorul numeric.....	2
1.2. Arhitectura calculatorului numeric	4
1.3. Organizarea generală a unui sistem uni – procesor	5
1.4. Tendințe ale tehnologiei.....	7
1.5. Dezvoltare a aplicațiilor calculatoarelor numerice	8
1.6. Software (programe de sistem și aplicații)	9
1.7. Clasificarea în ordine a sistemelor numerice	10
1.8. Reprezentarea funcțională a unui calculator	11
1.9. Reprezentarea structurală a unui calculator	15
1.10. Scurtă privire istorică asupra dezvoltării mașinilor de calcul	18
 2. MODALITĂȚI DE EVALUARE A PERFORMANȚELOR CALCULATOARELOR.....	 31
2.1. Timpul, ca măsură a performanțelor.....	32
2.2. Alte unități de măsură	37
2.3. Programe etalon de testare a performanțelor	39
2.4. Compararea performanțelor prin mediere.....	42
2.5. Legea lui Amdahl.....	46
 3. FUNDAMENTE PRIVIND ARHITECTURA UNITĂȚII CENTRALE DE PROCESARE (UCP)	 49
3.1. Structura de procesor	50
3.2. Structura unei Unități Centrale de Procesare elementare	56
3.3. Perfecționarea Unității Centrale de Procesare elementare.....	59
3.4. Alte registre interne semnificative pentru UCP	61
3.5. Exemple privind operațiile UCP	64
3.6. Semnale la interfața UCP cu exteriorul	67
3.6.1. Magistrala de adrese.....	67
3.6.2. Magistrala de date	70
3.6.3. Magistrala de control	70
3.7. Reprezentarea numerelor în calculator	77
3.7.1. Reprezentarea cu virgulă fixă	77
3.7.2. Coduri binare folosite pentru reprezentarea numerelor cu semn	78
3.8. Reprezentarea numerelor în virgulă mobilă.....	83
 4. ARHITECTURA SETULUI DE INSTRUCȚIUNI.....	 89
4.1. Introducere în arhitectura setului de instrucțiuni	90
4.2. Formatul instrucțiunilor	91
4.3. Interdependența set de instrucțiuni - organizare internă a procesorului	94
4.4. Scurtă privire comparativă între arhitecturile RISC și CISC.....	98
4.5. Tipuri de instrucțiuni	99
4.5.1. Instrucțiuni aritmetice	100
4.5.2. Instrucțiuni logice	100
4.5.3. Instrucțiuni pentru transferul informației.....	101
4.5.4. Instrucțiuni pentru deplasarea și rotirea datelor.....	102
4.5.5. Instrucțiuni de ramificare (pentru controlul secvenței de program) ..	103
4.5.6. Instrucțiuni pentru controlul procesorului	105
4.5.7. Instrucțiuni pentru lucrul cu șiruri	105

4.6. Moduri de adresare	106
4.6.1. Adresare imediată	107
4.6.2. Adresare directă	108
4.6.3. Adresare indirectă	110
4.6.4. Adresare relativă	112
4.6.5. Adresare indexată.....	113
5. UNITATEA DE CONTROL A UCP	115
5.1. Funcțiile principale ale unității de control a UCP.....	116
5.2. Control cablat.....	118
5.2.1. Metoda tabelului de tranziție a stărilor	119
5.2.2. Metoda sincronizării prin numărătoare de secvențe	120
5.3. Controlul microprogramat	122
5.3.1. Metode de optimizare a UC microprogramate.....	125
5.4. Paralelism în execuția instrucțiunilor prin tehnici pipeline	129
5.5. Întreruperi și excepții	132
5.6. Rolul memoriei stivă.....	139
5.7. Utilizarea ferestrelor de registre	143
6. SISTEMUL DE MEMORIE.....	147
6.1. Organizarea ierarhică a sistemului de memorie.....	148
6.2. Caracteristici principale ale dispozitivelor de memorie	151
6.3. Structura și organizarea memoriilor statice cu acces aleator (SRAM).....	154
6.4. Structura și funcționarea memoriilor dinamice cu acces aleator (DRAM)	159
6.4.1. Structura și modul de lucru al unei DRAM convențional.....	160
6.4.2. Moduri de lucru de mare viteză ale DRAM.....	162
6.4.3. Moduri de reîmprospătare.....	165
6.4.4. Exemple de circuite DRAM cu interfață asincronă	167
6.4.5. Circuite DRAM sincrone (SDRAM)	168
6.5. Memoria cache.....	169
6.6. Tehnici de adresare și alocare a memoriei.....	173
6.7. Tehnici de translatare a adreselor	175
6.7.1. Maparea adreselor folosind pagini.....	176
6.7.2. Mapare prin segmentare.....	178
6.7.3. Mapare segmentat-paginată	179
6.7.4. Memoria virtuală.....	181
6.7.5. Noțiuni privind protecția memoriei	184
7. SISTEMUL DE INTRARE – IEȘIRE	187
7.1. Circuite de interfață	188
7.2. Organizarea ierarhică a magistralelor	191
7.3. Transferuri asincrone de date.....	193
7.3.1. Transmisie asincronă cu un singur semnal de control	193
7.3.2. Transmisie asincronă cu două semnale de control.....	194
7.4. Modalități de transfer de intrare-ieșire.....	196
7.4.1. Transferul de I/O prin program.....	196
7.4.2. Transferul I/O prin întreruperi	198
7.4.3. Transferul prin acces direct la memorie.....	201
BIBLIOGRAFIE.....	204

CAPITOLUL 1

INTRODUCERE ÎN ARHITECTURA CALCULATOARELOR

Conținut:

- 1.1. Calculatorul numeric
- 1.2. Arhitectura calculatorului numeric
- 1.3. Organizarea generală a unui sistem uni - procesor
- 1.4. Tendințe ale tehnologiei
- 1.5. Dezvoltare a aplicațiilor calculatoarelor numerice
- 1.6. Software (programe de sistem și aplicații)
- 1.7. Clasificarea în ordine a sistemelor numerice
- 1.8. Reprezentarea funcțională a unui calculator
- 1.9. Reprezentarea structurală a unui calculator
- 1.10. Scurtă privire istorică asupra dezvoltării mașinilor de calcul

Obiective educaționale:

- a. Definirea calculatorului numeric, a modului general de organizare și identificarea principalelor aspecte privind noțiunea de “arhitectură” a calculatorului
- b. Clasificarea domeniilor de aplicație pentru calculatoarele numerice
- c. Definirea unor termeni cheie pentru arhitectura calculatorului
- d. Descrierea principalelor elemente privind evoluția calculatoarelor numerice.

1.1. CALCULATORUL NUMERIC

Un calculator numeric este constituit dintr-un ansamblu de resurse fizice (hardware) și de programe de sistem (software de sistem) care asigură prelucrarea automată a informațiilor, în conformitate cu algoritmi specificați de utilizator prin programele de aplicații (software utilizator).

Descrierea unui calculator, ca sistem digital complex, se poate face ținând cont de diferite aspecte [Nicula97]. Astfel, la nivel înalt, calculatorul poate fi definit ca o mașină care ușurează o anumită activitate umană. Acest aspect “*uman*” al calculatorului a condus la studii care vizează ușurarea utilizării calculatorului, ergonomia locului de muncă, creșterea accesibilității calculatorului pentru utilizatorul uman ce are o pregătire diversă.

Calculatorul poate fi definit, de asemenea, ca fiind o mașină *care rulează programe*. Studiile efectuate pe acest nivel vizează optimizarea calculatorului în scopul minimizării timpului de execuție a programelor (aplicațiilor). Modul de scriere al unui program trebuie să permită ca aplicațiile să fie portabile¹ de pe o mașină pe alta. Importanța aspectului “*software*” al calculatorului este determinată de necesitatea de a proteja investiția în programare, în condițiile mediului deosebit de dinamic al calculatoarelor.

Calculatorul poate fi definit ca fiind un sistem digital care realizează un set de funcțiuni. Studiile efectuate pe acest nivel vizează realizarea unor funcții date la viteze maxime și resurse fizice (suprafață pe chip) minime, într-o tehnologie accesibilă la un moment dat. Acesta este aspectul “*hardware*” al calculatorului. Hardware este termenul general care desemnează resursele fizice (circuitele, dispozitivele și echipamentele componente) ale unui calculator numeric;

Cele trei aspecte, uman, software și hardware, nu sunt independente. Perfecționarea unuia dintre aspecte determină și progresul celorlalte două, chiar dacă nu în același ritm. De exemplu, randamentul muncii unui utilizator de calculator (îmbunătățire a aspectului “*uman*”) va crește dacă acesta își schimbă tastatura cu una optimizată ergonomic, în condițiile păstrării resurselor hardware și a programelor rulate (software). Randamentul muncii va crește mai mult dacă se rulează, programe mai puternice, de exemplu prin algoritmi mai rapizi sau sisteme de operare mai performante (îmbunătățire software), în condițiile păstrării acelorași resurse hardware. O creștere remarcabilă a randamentului muncii se obține, însă, dacă se înlocuiește calculatorul cu unul mai performant (îmbunătățire a aspectului hardware), în condițiile păstrării tastaturii și a aplicațiilor rulate. Aspectele software și hardware se întrepătrund și nu pot exista independent. Studiul unui calculator numeric poate fi realizat la diferite niveluri, organizate ierarhic în schema din figura 1.1. De menționat că, pe toate aceste niveluri, aspectele software și hardware coexistă.

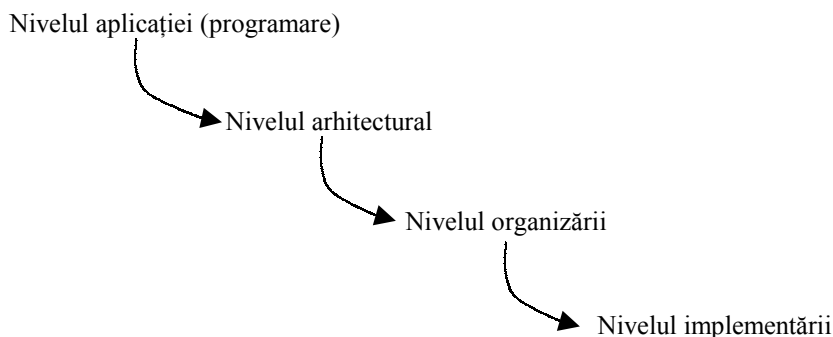


Figura 1.1. Reprezentare ierarhică simplificată a nivelurilor de descriere a unui calculator numeric.

¹ Portabil - aplicația este portabilă dacă este capabilă să funcționeze pe mai multe platforme hardware.

Prin nivelul de *programare (al aplicațiilor)* calculatorul este văzut ca un ansamblu de resurse care participă la executarea unui program. Este nivelul pe care lucrează programatorul într-un limbaj de nivel înalt, având la dispoziție un compilator care deține informațiile detaliate despre nivelele inferioare. Pe acest nivel se operează cu noțiuni ca *multiprocesoare, memorie distribuită, rețele de interconectare*.

Pe nivelul *arhitectural*, care materializează, în mod tradițional, *interfața software-hardware*, se încearcă maximizarea performanțelor prin optimizarea *setului de instrucțiuni, tratării întreruperilor, managementului și a protecției memoriei*. Este nivelul pe care programatorul poate accesa toate resursele microprocesorului², prin instrucțiuni specifice. Deoarece atât domeniul hardware cât și cel software evoluează rapid, în foarte puține cazuri arhitecții unui procesor au posibilitatea să înceapă "de la zero" proiectarea unui procesor. De cele mai multe ori, proiectanții primesc ca "date de intrare" arhitectura și chiar setul de instrucțiuni cu care noul procesor trebuie să fie compatibil. Menținerea compatibilității, deși îngreunează proiectarea, este obligatorie datorită faptului că nimeni nu este dispus să renunțe, de exemplu din trei în trei ani, la toate datele acumulate și toată investiția făcută în software. Pe acest nivel se definește noțiunea de *compatibilitate binară*, ca fiind posibilitatea unei mașini de a rula un program scris pentru un procesor dintr-o generație mai veche. Menținerea compatibilității binare implică, pe lângă moștenirea setului de instrucțiuni, și păstrarea modelului secvențial de execuție a instrucțiunilor. Aceste două constrângeri se dovedesc a fi foarte dure în cazul procesoarelor moderne, superscalare, care încearcă să elimine o mare parte din secvențialitatea programului și să îl transforme într-un program paralel, de înaltă performanță, dar care să păstreze *aparența* de program secvențial.

Pe nivelul *organizării*, denumit uneori și nivelul de *microarhitectură*, se precizează modul de implementare a arhitecturii microprocesorului. O arhitectură de microprocesor poate avea diferite organizări, obținându-se astfel procesoare cu performanțe optimizate pentru un anumit tip de aplicații dar compatibile la nivelul programatorului în limbaj de asamblare. Dintre noțiunile cu care se operează la acest nivel se pot preciza:

- Numărul, tipul și localizarea unităților funcționale;
- Descrierea setului de registre;
- Structura pipeline a căii de date;
- Ierarhia memoriei, localizarea și configurația memoriei cache.

Pe nivelul de *implementare*, microprocesorul este văzut ca un sistem digital implementat într-o tehnologie specifică. Aspectele tehnologice limitează deciziile de pe nivele superioare. Aici se stabilește bugetul de tranzistoare pus la dispoziție, buget ce trebuie cunoscut de către proiectanții de arhitecturi. Întrebările care își găsesc răspunsul pe acest nivel sunt:

- Cum se plasează unitățile funcționale pe chip (floorplan)
- Cum se poate implementa o unitate funcțională cu un număr minim de tranzistoare, pe o suprafață minimă de circuit ?

De-a lungul timpului, performanțele calculatoarelor nu au crescut în ritm constant. Există două forțe care stimulează dezvoltarea calculatoarelor:

- îmbunătățirea tehnologică și
- inventarea unor noi arhitecturi.

În primii 25 de ani ai calculatoarelor electronice (1945-1970), ambele forțe au determinat în egală măsură creșterea performanțelor. După 1970, tehnologia (numărul redus de tranzistoare posibil de utilizat) a limitat ritmul de creștere, care a continuat totuși să rămână la 25-30% pe an în domeniul minicalculatoarelor și mainframe³-urilor. Apariția microprocesorului a adus posibilitatea

² Microprocesor = Unitate Centrală de procesare implementată într-un singur circuit integrat. Conține pe aceeași așchie de siliciu: Unitate de control, Unitate aritmetică și logică și registre de uz general folosite ca memorie locală de mare viteză.

³mainframes = calculatoare de mari dimensiuni, cu putere mare de calcul, având de obicei o arhitectură multi-procesor și unități funcționale multiple.

de integrare a unui număr mare de tranzistoare ceea ce a permis un ritm de creștere a performanțelor de 35% pe an. Ritmul de creștere a fost și mai mare după 1980, atunci când a fost determinat de apariția unor idei arhitecturale noi, ce au condus la crearea procesoarelor RISC.

Între procesoarele existente în prezent pe piață și de asemenea între calculatoarele construite cu ajutorul acestor procesoare există o mare diversitate, diversitate ce se manifestă din mai multe puncte de vedere:

- al tehnologiei folosite pentru implementarea lor,
- al caracteristicilor funcționale (proprietăți și funcții observabile din exterior) și
- al organizării structurii⁴ lor interne.

Toate aceste trei aspecte principale constituie ceea ce este cunoscut sub numele de **arhitectura unui calculator**.

1.2. ARHITECTURA CALCULATORULUI NUMERIC

Termenul "arhitectură" este o abstractizare a proprietăților unui calculator. Ca urmare a evoluției funcțiilor și structurii calculatoarelor și noțiunea de arhitectura a evoluat, reprezentând în prezent mai mult decât "interfața" între hardware și software definită inițial de cercetătorii de la firma IBM⁵ în deceniul al șaselea al acestui secol. Un arhitect de calculatoare proiectează mașini care vor rula programe, iar sarcinile sale de proiectare vor include: proiectarea setului de instrucțiuni, organizarea funcțională, proiectarea logică și implementarea. Implementarea cuprinde totul începând de la circuitele integrate utilizate, până la aspectele privind puterea consumată și tipul de răcire.

Arhitectura, în sensul inițial de definire, ca interfață între hardware și software, îngloba funcțiuni în hardware, prin micro-programare (microcod). Orice mașină ulterioară unei mașini din aceeași serie, este obligată - prin arhitectura definită - să recunoască setul de instrucțiuni al celei vechi, chiar dacă are și funcțiuni suplimentare, [Patterson94]. La momentul introducerii sale, noțiunea de arhitectură a calculatorului se referea doar la ceea ce astăzi se înțelege prin **arhitectura setului de instrucțiuni**. Aceasta este o interfață între resursele hardware și resursele cele mai "rudimentare" de software (cod mașina - forma binară a programelor înțeleasă de mașină).

Arhitectura setului de instrucțiuni este ceea ce trebuie să știe un programator pentru a scrie programe în limbaj de asamblare, respectiv pentru a concepe și construi un program de tip compilator, sau rutine destinate sistemului de operare. Setul de instrucțiuni, ca și componentă arhitecturală, permite atât proiectantului cât și utilizatorului procesorului, să vorbească despre funcții, independent de suportul hardware specific care le realizează. Arhitectura setului de instrucțiuni (notată în continuare cu ASI⁶), ca interfață între hardware și software, permite mai multor implementări, cu costuri și performanțe diferite, să ruleze același software.

Noțiunea de arhitectură trebuie privită, prin analogie cu înțelesul clasic al noțiunii de arhitectură⁷, care se referea la construcții, nu numai în sens declarativ al ASI (care definește un set de proprietăți abstracte, un plan) dar și în sens procedural, al unei discipline de proiectare, care implică procesul de producere și implementare a acestor proprietăți abstracte. Aceasta a doua componentă a noțiunii de arhitectura, care se referea la aspectele de **implementare** a funcțiilor (proiectare și realizare a circuitelor ce realizează funcțiile), are la rândul său două componente: organizare și hardware.

⁴ Structură [MDE72] = Alcătuirea internă a unui corp sau sistem fizic, mod de dispunere a elementelor componente ale unui ansamblu, mod specific de organizare a elementelor constitutive ale unei limbi, mod de organizare a unei mulțimi de elemente, prin definirea uneia sau mai multor operații care au anumite proprietăți.

⁵ IBM = International Business Machine

⁶ ASI = Arhitectura Setului de Instrucțiuni (în limba engleză ISA = Instruction Set Architecture)

⁷ Arhitectură [MDE72] = Știința și arta de a proiecta și a construi clădiri.

Termenul **organizare** include aspectele de nivel înalt ale unui proiect de calculator, ca de exemplu organizarea sistemului de memorie, structura și organizarea magistralelor și proiectul intern al UCP. De exemplu, [Patterson90], două mașini cu aceeași arhitectură a setului de instrucțiuni, dar organizări diferite, sunt VAX - 11/780 și VAX 8600. Noțiunea de **hardware** (resurse fizice) e utilizată pentru a ne referi la aspectele specifice ale implementării mașinii. Acestea includ proiectul logic de detaliu și tehnologia de realizare a mașinii de calcul.

În concluzie arhitectura calculatorului cuprinde două componente principale:

1. arhitectura setului de instrucțiuni (ASI)
2. implementarea mașinii, cu cele două sub-componente:
 - organizare
 - hardware

1.3. ORGANIZAREA GENERALĂ A UNUI SISTEM UNI - PROCESOR

Ca organizare generală, independent de tehnologia de realizare, resursele fizice (hardware) ale unui calculator numeric uni-procesor cuprind (figura 1.2.):

- a. **Procesorul.** Procesorul central (de uz general) al unui calculator este numit **Unitate Centrală de Prelucrare (UCP)** și este format din cale de date și unitate de control. Calea de date cuprinde unitatea aritmetică și logică (**ALU**⁸), setul de registre interne, eventuale blocuri de memorie temporară și magistralele interne procesorului, necesare transferului informației. Componentele căii de date execută, în fiecare moment, operații elementare conform comenzilor primite de la Unitatea de Control (UC). Unitatea de control a procesorului este automatul care, în funcție de informațiile primite din exterior, comandă celelalte unități funcționale ale procesorului, cu scopul execuției instrucțiunilor;

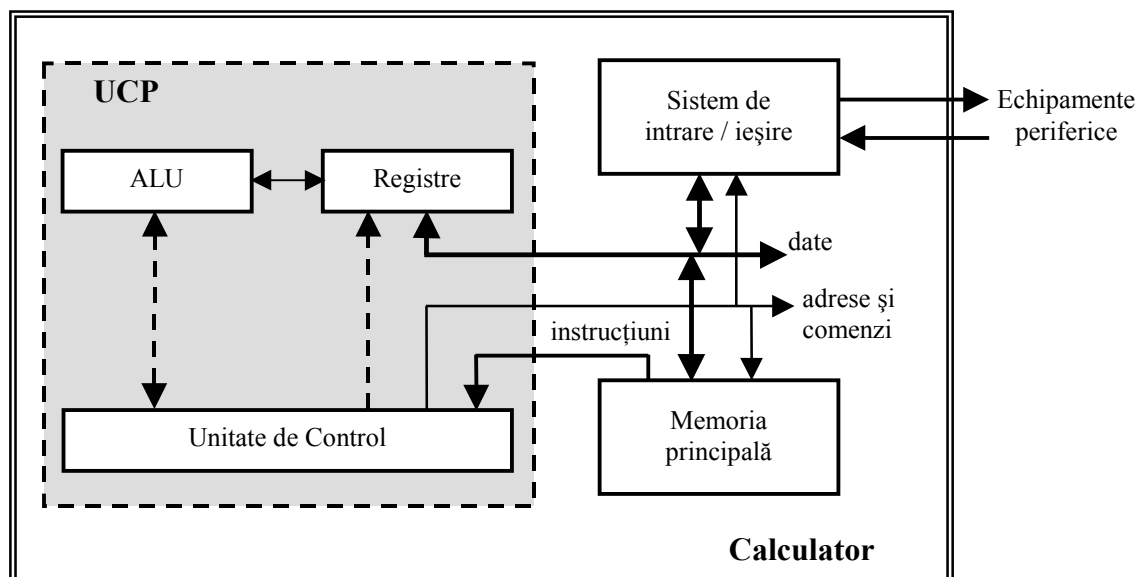


Figura 1.2. Organizarea generală a unui calculator uni-procesor. ALU = Unitate aritmetică și logică. Liniile punctate reprezintă semnale de comandă și sincronizare interne procesorului.

- b. **Memoria principală** (numită și memorie internă sau memorie operativă). Este adresabilă, prin adresă lansată de UCP, la nivel de cuvânt (octet sau multiplu de octet) și este selectată și prin semnale de comandă de către procesor. În memoria principală, dacă se

⁸ ALU - Arithmetic and Logic Unit (engl.)

consideră arhitectura cu memorie unică de date și instrucțiuni, se stochează instrucțiunile programelor rulate de procesor și se pot scrie / citi date aferente programelor. Este o memorie de tip semiconductor, putând fi memorie doar cu citire (ROM - Read Only Memory), sau memorie cu citire scriere (RAM - Random Access Memory). Din punctul de vedere al modului de funcționare și construcție al celulelor de memorie se pot folosi două tipuri de memorie RAM semiconductoare: RAM static și RAM dinamic.

- c. **Sistemul de intrare / ieșire.** Sistemul de intrare - ieșire este, "poarta" prin care se face legătura calculatorului cu lumea externă, pentru a citi și furniza date. Echipamentele cuplate la acest sistem sunt numite echipamente periferice, iar conversația între calculator și aceste echipamente se face prin **logică adresabilă**. Fiecare circuit de comunicare cu exteriorul adresabil printr-o adresă unică, este numit "**port**". Dintre echipamentele periferice standard pentru un calculator de uz general amintim: tastatura, echipamentul de afișare alfanumerică (display) și memoria externă (Hard-disc).

După cum se observă din figură Unitatea de Control și Calea de date (registre, ALU, registre de adresare, registre de interfață cu magistralele) formează Unitatea Centrală de Procesare - UCP (procesorul calculatorului numeric). Calculatorul numeric cuprinde cel puțin o unitate UCP și memoria principală. Un sistem de calcul este format dintr-un calculator numeric și diferite dispozitive periferice.

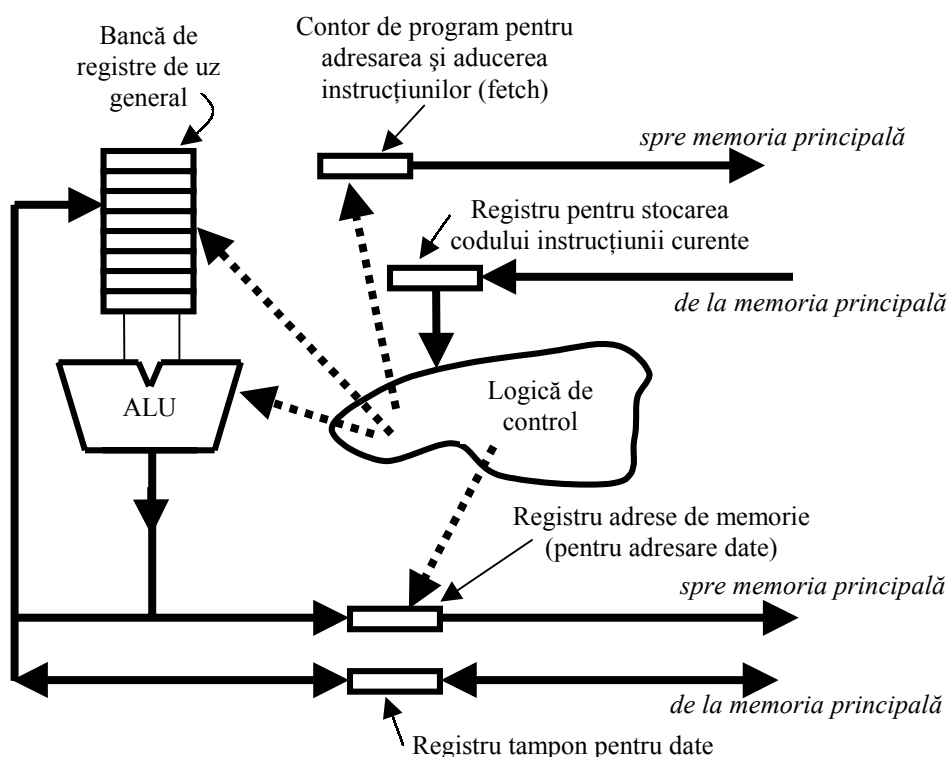


Figura 1.3. Organizarea generală a unui procesor, cu prezentarea interacțiunilor principale dintre unitățile funcționale (prin linie punctată s-au simbolizat comenzile interne procesorului)

Toată această structură hardware a procesorului (fixă - încastrată în siliciu) poate realiza funcțiuni pentru utilizatorul uman, prin aplicații dedicate, doar dacă pe suportul hardware se execută un set coerent de instrucțiuni care indică funcțiile elementare ce trebuie executate de suportul hardware. Funcțiile pot fi foarte diverse și de aceea se poate spune că instrucțiunile ce alcătuiesc un program realizează o *reconfigurare logică* periodică a căii de date a procesorului, pentru a se realiza

funcțiunile cerute. Pentru a prezenta la modul foarte general informații elementare ce se vehiculează între procesor și memoria principală în figura 1.3. se prezintă schematic interacțiunea dintre principalele componente ale unui procesor și memoria principală (externă procesorului).

Conform figurilor 1.2. și 1.3. informațiile principale pe care procesorul le schimbă cu exteriorul sunt: date (operandi și rezultate), instrucțiuni, adrese, informații de control. Toate acestea se transmit ca semnale electrice prin linii conductoare grupate în magistrale. În mod tradițional magistralele externe procesorului sunt clasificate în:

- magistrală de date (Bus de date)
- magistrală de adrese (Bus de adrese)
- magistrală de control (Bus de control)

Legarea diferitelor componente ale calculatorului la aceste trei magistrale este simbolizată în figura 1.4. Se va folosi și termenul împrumutat din limba engleză pentru magistrală: "bus"

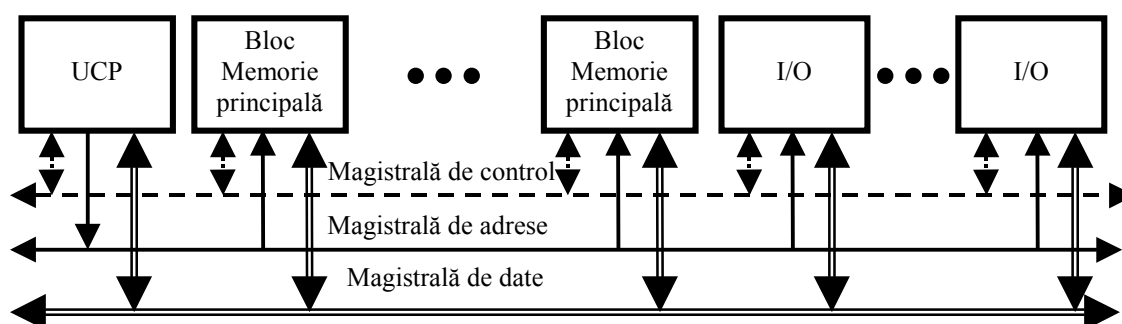


Figura 1.4. Schemă bloc generală ce indică conectarea componentelor calculatorului la cele trei tipuri de magistrale externe UCP. Prin I/O sau simbolizat circuitele de interfață cu echipamentele de intrare / ieșire.

1.4. TENDINȚE ALE TEHNOLOGIEI – LEGEA LUI MOORE

În 1965, Dr. Gordon E. Moore, la acel moment vicepreședinte executiv la Intel Corporation, a prezis că numărul de tranzistoare echivalente pe un singur chip va crește în ritm constant și se prevedea o creștere în același ritm pentru următoarele decenii. Această predicție, numită *Legea lui Moore*, spune că “puterea de calcul va crește exponențial, dublându-se la fiecare 18 - 24 luni, pentru un viitor previzibil”. Legea lui Moore este adesea criticată, pentru că utilizatorii sunt interesați în primul rând de creșterea performanțelor procesoarelor. Este greu să se echivaleze creșterea numărului de tranzistoare pe un chip cu creșterea performanței procesoarelor. În plus adesea creșterea numărului echivalent de tranzistoare nu a produs o creștere a performanțelor în aceeași măsură - de aceea criticii spun că în ultimele decenii îmbunătățirile aduse circuitelor (procesoare, memorii) au fost dictate mai mult de sistemul de operare Windows al firmei Microsoft, decât de cerințele de putere de calcul din partea utilizatorilor. Creșterea gradului de integrare reflectată în creșterea puterii procesoarelor reprezintă doar un procent al afacerilor din domeniul dezvoltării tehnologiei - procent care se reduce continuu - și de obicei există mai multă putere disponibilă decât cea necesară.

Tehnologia calculatoarelor electronice a evoluat într-un ritm incredibil în ultimii 60 de ani, de la conceperea primului calculator electronic. În prezent un calculator de tip PC⁹ cu preț mai mic de 1000 de dolari, are viteză și capacitate de memorare mult mai mare decât un calculator din anii

⁹ PC = Personal Computer

'80 al cărui preț era la nivelul sutelor de mii sau milioane de dolari. Așa cum s-a menționat anterior această dezvoltare rapidă s-a făcut nu numai pe baza dezvoltărilor tehnologice (ce au permis mărirea gradului de integrare și reducerea puterii consumate și a prețului) dar și pe baza inovațiilor în domeniul proiectării arhitecturale.

Apariția microprocesoarelor, în anii '70 ai secolului trecut, a permis apariția unor dezvoltări arhitecturale esențiale pentru performanța calculatoarelor dintre care amintim aici doar câteva: generalizarea utilizării limbajelor de programare de nivel înalt și a sistemelor de operare, organizări ce permit execuție paralelă a instrucțiunilor și generalizarea utilizării memoriilor temporare ("cache")

1.5. DEZVOLTARE A APLICAȚIILOR

În prezent lumea calculatoarelor numerice este dominată de unitățile centrale de procesare (UCP) numite microprocesoare. Acestea constituie UCP nu numai pentru calculatoarele personale, atât de răspândite, dar și pentru supercomputere, stații de lucru, calculatoare de bord la mijloacele de transport auto, aeronautice și spațiale, sau navale. Principalele domenii de aplicare a UCP sub formă de microprocesor integrat pe un singur circuit integrat sunt:

- Calculatoare desktop și stații de lucru (calculatoare de uz general ce se pot așeza "pe birou")
- Servere de rețele sau pentru Web
- Calculatoare integrate în sisteme dedicate aplicațiilor specifice¹⁰

Calculatoarele de tip desktop sunt construite pentru utilizare generală, cât mai largă, atât în domeniul procesării de tip text, sau baze de date cât și al realizării de calcule matematico-științifice, principalul scop urmărit fiind obținerea unui raport preț / performanță cât mai mic.

Calculatoarele de tip server, conțin procesoare cu putere mare de calcul, sunt relativ scumpe, iar principalele obiective ale proiectării sunt furnizarea de servicii de calcul și de transfer al fișierelor, cu asigurarea unei cantități cât mai mari de operații pe unitatea de timp. Aceste calculatoare trebuie să prezinte în plus o mare siguranță în funcționare și să aibă calitatea de scalabilitate (de extindere simplă a resurselor pe măsura creșterii cerințelor).

Sistemele cu calculator integrat aplicației (Embedded) sunt sisteme pe bază de microprocesor construite pentru a controla o funcție sau un domeniu de funcții particulare și nu sunt proiectate pentru a fi programate de către utilizatorul final, așa cum se întâmplă de exemplu cu un PC. Sigura interacțiune cu utilizatorul se face în scopul realizării funcțiilor impuse sistemului - aplicației. Pentru cele mai multe aplicații aceste sisteme trebuie să răspundă în timp real la evenimentele din mediul extern. Sistemele de acest tip au constrângeri mari de proiectare în ceea ce privește costurile, consumul de putere și gabaritul. Au un domeniu de aplicații extrem de vast:

- Sistemele de comunicații cu și fără fir (wireless),
- Sisteme de transport (auto, feroviar, naval, aeronautică)
- Aplicațiile casnice și electronica de larg consum,
- Sistemele de control în industrie, finanțe
- Echipamente medicale

Lucrarea de față nu se referă la domeniul Embedded systems ci doar la calculatoarele de uz general și la descrierea arhitecturii și organizării acestora.

¹⁰ Termenii consacrați în literatura științifică de limbă engleză sunt: Embedded Computers și Embedded Systems.

1.6. SOFTWARE

În ceea ce privește partea de program rulată pe “suportul” hardware există mai multe tipuri de programe rulate de un calculator numeric. Vom folosi termenul de "software", împrumutat din limba engleză prin care vom înțelege totalitatea programelor de sistem și aplicații. Seturile de programe dintr-un sistem de calcul pot fi denumite după utilizarea lor. Astfel programele ce furnizează servicii utile tuturor proceselor și resurselor sistemului de calcul, în mod general, sunt numite programe de sistem (software de sistem). Exemple de software de sistem sunt: sistemele de operare, compilatoare, asamblatoare etc.

Există însă și software - destinat programatorilor - software de asistență pentru dezvoltarea aplicațiilor (numit uneori și software de aplicații). Exemple de software de asistență pentru dezvoltarea aplicațiilor: editoare de text, programe de proiectare asistată, programe pentru baze de date etc.

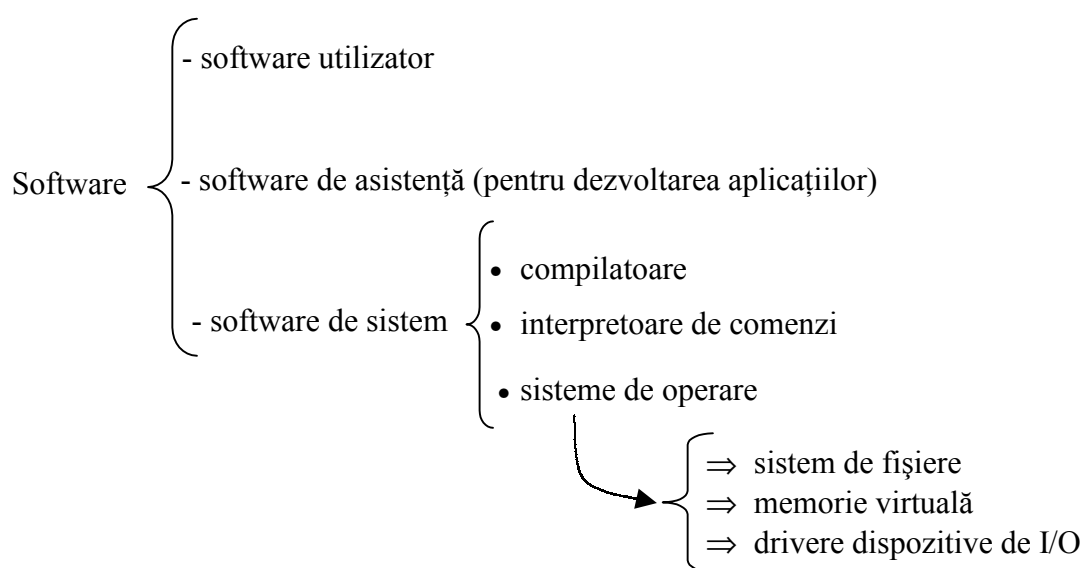


Figura 1.5. Clasificare generală a componentelor software ale unui calculator.

Ultimul tip de software în clasificarea prezentată în figura 1.5. este reprezentat de programele utilizator ce rezolvă anumite probleme specifice (software utilizator sau aplicativ). O clasificare foarte generală a sub-sistemului software al calculatorului numeric, sub forma de niveluri ierarhice, este prezentată în figura 1.5. De observat că la sistemul de operare s-au trecut doar funcțiile principale, "vizibile" utilizatorului:

- administrarea sistemului de fișiere (administrarea memoriei auxiliare - MA)
- gestiunea memoriei principale (MP) și asigurarea tehnicilor software pentru implementarea memoriei virtuale,
- gestionarea programelor de control (drivere) pentru dispozitivele de intrare / ieșire (I/O).

În cadrul programelor de sistem, în afara sistemului de operare s-au specificat programele de sistem ce oferă servicii generale programatorilor (compilatoare, asamblatoare, editoare de legături¹¹, programe de depanare și corectare a erorilor, bibliotecari, etc.)

Este important de observat ca aceste programe nu fac parte din sistemul de operare, chiar daca sunt furnizate în mod obișnuit de producătorii de calculatoare, împreună cu sistemul de

¹¹ linker (engl.): tradus prin link-editor, sau editor de legături

operare. Sistemul de operare este, în general, acea porțiune din software ce rulează în "kernel mode" (mod nucleu) sau în mod supervizor. El este astfel protejat (ignorăm aici procesoarele ce nu oferă suport hardware suficient pentru rularea unui sistem de operare în mod protejat, cum ar fi procesoarele Intel din seria 80x86 lucrând în "mod real") împotriva stricăciunilor ce ar putea fi provocate de programele de aplicații sau de programele utilizator.

Tipurile de software incluse în categoria "software de asistență" se referă la resursele logice numite adesea și programe utilitare pentru dezvoltarea de aplicații.

1.7. CLASIFICAREA ÎN ORDINE A SISTEMELOR NUMERICE

Calculatorul numeric este un sistem digital, cu o serie de funcții ce corespund organizării sale interne. În literatura românească a fost introdusă în 1983, [Stefan83], o modalitate sugestivă de **clasificare a sistemelor digitale** în ordine, clasificare care ne va ajuta să încadrăm calculatorul în clasa mai generală a sistemelor numerice și totodată să exemplificăm ulterior corelația ce există între gradul de dezvoltare structurală și funcțiile calculatoarelor numerice.

În mod tradițional, sistemele digitale sunt împărțite în:

- *sisteme logice combinaționale* (ieșirile la momentul de timp t , depind numai de valorile intrărilor la momentul t) și
- *sisteme logice secvențiale* (la care ieșirile la momentul t depind atât de intrările la momentul t cât și de starea sistemului la momentul t , stare determinată intrările și ieșirile circuitului la momente anterioare lui t).

Această clasificare generală nu evidențiază deosebiri de structură și funcționare a circuitelor ce sunt incluse în cadrul celor secvențiale. **Criteriul de clasificare** propus [Stefan93] împarte sistemele digitale în **ordine**, fiecărui ordin fiindu-i asociată o anumită clasă de funcții.

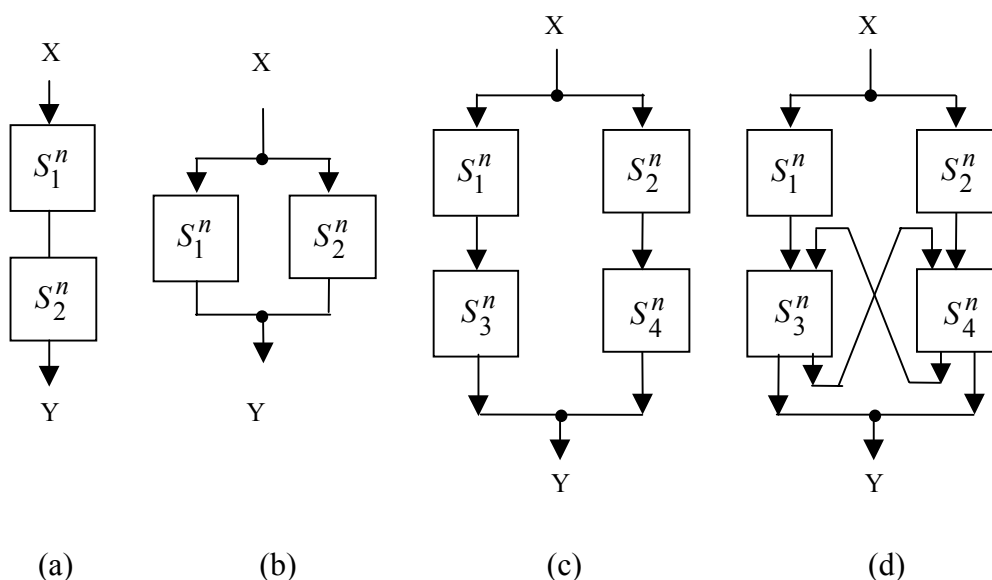


Figura 1.6. Exemplificare a interconectării sistemelor digitale: a) sistem de ordinul n serie, b) sistem de ordinul n paralel, c) sistem de ordinul n serie - paralel, d) sistem de ordinul $n+1$ (dezvoltare structurală prin introducerea reacției). Prin X s-a indicat intrarea în sistem iar prin Y ieșirea. Prin S_i^n , $i = 1...4$ în figură, s-a indicat sistemul digital de ordinul n cu indicele i .

Ordinele sunt determinate de gradul de structurare (organizare internă), numărul ordinului crescând pe măsura creșterii structurale.

Clasificarea se face conform următoarelor definiții:

1. Sistemul de ordinul zero cel mai simplu este reprezentat de o poartă logică (circuit combinațional elementar).
2. Extensiile serie (figura 1.6. a), paralel (figura 1.6.b) sau serie-paralel (figura 1.6.c), ale unor sisteme ce au ordinul maxim n generează tot un sistem de ordinul n ;
3. Un sistem de ordinul $n+1$ poate fi generat prin interconectarea unor sisteme cu ordin mai mic sau egal cu n , dar între care cel puțin unul este de ordinul n , într-o configurație ce prezintă o buclă de reacție (figura 1.6.d);

În clasificarea descrisă pe scurt mai sus, ordinele sunt numerotate de la 0 la n , teoretic n putând fi oricât de mare. Unui anumit ordin îi corespunde o anumită clasă de funcții, funcții ce pot fi realizate și pe oricare din sistemele de ordin superior.

Încercând să exemplificăm corespondența între ordine, funcții și circuite existente pe piață, vom observa că începând de la ordinul cinci în sus nu mai putem să dăm niște exemple concrete [Stefan91], nu numai pentru că realizările tehnologice actuale sunt limitate, ci și pentru că funcția nu mai este dictată strict de structura fizică ci și de posibile structuri informaționale:

Ordinul $n =$	Funcții	Exemplificare circuite
0	funcții combinaționale	circuite logice combinaționale
1	funcții de memorare	circuite de memorare, (latch RS, latch D, registre, RAM, structura master-slave)
2	funcții de automat finit	bistabile JK, T, numărătoare digitale
3	funcții de prelucrare	structura de procesor (automate cu stivă, automate microprogramabile);
4	funcții de prelucrare programabile	calculator
5	funcții de prelucrare în rețea	rețele de prelucrare digitală

1.8. REPREZENTAREA FUNCȚIONALĂ A UNUI CALCULATOR

Un calculator poate fi descris atât sub aspect funcțional cât și structural.

Circuitele electronice ale unui calculator recunosc și execută doar un set limitat de *instrucțiuni elementare*, codificate în formă binară. Aceste instrucțiuni sunt doar succesiuni de biți (1 și 0) pe care procesorul le înțelege - decodifică și le execută. Indiferent de tipul de mașină, instrucțiunile recunoscute sunt rareori mai complicate decât [Tanenbaum99]:

- adună două numere
- verifică dacă un număr este egal cu zero
- copiază date dintr-o zonă a memoriei calculatorului în altă zonă.

De exemplu, la procesoarele Intel din seria 80x86 codul binar al instrucțiunii următoare:

0000 0100 0000 0110

comandă adunarea conținutului unui registru intern de 8 biți (numit registrul al) cu valoarea imediată 6. Adesea, când utilizatorul uman este obligat să lucreze cu valori numerice binare, se folosește reprezentarea în hexazecimal, care este mai compactă și mai ușor de citit. Codul, în hexazecimal, al instrucțiunii corespunzătoare succesiunii binare de mai sus este 04 06 hex. Și acest mod de scriere a instrucțiunilor este însă complicat pentru programator.

Instrucțiunile elementare, recunoscute de un procesor formează **limbajul mașină** al procesorului. Vom nota acest limbaj cu **L0**, el corespunzând mașinii fizice **M0**. Limbajul mașină cuprinde însă doar instrucțiuni codate sub formă de cuvinte binare și de aceea este dificil de utilizat de către om.

Presupunem că programatorul poate scrie programele sale într-un limbaj L1, format din instrucțiuni simbolice (succesiuni de litere și cifre) mai apropiat de limbajul uman. Pentru a putea executa acest program pe mașina M0 fiecare instrucțiune din L1 a programului trebuie *translatată* în instrucțiuni din limbajul L0. De exemplu programatorul poate scrie în L1 instrucțiunea:

add al,6

iar programul de traducere va genera succesiunea binară, corespunzătoare lui L0:

0000 0100 0000 0110

Se poate spune că limbajul L1 corespunde unei mașini virtuale programabile numită M1. Numele acestui limbaj simbolic de programare este “limbaj de asamblare”, iar programul de traducere este numit “asamblor”.

Traducerea presupune că tot programul scris în L1 este mai întâi transformat în program scris în L0. Apoi programul în L0 este încărcat în memoria calculatorului și executat, iar programul în L1 este abandonat. Calculatorul este controlat de programul în L0, lansat în execuție. Exemple de programe de traducere: compilator, asamblor.

Există însă și varianta rulării programelor pe M0 prin *interpretare*, de către un program numit “interpretor”. Interpretorul este un program în L0 care rulează pe mașina M0 și care preia instrucțiunile programului în L1 ca pe date de intrare. Interpretorul citește și decodifică fiecare instrucțiune din L1 și apoi trece imediat la execuția acesteia. În cazul interpretorului nu se generează un program executabil ca la traducere. În acest caz calculatorul este controlat de programul interpretor.

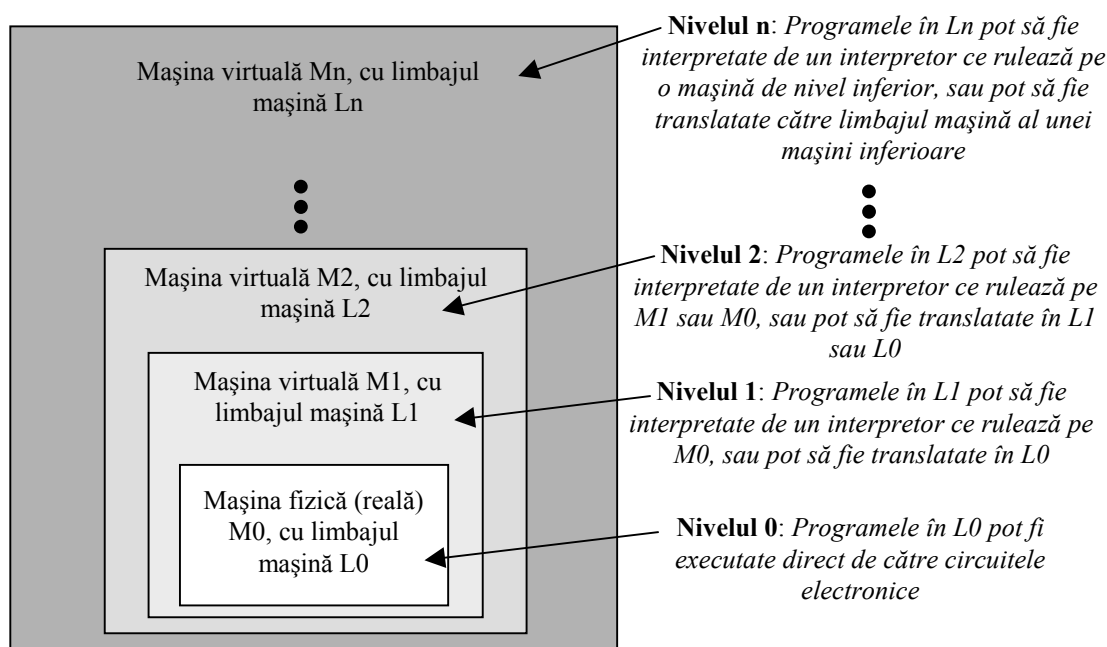


Figura 1.7. O mașină de calcul organizată din punct de vedere funcțional ca o succesiune de mai multe niveluri de mașini virtuale.

Programatorii doresc adesea să dispună de un limbaj de programare mult mai apropiat de limbajul uman, în care să existe comenzi de genul: “tipărește un șir la imprimantă”, “afișează un text pe ecranul calculatorului”, “extrage radical dintr-un număr” etc. Diferențele dintre un limbaj de acest fel și L0 sunt mari, iar la transformarea directă către L0 a instrucțiunilor noului limbaj fiecărei instrucțiuni îi va corespunde o succesiune de instrucțiuni elementare din L0. Dacă noul limbaj ar fi numit L1, iar transformarea s-ar face direct în L0 programul de traducere ar fi extrem de complex. De aceea programul scris în noul limbaj, să-l numim limbajul L2, corespunzător mașinii virtuale M2, va fi tradus mai întâi către un program în L1 și apoi către programul în L0.

Conform ierarhiei de limbaje descrise mai sus, un calculator poate fi privit, **din punct de vedere funcțional**, ca o succesiune de mașini virtuale M_i , fiecare corespunzătoare unui limbaj L_i ($i = 1, n$), ca în figura 1.7. Fiecare nivel are un set de funcțiuni specifice care prelucrează un set de intrări specifice și generează un set de ieșiri specifice. O mașină virtuală de pe un anumit nivel poate utiliza toate funcțiile oferite de mașina de pe nivelul inferior.

Structura ierarhică a mașinii de calcul indică că între fiecare două niveluri vecine există o interfață, deci există și o ierarhie verticală de interfețe. Pentru un calculator de uz general, cu procesor având unitatea de control microprogramată, această ierarhie de mașini virtuale și ierarhie de interfețe se indică în figura 1.8. În partea dreaptă a figurii s-a exemplificat aspectul programului la nivelul respectiv.

În figura 1.7. nivelurile succesive de mașini virtuale au fost imbricate. Aceasta pentru că o mașina virtuală¹² de pe un nivel superior folosește toate funcțiile oferite de nivelurile inferioare. Un nivel este constituit din mulțimea aplicațiilor asupra elementelor mulțimii de intrare pentru nivelul dat, cât și asupra elementelor mulțimilor de intrare și ieșire de la nivelul imediat inferior. Aplicațiile de la un nivel dat pot constitui aplicații și pentru nivelul superior următor. Cele mai multe calculatoare moderne au 2 sau mai multe niveluri. Există însă și mașini de calcul la care se pot identifica mai mult de 5 niveluri ierarhice de mașini virtuale.

În figura 1.8. se observă că la nivelul cel mai de jos se găsesc dispozitivele și circuitele electronice digitale care alcătuiesc structura hardware a calculatorului. La următoarele două niveluri se exemplifică existența unui nivel combinat hardware - software (firmware) care cuprinde pe lângă unitățile logice funcționale (set de registre, ALU, unitate de control, etc.) și un nivel de microprogram, încadrat într-o memorie locală de control. Acest nivel face parte din unitatea de control microprogramată, iar microprogramele se constituie ca un interpretor al instrucțiunilor de la nivelul inferior al unităților funcționale. În cazul în care unitatea de control este cablată, nivelul de microprogram nu apare explicit în ierarhia mașinilor virtuale.

La următorul nivel se observă nivelul instrucțiunilor elementare recunoscute de procesor. Este nivelul funcțiilor de bază pe care le oferă un anumit procesor de calculator și el este descris de către producători în manualele de descriere a procesoarelor. Acesta este nivelul ASI (*nivelul arhitecturii setului de instrucțiuni*), iar mașina virtuală *mașina de bază*.

Deși mașina "înțelege" setul specific de instrucțiuni binare și toate unitățile funcționale sunt asamblate, la acest nivel mașina poate fi folosită cu greu și numai dacă utilizatorul scrie în cod mașină nu numai programul de aplicație utilizator, dar și rutinele necesare pentru accesul la perifericele de intrare și de ieșire (tastatură, display¹³, disc magnetic, etc.). La acest nivel, dispozitivele de intrare / ieșire (I/O) pot fi controlate prin încărcarea de valori în registre speciale ale circuitelor de control ale acestor dispozitive I/O (controllere I/O). De exemplu, controlul unei unități de disc, pentru citirea informației de pe disc, se poate face prin încărcarea valorilor corespunzătoare adresei de pe disc, adresei memoriei principale, a numărului de octeți de transferat, și alte informații de stare și sincronizare. În practică, sunt necesari mult mai mulți parametri decât cei pomeniți, iar informația de stare returnată de circuitul de control al discului după o operație este foarte complexă.

¹² **virtuală**, pentru că utilizatorul unui anumit nivel lucrează cu instrucțiuni specifice, vede doar nivelul la care lucrează, fără să conteze pentru el ce se întâmplă mai jos

¹³ display (engl.) = dispozitiv de afișare

Mai mult, pentru multe dispozitive de I/O, timpul joacă un rol important în programare. Una din funcțiile majore ale unui sistem de operare este să ascundă toate aceste complexități și să pună la dispoziția programatorului un set de instrucțiuni simple cu care să lucreze. De exemplu o comandă de genul: "citește blocul X din fișierul Y" e mult mai simplă pentru că nu trebuie să ne mai facem probleme privind detaliile mișcării capetelor de citire/scriere și timpul de așteptare pentru stabilizarea lor.

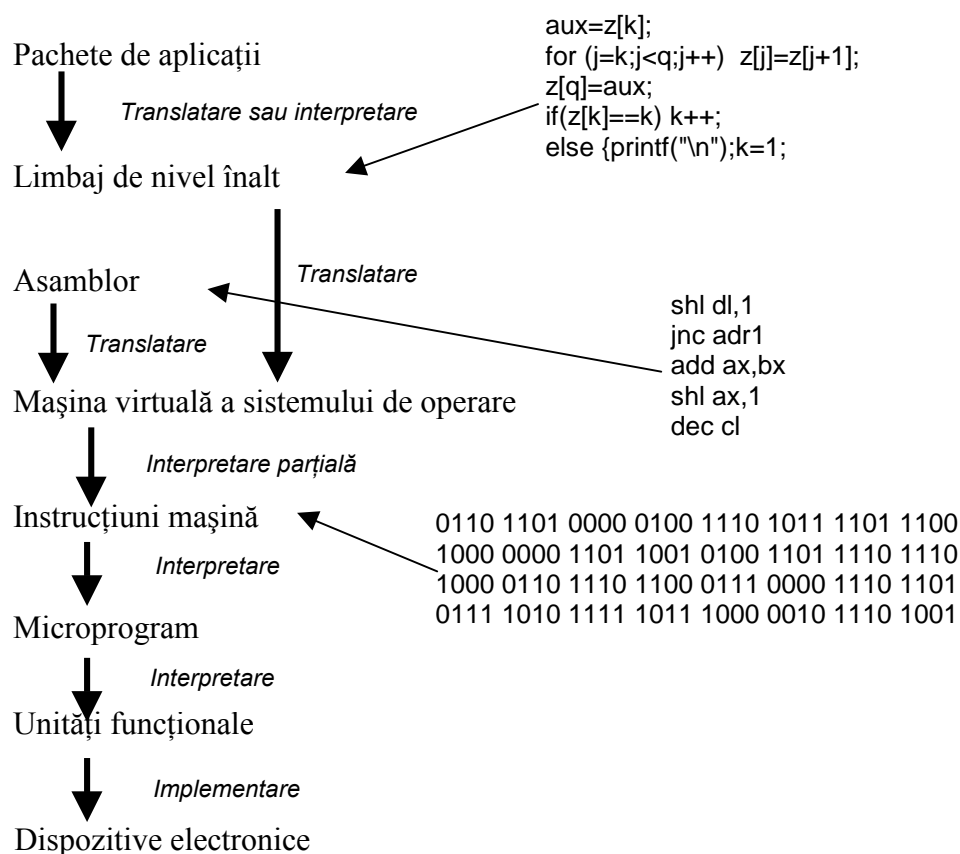


Figura 1.8. Ierarhia mașinilor virtuale și a interfețelor pentru un calculator de uz general cu procesor având control microprogramat.

Nivelul mașinii virtuale a sistemului de operare este de fapt un sistem hibrid. [Tanenbaum99], pentru că cele mai multe dintre instrucțiunile (funcțiile) oferite sunt de asemenea instrucțiuni și pentru nivelul imediat inferior. Funcțiile suplimentare oferite se referă la administrarea resurselor hardware și software ale calculatorului (alocarea memoriei principale, administrarea memoriei secundare, posibilitatea de execuție concurentă a programelor etc.). Uneori acest nivel este numit **nivelul mașinii extinse**, pentru că la acest nivel calculatorul este văzut de utilizator prin intermediul sistemului de operare (SO). Comunicarea utilizatorului, sau programului utilizator, cu SO se realizează pe două niveluri: prin intermediul limbajului de comandă cu ajutorul căruia utilizatorul solicită diferite resurse ale sistemului și prin intermediul unor instrucțiuni speciale (apeluri sistem) la execuția cărora sunt activate unele componente ale SO.

Sistemul de operare nu este doar o interfață între utilizator și mașina de calcul. El are și sarcina de *administrator* al resurselor calculatorului. El trebuie să administreze toate procesele (programele în cursul execuției, împreună cu contextul de lucru), memoriile internă și externă, discurile magnetice, dispozitive de intrare / ieșire, etc. Sarcina sistemului de operare este să facă o alocare ordonată și controlată a proceselor, blocurilor de memorie, dispozitivelor de I/O, între diferitele programe ce se află în competiție pentru a le folosi. Atunci când un calculator are

utilizatori multipli, sarcinile de *administrare și protecție* ale sistemului de operare sunt și mai evidente.

Sistemele de operare au o componentă de nivel inferior (un nucleu central, monitor, executiv, sau supervizor), care depinde de arhitectura mașinii de bază (de nivelul ASI) și care controlează mașina de bază tot timpul cu excepția momentelor când se rulează un program de aplicație (care însă poate folosi rutine ale executivului), iar la terminare, controlul e preluat din nou de executiv. Colecția de rutine ce formează nucleul SO este componenta cea mai apropiată de hardware care realizează gestiunea la nivel fizic a operațiilor de I/O, tratare a întreruperilor, încărcarea și lansarea programului, citirea/scrierea unui caracter pe disc sau pe monitor etc. Orice modificare a configurației mașinii de bază implică modificarea acestui nucleu al SO dar nu implică și modificarea nivelurilor superioare. Nucleul SO este păstrat de obicei în memorie de tip ROM, dar în unele cazuri se poate stoca pe discul magnetic. De exemplu, sistemul de operare MS-DOS, are o componentă numită BIOS (Basic Input Output System) stocată în memorie de tip ROM, sau memorie RAM cu baterie tampon pentru păstrarea datelor și la oprirea calculatorului. BIOS conține programe de control ("drivers") pentru dispozitivele standard de I/O, acestea putând oferi o serie de servicii care degreuează programatorul și rutinele sistemului DOS, de toate detaliile hardware ale circuitelor de interfață cu dispozitivele I/O. La PC aceste servicii pot fi apelate prin mecanismul întreruperilor software (INT nn), după o încărcare prealabilă cu valori adecvate, ale registrelor mașinii. Celelalte componente ale sistemului MS DOS numite "io.sys" (conține programele de control de I/O) și "msdos.sys" (care conține rutine de gestionare a proceselor, memoriei și a sistemului de fișiere de pe discul magnetic, interpretarea apelurilor sistem) se încarcă de pe discul magnetic. Programele aplicative pot cere servicii sistemului de operare DOS (pot apela funcțiile DOS), prin intermediul întreruperii soft INT 21H, codul corespunzător funcției dorite fiind încărcat în registrul AH.

1.9. REPREZENTAREA STRUCTURALĂ A UNUI CALCULATOR

Structural, un calculator este format din unități funcționale interconectate pentru a putea prelucra informația, pe baza comenzilor transmise prin program. Pentru un calculator uni-procesor, structura generală este cea din figura 1.9., ea corespunzând structurii propuse de von Neumann în 1945 pentru calculatorul secvențial, cu program memorat. În cadrul structurii calculatorului secvențial din figura 1.9. se disting următoarele unități [Sztojanov87]:

- **unitatea de intrare**, formată din echipamente *periferice de intrare* și *sistemul de intrare* al calculatorului,
- **unitatea centrală** a calculatorului, formată din *memoria principală* (memorie ce conține date și instrucțiuni) și *Unitatea Centrală de Procesare* (UCP),
- **unitatea de ieșire**, formată din echipamente *periferice de ieșire* și *sistemul de ieșire* al calculatorului.

Echipamentul periferic de intrare, preia datele din exterior (prin diverse traductoare) și le transformă ca natură fizică și format de reprezentare, în așa fel încât informația să fie compatibilă cu nivelurile de tensiuni și formatul de reprezentare binar folosit de sistemul de intrare al calculatorului.

Echipamentul periferic de ieșire are funcția inversă perifericului de intrare, el preluând datele de la sistemul de ieșire al calculatorului. Echipamentul periferic de ieșire transformă datele primite într-o formă corespunzătoare elementului de execuție comandat de calculator. Dacă este vorba de dispozitive de memorie externă informația se stochează pe suportul extern, conversia informației binare făcându-se în conformitate cu tipul circuitului de control și cu natura fizică a suportului de stocare (hârtie, semiconductor, material magnetic etc.).

Sistemul de intrare / ieșire (I/O) al calculatorului este locul prin care se face schimbul de informații între unitatea centrală și echipamentele periferice. Acest transfer se face prin intermediul unor locații adresabile de către procesor (UCP), numite porturi, ele făcând parte din sistemul I/O al calculatorului. Sistemul I/O realizează operații de genul: modifică formatul de reprezentare a datelor (din serie în paralel, sau invers), verifică corectitudinea informației transferate, asigură sincronizarea dintre echipamentul periferic și UCP din punctul de vedere al vitezelor de transfer (sincronizarea este necesară datorită vitezelor de lucru mult diferite între UCP și echipamentele periferice).

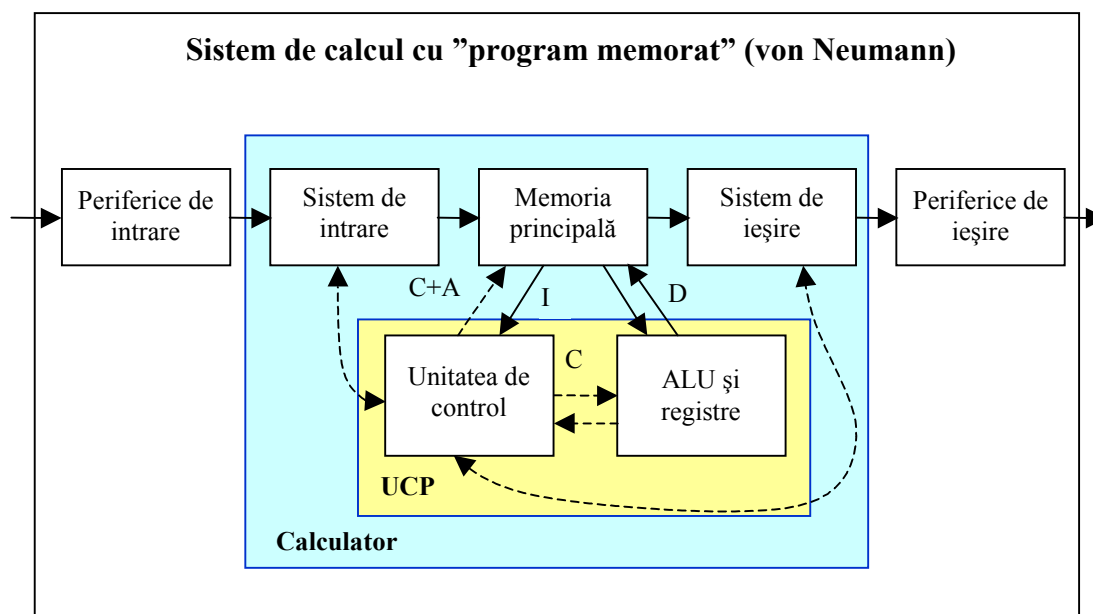


Figura 1.9. Reprezentare structurală a unui calculator uni-procesor de uz general. Cu linie punctată s-au reprezentat comenzile (C) transmise de la unitatea de control și semnalele de stare-sincronizare ca răspuns al unităților comandate către unitatea de control. În cadrul semnalelor de control pentru memoria principală a fost inclusă și informația de adresare (A) a memoriei. Căile de transfer pentru date (D) și instrucțiuni (I) sunt reprezentate cu linie plină. ALU = Unitate Aritmetică și Logică.

Unitatea centrală asigură prelucrarea automată a datelor, prin interpretarea instrucțiunilor unui program. Programul rulat (aflat în execuție) este stocat împreună cu datele aferente în memoria principală (internă calculatorului). Introducerea datelor inițiale, comanda lansării în execuție, afișarea, tipărirea sau transmisia la alte dispozitive a rezultatelor se face prin unitățile de intrare, respectiv de ieșire ale sistemului de calcul.

Ca structură internă, unitatea centrală este formată din memoria principală și din Unitatea Centrală de Procesare (UCP). La rândul ei UCP include calea de date (ALU, registre, magistrale interne pentru transferul informațiilor) și unitatea de control.

Unitatea de control (UC), adresează (A) și citește din memorie instrucțiunile binare ale programului, le interpretează și generează în consecință semnale de comandă către toate celelalte unități funcționale ale calculatorului. Ca urmare a acestor semnale instrucțiunea curentă (I) este executată. În plus UC analizează semnale de stare și sincronizare, ce provin de la celelalte unități funcționale și ca urmare poate schimba succesiunea semnalelor de control pe care le generează, sau poate modifica momentele de timp ale succesiunii semnalelor de comandă, pentru a realiza sincronizarea cu celelalte unități. Există două moduri de implementare a unității de control:

- **control cablat**: dacă unitatea de control este realizată ca un automat secvențial convențional, la care algoritmul de interpretare a fiecărei instrucțiuni binare este inclus în

circuitele fizice (hardware). Pentru modificarea setului de instrucțiuni recunoscut unitatea de control trebuie re-proiectată complet.

- **control microprogramat (micro-codat):** dacă succesiunea de semnale de comandă specifice fiecărei instrucțiuni (succesiune numită microprogram) se înscrie într-o memorie de control locală, de obicei, memorie numai cu citire. Pentru modificarea setului de instrucțiuni recunoscut trebuie doar rescris conținutul memoriei de control.

Tot în cadrul unității de control includem și circuitele pentru generarea adreselor, care calculează adresele corecte (adrese pentru memoria principală, pentru porturi de I/O, sau pentru registrele interne ale UCP) pe baza informațiilor binare din corpul instrucțiunilor.

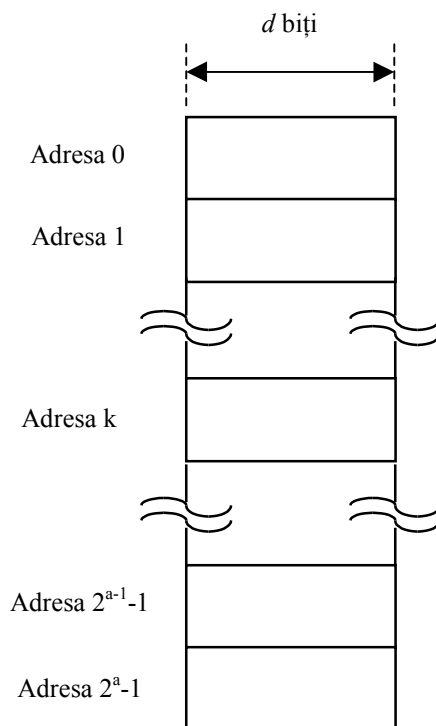


Figura 1.10. Memoria principală prezentată ca o succesiune de locații de memorie, fiecare locație având o adresă specifică. S-a presupus că adresa de memorie are a biți, iar în fiecare locație, cu adresă unică, se poate stoca un cuvânt binar cu lungimea de d biți.

Unitatea aritmetică și logică (ALU¹⁴), realizează operații logice sau aritmetice cu operanzii adresați de UC. Înainte de prelucrare, operanzii se stochează într-un set de registre de uz general, folosite ca memorie temporară. Registrele reprezintă o memorie locală UCP, de foarte mare viteză. Setul de registre de uz general poate fi folosit însă și pentru salvarea diferitelor informații privind adresarea memoriei principale.

În funcție de rezultatul operațiilor efectuate, ALU setează anumiți indicatori de condiții (indicatori de stare, fanioane¹⁵) care pot fi citiți de UC și pot astfel modifica secvența de tranziție a stărilor acestui automat.

Unitatea de memorie principală (sau memorie internă) are funcția de stocare a programelor și datelor aferente acestora. Transferul datelor cu registrele UCP se face conform comenzilor date de unitatea de control. Memoria principală este realizată în prezent exclusiv în tehnologie semiconductoare. Aceasta unitate de stocare a informației are, în principiu, o organizare liniară, constând din **locații (registre) de memorare**, fiecare de câte d biți, locația fiind selectabilă printr-o

¹⁴ ALU = Arithmetic and Logic Unit (engl.) - Unitate aritmetică și logică

¹⁵ flag (engl.) = fanion, indicator de condiții / de stare

adresă unică. Adresa poate lua valori cuprinse între 0 și 2^a-1 , unde prin a s-a notat numărul de biți ai cuvântului de adresă fizică.

Din punctul de vedere al denumirilor folosite facem următoarele observații:

- Ansamblul format din UC, ALU și registre este numit *Unitate Centrală de Procesare* (UCP) fiind o structură de *procesor* de uz general cu set de instrucțiuni.
- UCP împreună cu memoria principală formează Unitatea Centrală, iar
- Unitatea centrală împreună cu sistemul de I/O și setul de programe de sistem constituie structura de *calculator*.
- Un calculator împreună cu diversele echipamente periferice formează un *sistem de calcul*.

Structura logică de bază a calculatorului uni-procesor, prezentată anterior, conform figurii 1.9. corespunde, în bună măsură, celei stabilite de John von Neumann în 1945. În lucrarea "Prima schiță de Raport asupra lui EDVAC", el a definit structura logică de bază a *calculatorului cu program memorat*, menționând cinci criterii necesare a fi îndeplinite de acesta [Hayes88]:

1. să posede un mediu de intrare care să permită introducerea unui număr nelimitat de operanzi și instrucțiuni.
2. să posede o memorie din care să se citească operanzi și instrucțiuni și în care să se poată introduce, în ordinea dorită, rezultatele.
3. să dispună de o secțiune de calcul, capabilă să efectueze operații aritmetice și logice asupra operanzilor citați din memorie.
4. să posede un mediu de ieșire, care să permită livrarea unui număr nelimitat de rezultate către utilizator.
5. să posede o unitate de comandă capabilă să interpreteze instrucțiunile citite din memorie și să selecteze diverse variante de desfășurare a operațiilor, în funcție de rezultatele obținute pe parcurs.

Marea majoritate a calculatoarelor construite până în prezent se bazează pe principii rezumate mai sus, fiind numite calculatoare de tip von Neumann, sau cu arhitectură von Neumann.

1.10. SCURTĂ PRIVIRE ISTORICĂ ASUPRA DEZVOLTĂRII MAȘINILOR DE CALCUL

Un calculator modern reprezintă un sistem complex care înglobează în construcția sa mai multe tehnologii: electronice, magnetice, electromecanice etc. Evoluția calculatoarelor este strâns legată de progresele înregistrate în aceste tehnologii. Astfel, în prezent, pentru realizarea unității centrale a unui calculator se utilizează, cu precădere, tehnologii microelectronice. Atât memoria principală cât și UCP (microprocesorul) sunt construite ca circuite integrate pentru toate calculatoarele moderne. Pentru stocarea datelor și a programelor pe termen lung se folosesc suporturi magnetice sub forma de discuri sau benzi, bazate pe diverse tehnologii magnetice / optice. Pentru implementarea dispozitivelor periferice de intrare / ieșire sunt folosite în principal tehnologiile electromecanice.

În evoluția mijloacelor de calcul se pot evidenția patru etape [Hayes88]:

- etapa calculului *manual*
- etapa mașinilor *mecanice* de calcul
- etapa mașinilor *electromecanice* de calculat
- etapa mașinilor *electronice* de calcul

Nu ne vom referi aici la calculul manual și la eventualele ajutoare pentru acest calcul, ajutoare de tip abac (primul numărător de tip abac se pare că a apărut la chinezi cu 2600 de ani înainte de Cristos). Calcularea manuală are două limitări serioase [Hayes88]:

- a. viteza la care poate lucra “calculatorul” uman este limitată. Unele probleme ce presupun foarte multe operații nu pot fi rezolvate manual niciodată într-o perioadă rezonabilă sau la un cost rezonabil;
- b. ființele umane sunt înclinate să facă greșeli (în calcule !) astfel că rezultatul obținut cu mâna, în cazul unor calcule complexe nu prezintă încredere, decât dacă s-au luat precauții sofisticate de înlăturare a greșelilor.

1.10.1. Etapa mașinilor mecanice de calcul

La aceste mașini roata dințată a jucat rolul elementului cu mai multe stări stabile. Fiecare cifră zecimală poate fi exprimată printr-o poziție a roții. La reprezentarea numerelor zecimale se foloseau atâtea roți câte cifre erau necesare pentru reprezentarea unui număr. Ca evenimente semnificative ale mașinilor mecanice de calcul amintim:

- 1642 - filozoful francez Blaise Pascal (1623 - 1662) a proiectat și realizat mașina “automată” de adunat și scăzut. Avea 2 seturi a câte 6 roți numărătoare pentru cei doi operanzi reprezentați fiecare cu 6 cifre zecimale. Poziția specifică a roților indica valoarea zecimală stocată. Cele două seturi de roți erau interconectate prin angrenaje și roți dințate. Pentru realizarea acestei mașini Pascal a introdus mai multe inovații: clichet de blocare care făcea automat transferul transportului sau împrumutului între rangurile zecimale vecine, când roata trecea de la 9 la 0 sau invers. Numerele negative erau reprezentate printr-un cod complementar.
- În jurul anului 1671 filozoful și matematicianul german Gottfried Leibniz (1646 - 1716) a construit un calculator mecanic care adăuga la mașina de adunat și scăzut a lui Pascal mecanismele necesare pentru a realiza automat operațiile de înmulțire și împărțire. Înmulțirea a fost implementată prin intermediul unor scripeți și lanțuri de transmisie. Această mașină, predecesorul mașinilor mecanice cu patru operații, a rămas o curiozitate academică până în secolul XIX când s-au produs și comercializat primele mașini de calcul mecanice.
- În 1823 Charles Babbage profesor de matematică la Cambridge, Marea Britanie, începe proiectarea unei mașini de calcul, cu execuție automată a programului, numită “Mașina diferențială” (Difference Engine). Proiectul prevedea structura principalelor elemente ale calculatoarelor moderne (unitățile de memorie, de calcul, de intrare, ieșire și comandă) și era destinat calculului automat al tabelelor matematice. Mașina diferențială trebuia să calculeze automat valorile unor tabele matematice și să le transfere prin perforatoare, unor plăci litografice, de pe care ulterior tabelele puteau fi tipărite. Singura operație efectuată era adunarea, prin care se calculau o mulțime de funcții utile prin metoda diferențelor finite. Proiectul mașinii diferențiale a fost abandonat în 1842. Un motiv este că proiectul mașinii era mult prea avansat față de tehnologia mecanică existentă atunci. Al doilea motiv, a fost scăderea interesului lui Babbage care concepușe deja o mașină mult mai puternică și pe care o numise mașina analitică.
- Proiectul mașinii analitice (Analytical Engine) a început în 1834. Din diverse motive nici una din mașinile proiectate de Babbage nu a fost terminată în secolul său. Ulterior, în secolul XX, mașinile au fost construite pentru a fi expuse în muzee ale tehnicii.

Structura propusă pentru *Mașina Analitică* este cea din figura 1.11. ([Hayes88]):

Proiectul mașinii analitice cuprinde funcțiunile generale pe care le are și în prezent oricare mașină de calcul de uz general. Pentru a controla succesiunea operațiilor mașinii, Babbage a propus, pentru prima oară, folosirea de cartele perforate, de genul celor folosite la războaiele de țesut Jacquard. Cartelele cu operații ("operation cards") erau utilizate pentru controlul operațiilor unității

de calcul. Fiecare cartelă selecta una din cele 4 operații elementare (adunare, scădere, înmulțire, împărțire) ce urma să fie realizată la fiecare pas din program.

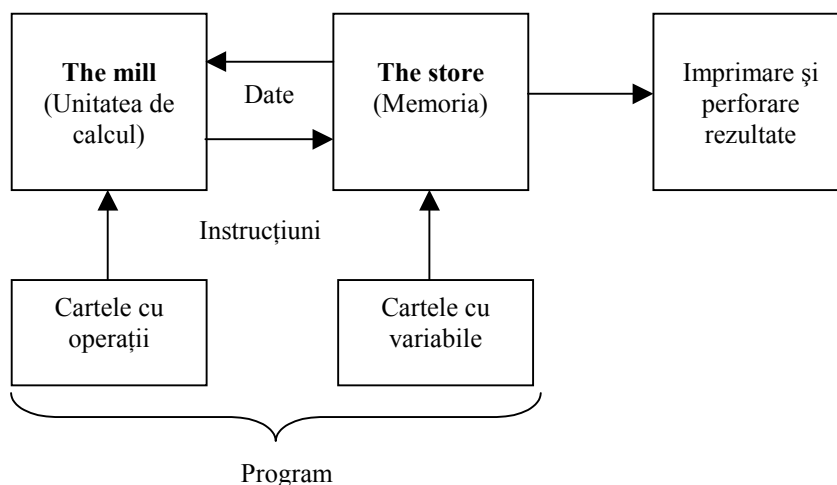


Figura 1.11. Structura "mașinii analitice" a lui Babbage. În figură s-au păstrat și denumirile originale, în limba engleză, ale unor componente ale mașinii, trecându-se în paranteză și funcția corespunzătoare în limba română. Unitate de calcul (the mill - moara) având funcțiuni similare cu unitatea aritmetică și logică din structurile moderne de procesoare. Memoria (the store - magazia) cu rol de stocare a datelor. Echipamentul de ieșire al mașinii de calcul este unitatea de imprimare și perforare rezultate. Cartele cu operații (de comandă) memorează codul operației cerute, iar cartelele de variabile furnizează valorile, sau adresele datelor ce vor fi prelucrate - date de intrare în tabele.

Cartelele cu variabile ("variable cards") se foloseau pentru a selecta locațiile de memorie utilizate de o anumită operație (de exemplu sursa operandului de intrare și destinația rezultatului). Informațiile numerice constante se puteau furniza fie prin cartele perforate, fie prin setarea manuală a roților dințate numărătoare.

Alta contribuție importantă a lui Babbage a fost concepția mecanismului care să permită unui program să-și modifice automat secvența liniară de operații. În termenii actuali putem spune ca el a conceput mecanismul de execuție a instrucțiunilor de salt condițional, prin testarea semnului unui număr.

- 1912. Începe producția industrială, de masă, a mașinilor mecanice de calculat, cu patru operații aritmetice.

1.10.2. Mașini electromecanice de calculat

Aceste mașini de calcul erau construite cu roți dințate, electromagneți și mașini electrice de acționare a roților dințate. Principalele momente în dezvoltare sunt următoarele:

- 1896 -Herman Hollerith, care anterior inventase o mașină comercială cu cartele perforate pentru calcul, sortare, și catalogare a cantităților mari de date, a înființat "The Tabulating Machine Company " pentru a fabrica echipamentul inventat de el. În 1911 această companie s-a unit cu altele și a format "Computing-Tabulating-Recording Company" redenumită în 1924 "International Business Machines" (IBM). Este interesant de menționat ca prototipul mașinii de sortare și catalogare construit de Hollerith, a fost folosit în calcularea rezultatelor pentru recensământul populației SUA din 1890.
- 1904 - inventarea tubului electronic (dioda) de către John Fleming, menționat aici ca prim dispozitiv electronic ce va permite apariția calculatoarelor electronice.

- 1930 - Producție de masă a mașinilor electromecanice de calculat cu cele patru operații aritmetice elementare.
- 1937 - 1945. În SUA s-au dezvoltat mai multe mașini electromecanice de calculat, bazate pe relee electromagnetice, majoritatea funcționând în aritmetică zecimală, deși electromagnetul putea fi suport pentru aritmetica binară. Astfel, Howard Aiken fizician la Universitatea Harvard a propus în 1937 proiectul unui calculator electromecanic de uz general, bazat pe principiile enunțate de Charles Babbage. Construcția calculatorului, numit MARK 1, a început în 1939 și s-a terminat în 1944. Era programat cu informații înscrise pe bandă de hârtie perforată, sau cu cartelele de comandă și variabile.
- În 1938 Konrad Zuse a construit, în Germania, calculatorul electromecanic, numit Z1, care utiliza aritmetica binară în loc de cea zecimală. În 1941 Zuse realizează o variantă îmbunătățită a calculatorului său, pe care o numește Z3 și care se consideră că a fost primul calculator operațional de uz general controlat prin program. Unitatea aritmetică era construită cu relee și utiliza reprezentarea numerelor în virgulă mobilă.

1.10.3. Mașini electronice de calcul

Mașinile electronice de calcul, s-au dezvoltat în strânsă legătură cu dezvoltarea tehnologiilor de realizare a dispozitivelor electronice și a circuitelor integrate și de asemenea pe măsură dezvoltării științei calculatoarelor (organizare, arhitectura setului de instrucțiuni, limbaje de programare, programe de sistem etc.). Un element revoluționar pentru dezvoltarea resurselor fizice ale sistemelor de calcul l-a constituit perfecționarea tehnologiilor de producere a circuitelor integrate și apariția circuitelor integrate pe scară foarte mare (VLSI). Amintim ca în funcție de gradul de integrare circuitele integrate pot fi clasificate în:

- circuite integrate pe scară mică (SSI- aproximativ 1-10 porți logice convenționale pe chip-ul de siliciu; o poartă se consideră a avea aproximativ 5 tranzistoare echivalente)
- circuite integrate pe scară medie (MSI - până la o sută de porți / chip)
- circuite integrate pe scară mare (LSI - mii de porți / chip)
- circuite integrate pe scară foarte mare (VLSI - sute de mii de porți / chip)

Pentru a clasifica mașinile de calcul electronice, în literatura de specialitate se vorbește adesea despre generații de calculatoare electronice. Această clasificare pe generații succesive, deși utilă, este totuși bazată pe criterii destul de vagi, astfel că pentru unele calculatoare este destul de dificilă încadrarea într-o generație sau alta. Trebuie ținut cont aici, că pe lângă dezvoltările tehnologice, inovațiile în proiectare, dezvoltarea limbajelor și a tehnicilor de programare, un efect important asupra dezvoltării calculatoarelor l-a avut piața - criteriul comercial.

Având în vedere tehnologiile utilizate în construcția calculatoarelor, începând cu anul 1945, se pot evidenția cinci generații de calculatoare, ce vor fi descrise pe scurt în continuare ([Hayes88]).

Generația I de calculatoare electronice (aproximativ 1945-1954)

Caracteristici principale ale calculatoarelor de generația I:

- Utilizarea tuburilor electronice pentru realizarea circuitelor electronice
- Folosirea memoriilor interne cu tub catodic (CRT¹⁶) și linii de întârziere cu mercur;
- Capacitatea memoriei principale de aproximativ 1-2 kilo-cuvinte
- Aritmetica în virgulă fixă
- Viteza de operare: mii sau zeci de mii de instrucțiuni pe secundă.
- Programarea se făcea inițial doar în cod mașina, iar apoi și în limbaje de asamblare.

¹⁶ CRT = Cathode Ray Tube (engl.) = tub catodic. Este aceeași tehnologie utilizată și în televiziune.

Ca evenimente principale menționăm în ordine cronologică:

- În 1943 la Universitatea din Pennsylvania a început construcția primului calculator bazat pe tuburi electronice, numit ENIAC (Electronic Numerical Integrator And Computer), ce va fi terminat în 1946. Echipa de proiectanți și constructori a fost condusă de Presper Eckert, John Mauchly și John von Neumann. Calculatorul avea 18000 de tuburi electronice, cântărea 30 tone și era folosit pentru generarea automată a tabelelor balistice pentru armata SUA. Era cel mai rapid calculator construit până atunci, putând realiza cam 5000 de operații de adunare într-o secundă. ENIAC a fost construit pentru a funcționa în aritmetică zecimală. Pentru memorarea în acumulatori a unei cifre zecimale se utiliza un cerc constând din 10 bistabile cu tuburi. Contorul în cerc cu tuburi electronice era echivalentul roților contoare mecanice din generațiile anterioare de mașini de calcul. Mașina putea efectua operații de adunare și scădere, dar avea unități suplimentare pentru înmulțire, împărțire și rădăcină pătrată. Programarea se făcea manual prin comutatoare și cuple pentru cabluri de legătură la diferite module. De exemplu pentru adunarea a două numere din acumuloarele A1 și A2 trebuia stabilită manual o cale de date între A1 și A2. Introducerea datelor se făcea prin cartele perforate. Mai cuprindea niște memorii speciale, numite tabele de funcții, utilizate pentru stocarea tabelor și constantelor. Rezultatele se obțineau fie pe cartele perforate, fie la mașina electrică de scris. La această mașină de calcul datele și programele erau stocate separat.
- În timpul realizării lui ENIAC, în 1945, John von Neumann a lansat ideea de a stoca în aceeași memorie atât datele cât și programul, descriind arhitectura unui "calculator cu program memorat".
- În 1945 a început construcția unui alt calculator electronic, EDVAC (Electronic Discrete Variable Automatic Computer) pe baza lucrării lui von Neumann "Prima schiță de Raport asupra lui EDVAC". Pe lângă facilitarea procesului de programare, conceptul de program stocat, făcea posibil ca un program să-și modifice propriile instrucțiuni, proprietate rar utilizată în prezent. Înainte ca programul să fie executat, toate instrucțiunile și datele se stocau în memoria principală.
- În 1946 un colectiv condus de von Neumann a început proiectarea unei noi mașini, numită IAS, la Institute for Advanced Studies din Princeton (calculatorul a fost terminat în 1951). Această mașină utiliza o memorie principală cu acces aleator cu tub catodic, care permitea să se acceseze într-o operație un întreg cuvânt. Spre deosebire de EDVAC se utilizau circuite binare paralele. Calculatorului IAS, avea posibilitatea de a-și modifica partea de adresă din instrucțiune. Aceasta permitea utilizarea aceluiași program pentru prelucrarea mai multor seturi de date, reducându-se astfel spațiul ocupat de program în memorie. UCP conținea câteva registre cu tuburi electronice, folosite ca locații de stocare rapidă, implicită a operanzilor și rezultatelor. Deși avea facilități limitate de intrare-ieșire, mașina IAS era destul de modernă în concepție. Ea poate fi privită ca un prototip pentru toate calculatoarele de uz general ce au urmat. În cadrul memoriei auxiliare, se folosea un dispozitiv cu fir magnetic și se puteau stoca aproximativ 20 k-cuvinte.
- În 1948 cercetătorii americani John Bardeen, Walter Brattain și William Shockley au inventat tranzistorul (cu contact punctiform), deschizând era miniaturizării în circuitele electronice. Pentru descoperirea lor au primit în 1956 premiul Nobel în domeniul fizică.
- Pe baza proiectului EDVAC, Eckert și Mauchly, în cadrul unei companii proprii (Eckert-Mauchly Computer Corporation), au produs în 1951, primul calculator comercial, numit UNIVAC¹⁷. Memoria principală era realizată cu linie de întârziere cu mercur, iar memoria auxiliară cu bandă magnetică.
- În 1953 firma IBM construiește calculatorul cu program memorat, IBM 701, după ce anterior construisese doar echipamente pentru cartele perforate și doar o mașină de calcul, modelul

¹⁷ UNIVAC = Universal Automatic Computer

Howard Mark I (relee electronice și program cablat). Memoria principală era de tip electrostatic (tub catodic), iar pentru memoria auxiliară se foloseau tamburul magnetic și banda magnetică. Programele erau scrise în limbaj mașină, iar prin anii '50 în limbaj de asamblare.

- În 1954 a început producția (la firma Texas Instruments) primelor tranzistoare cu siliciu destinate comercializării.
- 1958 - la firma Texas Instruments se produce primul circuit care integrează mai mult decât un singur tranzistor în același chip de siliciu.

Generația a II-a de calculatoare electronice (aproximativ anii 1955 - 1964)

Din punct de vedere al tehnologiei de realizare este generația caracterizată de folosirea tranzistoarelor discrete. Totodată în această generație se dezvoltă limbajele de nivel înalt pentru a ușura munca programatorilor, și se dezvoltă sisteme de operare, pentru a gestiona resursele hardware și software ale sistemului.

Caracteristici principale ale calculatoarelor din generația a doua:

- Utilizarea tranzistoarelor în locul tuburilor electronice
- Apariția cablajului imprimat.
- Apariția memoriilor interne cu miezuri de ferită și tambur magnetic;
- Utilizarea registrelor index și implementarea cablată (în hardware) pentru aritmetica în virgulă mobilă;
- Se răspândesc limbajele de nivel înalt, independente de mașină, care simplifică munca de programare: ALGOL, COBOL, FORTRAN;
- Introducerea procesoarelor de intrare-ieșire (IOP¹⁸) pentru a supraveghea operațiile de I/O și pentru a elibera UCP de aceste sarcini. Procesoarele de intrare-ieșire (au fost numite și "canale", sau "sincronizatoare de date") au acces direct la memoria principală (prin multiplexarea adreselor între IOP și UCP). Ele recunosc un set redus de instrucțiuni și rulează programe specifice, independent și simultan cu UCP, inițierea oricărui transfer având însă nevoie de inițializarea făcută de UCP;
- Producătorii de calculatoare au început să furnizeze și software de sistem, ca de exemplu: compilatoare, biblioteci de subrutine și monitoare pe loturi;
- Performanțe principale: capacitatea memoriei principale de aproximativ 32 kilo-octeți și viteză de operare de aproximativ 2 MIPS.

Multe din îmbunătățirile pomenite mai sus au apărut sporadic în unele calculatoare cu tuburi, deci încadrate formal în generația anterioară. De la generația a 2-a se vorbește despre sisteme de calcul, pentru că numărul unităților de memorie, al procesoarelor, al dispozitivelor de I/O și alte componente de sistem puteau varia între diferite instalații, chiar dacă se utiliza același calculator de bază. La unele calculatoarele din această generație apare pentru prima oară memoria stivă¹⁹ (Electric KDF-9 și Burroughs B5000). Pentru această memorie de tip LIFO²⁰, la mașina B5000 primele două cuvinte din stivă reprezintă registre rapide ale UCP în timp ce restul stivei se găsește în memoria principală. Memoria stivă era folosită în așa fel încât toate instrucțiunile și operanzii erau preluate din vârful stivei, iar rezultatele erau de asemenea plasate în vârful stivei. De aici rezultă că în cele mai multe din instrucțiuni nu este nevoie de specificarea adresei operanzilor.

O dată cu îmbunătățirea echipamentelor de I/O (din punctul de vedere al vitezei), a apărut necesitatea de a se pregăti loturi de lucrări²¹ pentru execuție. Înainte de rulare acestea erau stocate pe bandă magnetică, iar apoi calculatorul le prelucra într-o secvență continuă plasând rezultatele pe o altă bandă magnetică, de ieșire. Prin lucrare / job, înțelegem o înlănțuire a execuției mai multor

¹⁸ IOP = Input-Output Processor

¹⁹ stack / stack memory (engl.) = stivă / memoria stivă

²⁰ LIFO = Last-In First-Out; ultimul articol introdus este primul ieșit-citit din memorie.

²¹ batch of jobs (engl.) = loturi de lucrări.

programe: compilator, programe de legare și încărcare, programe utilitare, lansare în execuție, cu scopul satisfacerii cererilor de servicii formulate de utilizator. Acestea sunt exprimate cu ajutorul unor comenzi, interpretate de o componentă a sistemului de operare. Fiecare comandă generează o fază a lucrării: compilare, legare, încărcare, execuție, etc. Înainte de apariția sistemelor de operare denumirea s-a referit la cereri lansate de utilizatori către echipamentele sistemului de calcul, cum ar fi de exemplu cererea pentru o sarcină de tipărire de la compartimentul de tipărire. Acest mod de administrare a operațiilor de I/O e numit *prelucrare în loturi* (*batch processing*²²), sau prelucrare serială. Prelucrarea pe loturi cere utilizarea unui program supervisor numit monitor, care e permanent rezident în MP. Un *monitor de loturi* ("batch monitor") este un exemplu de program de sistem, proiectat pentru a furniza un set de servicii comune către toți utilizatorii unui calculator, el fiind un sistem de operare (SO) rudimentar. Prelucrarea pe loturi a permis utilizarea mai bună a timpului UCP, componenta cea mai scumpă la acel moment. Apoi, o dată cu scădere continuă a costurilor de realizare a UCP, prelucrarea batch (pe loturi) a fost înlocuită de sistemele cu *partajarea timpului UCP*, ca un ajutor al productivității programatorilor, existând un răspuns rapid pentru mai mulți utilizatori. Astfel, cu sprijinul sistemului de operare, se permitea unui singur procesor (UCP) să prelucreze concurrent, un set de programe independente, tehnica fiind numită *multiprogramare*. Aceasta se realizează în felul următor: UCP suspendă temporar execuția programului curent, începe execuția unui alt program și se întoarce la primul mai târziu. Se realizează deci o multiplexare în timp a resursei UCP între diferitele programe. Acolo unde este necesar programul suspendat este transferat către un IOP pentru un transfer I/O. În acest fel multiprogramarea ține UCP și unul sau mai multe IOP-uri ocupate prin suprapunerea operațiilor efectuate de UCP și IOP. Acest tip de sisteme multiprogramate se numesc sisteme *timesharing* (cu partajare în timp).

Tot în perioada de timp în care sunt încadrate calculatoarele de generația a doua, au apărut calculatoare foarte puternice, numite supercomputere, folosite pentru prevederea vremii, cercetare în domenii militare, tehnică spațială, tehnică nucleară, inteligență artificială, domenii la care se cere un volum foarte mare de pași de calcul (de exemplu UNIVAC LARC - Livermore Atomic Research Computer și IBM 7030 - numit și Stretch Computer). La supercalculatoare apare pentru prima oară noțiunea de procesare paralelă. Două din tehnicile de prelucrare paralelă dezvoltate la această generație de supercalculatoare sunt:

1. Suprapunerea operațiilor de aducere (fetch) și execuție a instrucțiunilor din cadrul unui program. Se realizează prin hardware special ce cuprinde procesoare pipeline (tip conductă, sau bandă de asamblare) sau unități ALU multiple;
2. Suprapunerea execuției diferitelor programe. Asta se face prin proiectarea unor sisteme multiprocesor (sistem cu mai multe unități UCP). Ca și multiprogramarea multiprocesarea necesită un program supervisor relativ complex.

Atât LARC cât și Stretch au constituit dezastru financiar-comercial, dar au influențat considerabil următoarele generații de calculatoare. Supercalculatoarele au introdus unele soluții de îmbunătățire a arhitecturii, pentru creșterea vitezei de calcul, proprii mai degrabă generației următoare decât generației din care fac parte din punctul de vedere al tehnologiei.

Generația a treia (aproximativ anii 1965-1978)

Anul 1965 poate fi considerat ca marcând începutul celei de-a treia generații de calculatoare, deși distincțiile dintre generațiile 2 și 3 nu sunt extrem de clare. Se amintesc adesea următoarele îmbunătățiri specifice acestei generații:

²² batch processing = prelucrare secvențială / pe loturi succesive.

- Circuitele integrate (C.I.²³) încep să înlocuiască circuitele cu tranzistoare discrete. Rezultă o reducere importantă a dimensiunilor, costurilor și consumului de putere, dar și o creștere a vitezei de lucru.
- Memoriile semiconductoare sub forma de C.I. au capacitate de memorare crescută și înlocuiesc treptat celelalte tipuri de dispozitive în cadrul memoriei principale.
- Se generalizează tehnica microprogramării pentru realizarea unității de control. Ca urmare se simplifică activitățile de proiectare a UCP. Este unul din elementele care au permis dezvoltarea seriilor de calculatoare compatibile din punctul de vedere al arhitecturii.
- Se dezvoltă o serie de tehnici menite a crește viteza de execuție a programelor: pipelining, multiprocesare.
- Se dezvoltă sisteme de operare, acestea furnizându-se o dată cu sistemul de calcul.
- Apar noi limbaje de programare (Pascal, C, LISP), grafica pe calculator, baze de date, etc.

În perioada de timp descrisă ca fiind a generației a treia apare un număr foarte mare de calculatoare, prețul lor scăzând semnificativ față de generația anterioară (de exemplu: IBM²⁴ System/360 (1965), IBM System/370 (1970), CDC 6600²⁵, CDC 7600, CYBER, ASC²⁶, ILLIAC IV²⁷).

Tot în această generație se produc și comercializează primele calculatoare multiprocesor.

O caracteristică importantă a calculatoarelor din această generație este utilizarea generalizată a sistemelor de operare. Se dezvoltă sistemele de operare multiprogramare²⁸ (multiprogramming), multiutilizator²⁹ și multi-procesare³⁰ (multiprocessing)

Generația a treia este generația în care se produce un fenomen deosebit pentru dezvoltarea și răspândirea tehnicii de calcul, prin producția de masă a unor calculatoare mici și de preț scăzut, numite *minicalculatoare*. Ca exemple se pomenesc seria de minicalculatoare PDP - Programmed Data Processor de la firma Digital Equipment Corporation (DEC) și seria LINC - Laboratory Instrument Computer al firmei MIT.

Datorită costului lor scăzut, minicalculatoarele erau adecvate pentru o mare varietate de aplicații, ca de exemplu controlul industrial, unde era nevoie de calculatoare dedicate permanent unei anumite aplicații. Unele minicalculatoare au performanțe comparabile, sau chiar mai bune, decât calculatoarele mari din generația a 2-a.

În această perioadă se implementează conceptul memoriei virtuale. *Memoria virtuală* presupune o tehnică de organizare ierarhică (pe cel puțin două niveluri) a memoriei, administrarea sistemului de memorie făcându-se de un sistem de operare, cu scopul de a mări capacitatea aparentă a memoriei la care UCP are acces direct³¹. Deoarece capacitatea memoriei principale este limitată, în primul rând din considerente de cost, este imposibil să se stocheze în memoria principală (MP) toate programele aflate în execuție și datele aferente lor. De aceea, este necesar să se aloce dinamic

²³ C.I. = Circuit Integrat, original în engleză Integrated Circuit - I.C.

²⁴ IBM = International Business Machines

²⁵ CDC = Control Data Corporation

²⁶ ASC = Advanced Scientific Computer, Texas Instruments

²⁷ ILLIAC = Illinois Automatic Computer

²⁸ multiprogramming = multiprogramare. Execuția simultană întrețesută a două sau mai multe programe sau sarcini. Întrețeserea se referă la faptul că la un anumit moment de timp se execută un singur program, dar există mai multe programe lansate în execuție. Cele suspendate temporar așteaptă producerea unor evenimente care să le relanseze în execuție.

²⁹ multiuser = multiutilizator. Se referă la un sistem de calcul ce poate fi folosit de mai mulți utilizatori ce au acces simultan la resursele sistemului, de obicei de la terminale diferite. În cazul sistemelor de calcul uni-procesor, accesul multiplu la resurse se asigură prin partajarea timpului UCP.

³⁰ multiprocessing = multiprocesare. Prelucrare a două sau mai multe programe sau sarcini de lucru de către două sau mai multe procesoare de uz general (UCP), într-un calculator numit multiprocesor.

³¹ acces direct = transferul datelor se face direct între memorie și registrele UCP, conform informațiilor de adresare și selecție generate de UCP.

spațiul de memorie între diferitele programe concurente și să se transfere (copieze) informația din memoria auxiliară (MA) în MP, pe măsură ce ea este cerută. De asemenea programarea este mult simplificată dacă calculatorul poate fi văzut ca având o singură memorie adresabilă direct, nelimitată ca dimensiuni, și la care să aibă acces fără restricții toți programatorii. Memoria virtuală a apărut pentru prima dată pe o mașină numită Atlas, proiectată în Anglia în 1962. IBM 360/85, a introdus-o la sfârșitul anilor '60, aceasta fiind prima mașină comercială ce utiliza și o memorie cache. Pentru a înțelege sistemele de memorie virtuală, trebuie să distingem între setul adreselor de memorie specificate de un program, numit spațiul adreselor logice L și setul adreselor reale utilizate de MP, ce formează spațiul adreselor fizice F. Spațiul L poate fi mult mai mare decât F și de aici termenul de "virtual". În timpul execuției programului, fiecare adresă logică în L e translatată într-o adresă fizică din F. Ca urmare este necesar un mecanism care implementează funcția de translatare din L în F ($f: L \rightarrow F$). Dacă la un moment dat se determină că informația specificată de UCP nu este în MP execuția programului este suspendată până când pagina³² ce conține respectiva informație este transferată din MA (disc magnetic) în MP.

În cadrul organizării ierarhice a memoriei, în această perioadă, apar memoriile tampon de mare viteză, (cache), între UCP și MP. Un cache este o memorie de mici dimensiuni, rapidă (viteza fiind apropiată de cea a procesorului), care păstrează cele mai recent accesate informații (coduri de program și date) din memoria principală.

Tehnica microprogramarii, răspândită începând cu generația a treia de calculatoare, este o tehnică sistematică și flexibilă de implementare a unității de control (UC) a unui procesor. După apariția seriei IBM System /360, tehnica de implementare microprogramată a controlului s-a generalizat, situație ce s-a păstrat până la sfârșitul anilor '80. La microprogramare, informația de control este înscrisă într-o memorie rapidă integrată în UC, numita memorie de control. Semnalele de control, în formă binară, sunt asamblate în vectori binari numite microinstrucțiuni. Fiecare instrucțiune a procesorului este "interpretată" de un microprogram format dintr-o secvență de microinstrucțiuni. Microinstrucțiunile controlează secvența de microoperații (în esență operații de transfer între registre) necesare pentru o fază a execuției instrucțiunii și totodată conțin informații privind adresa microinstrucțiunii următoare (informație de secvențiere). Din cauza faptului că microprogramele reprezintă software inclus în hardware procedeul poartă și numele de *firmware*.

Deși primul microprocesor, în tehnologie de integrare la nivel VLSI, a fost proiectat în 1969 și a apărut pe piață în 1971 (Intel 4004), microprocesoarele vor fi incluse în generația următoare de calculatoare.

Generația a patra (începând aproximativ din 1979)

Arhitecturile din aceasta generație de calculatoare se bazează în primul rând pe dezvoltarea accelerată a tehnologiei de fabricație a circuitelor integrate VLSI. A devenit astfel posibil ca într-un singur C.I.³³ să se fabrice o unitate centrală de procesare (UCP) sau un bloc de memorie principală. Aceste C.I. pot fi produse în cantitate mare și la un preț scăzut. Astfel a putut să apară noua clasă de calculatoare personale (PC) ieftine, de mare performanță, unele din ele conținând mai multe procesoare lucrând în paralel.

O Unitate Centrală de Procesare construită pe un singur C.I. este numit **microprocesor**. La început calculatorul construit pe bază de microprocesor a fost numit *microcalculator*. În prezent toate unitățile centrale folosite pentru calculatoarele de uz general pot fi încadrate în categoria microprocesoarelor. Pentru domeniul Embedded (pomenit mai sus) se fabrică și microcalculatoare pe un chip, care conțin pe un singur circuit integrat blocurile: UCP, memorie, sistem I/O. Pentru că

³² pagină de memorie = o zonă continuă de memorie de o anumită dimensiune

³³ C.I. = Circuit Integrat

aceste din urmă microcalculatoare sunt construite special pentru aplicații ce trebuie controlate, ele sunt apelate adesea cu numele “microcontroller”.

Microprocesorul este o unitate completă de prelucrare de uz general, cu ALU, registre, o anumită cantitate de memorie locală și logică de control, toate într-un singur chip. Microprocesorul este o componentă foarte eficientă din punctul de vedere al costului și, datorită programabilității sale, este și o componentă foarte flexibilă care poate fi utilizată într-o gamă vastă de produse (de exemplu: jocuri video, aplicații casnice, ceasuri electronice, sistem de diagnosticare și control în automobile sau aviație, instrumentație electronică medicală).

Microprocesorul este în prezent componenta utilizată pentru crearea de calculatoare cu mare putere de prelucrare a informațiilor, fiind o componentă inclusiv pentru construcția sistemelor multi-procesor și a super-calculatoarelor³⁴.

În generațiile anterioare instalarea unui calculator conducea la cheltuieli importante pentru amenajarea sălii, instalarea echipamentelor de protecție la incendii și a sursei stabilizate de tensiune, instalația de climatizare a sălii, angajarea colectivului de specialiști pentru exploatarea sistemului de calcul etc. Apariția microprocesorului, și a micro-calculatoarelor (și în special a celor numite calculatoare personale, PC), nepretențioase ca gabarit, preț, consum de putere, întreținere etc., a determinat pătrunderea informaticii în toate sectoarele de activitate.

În prezent se produce o mare diversitate de microprocesoare, microcalculatoare, microcontrollere, procesoare digitale de semnal, microprocesoare specializate pentru prelucrarea paralelă a informațiilor (“transputer”), și aceasta pentru că microprocesorul a pătruns în toate activitățile vieții economice și sociale. Peste tot unde este ceva de controlat, această unitate de procesare programabilă ieftină și de foarte mici dimensiuni a devenit indispensabilă. Aplicația este cea care determină tipul și caracteristicile cerute procesorului pe un chip.

În prezent microprocesoarele intră în construcția unităților centrale a sistemelor de calcul “mari”. Ele mai sunt numite microprocesoare doar datorită faptului că ocupă una sau câteva capsule integrate. De asemenea în sistemele de calcul mari, microprocesoarele pot executa diferite funcții de I/O.

Evoluția calculatoarelor din ultima decadă se confundă de fapt cu evoluția microprocesoarelor.

Circuitele integrate ce apar în construcția calculatorului pot fi standard, sau specifice aplicațiilor. Acestea din urmă sunt apelate de obicei prin prescurtarea **ASIC** (Application Specific Integrated Circuits - Circuite integrate cu aplicații specifice).

Circuitele ASIC pot fi de următoarele tipuri:

a) personalizate la cerere (semicustom):

PLD (dispozitive logice programabile, care sunt configurabile) și

GA (gate arrays, rețele de porți, care sunt predifuzate)

b) la comandă (custom), care pot fi împărțite la rândul lor în

b1) precaracterizate (Standard Cells)

b2) complet la cerere (full custom).

Circuitele semi-custom au o cantitate relativ mare de logică internă prefixată, ca circuite de bază, implementate în siliciu. Pentru a ajunge la rețeaua logică țintă funcțiile specifice țintei se programează pe circuit fie prin ardere la utilizator (la PLD-urile timpurii) fie prin proiectarea unor măști de realizare a legăturilor în siliciu; în acest ultim caz operația de realizare fizică se face la producătorul circuitului integrat.

³⁴ Supercomputer (engl.) = super-calculator. Denumirea se referă la un calculator cu putere extrem de mare de calcul, proiectat pentru a executa cu viteză maximă aplicații cum ar fi: prognoza meteo, tehnică spațială, tehnică militară, cercetare științifică etc.

Circuitele PLD (Programmable Logic Devices - Dispozitive logice programabile) cuprind în principal tipurile: PROM, PLA și PAL. Aceste circuite programabile implementează funcții logice disjunctive (sume de produse logice), pentru fiecare variantă de circuit programabil fiind specifice gradul de integrare, și nivelul logic care poate fi programat.

Circuitele PROM sunt memorii doar cu citire ce pot fi programate o singură dată pe nivelul logic de SAU (nivelul logic SI, care este nivelul de intrare, este prefixat, el fiind reprezentat de un decodificator - DCD).

Circuitul PLA, o matrice logică programabilă, are ambele niveluri (SI și SAU) programabile. Un circuit integrat PLA (programabil cu fuzibil) conține tipic 50 - 250 porți convenționale și poate înlocui până la 4-6 chip-uri SSI sau MSI.

Circuitele PAL, au gradul de integrare cel mai mare dintre cele trei tipuri de PLD-uri, aproximativ 300 de porți pe chip, situându-se ca tehnologie între MSI și LSI. Ele pot conține pe chip atât porți logice cit și bistabile, dar la aceste circuite este programabil doar nivelul de SI, cel de SAU fiind prefixat.

O altă abordare a proiectării cu logică semi-custom, ce a câștigat importanță în ultimii ani, este utilizarea rețelelor de porți (GA - gate array), numite de asemenea și ULA (uncommitted logic array - rețea logică neconectată). Circuitul GA este un circuit LSI ce conține o rețea de celule sau porți logice identice, neconectate, dar înconjurate de o rețea de interconectare. Intersecțiile rețelei sunt programabile (se face legătură, sau nu) prin masca creată de viitorul utilizator cu ajutorul unor programe de proiectare asistată de calculator. O funcție țintă poate fi implementată în GA prin plasarea elementelor specifice funcțiilor urmărite în porțile rețelei (așa numita problemă a *plasării*) și conectarea adecvată a porților selectate (așa numita problema a *rutării*). Pe un circuit de tip GA ne-personalizat, celulele de bază, identice, sunt înconjurate de un număr de porți de I/O. Avantajul major al circuitelor GA constă în faptul că realizarea unei funcții logice particulare presupune doar sarcinile de plasare și rutare. Mai mult, din cauza naturii determinate a poziției celulelor, plasarea și rutarea pot fi eficient rezolvate utilizând programe CAD (computer-aided design). Gate arrays (în prezent, mai ales sub forma lor extinsă FPGA - Field Programmable Gate Arrays) sunt utilizate în primul rând din cauza timpului scăzut cerut pentru proiectare și în consecință a prețului scăzut de proiectare. În același timp, trebuie specificat că chip-urile GA necesită suprafețe mai mari în comparație cu circuitele full-custom și nu permit ușor realizarea unor niveluri foarte mari de integrare. GA sunt produse într-o mare varietate de tehnologii (ECL, TTL, I²L, NMOS, CMOS) și în consecință există o mare varietate a densității de porți. Circuitele GA au în jur de 1200 - 6000 de porți și se încadrează tehnologic între LSI și VLSI. Se poate spune că o singura rețea de porți (GA) poate înlocui câteva circuite PLA. La GA se pot defini și funcțiunile pinilor capsulei, pe când la PLA și PAL aceste funcții sunt pre-fixate.

Circuitele precaracterizate (pe baza de macrocelule - Standard Cells), folosesc ca celule standard circuite proiectate anterior, aflate într-o bibliotecă. În aceste macrocelule standard sunt incluse porți logice, bistabile, numărătoare, dar și dispozitive logice programabile, cum sunt microprocesoarele. Proiectantul și apoi producătorul la turnătorie de siliciu realizează doar interconectările între macrocelule conform dorinței beneficiarului.

Circuitele full-custom, sunt circuite specifice concepute în strictă concordanță cu cerințele beneficiarului. Implementarea full-custom LSI sau VLSI a unei arhitecturi conduce la cea mai bună exploatare a suprafeței chip-ului de semiconductor, dar dacă numărul de circuite produse și vândute este mic, prețul specific va fi foarte mare, mult mai mare decât la toate celelalte variante de circuite.

Caracteristicile cele mai importante ale circuitelor full-custom pot fi rezumate în cele ce urmează:

1. Pentru un set de performanțe și caracteristici funcționale date, abordarea full-custom conduce la cea mai compactă implementare, cu cel mai mic produs viteză-putere. Fiabilitatea circuitului va fi de asemenea foarte mare.

2. Totuși, personalizarea completă cere o investiție foarte mare în timpul de proiectare. Proiectarea, verificarea și mai ales obținerea formei finale de organizare a chipurilor VLSI, sunt consumatoare extreme de timp.
3. Din cauza investiției inițiale mari, abordarea full-custom devine eficientă ca și costuri dacă volumul producției este foarte mare, de ordinul sute de mii sau milioane de exemplare. Acest factor economic formează bazele producției fabricanților diferitelor microprocesoare.

Cu programe CAD se proiectează arhitectura circuitelor ASIC și totodată se face testarea circuitului simulat înainte ca acesta să fie produs. Deci plecând de la calculatorul universal programabil folosit pentru realizarea anumite funcții prin program (logică programată), datorită dezvoltării metodelor CAD și a perfecționării tehnologiei circuitelor VLSI, se remarcă revenirea la structuri hardware dedicate funcțional, foarte eficiente ca viteza de lucru, număr de componente, consum energetic (implementarea prin hardware a funcțiilor, sau logica cablata).

Dacă analizăm piața de microprocesoare de uz general, vom observa că ea este dominată de două familii de procesoare diferite. Este vorba de familia 80x86 de la Intel și seria 68000 de la Motorola. [Mesmer] Este mare păcat că cele două familii nu sunt compatibile: modul în care instrucțiunile sunt manipulate intern, limbajul la nivelul mașinii de bază și modul în care sunt alocate adresele de memorie RAM sau zonele de adrese de I/O sunt complet diferite. Procesoarele Intel, inclusiv cele mai noi din seria Pentium, au fost utilizate pentru realizarea mașinilor numite Calculator Personal (PC - Personal Computer). Seria 68000 a fost utilizată mai ales în realizarea mașinilor Apple Macintosh, recent echipate însă cu procesoare cu set redus de instrucțiuni (mașini RISC).

Dezvoltarea microprocesoarelor este caracterizată de dezvoltări arhitecturale specifice, dar și de dezvoltări tehnologice, care au permis includerea pe același circuit integrat a tuturor componentelor necesare unui calculator numeric. Istoria calculatoarelor, începând cu anii '70 ai secolului XX se confundă cu istoria microprocesoarelor și a fost extrem de agitată, prin marcarea unui mare număr de "evenimente". Amintim doar câteva, în ordine cronologică:

- 1971 - Intel a produs primul microprocesor de 4 biți (Intel 4004)
- 1974 - Intel produce microprocesorul de 8 biți I8080
- 1974 - Motorola produce microprocesorul de 8 biți MC 6800
- 1976 - Apare primul microcalculator, destinat în special activităților recreative, produs de noua firmă Apple. Ca UCP s-a folosit un microprocesor 6502 (asemănător cu MC6800) de la firma MOS Technologies, proiectanții microcalculatorului fiind doi americani entuziaști - Stephen Jobs și Steven Wozniak.
- 1976 - Zilog produce popularul microprocesor de 8 biți numit Z80
- 1978 - Intel produce microprocesorul de 16 biți I8086
- 1979 - Motorola produce microprocesorul de 16 biți MC 68000, la care apare pentru prima oară o structură de tip pipeline rudimentară cu trei etaje.
- 1981 - IBM lansează pe piață Personal Computer (IBM PC), moment extrem de important în răspândirea calculatoarelor de uz general construite pe bază de microprocesor.
- 1981 - apariția primelor proiecte de arhitecturi RISC: RISC 1 și RISC 2 la Universitatea din Berkeley, respectiv MIPS la Universitatea Stanford
- 1984 - Motorola MC68020, care introduce paralelismul la nivelul executării instrucțiunilor printr-o structură pipeline cu cinci etaje.
- 1985 - procesor ARM (proiectat în prezent la Advanced RISC Machine, inițial denumită Acorn RISC Machine) - procesoare RISC destinate în special domeniului Embedded, pentru dispozitive portabile. Dacă în 1985 a apărut ARM1, în 2002 a fost lansat ARM 11, un RISC pe 32 biți cu extensii arhitecturale pentru instrucțiuni codate pe 16 biți (TDMI) sau set de

instrucțiuni Java. Este considerat un procesor extrem de reușit, astfel că nuclee ARM dotează majoritatea telefoanelor mobile digitale și agendele electronice din prezent.

- 1990 - mașini RISC de la Motorola (MC 88110) și Intel (80960)
- 1990 - Apariția arhitecturilor superscalare
- 1991 - R4000 un procesor RISC de 64 biți de la MIPS Computer Systems
- 1994 - Intel produce primul procesor Pentium la 133 MHz
- 1999 - Intel Pentium III la 500 MHz
- 2000 - Apariția primelor procesoare cu frecvență de ceas peste 1 GHz.
- 2002 - Apariția primelor procesoare cu frecvență de ceas peste 2 GHz.

CAPITOLUL 2

MODALITĂȚI DE EVALUARE A PERFORMANȚELOR CALCULATOARELOR

Conținut:

- 2.1. Timpul, ca măsură a performanțelor
- 2.2. Alte unități de măsură
- 2.3. Programe etalon de testare a performanțelor
- 2.4. Compararea performanțelor prin mediere
- 2.5. Legea lui Amdahl

Obiective educaționale:

- a. Stabilirea și descrierea factorilor ce contribuie la performanțele calculatoarelor
- b. Analiza și clasificarea diferitelor metrici și programe de testare a performanțelor
- c. Aplicarea legii lui Amdahl pentru analiza creșterilor de performanță

2.1. TIMPUL, CA MĂSURĂ A PERFORMANTELOR CALCULATORULUI NUMERIC

Performanțele procesorului unui calculator numeric se evaluează în primul rând prin determinarea timpului de execuție al programelor. *Timpul* este măsura ce indică performanța. Comparând două mașini de calcul, concluzia de genul “Mașina A este mai rapidă decât mașina B” se referă la faptul că timpul de execuție al aceluiași program pe mașina A este mai mic decât pe mașina B. Dacă mașina A este *de n ori* mai rapidă decât mașina B putem scrie:

$$\frac{Timp_executie_B}{Timp_executie_A} = n \quad (2.1)$$

sau considerând că performanța (ca viteză de execuție) este inversul timpului:

$$\frac{Timp_executie_B}{Timp_executie_A} = \frac{1/Performanta_B}{1/Performanta_A} = \frac{Performanta_A}{Performanta_B} = n \quad (2.2)$$

Comparația între cele două mașini poate fi exprimată și procentual: “Mașina A este cu p% mai rapidă decât mașina B”. Aceasta se poate exprima analitic astfel:

$$\frac{Timp_executie_B}{Timp_executie_A} = 1 + \frac{p}{100} \quad (2.3)$$

sau

$$\frac{Performanta_A}{Performanta_B} = 1 + \frac{p}{100} \quad (2.4)$$

adică

$$p = 100 \cdot \left(\frac{Performanta_A - Performanta_B}{Performanta_B} \right) \quad (2.5)$$

O expresie de genul “Mașina A este mai performantă cu 50% față de mașina B” indică că numărul de sarcini de lucru terminate în unitatea de timp pe mașina A este de 1.5 ori mai mare decât pe mașina B.

Cele spuse mai sus sunt adevărate pentru un calculator care execută un singur program de testare, fără întreruperi de la lansare, până la terminarea sarcinii de lucru. În alte cazuri, cum ar fi de exemplu un sistem cu multi-programare, UCP poate abandona temporar un program (întreruperea execuției) în timp ce se așteaptă operații de I/O și să se treacă la execuția altui program. Ca urmare, în măsurarea timpului de execuție al unui program pe un anumit procesor trebuie ținut cont și de eventuala întrerupere temporară a programului. Evident că din punctul de vedere al utilizatorului timpul este un *timp de răspuns* (de la lansare până la terminarea execuției) sau un *timp de așteptare*, care include și intervalele de timp când se rulează altceva, prin partajarea timpului procesorului. De aceea, în [Patterson96] se introduce noțiunea de *timp al UCP* (pe care îl vom nota în continuare T_{UCP}), care este o măsură a timpului efectiv cât procesorul (UCP) rulează programul de testare a

performanțelor. Acest timp reprezintă timpul cât UCP realizează prelucrări fără a include așteptarea operațiilor de I/O și rularea altor programe. Timpul T_{UCP} include însă și intervalele de timp ce corespund timpului UCP consumat de către sistemul de operare pentru realizarea unor sarcini cerute de programul de test.

Evaluarea performanțelor unui calculator se face pentru a putea compara aceste performanțe cu performanțele obținute de alte calculatoare. Un utilizator al calculatorului poate spune că un calculator este mai rapid decât altul dacă acesta rulează un program într-un timp mai scurt, pe când administratorul unui centru de calcul poate spune că un calculator este mai rapid dacă acesta termină mai multe sarcini de lucru într-un anumit interval de timp. Utilizatorul calculatorului este interesat în primul rând de reducerea *timpului de execuție* (sau a *timpului de răspuns* între momentele startului și terminării unei sarcini de lucru), sau altfel spus în micșorarea timpului de așteptare (exprimat adesea ca latență). Managerul centrului de calcul este interesat în creșterea *performanțelor globale* ale sistemului de calcul exprimate prin denumirea “*throughput*”¹ (putere de calcul).

De obicei termenii “timp de răspuns”² și “throughput” se utilizează când se discută o întreagă sarcină de calcul.

Atunci când se analizează performanțele sistemului de memorie se folosesc termenii “*bandwidth*” (*lărgime de bandă*) și “*latency*” (*timp de așteptare, întârziere în răspuns*)

Pentru a exprima performanțele calculatoarelor cercetătorii în domeniu au introdus mai întâi unități de măsură care țin seama de perioada de ceas a UCP. Astfel, timpul de execuție al unui program, specific unei UCP, poate fi exprimat ca [Patterson94]:

$$T_{UCP} = NrI \times CPI \times T_{clk} \quad (2.6)$$

unde:

- NrI = numărul de instrucțiuni din program
- CPI ³ = numărul mediu de perioade de ceas pe instrucțiune; poate fi calculat prin împărțirea numărului de perioade de ceas pentru un program, la numărul de instrucțiuni din program.
- T_{clk} = perioada impulsului de ceas a procesorului.

Exemplu numeric:

Dacă frecvența ceasului unui procesor este de 1 MHz, iar rularea unui program necesită 5 milioane de perioade de tact pentru execuție completă, care este timpul de execuție al programului?

R:

$$\begin{aligned} T_{UCP} &= NrI \times CPI \times T_{clk} \\ &= \text{Perioade_tact_pe_program} \times T_{clk} \\ &= \frac{\text{Perioade_tact_pe_program}}{f_{clk}} \end{aligned}$$

¹ throughput - este termenul folosit pentru a descrie performanțele unui calculator prin cantitatea de date pe care le poate prelucra în unitatea de timp, sau performanțele globale ale unui calculator, exprimate prin capacitatea de a transmite / recepționa cât mai multe date în unitatea de timp cu toate componentele sistemului, inclusiv cu dispozitivele de stocare cum sunt unitățile de disc

² response time (engl.)

³ CPI = Clock Cycles Per Instruction

$$T_{UCP} = \frac{5 \times 10^6}{1 \times 10^6} = 5 \text{ secunde}$$

Încercând să comparăm diferite sisteme, apar următoarele problemele:

- frecvențele de ceas sunt diferite pentru diferite calculatoare
- arhitecturile celor două calculatoare comparate pot fi diferite, conducând la valori diferite pentru componentele din relația timpului de execuție pe UCP.

De aceea, pentru a putea face o comparație cât de cât acceptabilă, trebuie rulat același program de testare a performanțelor pentru ambele calculatoare care se compară.

Pentru a micșora timpul de execuție ar trebui să micșorăm cele trei componente din relația (2.6), componente care nu sunt însă independente.

Numărul mediu de perioade de ceas pe instrucțiune (*CPI*) este determinat de:

- arhitectura procesorului, prin componentele: set de instrucțiuni și modul de organizare
- viteza de lucru a procesorului cu memoria principală, datorită faptului că instrucțiunile și datele inițiale / finale se stochează în această memorie.

Valoarea medie *CPI* poate fi exprimată fie ca raport între numărul total de cicluri de ceas și numărul de instrucțiuni, fie în funcție de numărul mediu al perioadelor de ceas, (*P*), necesare procesorului pentru decodificarea și execuția instrucțiunilor, numărul mediu de referințe-accesări la memorie (*A*) și raportul *k* dintre timpul de acces la memorie și perioada de ceas a procesorului, conform relației:

$$CPI = P + A \times k \quad (2.7)$$

Din cele pomenite se conturează câteva direcții ce pot fi folosite pentru a micșora *CPI* și implicit și T_{UCP} :

- proiectare adecvată a setului de instrucțiuni
- organizare ce permite operații paralele în execuția instrucțiunilor
- proiectare a unității de control în scopul micșorării timpului de decodificare și a numărului de pași (stări ale automatului) utilizați în ciclul de execuție a instrucțiunilor
- micșorarea timpului de acces la memoria principală prin diverse metode (tehnologia de realizare și organizarea internă a circuitelor de memorare, introducerea unor niveluri intermediare de memorie de mare viteză, optimizarea organizării spațiului de adresare)

A doua componentă din relația (2.6), numărul de instrucțiuni (*NrI*), depinde de tipul programului rulat, de arhitectura setului de instrucțiuni a procesorului și de structura și calitatea compilatorului folosit la translatarea din program sursă (în limbaj de programare de nivel înalt), în program obiect. Remarcăm că între direcțiile de micșorare a apărut din nou arhitectura setului de instrucțiuni, dar în sens contrar celui de la *CPI*. Aceasta pentru că un număr mai mic de instrucțiuni pe program înseamnă, în general, instrucțiuni mai complexe, deci care durează un număr mai mare de cicluri de ceas. Dar un număr mai mic de instrucțiuni pe program, presupune și un număr total mai mic de instrucțiuni recunoscute de procesor (funcții implementate în setul de instrucțiuni). Aceasta înseamnă un număr mai mic de biți necesari pentru codificarea instrucțiunilor și ca urmare

o posibilitate de micșorare a lungimii instrucțiunilor, cu efect și asupra dimensiunilor necesare în memoria locală, intermediară, sau principală.

Frecvența impulsului de ceas (clock) a UCP depinde în primul rând de tehnologia de realizare a procesorului, dar și de organizarea internă a acestuia. Din păcate creșterea frecvenței de oscilație a impulsului de ceas conduce la creșterea puterii consumate și a căldurii disipate de circuite.

Îmbunătățirea tehnologică este o condiție necesară, dar nu și suficientă pentru creșterea spectaculoasă a performanțelor. *Procesarea paralelă* a informației conduce la îmbunătățirea performanțelor globale ale mașinii, fără a modifica ciclurile de bază ale procesorului. Condiția de funcționare a arhitecturilor care permit execuția în paralel a mai multor instrucțiuni este ca instrucțiunile să fie independente. Procesarea paralelă a instrucțiunilor într-o arhitectură *pipeline*, poate conduce, în cazul ideal, la terminarea unei instrucțiuni la fiecare perioadă de ceas. În cazul ideal performanța se va îmbunătăți de un număr de ori egal cu numărul de nivele ale arhitecturii pipeline.

Îmbunătățirea performanțelor dincolo de nivelul obținut prin pipeline (ideal - o instrucțiune pe ciclu de ceas) se poate face în două moduri [Nicula97]:

- creșterea numărului de etaje pipeline și lansarea *succesivă* a mai multor instrucțiuni pe o perioadă de ceas (*arhitectură superpipeline*);
- lansarea *simultană* a mai multor instrucțiuni pe o perioadă de ceas, spre unități de execuție multiple (*arhitectură superscalară*).

Ambele variante se bazează pe aceeași idee: “lansarea mai multor instrucțiuni într-o perioadă de ceas nu garantează terminarea mai multor instrucțiuni pe perioadă de ceas; dar, lansarea unei singure instrucțiuni pe perioadă de ceas *în mod sigur nu* poate determina terminarea mai multor instrucțiuni pe perioadă de ceas”.

Cu privire la termenul CPI, care este o valoare medie, calculată pentru un anumit program, a numărului de perioade de ceas pe instrucțiune, trebuie să observăm următoarele:

- într-un program există mai multe tipuri de instrucțiuni, fiecare tip de instrucțiune putând avea un număr diferit de cicluri de ceas pentru execuție completă. De exemplu o instrucțiune de transfer între două registre va dura mult mai puțin decât o instrucțiune de înmulțire a două numere.
- arhitectura setului de instrucțiuni, prin modurile de adresare folosite, poate conduce la diferențe de timp, chiar între instrucțiuni similare. Astfel o instrucțiune de scădere a doi operanzi aflați în registrele interne ale UCP va dura mai puține cicluri de ceas decât o instrucțiune la care unul din operanzi se găsește în memorie.

Exemplu numeric:

Într-un program de testare a performanțelor (benchmark) cu 250 instrucțiuni, există următoarele tipuri de instrucțiuni:

- 100 instrucțiuni de tip load/store⁴, fiecare consumând 5 perioade de ceas.
- 80 instrucțiuni de transfer între registre interne ale UCP, fiecare consumând 1 perioadă de ceas.
- 50 instrucțiuni de adunare sau scădere, fiecare consumând 3 perioade de ceas.
- 20 instrucțiuni de înmulțire cu întregi, fiecare consumând 50 perioade de ceas.

Să se calculeze CPI pentru programul de test.

⁴ load/store = încarcă - citește din memorie, respectiv stochează în memorie. Sunt operații de transfer între locații ale memoriei principale și registre interne ale UCP.

R:

$$CPI = \frac{(100 \times 5) + (80 \times 1) + (50 \times 3) + (20 \times 50)}{250} = \frac{1730}{250} = 6.92$$

La calculul CPI, ca valoare medie, este adesea util să se calculeze numărul total de perioade de ceas ale CPU ($NrClk_p$), pentru un program, conform relației:

$$NrClk_p = \sum_{i=1}^n (CPI_i * I_i) \quad (2.8)$$

unde:

I_i - indică de câte ori este executată instrucțiunea i într-un program

CPI_i - numărul mediu de perioade de ceas necesare pentru execuția instrucțiunii i .

Ca urmare timpul de execuție de către UCP al unui program, cu n tipuri de instrucțiuni, poate fi exprimat ca:

$$T_{UCP} = T_{clk} * \sum_{i=1}^n CPI_i * I_i \quad (2.9)$$

Dacă se introduce noțiunea de “frecvență a instrucțiunii” pentru a indica care este procentul reprezentat de instrucțiunea de tip i din numărul total de instrucțiuni ale unui program, expresia termenului CPI total, mediu pentru un program poate fi scrisă și ca:

$$CPI = \frac{NrClk_p}{NrI} = \frac{\sum_{i=1}^n (CPI_i * I_i)}{NrI} \quad (2.10)$$

sau

$$CPI = \sum_{i=1}^n \left(CPI_i * \frac{I_i}{NrI} \right)$$

Ultima formă a expresiei CPI, indică ca mod de calcul înmulțirea fiecărei valori CPI_i cu frecvența de producere a instrucțiunii de tip i într-un program, expresie care se scrie de obicei ca:

$$CPI = \sum_{i=1}^n CPI_i * F_i \quad (2.11)$$

$$\text{unde} \quad F_i = \frac{I_i}{NrI} \quad (2.12)$$

Importanța practică a acestei ultime forme de exprimare a CPI constă în indicarea locului unde proiectantul trebuie să investească mai mult efort și resurse pentru a scădea valoarea CPI și ca urmare să mărească performanțele procesorului. Concluzia poate fi exprimată în două moduri:

- trebuie investit (optimizat) acolo unde se pierde timpul, dacă frecvența de apariție a instrucțiunilor respective este semnificativă în programe.
- trebuie investit în instrucțiunile cu frecvență mare și nu în cele ce se folosesc rar.

Exemplu numeric:

Se va calcula CPI, pentru un program de test ce rulează pe o mașină RISC (arhitectură load / store), conform datelor din tabelul 2.1. și al relațiilor 2.11 și 2.12.

Tabelul 2.1.

Tip operație	CPI _i	Frecvența	CPI _i *F _i	(% Timp)
Operație ALU	1	60%	0.60	39%
Load (citire memorie)	2	22%	0.44	28%
Store (scriere memorie)	2	3%	0.06	4%
Ramificare	3	15%	0.45	29%
Total:		100%		100%
CPI mediu total:			1.55	

2.2. ALTE UNITĂȚI DE MĂSURĂ

În scopul măsurării și comparării performanțelor s-a introdus mărimea **MIPS**⁵ (Milioane de Instrucțiuni Pe Secundă) exprimată ca:

$$MIPS = \frac{NrI}{T_{UCP} \times 10^6} \quad (2.13)$$

sau prin înlocuirea lui T_{UCP} din ecuația (2.6) și simplificare:

$$MIPS = \frac{f_{clk}}{CPI \times 10^6} \quad (2.14)$$

unde $f_{clk} = 1/T_{clk}$ este frecvența oscilatorului de ceas.

Ultima relație (2.14) este mai des folosită, fiind mai ușor de calculat, pentru că depinde de frecvența de ceas și nu de numărul de instrucțiuni din program. De asemenea relația (2.14) pare să indice corect faptul că un procesor cu frecvență de ceas mai mare are o valoare MIPS mai mare. De fapt nu este chiar așa, pentru că frecvența ceasului este inclusă și în relația termenului CPI. Avantajul introducerii parametrului MIPS constă în exprimarea ușoară și în faptul că este deosebit de sugestiv pentru clienții cumpărători de mașini de calcul. Măsura prezintă însă dezavantaje atunci când este folosită pentru a compara diferite mașini, pentru că: [Patterson94]:

- valoarea MIPS este dependentă de setul de instrucțiuni, făcând dificilă și neconcludentă compararea între calculatoare cu seturi diferite de instrucțiuni;

⁵ MIPS = Millions of Instructions per Second

- valoarea MIPS, pentru același calculator, variază în funcție de programele rulate. La programele cu operații complexe valoarea medie CPI este mare;
- valoarea MIPS poate varia uneori invers proporțional cu performanța. Exemplul clasic folosit pentru a demonstra aceasta, se referă la operațiile cu numere reale, reprezentate în virgulă mobilă. Calculatoarele care au (co-)procesor aritmetic și care pot efectua operații în virgulă mobilă, opțional, fie prin hardware (operații cablate, prin intermediul procesorului aritmetic), fie prin software, (prin emularea operațiilor în virgulă mobilă prin unități de prelucrare pe întregi), conduc la valori MIPS invers proporționale cu performanța. În cazul utilizării variantei cablate, numărul de instrucțiuni va fi mai mic, dar o instrucțiune în virgulă mobilă necesită mai multe perioade de ceas (CPI mare) decât una în virgulă fixă (întregi). Cu toate că timpul de execuție al programului ce operează pe numere reale va fi mai scurt decât în cazul folosirii rutinelor de emulare, valoarea lui MIPS va rezulta mică. În varianta de efectuare prin software a operațiilor în virgulă mobilă, se execută mai multe instrucțiuni simple, rezultă un MIPS mai mare, deși timpul total de execuție va fi mai mare.

Exemplu de interpretare greșită:

Se compară două mașini, notate cu A și B. Mașina A recunoaște o instrucțiune specială de înmulțire a întregilor cu semn, care consumă 50 de perioade de ceas pentru a fi executată. Mașina B nu are instrucțiune specială de înmulțire, dar o realizează prin instrucțiuni simple de adunare, deplasare, comparare, fiecare dintre instrucțiunile simple consumând două perioade de ceas. Dacă ne referim doar la operația de înmulțire, iar frecvența de ceas pentru ambele mașini este de 200 MHz, rezultă valorile MIPS astfel:

$$\begin{aligned}\text{Mașina A} &= 200/50 \text{ MIPS} = 4 \text{ MIPS} \\ \text{Mașina B} &= 200/2 = 100 \text{ MIPS}\end{aligned}$$

ceea ce ar indica că mașina B este de 25 ori mai rapidă, deși nu aceasta este realitatea.

Următoarea măsură introdusă a încercat să se refere doar la operațiile în virgulă mobilă: **MFLOPS**⁶ = milioane de operații în virgulă mobilă pe secundă. Bazat pe operații și nu pe instrucțiuni, MFLOPS are intenția să facă o comparație corectă între diferite mașini, cu referire nu la instrucțiuni ci doar la operații în virgulă mobilă. Ideea de la care s-a pornit a fost că același program, rulat pe diferite calculatoare, va executa un număr diferit de instrucțiuni, dar același număr de operații în virgulă mobilă. Relația folosită pentru calcul este:

$$MFLOPS = \frac{\text{Nr. operatii în VM din program}}{T_{UCP} \times 10^6} \quad (2.15)$$

Din păcate nici această măsură nu este pe deplin edificatoare în compararea performanțelor. Operațiile în virgulă mobilă implementate pe diferite mașini nu sunt aceleași. Unele procesoare pot executa, prin implementare cablată, operații în virgulă mobilă de împărțire, radical, sinus etc., iar altele pot recunoaște doar instrucțiuni aritmetice simple ca adunare și scădere cu numere reale. Mai mult, operațiile în virgulă mobilă nu au aceeași complexitate; unele se efectuează mai rapid, altele mai lent. De exemplu, pe aceeași mașină, vom obține o valoare mai mare pentru MFLOPS, dacă

⁶ MFLOPS = Millions FLoating-point Operations Per Second

programul de test are doar adunări în virgulă mobilă, față de cazul când programul de test are operații complexe de împărțire, sinus, radical ș.a.m.d.

2.3. PROGRAME ETALON DE TESTARE A PERFORMANȚELOR

Altă direcție în crearea de măsurători în scopul comparării performanțelor diferitelor mașini este rularea unor *programe etalon de testare a performanțelor (benchmarks)*. Utilizarea programelor “benchmarks” dorește să facă evaluarea și compararea performanțelor mașinilor, prin rularea acelorași programe, analizând în final timpul de execuție.

Programele etalon de testare se pot referi la diferite componente ale sistemului de calcul:

- UCP (Unitatea centrală de procesare)
- aritmetică în virgulă fixă / mobilă
- sistemul de memorie
- sistemul de I/O
- sistemul de operare

Programele de test pot fi împărțite în următoarele categorii [Patterson94]:

1. **Programe reale:** de exemplu compilatoare de C, programe de procesare a textelor, programe de proiectare asistată (Spice). Testarea se face în condiții similare privind intrările, ieșirile și setarea opțiunilor. Aplicațiile reale folosite pentru testarea performanțelor și comparare au dezavantajul că uneori se pot întâlni probleme de portabilitate, mai ales datorită dependenței de sistemul de operare folosit pe calculator.
2. **Aplicații modificate:** adesea aplicațiile reale sunt folosite ca și blocuri componente ale unor programe de testare. Modificările constau în eliminarea unor componente (de exemplu eliminarea componentelor de I/O dacă programul de test este destinat performanțelor UCP) sau prin introducerea unor componente care măresc gradul de portabilitate al programelor. În această categorie sunt incluse și aplicațiile controlate prin fișiere de comenzi (“scripted⁷ applications”). Script-urile sunt folosite pentru a simula programe de aplicație și de asemenea cu scopul de a reproduce o comportare interactivă, de exemplu prin afișări pe ecranul calculatorului, sau pentru a simula interacțiunea multi-user produsă pe un sistem de tip server.
3. **Nuclee (kernels)** din programe reale: s-au extras porțiuni semnificative, numite nuclee, din programe reale, pentru a fi folosite ca rutine de test. Ele sunt utile pentru a indica performanța, referită la caracteristici individuale ale unei mașini, permițând să se explice motivele diferențelor în performanță pentru programele reale. Exemple: Livermore Loops, Linpack.
4. **Toy benchmarks** (programe de test amuzante - jucărie): sunt programe scurte (maxim 100 de linii de cod) care produc un rezultat cunoscut înainte de rulare, folosite doar pentru a testa viteza de execuție. Exemplu Sieve of Erastosthenes, Puzzle, Quicksort. Nu

⁷ Script = un fișier de comenzi. O serie de instrucțiuni care spun unui program cum să execute o anumită procedură, cum ar fi deschiderea unei sesiuni de lucru pe un sistem de poștă electronică.

sunt foarte concludente, acesta fiind un motiv suplimentar pentru care sunt privite ca jucării.

5. **Programe de test sintetice (synthetic benchmarks):** sunt programe artificial create, dar asemănătoare din punctul de vedere al intențiilor cu programele de test de tip “kernels”, încercând să facă testarea pe baza unor operații și operanți ce se consideră a fi caracteristici pentru frecvența medie de apariție a acestora în programele reale. Dintre testele des folosite, amintim *Drystone* și *Whetstone*. Programul de test Dhrystone se bazează pe analiză statistică și este utilizat pentru testarea performanțelor la aritmetica cu numere întregi și pentru modul de transmitere a parametrilor la apelul unor funcții. A fost implementat în limbajele Ada și C. Programul de test raportează un număr de Dhrystones/secundă - o valoare utilă doar în compararea cu alte valori obținute la execuția programului de test. Numele programului Dhrystone s-a dat comparativ cu numele celui alt test sintetic numit Whetstone (traducerea din engleză a numelui *Whetstone* înseamnă *tocilă, piatră de ascuțit*). Faptul că Dhrystone nu folosește operații în virgulă mobilă (în engleză “floating point”, iar “float” înseamnă *a pluti*) adică, dacă “nu plutește” trebuie să fie *uscat* (în engleză *Dry*). Pentru aritmetica în virgulă mobilă se folosește testul Whetstone. Whetstone este o colecție de coduri care testează performanța la rularea unor biblioteci matematice în virgulă mobilă. A fost scris inițial în Algol, dar ulterior a fost implementat în mai multe limbaje de programare de nivel înalt. Programul de test raportează un număr de Whetstones/secundă - o valoare intenționat fără valoare intrinsecă, utilă doar în compararea cu alte valori obținute la execuția programului de test. Sunt utilizate în calcule funcții sinus, cosinus și funcții transcendente, pentru a măsura performanța coprocesorului sau a simulării acestuia de către UCP prin unitatea de prelucrare a întregilor (ALU).

Teste ca Drystone și Whetstone, testează amănunțit doar UCP, dar nu testează performanța celorlalte componente ale sistemului, cum ar fi unitățile de disc, dispozitivul de afișare, etc. De aceea au apărut rutine de test specializate pentru testarea periferiei și a interfețelor cu periferia sistemului de calcul.

În 1988 s-a înființat corporația SPEC (**The Standard Performance Evaluation Corporation**) ca organizație non-profit formată cu scopul de a stabili, menține și garanta seturi standardizate de programe de test (benchmarks) ce pot fi aplicate la noile generații de calculatoare de înaltă performanță. Organizația s-a numit inițial “System Performance Evaluation Cooperative” și ea cuprindea firmele HP, DEC, MIPS și Sun. SPEC⁸ a crescut în timp, incluzând în prezent peste 60 de companii, și a devenit una dintre organizațiile de succes privind standardizarea programelor de test în multe domenii ale sistemelor de calcul, al rețelelor de calculatoare, al sistemelor cu calculator integrat (embedded systems”) și al prelucrării datelor.

Organizația SPEC nu realizează testare de mașini de calcul, ea doar dezvoltă seturi de programe de test și de asemenea analizează și publică rezultatele propuse/obținute de membrii organizației și alte instituții licențiate pentru aceasta. Rezultatele ce se găsesc pe web-site-ul organizației au fost transmise de parteneri. SPEC nu rulează nici un program de test, ci doar dezvoltă programe de test, analizează și publică rezultatele obținute și transmise către SPEC de membrii organizației și alte instituții licențiate pentru aceasta. SPEC nu are control asupra companiilor partenere și nici asupra sistemelor sau configurațiilor pe care s-au făcut testele, dar SPEC analizează și validează doar rezultatele ce apar pe web-site-ul lor.

⁸ vezi <http://www.spec.org>

Consortiul stabilește *setul de programe de test* (doar programe reale, nuclee, sau aplicații modificate) și intrările folosite la rularea acestor programe. Principala caracteristică a programelor de test SPEC este că sunt formate din *seturi (suite)* de aplicații pentru testarea procesoarelor. Avantajul setului de programe de test este dat de faptul că eventuala slăbiciune a uneia dintre aplicațiile de testare poate fi compensată rulând celelalte aplicații din setul etalon de testare (benchmark).

În cazul programelor pentru testarea UCP, există în acest moment cinci generații de seturi de programe de test, pentru operare în virgulă fixă (INT - întregi) și virgulă mobilă (reale, FP - Floating Point): SPEC89, SPEC92, SPEC95, SPEC2000. De exemplu SPEC CPU2000 [SPEC] este un produs software pentru testare realizat de SPEC împreună cu un grup non-profit format din comercianți de calculatoare, integratori de sisteme, universități, organizații de cercetare, edituri și consultanți din întreaga lume. A fost proiectat pentru a face măsurători de performanță ce se pot utiliza în compararea sarcinilor intense de calcul pe diferite sisteme. SPEC CPU2000 conține două serii de programe de test: CINT2000 pentru măsurarea și compararea performanțelor sarcinilor intense de calcul cu întregi și CFP2000 pentru măsurarea performanțelor sarcinilor intense de calcul în virgulă mobilă. Cele două componente măsoară performanța pentru procesorul calculatorului, pentru arhitectura memoriei și pentru compilator.

SPEC cuprinde în prezent trei grupuri de lucru, fiecare lucrând la elaborarea propriilor seturi de programe de test:

1. **Open Systems Group (OSG):** elaborează programe de test la nivel de componente și de sistem pentru sisteme de operare UNIX / NT și pentru operații în virgulă mobilă. Există subcomitete de lucru pentru testare: UCP (testare procesor), Java (programe de test la nivel client și nivel server), Mail (programe pentru testare utilitare E-mail și performanțe la ISP⁹), testare servere de fișiere, testare pentru sisteme UNIX multi-utilizator, WEB (programe testare pentru web server).
2. **High Performance Group (HPG):** Grupul elaborează și întreține programe de test pentru aplicații de calcul de înaltă performanță, pentru evaluare a performanțelor de calcul inter-platforme. Se adresează sistemelor cu arhitectură de înaltă performanță cum ar fi: sisteme multiprocesor simetrice, grupuri integrate ("clusters") de stații de lucru, sisteme paralele cu memorie distribuită și supercalculatoare tradiționale vectoriale și paralel vectoriale.
3. **Graphics Performance Characterization Group (GPCG):** Elaborează programe de test pentru subsisteme grafice, pentru OpenGL și pentru Xwindows. Cuprinde de asemenea mai multe grupuri, pentru elaborare de programe standard de test în domenii ca: aplicații cu grafică intensivă, multi-media (în special pe codare, recunoaștere sunete, sunete 3D, multi-media pentru jocuri și timp liber).

Un alt set de programe de test standardizate sunt programele de test TP ("Transaction-Processing"). Acestea măsoară capacitățile unui sistem în realizarea tranzacțiilor, tranzacții ce constau în accesări și actualizări de baze de date. De exemplu, sisteme simple TP sunt: un sistem de rezervare de bilete de avion, sau un bancomat (ATM). În anii '80 s-a creat o organizație non-profit numită Transaction Processing Council (TPC) cu scopul de a crea un set de programe de test TP de bună calitate. Primul program de test a apărut în 1985 și s-a numit TPC-A. Au apărut multe alte

⁹ ISP = Internet Service Provider

variante de programe de test, inclusiv TPC-W pentru testarea performanțelor tranzacțiilor pe bază de Web. Programele de test TPC sunt descrise la adresa www.tpc.org/.

În domeniul sistemelor cu calculator integrat, (Embedded Systems) care au căpătat o extrem de mare răspândire în ultimul deceniu, s-au dezvoltat de asemenea seturi de programe de test. Acestea au apărut în cadrul organizației EDN Embedded Microprocessor Benchmark Consortium (EEMBC - pronunțat “embassy”). Există cinci tipuri de seturi de programe de test în funcție de domeniile de aplicație:

- auto și industrial
- bunuri de consum
- interconectare
- automatizări de birou
- telecomunicații

În plus față de programele etalon de test pomenite mai sus, există o sumedenie de programe de testare realizate prin cooperare universitară, sau realizate de către marile periodice din domeniul calculatoarelor.

2.4. COMPARAREA PERFORMANTELOR PRIN MEDIERE

Modul de comparare a performanțelor diferitelor mașini de calcul a fost și va fi un permanent subiect de discuție pentru cercetători. Aceasta pentru că există nu numai interese financiare și de carieră la mijloc, dar și din cauză că puțini vor putea spune întotdeauna și fără nici o urmă de îndoială, că o mașină testată este mai bună decât alta. Trebuie luate în considerare foarte multe elemente ale sistemului de calcul, dar în final, cel mai bun calculator este cel care prezintă cel mai bun raport performanță / preț. Dar performanța este strâns legată de aplicațiile ce vor fi rulate pe mașina respectivă.

Pentru compararea mai multor mașini, cea mai simplă metodă este să se ruleze același program pe fiecare dintre mașini și să se determine timpul de execuție. Dar dacă una dintre mașini are performanțele cele mai bune pentru un program de test rulat, iar dacă se alege un alt program de test are rezultate mai slabe decât competitorii? Pentru a evita o asemenea situație se face testarea prin rularea mai multor programe diferite pe fiecare din mașinile testate și apoi să se ia o decizie cu privire la performanțe.

Cel mai simplu mod de comparare consideră fiecare program și fiecare mașină în parte, ca în exemplul următor:

Exemplul 1 privind compararea performanțelor

Se dau trei mașini notate A, B și C și trei programe de test notate P1, P2 și P3
După rulare se obțin timpuri din tabelul de mai jos:

Tabelul 2.2.

	Calculator A	Calculator B	Calculator C
Program P1 (secunde)	5	10	15
Program P2 (secunde)	900	100	20
Program P3 (secunde)	200	300	30
Timp total (secunde)	1105	410	65

Putem observa că:

- A este de 2 ori mai rapidă decât B pentru programul P1
- B este de 1,5 ori mai rapidă decât C pentru programul P2
- A este de 3 ori mai rapidă decât C pentru programul P1
- C este de 45 ori mai rapidă decât A pentru programul P2
- s.a.m.d.

Dorim însă să obținem un indicator, o măsură sintetică pentru fiecare mașină, indiferent de numărul de programe de test rulare. Vom începe cu valoarea timpului total de execuție pentru fiecare mașină (așa cum apare în tabel). Vom putea scrie:

- B este de 2,7 ori mai rapidă decât A pentru programele P1, P2 și P3 (din 1105/410)
- C este de 6,31 ori mai rapidă decât B pentru programele P1, P2 și P3 (din 410/65)
- C este de 17 ori mai rapidă decât A pentru programele P1, P2 și P3 (din 1105/65)

Se pare că deja avem un câștigător: mașina C. Dar dacă sunt mai multe mașini, sistemul acesta de comparare este extrem de dificil. Dacă se rulează mai multe programe de test pe o mașină și se obțin diverse măsuri ale performanței pentru fiecare program (de exemplu timp, MIPS, sau raport SPEC), vom dori să utilizăm valorile obținute pentru a calcula o viteză medie de execuție pentru mașină și nu un total al timpului. La început am fi tentați să facem pur și simplu media aritmetică a timpilor / vitezelor de execuție pe acea mașină.

Măsura 1: Media aritmetică

Dacă sarcina de lucru a constat în rularea programelor P1, P2 și P3 de un număr egal de ori, declarațiile de mai sus indică timpul de execuție relativ pentru sarcina de lucru pe fiecare mașină. O medie a timpului de execuție care urmărește timpul total de execuție este media aritmetică:

$$\frac{1}{n} \sum_{i=1}^n Timp_i \quad (2.16)$$

unde timpul $Timp_i$ este timpul de execuție pentru programul i din numărul n total de sarcini de lucru.

Pentru același exemplu numeric, mediile aritmetice sunt trecute în ultima linie a tabelului 2.3. Media aritmetică este simplu de calculat și ușor de înțeles și comparat. Compararea este însă corectă doar dacă programele de test sunt rulate în mod egal în sarcina de lucru, așa cum s-a presupus prin realizarea mediei aritmetice. În caz contrar va trebui calculată o medie ponderată. Acest lucru este necesar pentru că valoarea obținută pentru media aritmetică nu este proporțională cu timpul de execuție pentru întregul set de programe de test - ea este doar media timpilor de execuție obținuți individual.

Tabelul 2.3.

	Calculator A	Calculator B	Calculator C
Program P1 (secunde)	5	10	15
Program P2 (secunde)	900	100	20
Program P3 (secunde)	200	300	30
Media aritmetică (secunde)	368.33	136.67	21.67

Dacă toate programele de test ar fi procesat același tip și același număr de valori, am putea să calculăm un timp general (respectiv o performanță de execuție generală) din timpul total necesar pentru setul de programe de test. Dar pentru că setul de programe cuprinde în general o mare diversitate de programe, este nevoie de o altă abordare. Pentru programele de test SPEC se utilizează media geometrică calculată ca:

Măsura 2: Media aritmetică ponderată

Dacă programele P1, P2 și P3 nu sunt rulate în mod egal în sarcina de lucru de testare a performanțelor, există două abordări pentru rezumarea performanței.

Prima abordare atunci când există o combinație inegală de programe în sarcina de lucru este să se atribuie factori de ponderare w_i pentru fiecare program, pentru a indica frecvența relativă a programului în acea sarcină de lucru. Dacă, de exemplu, 20% din sarcină a constat în programul P1 și 70% din sarcină a fost programul P2, atunci factorii de ponderare vor fi 0,2 pentru P1, 0,7 pentru P2 și 0,1 pentru P3 (ei însumați trebuie să fie 1). Rezultă media aritmetică ponderată:

$$\sum_{i=1}^n Pondere_i \times Timp_i \quad \sum_{i=1}^n Pondere_i = 1 \quad (2.17)$$

unde $Pondere_i$ este frecvența programului i în sarcina de lucru iar $Timp_i$ este timpul de execuție al acelui program. În tabelul 2.4. se indică trei rânduri de ponderi pentru același exemplu numeric din tabelul 2.2.

Tabelul 2.4.

	Program A	Program B	Program C	Observații
Program P1 (secunde)	5	10	15	
Program P2 (secunde)	900	100	20	
Program P3 (secunde)	200	300	30	
Set 1 de ponderi	0,333	0,333	0,333	
Media aritmetică pentru set 1	368.33	136.67	21.67	media aritmetică neponderată
Set 2 de ponderi	0,960	0,039	0,001	
Media aritmetică pentru set 2	40.10	13.80	15.21	
Set 3 de ponderi	0,959	0,001	0,040	
Media aritmetică pentru set 3	13.70	21.69	15.61	

Se observă că media ponderată pentru primul set de ponderi este identică cu media aritmetică, pentru că s-au ales ponderi egale pentru toate cele trei programe. Ponderile au fost în așa fel modificate pentru ca la fiecare set să se obțină valoarea mediei minime pentru altă mașină.

Efectul vizibil al ponderării este de “normalizare” a ponderilor la timpii de execuție ai programelor ce rulează pe acele mașini.

A doua abordare în cazul unei combinații neuniforme de programe într-o sarcină de testare este să se normalizeze timpii de execuție față de o mașină de referință și să se folosească media timpilor de execuție normalizați. Aceasta este și abordarea folosită de seturile de programe de test de la SPEC unde se folosește o mașină de referință de tip SPARCstation 10. Această măsură sugerează că performanța noilor programe poate fi prezisă simplu prin înmulțirea numărului rezultat cu performanța sa pe mașina de referință.

Timpul de execuție normalizat se poate exprima fie ca o medie aritmetică, fie ca o medie geometrică. Media geometrică se poate calcula cu:

$$\sqrt[n]{\prod_{i=1}^n Raport_Timp_Executie_i} \quad (2.18)$$

unde $Raport_Timp_Executie_i$ este timpul de execuție normalizat (raportat) la mașina de referință, pentru programul cu numărul i din numărul total de n ale sarcinii de lucru. Raportul se poate calcula ușor, pentru că media geometrică are o proprietate avantajoasă pentru două eșantioane A_i și B_i , astfel că putem face fie raportul mediilor, fie media rapoartelor:

$$\frac{Medie_Geometrica(A_i)}{Medie_Geometrica(B_i)} = Medie_Geometrica\left(\frac{A_i}{B_i}\right) \quad (2.19)$$

Valorile numerice pentru exemplul inițial din tabelul 2.2, împreună cu mediile aritmetice normalizate și mediile geometrice normalizate sunt prezentate în tabelul 2.5. Se observă că spre deosebire de media aritmetică, unde ierarhizarea performanțelor mașinilor este diferită, performanțele mașinilor pe baza valorilor obținute la media geometrică sunt consecvente, indiferent de mașina folosită ca referință (s-au întărit rezultatele numerice care indică timp minim, performanță maximă). În tabelul 2.5. se indică câteva variante de medii aritmetice și geometrice normalizate, pentru a indica faptul că media aritmetică nu este indicat a fi folosită pentru timpi de execuție normalizați.

Media geometrică a timpilor de execuție normalizați este independentă de timpii de rulare pentru programele individuale, și nu are importanță care mașină este luată ca referință pentru normalizare. Principalul dezavantaj al mediilor geometrice a timpilor de execuție normalizați este că ele nu prezic un timp de execuție ci indică doar performanțele mai bune sau mai rele comparativ cu referința.

Tabelul 2.5.

	Normalizat la A			Normalizat la B			Normalizat la C		
Program P1	1.00	2.00	3.00	0.50	1.00	1.50	0.33	0.67	1.00
Program P2	1.00	0.11	0.02	9.00	1.00	0.20	45.00	5.00	1.00
Program P3	1.00	1.50	0.15	0.67	1.00	0.10	6.67	10.00	1.00
Medie aritmetică	1.00	1.20	1.06	3.39	1.00	0.60	17.33	5.22	1.00
Medie geometrică	1.00	0.69	0.22	1.44	1.00	0.31	4.64	3.22	1.00
Timp total	1.00	0.37	0.06	2.70	1.00	0.16	17.00	6.31	1.00

2.5. LEGEA LUI AMDAHL

Câștigul în performanță ce se poate obține prin îmbunătățirea unei componente a unui calculator poate fi calculat cu ajutorul Legii lui Amdahl. Legea stabilește valoarea creșterii globale a vitezei sistemului în condițiile utilizării unei metode de creștere a vitezei doar pentru una din componentele sistemului, creșterea globală depinzând de fracțiunea de timp cât acea componentă este utilizată. Legea permite calculul valorii numerice a creșterii în viteză a sistemului pe baza îmbunătățirii unei caracteristici particulare (“speedup¹⁰”, ca raport între timpul consumat înainte de îmbunătățire și timpul consumat după îmbunătățire).

În legătură cu legea lui Amdahl trebuie observat că:

- dacă se crește viteza de execuție de n ori doar pentru o fracțiune F din timpul de execuție al unui program nu înseamnă că am crescut viteza de execuție a programului de n ori.
- dacă se știe frecvența cu care se utilizează o anumită îmbunătățire locală dintr-un sistem, se poate calcula cu cât crește viteza întregului sistem datorită îmbunătățirii. Îmbunătățirea locală pentru o UCP se referă de exemplu la: dublarea frecvenței de ceas, reducerea la jumătate a timpului mediu de acces la memoria principală, dublarea vitezei de lucru la circuitul înmulțitor, etc.
- dacă se dorește o creștere globală cât mai mare a performanțelor unui sistem trebuie în primul rând crescute performanțele subsistemelor utilizate în majoritatea timpului de execuție (“alege cazul comun!”).

Pentru a exprima analitic legea lui Amdahl vom nota cu T_{nou} timpul de execuție consumat după îmbunătățirea unei componente și cu T_{vechi} timpul de execuție înainte de îmbunătățire, raportul lor reprezentând creșterea în viteză a sistemului.

$$Speedup = \frac{T_{vechi}}{T_{nou}} = \frac{Performanta_{noua}}{Performanta_{veche}} \quad (2.20)$$

adică

$$Speedup = \frac{T_{vechi}}{T_{nou}} = \frac{n}{n - nF + F} \quad (2.21)$$

sau expresia finală:

$$Speedup = \frac{1}{(1 - F) + \frac{F}{n}} \quad (2.22)$$

unde

T_{nou} = timpul de execuție consumat după îmbunătățirea de n ori a unei componente

T_{vechi} = timpul de execuție înainte de îmbunătățire

F = fracțiunea de timp cât este utilizată componenta îmbunătățită

n = de câte ori au fost crescute performanțele unei componente.

Relația legii lui Amdahl, conform ecuației (2.22) poate fi dedusă pe baza unui exemplu, pe care-l prezentăm în continuare. (vezi și figura 2.1.). Se presupune că un sistem execută un program de test

¹⁰ speedup (engl.) = creștere în viteză.

în 100 de secunde. Dacă se crește de 10 ori viteza unei componente a sistemului, iar această componentă este utilizată doar 20% din timpul total de execuție al programului, să se determine cu cât crește viteza întregului sistem.

Pentru timpul de rulare înainte de îmbunătățirea unei componente putem scrie

$$T_{vechi} = t_1 + t_2 \quad (2.23)$$

în care $t_1 = F \times T_{vechi}$, $t_2 = (1 - F) \times T_{vechi}$, iar F = fracțiunea de timp cât este utilizată componenta ce va fi îmbunătățită. Pentru exemplul numeric din figura 2.1., $F = 20\%$.

După realizarea îmbunătățirii, prin creșterea vitezei componentei de $n = 10$ ori putem scrie:

$$\begin{aligned} T_{nou} &= t_{11} + t_2 \\ &= \frac{t_1}{n} + t_2 \end{aligned} \quad (2.24)$$

sau

$$T_{nou} = \frac{F \times T_{vechi}}{n} + (1 - F) \times T_{vechi} \quad (2.25)$$

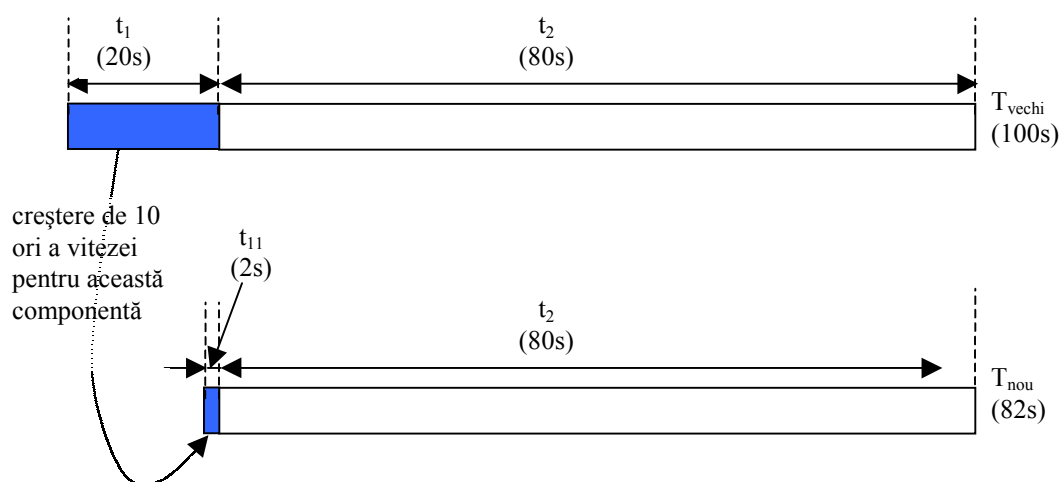


Figura 2.1. Exemplificare, pentru justificarea expresiei analitice a Legii lui Amdhal.

și aranjată pentru comparație cu ecuația (2.22):

$$\frac{T_{nou}}{T_{vechi}} = \frac{F}{n} + (1 - F) \quad (2.26)$$

Exemple numerice:**Exemplul 1.**

Un program de test rulează timp de 200 milisecunde pe un sistem uni-procesor. Doar 20% din program poate fi paralelizat pentru a rula pe un sistem multiprocesor. Presupunând că se utilizează un multiprocesor cu 10 procesoare să se calculeze care este creșterea în viteză și timpul de execuție al programului de test.

$$\begin{aligned} \text{R: } F &= 20\% = 0.2 \\ n &= 10 \end{aligned}$$

$$Speedup = \frac{T_{vechi}}{T_{nou}} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.8 + 0.02} = \frac{1}{0.82} = 1.22$$

$$T_{nou} = \frac{T_{vechi}}{1.22} = 163.9 \text{ milisecunde}$$

Exemplul 2.

La un procesor unitatea aritmetică în virgulă mobilă este îmbunătățită, astfel că rulează de 8 ori mai repede. Calculați creșterea în viteză a procesorului, pentru un program la care doar 20% din instrucțiuni folosesc operanzi în virgulă mobilă.

$$\begin{aligned} \text{R: } F &= 20\% \\ n &= 8 \end{aligned}$$

$$Speedup = \frac{1}{(1 - 0.2) + \frac{0.2}{8}} = 1.212$$

CAPITOLUL 3

FUNDAMENTE PRIVIND ARHITECTURA

UNITĂȚII CENTRALE DE PROCESARE (UCP)

Conținut:

- 3.1. Structura de procesor
- 3.2. Structura unei Unități Centrale de Procesare elementare
- 3.3. Perfecționarea Unității Centrale de Procesare elementare
- 3.4. Alte registre interne semnificative pentru UCP
- 3.5. Exemple privind operațiile UCP
- 3.6. Semnale la interfața UCP cu exteriorul
 - 3.6.1. Magistrala de adrese
 - 3.6.2. Magistrala de date
 - 3.6.3. Magistrala de control
- 3.7. Reprezentarea numerelor în calculator
 - 3.7.1. Reprezentarea cu virgulă fixă
 - 3.7.2. Coduri binare folosite pentru reprezentarea numerelor cu semn
- 3.8. Reprezentarea numerelor în virgulă mobilă

Obiective educaționale:

- a. Descrierea și analiza funcționării pentru o structură de procesor de uz general, evidențierea căilor de date și de control, enumerarea ciclurilor caracteristice de funcționare ale microprocesorului;
- b. Descrierea și exemplificarea modului de aducere din memoria principală și de execuție a instrucțiunilor;
- c. Clasificarea, analiza și exemplificarea unor direcții de îmbunătățire a structurii și performanțelor procesorului;
- d. Explicarea rolul registrelor specifice arhitecturii microprocesoarelor cu arhitectură von Neumann;
- e. Clasificarea și explicarea rolului principalelor semnale de la interfața microprocesorului cu exteriorul.
- f. Enumerarea și clasificarea metodelor de codificare a numerelor în calculatoare.

3.1. STRUCTURA DE PROCESOR

Unitatea Centrală de Prelucrare (UCP) a calculatorului numeric este o structură de procesor de uz general, cu set de instrucțiuni. UCP este un procesor *de uz general*, spre deosebire de alte procesoare cu set de instrucțiuni, cum ar fi procesoarele de I/O, sau procesoarele aritmetice, care îndeplinesc doar funcții limitate, specifice. De asemenea UCP este un procesor *cu set de instrucțiuni*, pentru că UCP recunoaște și execută un set specific de instrucțiuni, furnizate din exteriorul său, sub formă binară.

În cadrul unui sistem de calcul, UCP are responsabilitatea generală de *interpretare* și *execuție* a instrucțiunilor unui program.

În accepțiunea clasică (von Neumann) a unui sistem de calcul, mașina conține o singură Unitate Centrală de Prelucrare. Un astfel de calculator este numit *uniprocessor*, pentru a-l deosebi de calculatoarele *multiprocessor*, care conțin două sau mai multe UCP.

Cele mai importante considerații privind arhitectura și organizarea unei UCP se referă la următoarele:

- Tipurile de instrucțiuni, ce formează setul de instrucțiuni al UCP;
- Organizarea la nivel de registru a unității aritmetice și logice;
- Organizarea la nivel de registru a unității de control a UCP;
- Modul în care UCP comunică cu dispozitivele externe.

Ca structură generală, un procesor cuprinde o unitate de prelucrare a datelor (care conține o unitate aritmetică și logică și registrele folosite ca memorie locală) și o unitate de control. Conform modalității de clasificare a circuitelor digitale în ordine, fiecare dintre cele două componente amintite se constituie ca un automat cu ordinul cel puțin egal cu 2. Structura procesorului, presupune o buclă de reacție între cele două automate componente, deci procesorul este, conform clasificării în ordine descrisă în capitolul 1, un automat de ordinul 3.

Unitatea Aritmetică și Logică (ALU), efectuează operații cu datele de intrare (operanzi) și conține pe lângă structurile combinaționale pentru efectuarea acestor operații, circuite de tip registru pentru memorarea locală, sau prelucrarea datelor și circuite necesare transferului de informație între registre și cu exteriorul ALU (circuite de decodificare, codificare, multiplexare, comparare, circuite tampon, etc.). Registrele locale folosite ca memorie de mare viteză, pot fi folosite și pentru efectuarea unor operații aritmetice sau logice. Principalele operații realizate de Unitatea Aritmetică și Logică (ALU) sunt:

- operații aritmetice și logice cu operanzi de intrare codificați pe n biți. Valoarea lui n este egală de obicei cu numărul de biți ai magistralei interne de date a procesorului, dar există și excepții de la aceasta regulă.
- furnizează indicații privind transportul, împrumutul, sau alte caracteristici privind rezultatul operațiilor efectuate, prin intermediul unor valori binare numite indicatori de condiții (FI = flags).

În figura 3.1. datele de intrare în unitatea de prelucrare a datelor sunt intrările pentru operanzii citiți din exterior. Acești operanzi se citesc ca și cuvinte binare (deci cu n biți în paralel) și se stochează în registre temporare, interne unității de prelucrare a datelor. Rezultatul operațiilor efectuate, sau conținutul unor registre de stocare temporară, se poate furniza în exterior prin liniile notate "date ieșire".

Intrările în unitatea de prelucrare notate "comenzi interne procesorului", reprezintă comenzi către ALU generate de unitatea de control. Conform acestor comenzi ALU va prelucra datele de la intrare și va furniza datele la ieșire (registre interne sau externe). Aceste comenzi (în formă binară) pot reprezenta:

- codul operației pe care trebuie să o execute ALU;
- informații de adresare a registrelor interne unității de prelucrare, (de exemplu pentru a se indica registrele sursă și / sau destinație ale operațiilor efectuate de ALU, pentru a se selecta / valida registre tampon de memorare a valorilor binare de la "Intrare date " sau la "Ieșire date ");
- semnale de sincronizare (destinată logicii secvențiale interne a unității de prelucrare); aceste semnale indică momentele de timp când se face transferul între registre, respectiv momentele de încărcare a unor registre.

Indicatorii de condiții sunt poziționați de unitatea de prelucrare în conformitate cu "evenimentele" ce s-au produs la efectuarea operației comandate anterior pentru ALU. Acești indicatori (numiți și fanioane) sunt memorați de obicei în bistabile, incluse ca celule ale unor registre de indicatori, registre interne UCP. Indicatorii de condiții pot reprezenta, de exemplu, semnul rezultatului, rezultat zero, paritatea rezultatului etc.

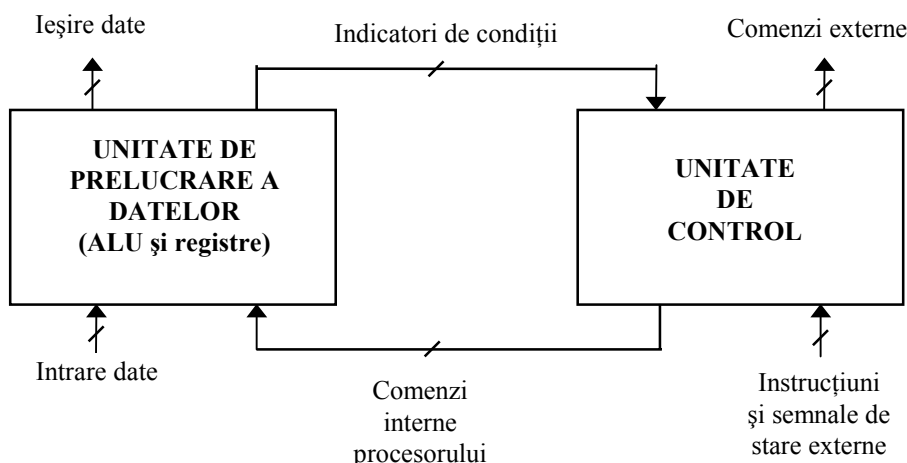


Figura 3.1. Structura de procesor, ca automat de ordinul 3. (liniile oblice pe căile de intrare și ieșire a semnalelor indică existența mai multor linii de semnal, simbolizate prin linie simplă în figură)

Succesiunea de comenzi primită de ALU este generată de către un automat cu algoritm de stare încorporat, numit *Unitate de Control* (UC). Algoritmul de stare definește succesiunea stărilor automatului, respectiv succesiunea și natura comenzilor. Algoritmul de stare poate fi implementat fie în structura fizică (hardware) a UC, fie prin înscrierea comenzilor binare secvențiale într-o memorie de control de mare viteză. În primul caz vorbim de unitate de *control cablată*, iar în al doilea caz de unitate de control *microprogramată*.

*Sistemul digital format dintr-un automat de prelucrare aritmetică și logică și o unitate de control este numit **procesor**.*

Datorită reacției pe care UC o primește de la Unitatea de prelucrare, UC examinează caracteristicile rezultatelor (prin testarea valorii indicatorilor de condiții) și astfel poate modifica

secvența de comenzi în mod corespunzător. UC primește din exterior "instrucțiuni", fiecare din acestea producând generarea unei secvențe de comenzi elementare către unitatea de prelucrare și către exteriorul procesorului. Aceste comenzi controlează succesiunea micro-operațiilor pe care le efectuează procesorul pentru a *executa* operația ce corespunde instrucțiunii primite. Instrucțiunile se păstrează într-o memorie externă procesorului. În cazul în care procesorul este "creierul" unui calculator numeric, el trebuie să decidă și succesiunea în care instrucțiunile vor fi aduse din memoria operativă (funcția de *secvențiere*). Astfel că UC trebuie să genereze în exterior și comenzi (informații) de adresare a instrucțiunilor în memorie. Pentru ca transferul instrucțiunilor binare să se facă corect, alte comenzi de la UC vor activa și circuitele ce comandă liniile de legătură între intrările UCP și ieșirile circuitelor de memorie. Ordinea și momentele de timp la care se generează comenzile externe poate fi influențată de semnalele de stare externe pe care le primește UC (de exemplu, semnale care indică unității de control că un circuit comandat nu poate răspunde în timp la aceste comenzi, deci solicită o întârziere). De aceea, se poate spune că unitatea de control poate fi împărțită funcțional, în două sub-unități de control: prima sub-unitate face controlul la nivel de micro-operații, iar cea de-a doua face control la nivel de secvențiere a instrucțiunilor din memoria externă (automat de adresare).

Interacțiunea unității de control cu dispozitive externe necesită "sincronizarea" acțiunilor. **Sincronizarea** operațiilor UC (și implicit a UCP) cu exteriorul poate fi clasificată în *trei niveluri de sincronizare*:

1. nivelul elementar al semnalului de *ceas* aplicat din exteriorul UCP;
2. nivelul semnalelor de sincronizare și control pentru *transferurile de date* cu memoria principală și porturile de I/O;
3. nivelul semnalelor de *întrerupere* pentru sincronizarea UCP cu evenimente externe.

Funcționarea procesorului este ciclică, conform schemei bloc din figura 3.2, până la terminarea execuției tuturor instrucțiunilor programului.

În prima fază unitatea de control adresează și controlează aducerea unei instrucțiuni din memoria principală. Apoi UC decodifică codul instrucțiunii și generează secvența de comenzi specifică instrucțiunii către unitatea de prelucrare și către exterior (de exemplu pentru citirea operanzilor). În faza de execuție ALU execută comenzile (realizând operații aritmetice sau logice asupra operanzilor) și setează indicatori de condiții, care sunt citiți de UC. Stocarea rezultatului prelucrării se poate face în registre interne ale ALU, sau în registre externe UCP (de exemplu memorie, sau port de intrare / ieșire). Dacă mai sunt instrucțiuni în program unitatea de control revine la prima fază, de aducere a instrucțiunii următoare (ciclul de fetch).

Funcția principală a unui procesor de uz general, ca Unitate Centrală de Procesare (UCP) a calculatorului, este să execute secvența de instrucțiuni stocată într-o memorie externă (numită **memoria principală (MP)** a calculatorului). Secvența de operații implicate în aducerea și execuția unei instrucțiuni, descrisă în figura 3.2, se produce într-un interval de timp, multiplu al perioadei impulsurilor de ceas, interval numit **ciclu instrucțiune**.

Comportarea UCP în timpul ciclului instrucțiune e definită printr-o secvență de micro-operații, fiecare din acestea implicând operații de transfer între registre. Timpul t_{UCP} cerut pentru cea mai scurtă și bine definită micro-operație a UCP este numit **timp de ciclu al UCP** (sau **stare a UCP**, corespunzătoare unei stări a automatului UC) și este unitatea de timp de bază pentru măsurarea tuturor acțiunilor UCP. Inversul acestui timp este frecvența de ceas (clock) la care funcționează UCP, măsurată în MHz sau GHz.

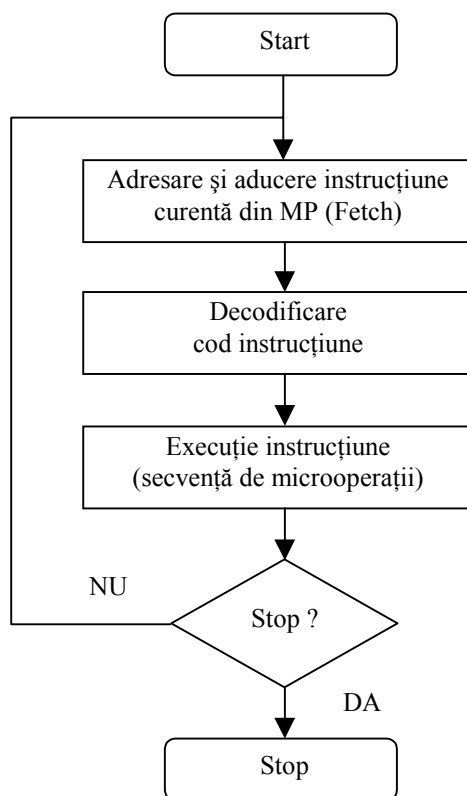


Figura 3.2. Schema bloc care indică funcționarea ciclică a procesorului. (MP = memoria principală)

Această frecvență este dependentă direct de arhitectura și de tehnologia de fabricație a UCP. În cele mai multe dintre (micro-)procesoarele implementate frecvența la care funcționează UCP nu este identică cu frecvența oscilatorului de ceas cu cuarț, care fixează baza de timp a microprocesorului. Frecvența de lucru a UCP este un multiplu, sau submultiplu al frecvenței oscilatorului de ceas.

În plus, pentru a executa programele stocate în memoria principală, UCP controlează celelalte componente ale sistemului, prin linii de comandă și stare speciale. De exemplu, UCP controlează direct sau indirect operațiile de I/O, cum ar fi transferul datelor între dispozitivele de I/O și memoria principală (MP).

De multe ori operațiile cu porturile de I/O cer rar intervenția UCP și de aceea este mai eficient pentru UCP să ignore dispozitivele de I/O până când acestea cer servicii de la UCP. Asemenea cereri sunt numite **întreruperi**. Dacă apare o cerere de întrerupere, UCP poate suspenda execuția programului executat pentru a trece temporar controlul la o rutină de tratare a cererii de întrerupere. Un test al prezentei cererilor de întrerupere se face la sfârșitul fiecărui ciclu instrucțiune.

În cadrul unui calculator pot însă să existe și cereri simultane de acces la un anume dispozitiv sau la o magistrală. Motivul principal îl constituie comunicarea, în general asincronă (în sensul că diferitele dispozitive nu pot fi sincronizate direct de un semnal comun de ceas), dintre procesor și alte dispozitive. Comunicarea asincronă este dictată de mai multe cauze:

- Există un mare grad de independență între componentele sistemului. De exemplu, într-un calculator, UCP și procesoarele de intrare-ieșire (IOP) pot executa programe diferite, ceea ce înseamnă că cele două componente interacționează relativ rar și la momente de timp nepredictibile.

- Viteza de funcționare a componentelor fizice ale calculatorului variază în limite largi. De exemplu UCP poate lucra la viteze de zeci de ori mai mari față de dispozitivele de memorie principală, în timp ce viteza MP poate fi cu câteva ordine de mărime mai mare ca viteza dispozitivelor de I/O.
- Distanța fizică între componente poate fi prea mare pentru a permite transmitia sincronă a informației între ele.

Operațiile efectuate într-un ciclu instrucțiune pot fi împărțite în operații intermediare de tip: aducere cod instrucțiune (citire memorie), aducere operand (citire memorie sau registru I/O), scriere memorie sau registru I/O, etc. Ciclul ce corespunde desfășurării în timp a acestor operații intermediare va fi numit **ciclu mașină (CM)**. El cuprinde deci suma activităților ce rezolvă o operație intermediară bine definită, cu o finalitate clară. Primul ciclu mașină al oricărui ciclu instrucțiune este ciclul de **fetch** (extragere-aducere a codului operației).

Concluzionând putem spune că un ciclu instrucțiune (CI) cuprinde mai multe cicluri mașină, fiecare ciclu mașină desfășurându-se de-a lungul mai multor stări ale UCP.

Din punctul de vedere al liniilor de legătură cu exteriorul UCP (blocul procesor a fost redesenat în figura 3.3.), de obicei liniile de intrare date (D_{in}) și ieșire date (D_{out}) se unifică în linii comune de date de intrare-ieșire ce sunt legate la o magistrală (bus) externă bidirecțională (fig. 3.3b).

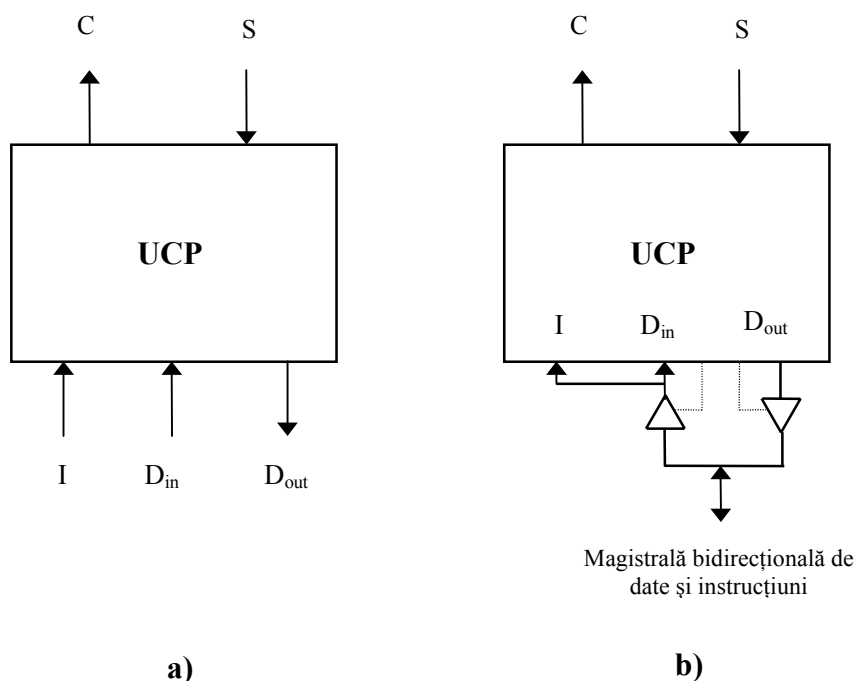


Figura 3.3. Blocul procesor, cu indicarea legăturilor externe. D_{in} - date de intrare, D_{out} - date de ieșire, C - comenzi externe-inclusiv adrese de MP, I - instrucțiuni, S - semnale de stare din exterior

În acest caz UC trebuie să comande (liniile punctate) și sensul de transfer a datelor prin intermediul unor circuite tampon (buffer) de interfață pentru a elimina conflictele electrice pe liniile bidirecționale (circuitul are posibilitatea invalidării - blocării ieșirilor pe baza circuitelor de tip TSL - trei stări logice, starea blocat corespunzând unei înalte impedanțe (HiZ) la ieșire).

La majoritatea procesoarelor pe magistrala bidirecțională de date se aduc și instrucțiunile (I), reducând astfel numărul de conexiuni către exterior. Este clar, că din punctul de vedere al vitezei de funcționare, o organizare cu magistrale separate pentru date și instrucțiuni este mai rapidă. Acest ultim mod de organizare corespunde așa-numitei *arhitecturi Harvard* a procesorului. Arhitecturile Harvard construite în prezent au o magistrală externă unică de date și instrucțiuni, dar magistrale

interne (și circuite de memorie tampon locale) separate pentru date și instrucțiuni.

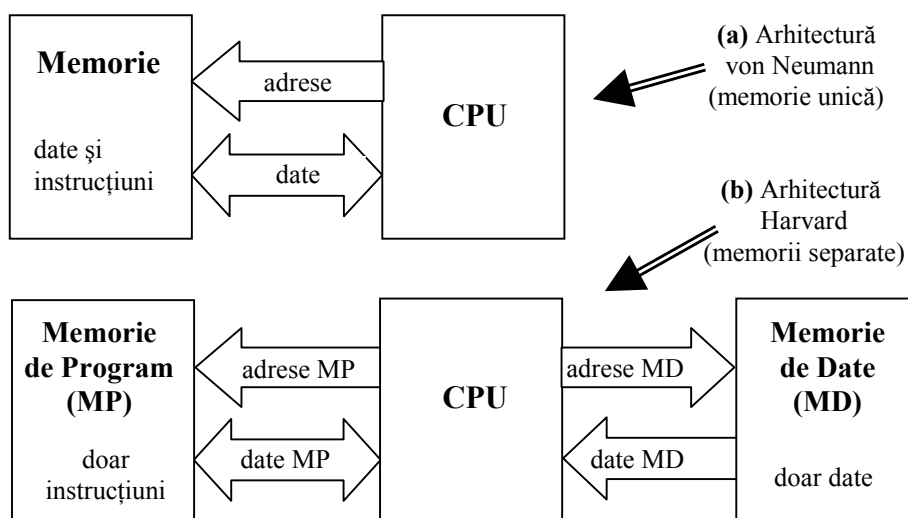


Figura 3.4. Diagrame bloc pentru exemplificarea deosebirilor dintre arhitecturile: (a) von Neumann și (b) Harvard.

Magistrala unică de date și instrucțiuni este numită magistrală de date bidirecțională și ea oferă anumite avantaje sistemului, din punctul de vedere al flexibilității modului de lucru cu exteriorul. Câteva din aceste avantaje ale unei magistrale unice de date și instrucțiuni sunt:

- se folosește o memorie unică pentru stocarea datelor și instrucțiunilor (memoria principală, externă procesorului). Rezultă simplificarea UCP din punctul de vedere al automatului de adresare a memoriei și al numărului de registre implicate.
- folosirea unei memorii unice, unde se stochează atât date cât și instrucțiuni crește flexibilitatea sistemului, pentru că nu există restricții privind adresele de stocare a datelor și instrucțiunilor în memoria principală.

Acest mod de organizare are și dezavantaje, din punctul de vedere al vitezei de lucru și din punctul de vedere al unei relative creșteri în complexitate a unității de control, care trebuie să interpreteze în mod corect cuvintele citite din memorie ca fiind date, sau instrucțiuni.

În cadrul liniilor notate ca și comenzi de ieșire (C) am inclus și adresele binare pe care procesorul le trimite către memorie sau circuitele de intrare - ieșire ale calculatorului. Acestea sunt grupate de obicei într-o magistrală separată, numită *magistrală de adrese*.

Deosebirile dintre arhitecturile von-Neumann și Harvard pentru organizarea memoriei principale și a magistrelor de acces cu această memorie sunt indicate în schemele bloc din figura 3.4.

Prin conectarea procesorului cu un circuit de memorie, în care se stochează programul de executat (ansamblul instrucțiunilor binare) împreună cu datele eferente programului, se realizează o *structură de calculator*. Aici reacția, care conduce la automatul de ordin superior (ordinul 4) se încheie prin programul stocat în memorie, program care controlează comportarea în timp a procesorului.

Schema bloc din figura 3.5. indică faptul că procesorul are circuite de interfață (tampoane) pe toate liniile de conectare cu exteriorul. Acestea permit "deconectarea" procesorului de la

magistralele externe de date și adrese și de asemenea eliberarea unor linii de control externe, astfel încât alte dispozitive active (procesoare) din sistem să poată prelua temporar controlul magistrelor. Uneori tampoanele de la interfața procesorului cu exteriorul, au în afara capacității de trecere în stare de înaltă impedanță (HiZ) și capacitatea de memorare (latch).

Trebuie menționat că rolul principal al tampoanelor la interfața procesorului cu exteriorul nu constă în posibilitatea de HiZ, ci în capacitatea de a asigura factorii de încărcare de ieșire (fan-out¹) și intrare (fan-in²) necesari circuitelor logice, pentru asigurarea nivelurilor corecte de tensiune. Circuitele tampon au de asemenea rol de izolare între circuitele interne și cele externe procesorului și rolul de a permite controlul unor valori de capacitate parazită limită, pe fiecare linie.

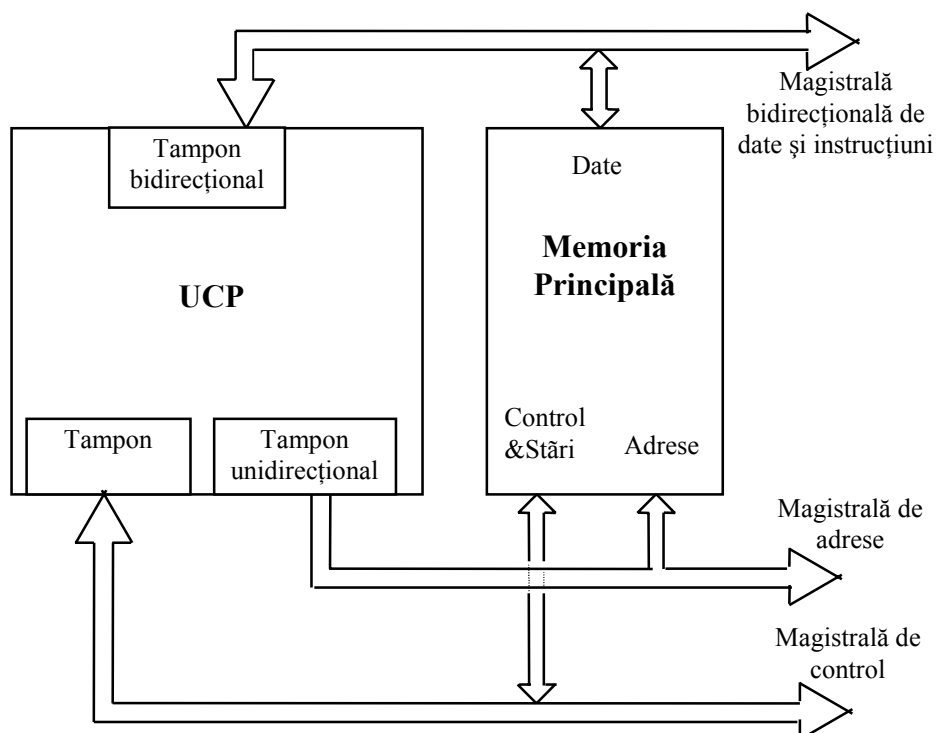


Figura 3.5. Structură de calculator, formată din UCP și memorie principală, cu indicarea circuitelor tampon la interfața procesorului cu exteriorul.

3.2. STRUCTURA UNEI UNITĂȚI CENTRALE DE PROCESARE ELEMENTARE

Cea mai simplă structură de UCP, bazată pe registru acumulator poate fi descrisă ca în figura 3.6. În fiecare dintre unitățile componente ale procesorului s-a figurat elementele principale ce permit interpretarea instrucțiunilor și execuția acestora.

În Unitatea de prelucrare a datelor s-au inclus Unitatea Aritmetică și Logică (ALU), registrul Acumulator (AC) și căile interne de transfer a datelor. În afara acestor elemente pot exista și registre de memorare temporară, registre folosite de procesor pentru operații intermediare, dar inaccesibile programatorului. ALU este un circuit combinațional și, așa cum s-a pomenit anterior, realizează

¹ Fan-out = este capacitatea maximă de încărcare pentru ieșirea unui circuit digital. Valoarea sa depinde de curentul maxim ce poate fi generat / absorbit pentru nivel logic 1 / 0 la ieșire și se exprimă ca numărul maxim de porți ce poate fi comandat de ieșire, cu păstrarea nivelurilor corecte de tensiune la ieșire.

² Fan-in = este capacitatea maximă de încărcare la intrare, definit similar cu fan-out pe baza curentului maxim la intrare, deci număr maxim de porți conectate la intrare, astfel ca nivelurile de tensiune corespunzătoare lui 1 și 0 logic să se păstreze.

operații aritmetice sau logice cu operanzii de la intrare. Operația este comandată de semnale de control intern provenite la unitatea de control. De exemplu, dacă operația ALU este selectată prin 4 biți, se pot selecta până la 16 operații elementare (adunare, complementare, ȘI logic, SAU, scădere etc.).

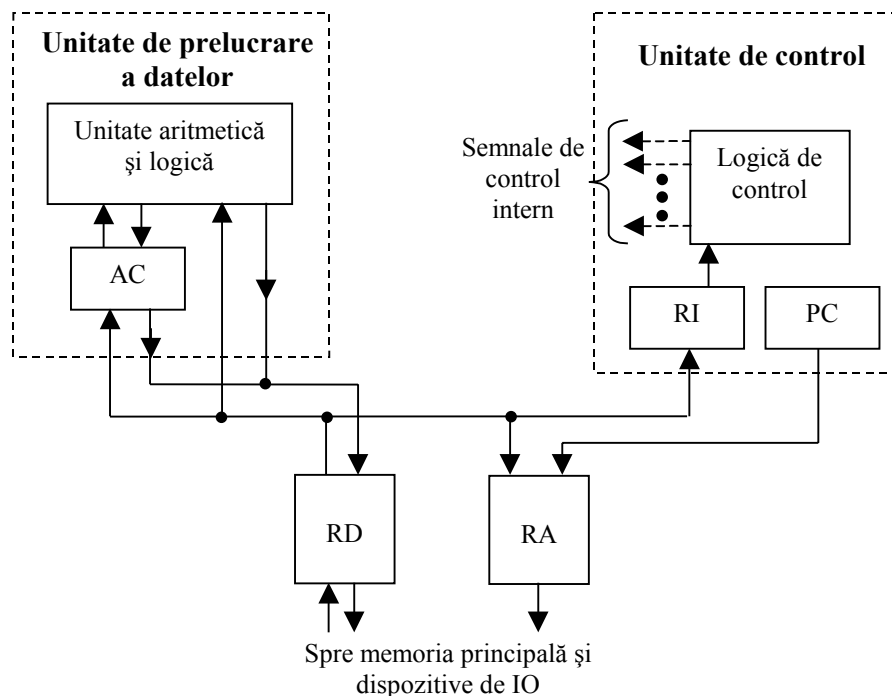


Figura 3.6. O structură simplă de procesor de uz general, pe bază de acumulator:

Pentru un procesor de uz general, în structura unității de prelucrare a datelor pe lângă logica combinațională de prelucrare, decodificare, multiplexare, există de obicei și un set de registre de uz general, vizibile programatorului și folosite pentru stocarea temporară a operanzilor și a rezultatelor. Aceste registre sunt selectate cu ajutorul unor câmpuri binare din cadrul codului fiecărei instrucțiuni. În structura UCP elementare se consideră, pentru moment, doar existența registrului *acumulator*. Acumulatorul este prezent în arhitectura majorității microprocesoarelor de 8 biți și este un registru cu funcție specială, el fiind atât registru de stocare pentru un operand sursă, cât și registru de stocare pentru operandul destinație, pentru multe din instrucțiunile microprocesorului. În codul acestor instrucțiuni, selecția acumulatorului se face implicit (deci fără câmp binar de adresare explicită). Folosirea acumulatorului prezintă avantaje din punctul de vedere al simplificării arhitecturii (adresare implicită a acumulatorului pentru multe din instrucțiuni, deci cod mai scurt al acestora), dar prezintă dezavantaje din punctul de vedere al vitezei de execuție și al flexibilității instrucțiunilor (acumulatorul este un loc îngust, o "ștrangulare", prin care trebuie să treacă fluxul informațional pentru toate instrucțiunile executate). Dacă se notează cu săgeată sensul de transfer al datelor și cu f operația efectuată de ALU, funcționarea AC poate fi simbolizată prin relația:

$$AC \leftarrow f(AC, RD) \quad (3.1)$$

unde RD este registrul tampon de date, la interfața cu magistrala externă de date.

În cadrul unității de control s-a simbolizat blocul ce realizează generarea semnalelor de control (logica secvențială de control) în care se implementează algoritmul de tranziție al stărilor automatului de control. Succesiunea specifică a stărilor corespunde codului instrucțiunii curente

care se stochează într-un registru de instrucțiuni (RI) pe toată durata execuției instrucțiunii. Un al doilea registru important al unității de control este numit Contor de Program (simbolizat prin PC³).

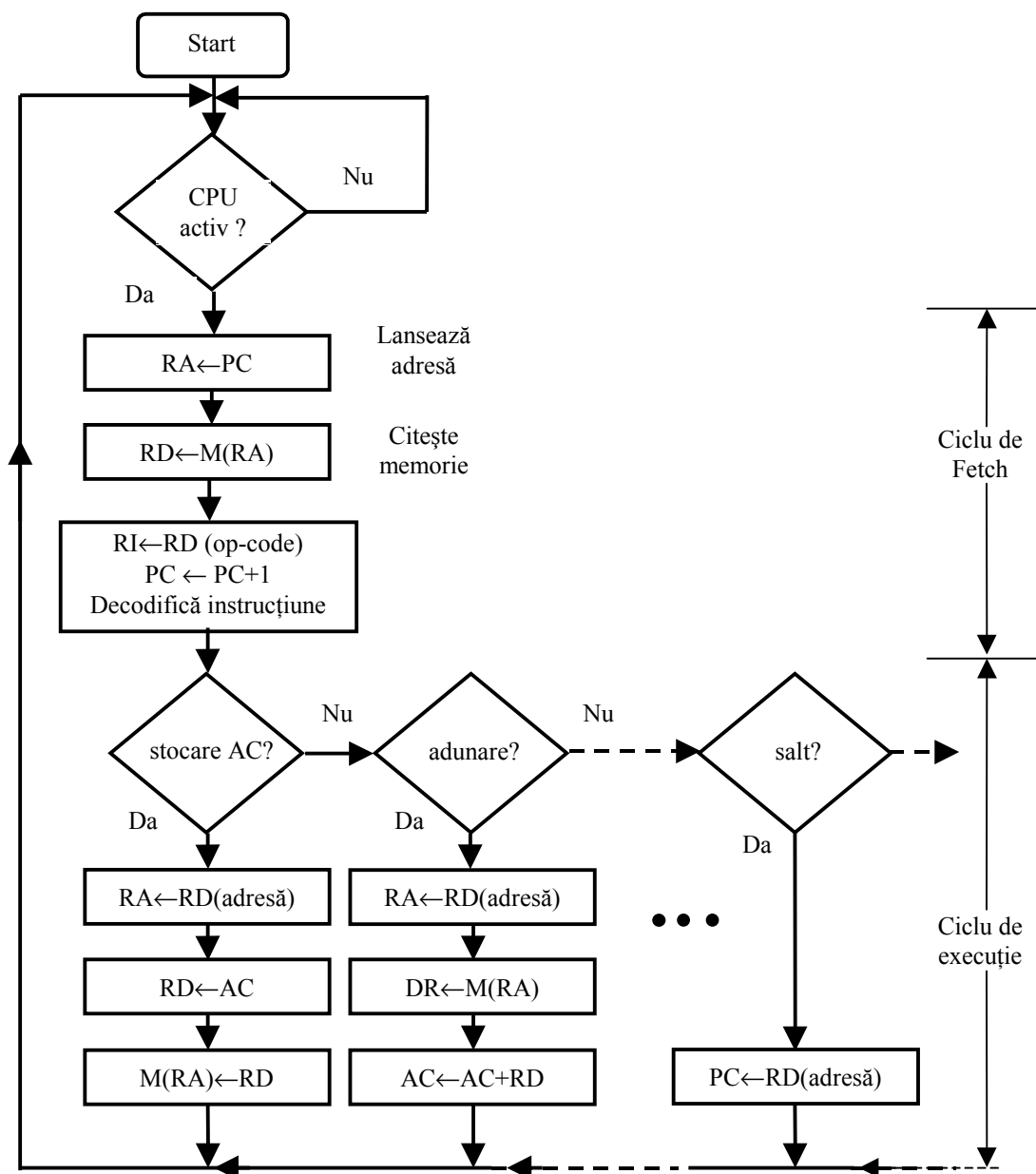


Figura 3.7. Organigramă privind funcționarea UCP elementare din figura 3.6

Acest registru are capacitatea de autoincrementare. El conține adresa instrucțiunii următoare din program (sau informație pe baza căreia se construiește adresa din memoria principală unde se găsește instrucțiunea următoare). Ori de câte ori s-a terminat execuția unei instrucțiuni, conținutul PC se transmite către registrul tampon de adrese (RA – transparent pentru programator), iar PC se incrementează pentru a pregăti adresa către următoarea instrucțiune. Se consideră că instrucțiunile programului sunt aranjate secvențial, în adrese succesive ale memoriei principale. Registrul PC, pentru adresarea instrucțiunilor funcționează în majoritatea timpului ca un contor, cu incrementare, (pentru secvențe liniare de instrucțiuni). La majoritatea microprocesoarelor de 8 biți conținutul acestui registru se transmite pe magistrala de adrese, indicând adresa cuvântului (octet, sau multi-

³ PC = Program Counter (engl.) – Contor de program

octet) ce trebuie citit din memorie, cuvânt ce face parte din câmpul binar al unei instrucțiuni. După fiecare ciclu de extragere (fetch) a unui cuvânt din memorie, PC se auto-incrementează ($PC := PC + 1$), pentru a pregăti extragerea următorului cuvânt. Dacă lărgimea magistralei de date a procesorului permite să se aducă simultan mai mult de un cuvânt operația poate fi scrisă: $PC = PC + N$, unde N este numărul de cuvinte citite simultan. La pornirea execuției unui program, sau ori de câte ori se face un salt în program conținutul PC se încarcă automat cu o valoare de adresă, urmând ca în continuare să funcționeze ca și contor de instrucțiuni.

În figura 3.7. se indică un exemplu de secvență a stărilor (microoperații) pentru aducerea (fetch) și execuția instrucțiunilor pentru structura elementară a UCP din figura 3.6. Fiecărei stări (dreptunghi în figură) îi sunt asociate semnale de control intern UCP, pentru realizarea operațiilor simbolizate pe figură. Orice transfer este indicat cu săgeată, pe poziția din stânga fiind indicată destinația, iar pe cea din dreapta sursa. Prin notația $M(RA)$ s-a indicat o adresare la memoria principală, adresa fiind cea conținută în registrul de adrese RA. La unele dintre operații s-a folosit notația RD (*câmp*) care semnifică faptul că se citește doar *câmpul* de adresă sau cod al operației (op-code) din formatul binar al instrucțiunii aduse în registrul RD.

Secvența din figura 3.7 este doar teoretică, funcționarea procesorului elementar nefiind optimizată din punctul de vedere al minimizării stărilor mașinii cu algoritm de stare. De exemplu blocurile de comparare, pentru testarea codului instrucțiunii (op-code⁴) funcționează secvențial și ar însemna că pentru ultima instrucțiune trecută în lista de comparări s-ar parcurge toate operațiile de comparare. Operația de tip *switch* (sau *case*) în limbaj de nivel înalt poate fi implementată în hardware, pentru saltul direct la ramura corespunzătoare execuției instrucțiunii.

3.3. PERFECTIONAREA UNITĂȚII CENTRALE DE PROCESARE ELEMENTARE

La structura elementară de UCP din figura 3.6 se pot aduce câteva tipuri de îmbunătățiri ce pot fi aduse structurii pentru a crește performanțele și flexibilitatea în ceea ce privește performanțele procesorului. O parte se vor discuta în acest paragraf, altele în paragrafele și capitolele următoare.

Principalele tipuri de extindere în organizarea UCP elementară sunt următoarele:

1. Introducerea unor registre speciale pentru stocarea stării procesorului în care să se salveze anumiți indicatori de condiții și de control. Adesea acest tip de registru este numit registru pentru cuvântul de stare al programului⁵. Indicatorii de condiții⁶ pot fi testați de instrucțiuni de salt condiționat și astfel pot modifica succesiunea instrucțiunilor executate de procesor în cadrul unui program. Indicatorii sunt importanți pentru că permit implementarea instrucțiunilor de testare și decizie în limbaj de nivel înalt. Indicatorii de control au un alt rol. Așa cum le spune și numele ei sunt destinați pentru a modifica modul de funcționare al unității de control, sau al unor registre de adresare ale acestei unități, în funcție de valoarea introdusă în indicatori. Cei mai simpli dintre acești indicatori pot indica de exemplu: mod de lucru supervisor / utilizator, mod de lucru cu auto-incrementarea sau auto-decrementarea unor registre index, blocarea cererilor de întrerupere mascabile, informații privind drepturile de acces la unele zone de memorie etc. În general indicatorii pot fi setați prin instrucțiuni recunoscute de procesor.
2. Introducerea unor registre suplimentare de adresare. Adesea se folosesc și registre pentru moduri complexe de adresare (indexată, bazată, cu bază și index).
3. Introducerea unei bănci de registre vizibile programatorilor (sau compilatorului), pentru

⁴ op-code (engl.) este prescurtarea de la operation code (codul operației)

⁵ PSW - program status word

⁶ flags în limba engleză

stocarea temporară și manipularea datelor.

4. Adăugarea unei unități aritmetice pentru numere în virgulă mobilă (numere reale), în plus față de unitatea ALU pe întregi.
5. Introducerea de suport hardware (logică plus registre de adresare specializate) pentru implementarea memorie stivă. Memoria stivă este utilă în primul rând utilă în operațiile de apelare și revenire din proceduri, dar are și alte utilizări. Uneori memoria stivă este implementată hardware, prin registre construite în cadrul UCP, dar cel mai adesea ea este organizată software (stocare în memoria principală și administrare printr-un registru indicator al vârfului stivei – SP = stack pointer).

O structură de UCP îmbunătățită cu o parte din extensiile pomenite mai sus se prezintă în figura 3.8.

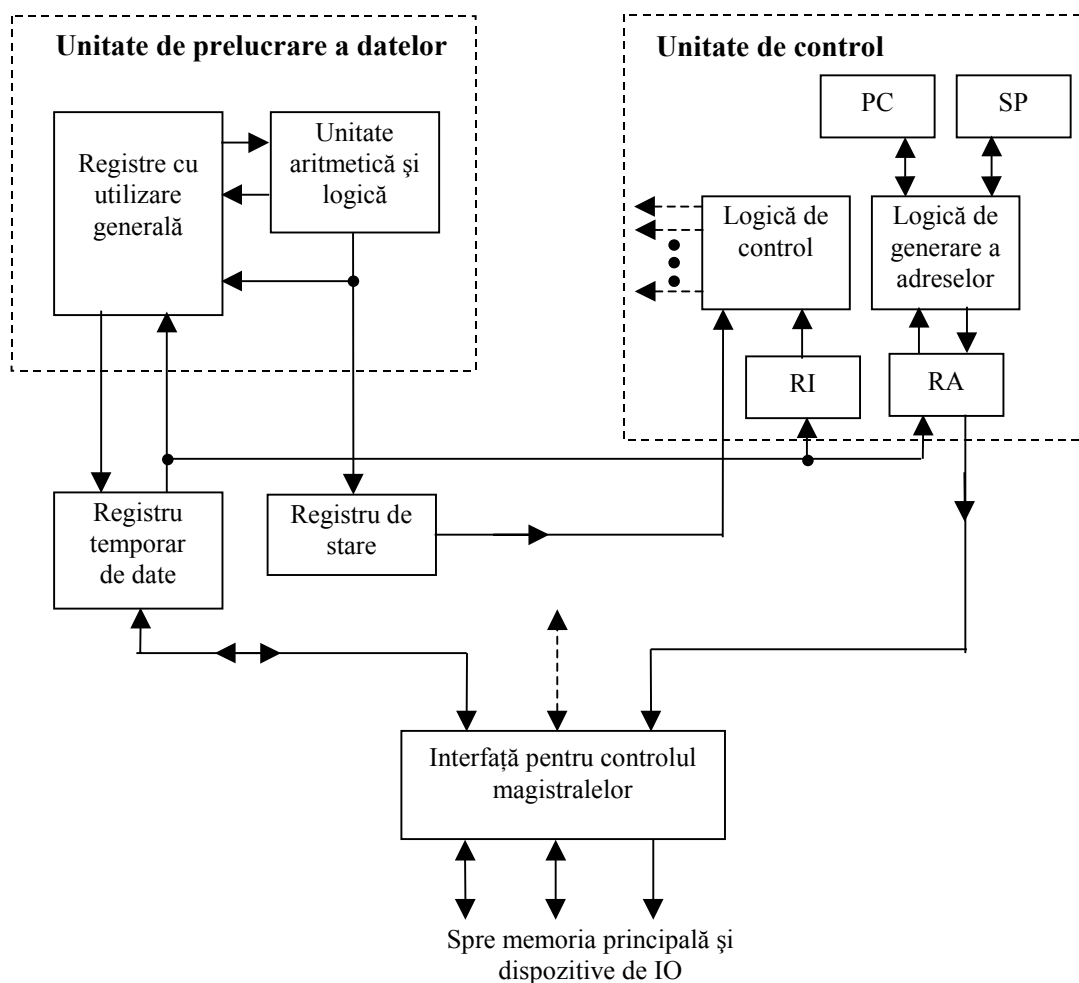


Figura 3.8. UCP tipică pentru organizarea pe bază de registre de uz general.

În unitatea de prelucrare a datelor s-a introdus o bancă de registre de uz general, adresabile individual de către program. În același bloc se consideră a fi inclusă și logica de selecție (adresare) și de multiplexare a ieșirilor către căile interne de date. Acestea sunt transparente pentru utilizator (deci inaccesibile explicit prin instrucțiuni). S-a figurat de asemenea un registru pentru stocarea indicatorilor de condiții (registru de stare). Indicatorii sunt setați în funcție de rezultatul operațiilor efectuate de ALU (unitatea aritmetică și logică) și sunt testați de unitatea de control, reacție simbolizată în figură prin liniile de intrare în blocul de logică de control. Toate circuitele tampon cu

magistralele externe au fost introduse într-un bloc numit “interfață pentru controlul magistralelor”. Pe calea de date apare de asemenea un registru temporar pentru stocarea operanzilor și a rezultatelor care se citesc / stochează în memoria principală (registru temporar nu este vizibil de instrucțiuni – programator). Pentru că se consideră un procesor cu magistrală bidirecțională de date și instrucțiuni registrul de instrucțiuni recepționează codul instrucțiunii curente din același registru temporar de date. În cazul în care instrucțiunile conțin adrese (sau informații de adresare) pentru operanzi din memoria principală, acestea se pot citi în registrul de adrese (RA) care poate citi registrul temporar de date.

Pentru automatul de adresare al unității de control s-a prevăzut logica necesară pentru generarea adreselor din memoria principală. Informația de adresare poate proveni de la contorul de program (PC) pentru instrucțiuni, sau de la registrul indicator de stivă (SP⁷) în cazul când se efectuează operații cu memoria stivă. Registrul SP indică întotdeauna adresa vârfului stivei, memoria stivă lucrând pe principiul LIFO⁸. Vârful stivei conține ultimul articol (cuvânt) înscris în stivă.

Nu s-au desenat căile interne de control, dar s-a indicat prin linie punctată existența semnalelor de control și sincronizare cu exteriorul UCP, care prin interfața cu magistralele vor forma magistrala de control externă procesorului.

3.4. ALTE REGISTRE INTERNE SEMNIFICATIVE PENTRU UCP

Așa cum s-a spus în paragraful anterior, unitatea aritmetică și logică setează biții unui registru special, numit *registru de stare*, sau *registru al indicatorilor de condiții*. Acești indicatori (Flags - fanioane) sunt numiți “*de condiții*” pentru că ei reflectă anumite stări ale rezultatului operațiilor aritmetico-logice, fiind testați de către Unitatea de Control, care poate astfel efectua devierea secvenței programului prin așa-numitele salturi condiționate. Indicatorii de condiții (stare) sunt modificați ca urmare a execuției instrucțiunilor aritmetice, logice sau prin execuția unor instrucțiuni de control, a căror singură funcție este modificarea indicatorilor. Conținutul lor, în general, nu poate fi citit direct, ci doar testat indirect prin instrucțiuni de salt condiționat ce transferă controlul programului la diferite ramuri, în funcție de valoarea indicatorului testat.

Vom descrie în continuare un *set tipic de indicatori* de condiții și apoi vom da câteva exemple pentru microprocesoare existente pe piață:

- **Semn (S)** - Indicatorul de semn ia valoarea 1 logic dacă rezultatul unei operații cu numere cu semn este un număr negativ. În cazul în care se lucrează cu numere fără semn, de obicei, indicatorul va copia valoarea logică a celui mai semnificativ bit (MSb) al rezultatului (dar atunci nu mai are interpretarea de indicator de semn).
- **Zero (Z)** - acest indicator ia valoarea 1 logic dacă toți biții rezultatului unei operații aritmetice sau logice sunt zero; altfel Z = 0.
- **Paritate (P)** - indică paritatea rezultatului în convenția de imparitate (suma modulo 2 a tuturor biților rezultatului, inclusiv a bitului P, este întotdeauna 1). Acest indicator este deci setat (poziționat pe 1 logic) dacă rezultatul conține un număr par de biți de valoare "1". Dacă numărul este impar, atunci P=0. Se setează, în general, în urma operațiilor logice.
- **Depășire (V, oVerflow)** a capacității de reprezentare. Prin depășire a capacității de reprezentare se înțelege faptul că rezultatul unei operații aritmetice nu poate fi reprezentat pe n biți ca număr cu semn. Indicatorul V se poziționează la 1 logic în urma operațiilor aritmetice, indicând depășirea capacității de reprezentare în cazul operării cu numere cu semn, depășire ce se manifestă prin alterarea bitului de semn (deci V=1 indică rezultat

⁷ SP – Stack Pointer (engl.) = Indicator sau pointer de stivă.

⁸ LIFO – Last In First Out (engl.) – Ultimul intrat, primul ieșit din memoria stivă

eronat). Valoarea indicatorului poate fi descrisă printr-o operație "SAU EXCLUSIV" între transporturile de la rangul n-2 (MSb) și respectiv n-1 (poziție bit de semn), pentru numere cu semn reprezentate pe n biți, în cod complementar (complement față de doi).

- **Transport (CY, Carry)** - indicator de transport. Se poziționează pe 1 logic dacă operația anterioară a produs un transport sau un împrumut la/de la rangul n-1.
- **Transport auxiliar (AC, Auxiliary Carry)** - Se poziționează pe 1 dacă o operație de adunare în cod NBCD a produs transport de la o cifră zecimală codificată în NBCD către cifra zecimală de rang superior (transport de la o tetradă binară inferioară către o tetradă binară superioară). În cazul operațiilor de scădere în NBCD acest indicator indică împrumutul.
- **Adunare sau scădere în NBCD (N)** - apare doar la unele microprocesoare. Prin valoarea 1 arată că operația anterioară în NBCD a fost o scădere. La microprocesoarele care folosesc acest tip de indicator există o singură instrucțiune de corecție zecimală, folosită atât pentru adunare cât și pentru scădere, spre deosebire de microprocesoarele ce nu au indicator de tipul N și care au instrucțiuni de corecție zecimală separate pentru adunare și scădere. Indicatorii specifici operațiilor în NBCD (AC și N) nu pot fi testați prin program, ei fiind destinați doar circuitului de corecție zecimală inclus în unitatea aritmetică și logică.

Exemple de registre de indicatori, la diferite microprocesoare:

Tabelul 3.1.

Tip procesor	Poziția binară în registrul de indicatori															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
18080									S	Z	*	AC	*	P	*	CY
Z80									S	Z	*	H	*	P/V	N	C
MC6800									*	*	AC	I	S	Z	V	C
18086	*	*	*	*	OF	DF	IF	TF	SF	ZF	*	AF	*	PF	*	CF
MC68000	T	*	S	*	*	I2	I1	I0	*	*	*	X	N	Z	V	C

Peste tot, biții liberi (care nu reprezintă indicatori) sunt notați cu steluță (*). Pentru unele din exemplele de mai sus se observă că în registrul de stare sunt conținuți nu numai *indicatori de condiții* ci și alte tipuri de indicatori; aceștia sunt *indicatori de control* ai procesorului, influențând funcționarea acestuia.

La microprocesorul Z80 există un indicator comun paritate / depășire (P/V), iar indicatorul de transport auxiliar e notat cu H. Indicatorul N este folosit de circuitul de corecție zecimală (la operații în NBCD) el indicând scădere (N=1) sau adunare (N=0) la operația anterioară în NBCD. La procesorul MC6800 al firmei Motorola în registrul de indicatori apare și un indicator de control al întreruperilor mascabile (permise pentru I=1). La fel, în cazul microprocesorului Intel 8086 registrul de indicatori conține și 3 indicatori de control (controlează unele operații ale UCP). Notăția indicatorilor are adăugată și litera F (de la flag), AF fiind indicatorul de transport auxiliar, iar OF cel de depășire. În privința indicatorilor de control la Intel 8086 menționăm următoarele:

- indicatorul de întrerupere (Interrupt Flag = IF) determină modul în care procesorul 8086 reacționează la cererile de întrerupere mascabile aplicate pe pinul INT. Valoarea IF = 1 arată că cererile de întrerupere sunt validate. Indicatorul poate fi poziționat pe 1 sau șters prin instrucțiuni ale programului.
- indicatorul de direcție (Direction Flag = DF) este utilizat în operațiile cu șiruri, când este necesar ca șirul să fie definit prin adresa de bază și prin sensul descrescător (DF=1), sau crescător (DF=0) al adreselor index (indice) care definesc fiecare element din șir față de o adresă de bază;
- indicatorul de mod "pas cu pas" (Trap Flag = TF) permite implementarea unui mod de

funcționare util în depanarea programelor. Astfel, dacă $TF = 1$, după execuția fiecărei instrucțiuni se inițiază o întrerupere care determină transferul controlului într-o zonă de memorie definită de utilizator, unde sub acțiunea unui program adecvat, se poate face analiza stării interne a procesorului. Acest indicator este folosit de programele depanatoare⁹.

La MC68000, octetul mai semnificativ al registrului de stare este octetul de control al sistemului (System Byte conține Trace mode, Supervisor state, Interrupt mask: **I2,I1,I0**), iar octetul mai puțin semnificativ (b_7-b_0) este octetul indicatorilor pentru programul utilizator (User Byte care conține indicatorii eXtend, Negative, Zero, oVerflow, Carry). În funcție de valoarea lui S există indicator de stivă (SP) separat pentru supervisor și pentru utilizator; dacă $S=1$ se folosește SSP (=A7' Supervisor stack pointer) iar dacă $S=0$ se folosește USP (=A7, User stack pointer). SSP și implicit stiva sunt protejate în acest fel față de modificările utilizatorilor. Maska de întrerupere indică nivelul de întrerupere curent. Orice întrerupere cu nivel mai mare decât nivelul măștii fiind recunoscută. Există 14 combinații ale indicatorilor C, Z, N și V ce pot fi testate pentru salturi condiționale. Indicatorul X este rezervat exclusiv pentru aritmetica în precizie multiplă.

Pentru I8080, al firmei Intel, notația indicatorilor corespunde cu cea din prezentarea setului tipic de indicatori.

Cu privire la indicatorul de depășire (V – oVerflow) remarcăm următoarele aspecte importante. Dacă rezultatul unei operații aritmetice cu numere cu semn reprezentate pe n biți este prea mare ca să poată fi reprezentat pe n biți se spune că se produce depășire a capacității de reprezentare. Aceste depășiri trebuie determinate pentru că adesea ele sunt o indicație a erorilor de programare. Fie exemplul următor de adunare a două numere binare cu semn, reprezentate în cod complementar:

$$\begin{array}{r} x_{n-1}x_{n-2}\dots x_0 + \\ y_{n-1}y_{n-2}\dots y_0 \\ \hline z_{n-1}z_{n-2}\dots z_0 \end{array} \quad (3.2)$$

Pentru fiecare rang i , prin adunarea biților x_i , y_i și c_i se generează un bit de sumă z_i și un bit de transport (carry bit) c_{i+1} către rangul următor, unde z_i și c_i se consideră pentru $0 \leq i \leq n-1$, $c_0=0$, iar valorile lor se calculează conform relațiilor unui sumator binar complet:

$$z_i = x_i \oplus y_i \oplus c_i \quad (3.3)$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \quad (3.4)$$

Notăm cu V variabila binară care indică depășirea (oVerflow) atunci când $V = 1$. Tabelul 3.2. indică modul de determinare al bitului de semn z_{n-1} și al variabilei V în funcție de biții de semn ai celor doi operanzi, x_{n-1} , y_{n-1} , și bitul de transport c_{n-1} de la rangul $n-2$, care este MSb¹⁰.

Rezultă că indicatorul de depășire poate fi definit prin ecuația:

$$V = \overline{x_{n-1}} \overline{y_{n-1}} c_{n-1} + x_{n-1} y_{n-1} \overline{c_{n-1}} \quad (3.5)$$

Dacă din tabelul 3.2 se elimină liniile pentru care $V = 1$ va rezulta definirea corectă, conform ecuației (3.3)) și pentru bitul de semn al sumei:

$$z_{n-1} = x_{n-1} \oplus y_{n-1} \oplus c_{n-1} \quad (3.6)$$

⁹ Debugger (engl.) = Program depanator

¹⁰ MSb – Most significant bit (engl.) = cel mai semnificativ bit

Acest lucru este deosebit de important, pentru că arată că la operațiile în cod complement față de 2 biții de semn ai operanzilor pot fi tratați în același mod ca și biții ce indică mărimea numărului.

Tabelul 3.2.

Intrare			Ieșire		Observații
x_{n-1}	y_{n-1}	c_{n-1}	z_{n-1}	V	
0	0	0	0	0	
0	0	1	0	1	Se adună două numere pozitive, deci suma trebuie să fie pozitivă
0	1	0	1	0	
0	1	1	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	1	0	1	1	Se adună două numere negative, deci suma trebuie să fie negativă
1	1	1	1	0	

3.5. EXEMPLE PRIVIND OPERAȚIILE UCP ȘI MODUL DE SETARE AL INDICATORILOR DE CONDIȚII

1. Dacă se adună, sau se scad, două numere reprezentate în cod NBCD, corecția zecimală (adunarea, respectiv scăderea codului lui 6) se aplică fie dacă rezultatul nu este un cod NBCD, fie dacă se produce transport de la bitul cel mai semnificativ al codului ($AC=1$):

a)

<u>zecimal</u>	<u>binar în cod NBCD</u>
07+	0000 0111+
05	0000 0101
---	-----
12	0000 1100 + (>1001) S=0, Z=0, P=1, V=0, C=0, AC=0
	0110 corecție zecimală

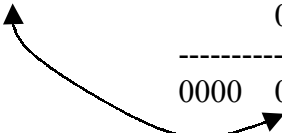
	0001 0010

Se observă că pentru codul cifrei zecimale de pondere mai mică (tetradă inferioară) s-a obținut un cod nepermis (1100 care este mai mare decât codul lui 9, ce are valoare maximă); de aceea se aplică corecția zecimală, prin adunarea codului cifrei 6. Se observă de asemenea că indicatorii de condiții sunt setați, chiar dacă unii din ei nu au semnificație pentru operația efectuată.

b)

<u>zecimal</u>	<u>binar în cod NBCD</u>	
12-	0001	0010-
08		1000
---	-----	
04	0000	1010 -
		0110
		corecție zecimală

	0000	0100



Aici corecția zecimală se aplică unei operații de scădere a două numere zecimale codificate în NBCD și de aceea se scade codul lui 6 din cauza apariției împrumutului ($AC=1$).

2. Dacă se adună două numere binare cu lungimea de un octet, se scriu mai jos, pe prima coloana reprezentările în zecimal, numerele fiind interpretate ca numere cu semn (corespunzând la valoarea binară în cod complementar), pe a doua coloană zecimal, numerele fiind interpretate ca numere fără semn, pe a treia în binar și pe ultima coloană valorile indicatorilor de condiții:

a)

<u>zecimal</u>		<u>binar</u>
<u>(cu semn)</u>	<u>(fără semn)</u>	
+ 8 +	8 +	0000 1000 +
- 128	128	1000 0000
----	----	-----
- 120	136	1000 1000
		S=1, Z=0, P=1, V=0, C=0, AC=0

Dacă numerele sunt considerate cu semn, rezultatul obținut este un număr cu semn, corect, pentru că nu s-a produs depășire a capacității de reprezentare ($V=0$). De asemenea rezultatul este corect și pentru numere fără semn, căci $CY=0$. La aceasta operație AC nu are nici o semnificație. Indicatorul de paritate $P=1$ indică un rezultat cu număr par de biți 1.

b)

<u>zecimal</u>		<u>binar</u>
<u>(cu semn)</u>	<u>(fără semn)</u>	
- 128 +	128 +	1000 0000 +
+ 127	127	0111 1111
----	----	-----
- 1	255	1111 1111
		S=1, Z=0, P=1, V=0, CY=0, AC=0

Indicatorii arată rezultat corect pe 8 biți, atât pentru numere cu semn cât și pentru numere

fără semn.

c)

zecimal		binar
(cu semn)	(fără semn)	
- 72 +	184	1011 1000 +
+ 104	104	0110 1000
----	----	-----
+ 32	288	0010 0000

S=0, Z=0, P=0, V=0, CY=1, AC=1

Aici V=0 indică că pentru operație în cod complementar (cu semn) nu s-a produs depășire a capacității de reprezentare, deci rezultatul este corect (+32 în zecimal). Dacă numerele binare sunt considerate fără semn, rezultatul pe 8 biți nu este corect pentru că indicatorul CY=1, deci s-a produs o depășire a capacității de reprezentare pe 8 biți (rezultatul fără semn este mai mare cu 32 decât 255, valoarea maximă reprezentabilă). Din nou AC=1 nu are nici o semnificație aici.

Din exemplele anterioare se observă importanța definirii corecte a operanzilor cu care lucrează instrucțiunile procesorului. Interpretarea corectă a numerelor, într-un anumit cod intern (binar cu semn, fără semn, NBCD, ASCII, etc.) se face de către programul compilator, sau de către programator, dacă se lucrează în limbaj de asamblare.

În ceea ce privește registrele suplimentare introduse în unitatea de adresare, exemplificăm pe scurt următoarele tipuri de registre:

- *Registre index*: Registrele index realizează modificarea unei adrese de operand. În unele cazuri funcțiile aritmetice ale registrelor index, pot fi la fel de generale ca și al registrelor aritmetice (adunări, scăderi, înmulțiri, inversări ale biților conținuți pentru realizarea unor funcții speciale), iar în altele, aceste funcții sunt limitate sever. Microprocesoarele simple nu au registre specializate pentru adresarea indexată, ele folosind pentru acest scop registrele de uz general.
- *Registre (de adresă) de bază*: Aceste registre sunt în mod obișnuit folosite în calculatoare pentru tehnicile de adresare ce permit adresarea unui spațiu de memorie mai mare, utilizând un număr limitat de biți pentru a reprezenta adresa operandului în instrucțiune.
- *Registre de protecție a memoriei (sau registre limită)* se folosesc în cadrul unităților de adresare din calculatoarele destinate să faciliteze utilizarea unor sisteme de operare de tip multi-tasking, multi-programming. Pentru că la acestea există mai mult decât un program rezident (sistemul de operare plus programele utilizator), trebuie prevăzut un anumit mod de control pentru a preveni accesele întâmplătoare sau intenționate la locații de memorie ce nu aparțin programului utilizator. Într-un sistem cu uni-programare (cum este și MS-DOS), este de obicei suficient să se interzică programului utilizator să acceseze locațiile sistemului de operare, situat la adresele mici din memoria principală.

3.6. SEMNALE LA INTERFAȚA UCP CU EXTERIORUL

Semnalele de la interfața Unității Centrale de Procesare (UCP) cu exteriorul pot fi grupate, funcțional, în 3 tipuri de magistrale:

1. magistrala de adrese
2. magistrala de date
3. magistrala de control

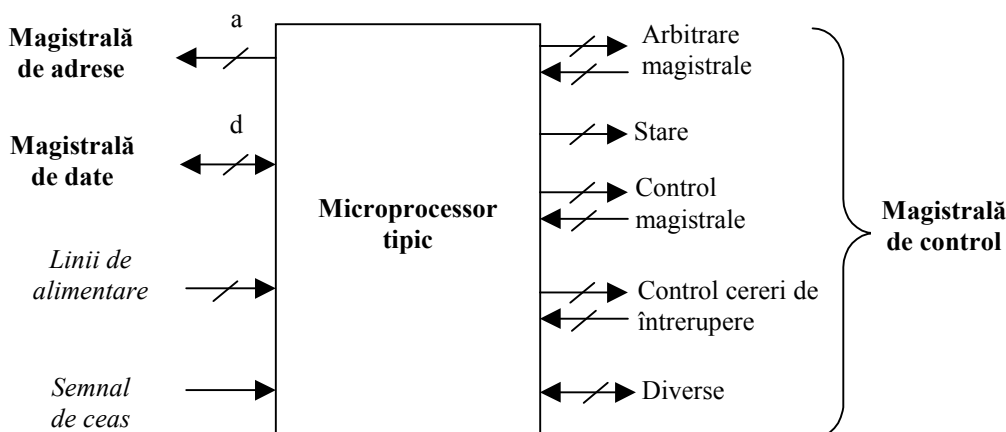


Figura 3.9. Semnale tipice la interfața unui microprocesor cu exteriorul

3.6.1. Magistrala de adrese

Această magistrală cuprinde linii de adresă, ce transmit doar semnale de ieșire din microprocesor, fiind deci o magistrală unidirecțională. Ieșirile UCP spre această magistrală externă de adrese pot trece în starea de înaltă impedanță (HiZ) la o cerere externă, când UCP cedează controlul magistrelor către alt dispozitiv. Liniile de pe această magistrală se folosesc pentru adresarea locațiilor de memorie și a porturilor de intrare-ieșire. De obicei valorile binare cele mai semnificative (cu ponderea binară maximă) din adresă se folosesc pentru selectarea blocurilor de memorie. Există trei moduri în care se pot obține aceste semnale de selecție, pe baza informației de adresare:

a. Adresare liniară

Utilizată doar pentru sistemele mici, unde se utilizează doar o parte din spațiul posibil de adresare. În aceste cazuri linii ale magistralei de adrese pot fi utilizate împreună cu semnale de control pentru a selecta direct blocuri de memorie. Pentru exemplificare, se presupune că pentru un microprocesor numărul de biți ai adresei este $a = 16$, deci spațiul maxim de adresare este de 2^{16} locații de memorie (64 k locații). Presupunem că un sistem simplu folosește doar 12 k de memorie, format din 3 blocuri de memorie a câte 4 k. Fiecare bloc poate fi adresat cu 12 biți de adresă ($2^{12} = 4096 = 4 \text{ k}$).

Dacă considerăm cei 12 biți mai puțini semnificativi pentru adresarea în cadrul blocurilor, ceilalți 4 biți din magistrala de adrese pot fi folosiți pentru selecția blocurilor de memorie. O variantă a acestui mod de selecție liniară se prezintă în figura 3.10, în care s-a considerat că

există două blocuri de ROM¹¹ a câte 4 kilo-octeți (4 kB) și un bloc de RAM¹².

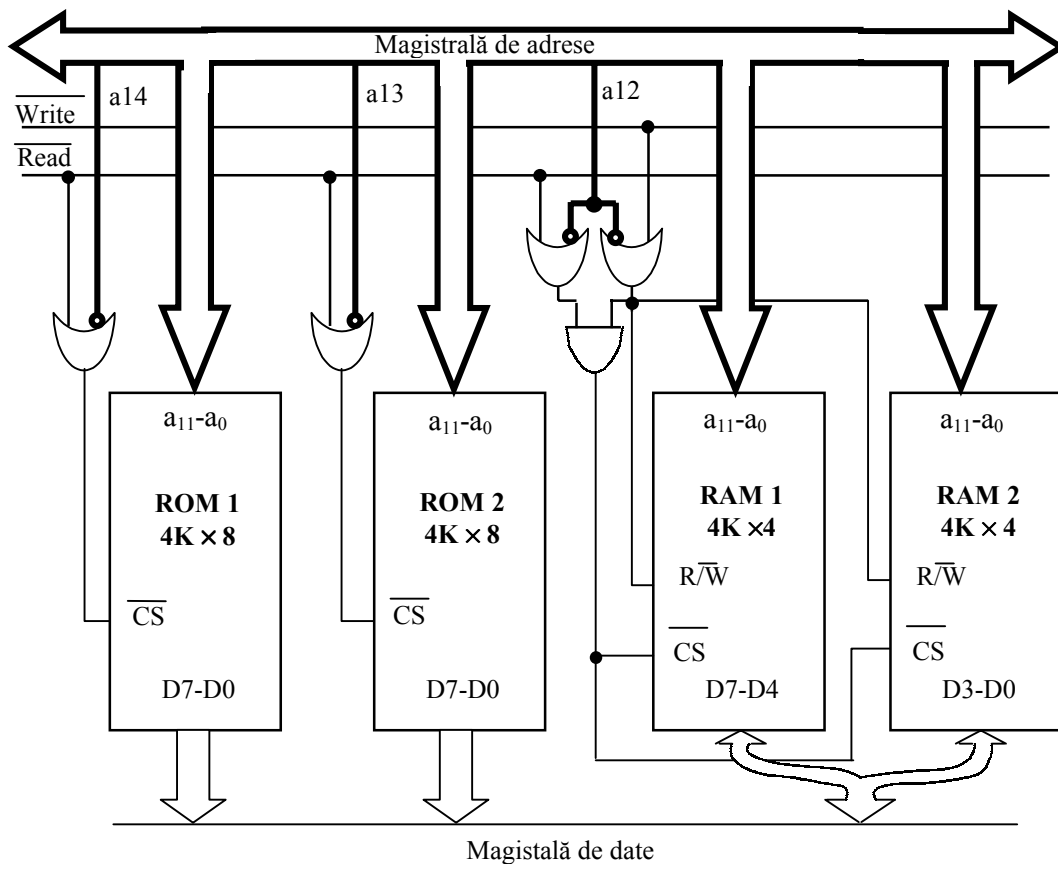


Figura 3.10. Un exemplu de selecție liniară pentru o memorie de 12kx8, din care 8kx8 memorie ROM

La rândul său blocul de RAM 4kx8 este format din două circuite RAM cu capacitatea 4Kx4 biți. Pentru selecția liniară se folosesc biții a₁₄, a₁₃ și a₁₂ de pe magistrala de adrese, împreună cu semnalele de control de citire (Read) și scriere (Write), iar biții a₁₁ - a₀ sunt folosiți pentru adresarea locațiilor interne circuitelor de memorie. Semnalele de control de citire și scriere sunt active la nivel logic JOS (de aceea au bară deasupra).

Pentru a înțelege logica de selecție (intrările \overline{CS} ¹³ - selecție circuit) pentru circuitele de memorie se indică mai jos ecuațiile care definesc porțile logice.

$$\begin{aligned} CS &= RD \cdot A14 \\ \overline{CS} &= \overline{RD} \cdot \overline{A14} \\ \overline{CS} &= \overline{RD} + \overline{A14} \end{aligned} \quad (3.7)$$

$$R / \overline{W} = \overline{WR} \cdot \overline{A14} = \overline{WR} + \overline{A14} \quad (3.8)$$

¹¹ ROM = Read Only Memory (engl.) - memorie doar cu citire

¹² RAM = Random Access Memory (engl.) - memorie cu citire scriere (traducerea exactă este "memorie cu acces aleator")

¹³ CS = Chip Select

Adresa (Hex)	Circuit	Selecție circuit	Selecție locații memorie			
		a ₁₅ a ₁₄ a ₁₃ a ₁₂	a ₁₁ a ₁₀ a ₉ a ₈	a ₇ a ₆ a ₅ a ₄	a ₃ a ₂ a ₁ a ₀	
0000	Fără circuit	0000	0000	0000	0000	0000
:		:	:	:	:	:
0FFF		0000	1111	1111	1111	1111
1000	RAM (1+2)	0001	0000	0000	0000	0000
:		:	:	:	:	:
1FFF		0001	1111	1111	1111	1111
2000	ROM 2	0010	0000	0000	0000	0000
:		:	:	:	:	:
2FFF		0010	1111	1111	1111	1111
3000	Fără circuit	0011	0000	0000	0000	0000
:		:	:	:	:	:
3FFF		0011	1111	1111	1111	1111
4000	ROM 1	0100	0000	0000	0000	0000
:		:	:	:	:	:
4FFF		0100	1111	1111	1111	1111
5000	Fără circuit	0101	0000	0000	0000	0000
:		:	:	:	:	:
:		:	:	:	:	:
FFFF		1111	1111	1111	1111	1111

Figura 3.11. Harta memoriei selectată liniar ca în figura 3.10.

$$\begin{aligned}\overline{CS} &= \overline{A_{12} \cdot (RD + WR)} = \overline{A_{12} \cdot RD} + \overline{A_{12} \cdot WR} = \overline{A_{12} \cdot RD} \cdot \overline{A_{12} \cdot WR} \\ &= (\overline{A_{12}} + \overline{RD}) \cdot (\overline{A_{12}} + \overline{WR})\end{aligned}\quad (3.9)$$

Pentru memoria din figura 3.10, în figura 3.11 se indică harta memoriei ce conține alocarea adreselor pe circuite de memorie, adrese minime maxime pentru fiecare circuit, iar în cazul unei adresări liniare și spațiile libere - neocupate.

b. Adresă decodificată complet.

La acest mod de selectare a circuitelor de memorie toate liniile de adresă (toți cei a biți) sunt folosite pentru adresare, deci se utilizează întreg spațiul de adresare disponibil. Biții mai semnificativi ai adresei sunt decodificați astfel încât să se genereze semnale de selecție pentru toate dispozitivele de memorie utilizate.

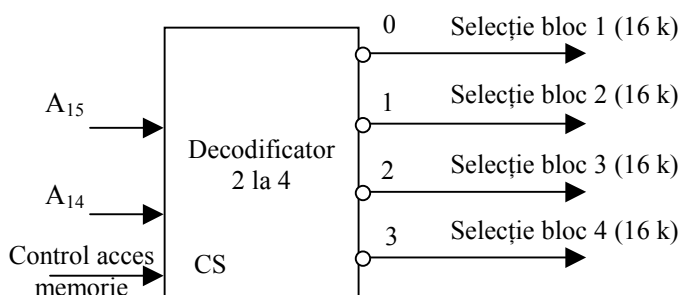


Figura 3.12. Decodificare pentru generarea semnalelor de selecție și adresarea unei memorii de 64 k prin decodificare completă. În cazul utilizării în sistem și a unor circuite mai mici de 16k se mai poate introduce un bloc de decodificare circuitul decodificator fiind selectat de ieșirile decodificatorului 2:4, iar pe intrările de adrese se introduc următorii biți mai semnificativi ai adresei, de la 13 în jos.

Ca exemplificare, în figura 3.12, se prezintă modul de decodificare și generare de semnale de selecție pentru un microprocesor similar cu cel de la adresarea liniară unde $a = 16$ biți. Prin decodificare completă se pot adresa 2^{16} locații de memorie.

- c. **Adresare combinată.** Combinație între cele două moduri descrise anterior.

3.6.2. Magistrala de date

Lărgimea magistralei de date este de obicei multiplu de octet ($d = 8, 16, 32, 64...$). Cele d linii ale magistralei de date au posibilitatea de transmitere bidirecțională a informațiilor (intrare sau ieșire din UCP), cu posibilitatea de trecere în starea de înaltă impedanță (HiZ). Pentru economie de pini, unele microprocesoare multiplexează în timp liniile magistralei de date, astfel că în primul ciclu mașină al fiecărei instrucțiuni pe magistrala de date se pot transmite informații de adresă sau informații de control. În acest caz, în prima parte a ciclului mașină, pe pinii multiplexați, se generează semnalele de adresă sau control, însoțite de un semnal indicator pe magistrala de control care servește pentru memorarea informației în registre externe procesorului. De exemplu la procesoarele Intel 8085 și 8086 pini ai magistralei de date sunt multiplexați pentru a se putea transmite și informație de adresă. Semnalul de control care comandă stocarea adresei într-un registru extern este numit ALE (Address Latch Enable). Apoi, pentru tot restul ciclului instrucțiune liniile magistralei de date transferă date propriu-zise sau instrucțiuni.

3.6.3. Magistrala de control

Magistrală de control conține o diversitate de linii de control și sincronizare, unele fiind doar uni-direcționale (intrare sau ieșire din UCP), iar altele permițând transferul bidirecțional de semnale. De aceea adesea se spune că magistrala de control este bidirecțională. Această magistrală cuprinde semnalele de control cu funcție diferită de la un tip de microprocesor la altul. Cu toate acestea, funcțional, liniile magistralei de control pot fi clasificate în următoarele categorii generale [Sztojanov87]:

- Semnale de control și sincronizare pentru transferuri de date cu memoria și dispozitivele de I/O;
- Semnale de control și sincronizare a cererilor de cedare a controlului magistralelor;
- Semnale de control și sincronizare cu evenimente externe ce generează cereri de întrerupere;
- Semnale indicatoare de stare a UCP;
- Semnale utilitare, cum ar fi reset, clock, alimentare cu tensiune;
- Semnale diverse, specifice tipurilor de microprocesoare și scopului pentru care au fost proiectate. În categoria semnalelor diverse se încadrează de exemplu: intrări testabile prin software, intrări pentru comanda activității pas cu pas a UCP, intrări pentru semnalizarea unor erori, intrări / ieșiri pentru lucrul în sistem multi-procesor etc.

3.6.3.a Semnale de control pentru transferul datelor cu memoria și dispozitivele de I/O

Rolul acestor semnale este de control și sincronizare al transferurilor pe magistrala de date. Sensul de circulație al informației pe magistrala de date este controlat de semnale care indică operație de citire, sau scriere. Sensul, pentru ambele operații sunt privite din punctul de vedere al UCP. În general semnalele de control sunt active pe nivel JOS, în așa fel încât nivelul SUS să corespundă stării inactive a semnalului. Situația este avantajoasă, pentru că durata semnalelor de

control activate este mult mai mică decât durata când ele nu sunt active, iar “în pauza” semnalului se folosește avantajul marginii de zgomot mai mare pentru nivelul logic 1 (SUS în logică pozitivă). Vom nota aceste două semnale cu prescurtările lor din limba engleză, bara de deasupra reprezentând faptul că sunt active pe nivelul logic JOS (“0”):

- \overline{RD} (ReaD, citește), semnal ce indică că se efectuează o operație de citire a datelor de către UCP. De obicei frontul posterior al semnalului de citire (aici frontul crescător) este utilizat de UCP pentru citirea propriu-zisă a datelor depuse pe magistrala de date de către memorie sau un dispozitiv de I/O.
- \overline{WR} (WRite, scrie), semnal ce indică că se efectuează o operație de scriere a datelor de către UCP. De obicei frontul posterior al semnalului de scriere este utilizat de dispozitivul adresat și selectat pentru ciclul de scriere, pentru a stoca datele într-un registru propriu.

La unele microprocesoare (de exemplu microprocesorul Motorola MC68000) există un singur semnal de ieșire (R/\overline{W}), care îndeplinește ambele funcții, pentru nivel “1” făcându-se citire, iar pentru nivel “0” scriere.

Transferurile între UCP și un dispozitiv interlocutor lent (memorie sau port) se fac de obicei asincron, pentru ca UCP să poată lucra la viteza maximă cu dispozitivele rapide și lent cu cele lente. Pentru aceasta sunt necesare semnale pentru realizarea protocolului asincron de comunicație (de tip handshake). Setul minimal de semnale de sincronizare este constituit din următoarele:

- un semnal generat de UCP ce indică dispozitivelor interlocutoare, că UCP a furnizat (pe magistrala de adrese) o informație validă de adresă. La locația cu această adresă se va face scriere sau citire la un moment imediat următor. Acest semnal, împreună cu informațiile de adresă și de sens (\overline{RD} sau \overline{WR}) al circulației pe magistrala de date, produc selecția/activarea dispozitivului interlocutor. Vom nota semnalul de validare al adresei cu \overline{AS} (Address Strobe = validare adresă)
- un semnal recepționat de UCP, care să indice faptul că interlocutorul este gata pentru efectuarea transferului. Vom nota acest semnal cu $GATA$. Acest semnal informează UCP că dispozitivul a livrat cuvântul de date (în cazul unui ciclu de citire) sau că este în măsură să primească date (în cazul ciclului de scriere). Dacă dispozitivul nu poate răspunde într-un anumit interval de timp, corespunzător unui număr de perioade ale impulsului de ceas, el nu poate activa $GATA$, iar UCP va introduce stări suplimentare de așteptare, până când transferul este posibil; semnalul $GATA$ este deci folosit pentru sincronizarea celor doi interlocutori. Recapitulând: dacă $GATA$ este inactiv (= 0 logic), UCP va prelungi ciclul mașină curent, "înghețând" toate celelalte semnale de comandă, prin adăugarea de stări de așteptare (WAIT) până când recepționează $GATA=1$, după care ciclul mașină de scriere (sau citire) se încheie.

Sincronizările cu memoria și cu dispozitivele de intrare-ieșire sunt asemănătoare. La multe dintre calculatoare însă, prin construcție, se cunoaște viteza de lucru a memoriei principale, iar lucrul cu memoria principală poate fi privit ca fiind sincron, nefiind necesar semnalul de tip $GATA$. În această situație transferul se desfășoară strict în cadrul unui interval pre-specificat de timp (ca număr de stări ale UCP). În toate celelalte cazuri transferurile sunt asincrone, UCP prelungindu-și ciclul mașină curent cu stări de așteptare, până când semnalul $GATA$ devine activ sau până când un alt eveniment (excepție sau reset) întrerupe acest ciclu mașină. Generarea sau nu a unui semnal de "GATA" ține de modul de lucru asincron sau sincron. Într-un mod de lucru sincron toate evenimentele se desfășoară în cadrul unui interval specificat de timp.

UCP eșantionează intrarea $GATA$, de obicei într-una din primele stări ale ciclului mașină.

Exemple de semnale de control al transferurilor la diferite microprocesoare se indică în tabelul 3.3. Pentru \overline{RD} și \overline{WR} nu s-au exemplificat denumirile semnalelor la procesoare reale, pentru că în majoritate se păstrează aceeași notație pe care am folosit-o și noi în acest paragraf.

Tabelul 3.3.

Tip microprocesor	Echivalent \overline{AS}	Echivalent GATA
I8080+ I8228	\overline{IOW} , \overline{IOR} , \overline{MEMR} , \overline{MEMW}	\overline{READY}
Z80	\overline{MREQ} , \overline{IOREQ} , \overline{RFSH}	\overline{WAIT}
I8086, I8088, I80x86	\overline{ALE} , $\overline{M/\overline{IO}}$, $\overline{DT/\overline{R}}$, \overline{DEN}	\overline{READY}
MC680x0	\overline{AS} , \overline{UDS} , \overline{LDS}	\overline{DTACK}

La multe dintre microprocesoare semnalul de tip \overline{AS} depinde de tipul ciclului mașină curent executat de procesor. Din exemplele date în tabelul 3.3 se vede că aceste semnale, din categoria validare adresă (\overline{AS}), pot fi separate pentru adresele din spațiul memoriei respectiv spațiul de I/O. Astfel la Intel 8080 cu controller de sistem există scriere/citire pentru dispozitiv de intrare-ieșire (\overline{IOW} ¹⁴ și \overline{IOR} ¹⁵) sau scriere/citire adresată memoriei principale (\overline{MEMW} / \overline{MEMR} ¹⁶). Se observă că la acest procesor semnalele de validare a adresei sunt combinate cu cele ce indică scrierea sau citirea. La microprocesorul Zilog, Z80, în afara semnalelor \overline{RD} și \overline{WR} , există semnale diferite de validare a adreselor pentru memorie (\overline{MREQ} ¹⁷ indică existența unei adrese de memorie pe magistrala de adrese) și spațiu de I/O (\overline{IOREQ} indică existența unei adrese de intrare - ieșire pe magistrala de adrese). Aceste tipuri de semnale pot fi privite și ca semnale indicatoare de stare, pentru că ele indică existența unui ciclu mașină de lucru cu memoria, respectiv cu porturile de I/O. În cazul microprocesorului I8086, în modul minim, există un singur semnal, numit $\overline{M/\overline{IO}}$, care indică transfer cu memoria pentru valoarea 1 logic și cu porturile de I/O pentru 0 logic. Semnalul \overline{ALE} ¹⁸ de la I8086 (mod minim) validează existența informațiilor de adresă pe pinii magistralei de date multiplexate. Pentru că microprocesoarele de 16 și 32 de biți pot face și transferuri pe octeți au fost necesare semnale de tip \overline{BHE} (la Intel 8086, validarea celor 8 linii mai semnificative de pe magistrala de date), sau \overline{UDS} (Upper Data Strobe - validare a octetului mai semnificativ de date) și \overline{LDS} (Lower) la MC68000. La procesorul Motorola 68000 nu există semnal de tipul $\overline{M/\overline{IO}}$ ca la I8086, pentru că 68000 nu are spațiu separat pentru memorie și dispozitive I/O (deși există un semnal numit \overline{VMA} care contribuie la selecția porturilor programabile de 8 biți din familia MC6800). Specific microprocesorului Zilog Z80, (care are inclus în UCP și controlerul de reîmprospătare a memoriei dinamice) este semnalul \overline{RFSH} ¹⁹ care validează adresa de reîmprospătare a DRAM pe biții A6-A0 ai magistralei de adrese.

În figura 3.13 se prezintă un exemplu de sincronizare cu semnal de tip GATA între un microprocesor (UCP) și un dispozitiv lent de memorie, pentru un ciclu de citire.

¹⁴ Input Output Write¹⁵ Input Output Read¹⁶ Memory Write / Read¹⁷ Memory Request¹⁸ Address Latch Enable¹⁹ Refresh

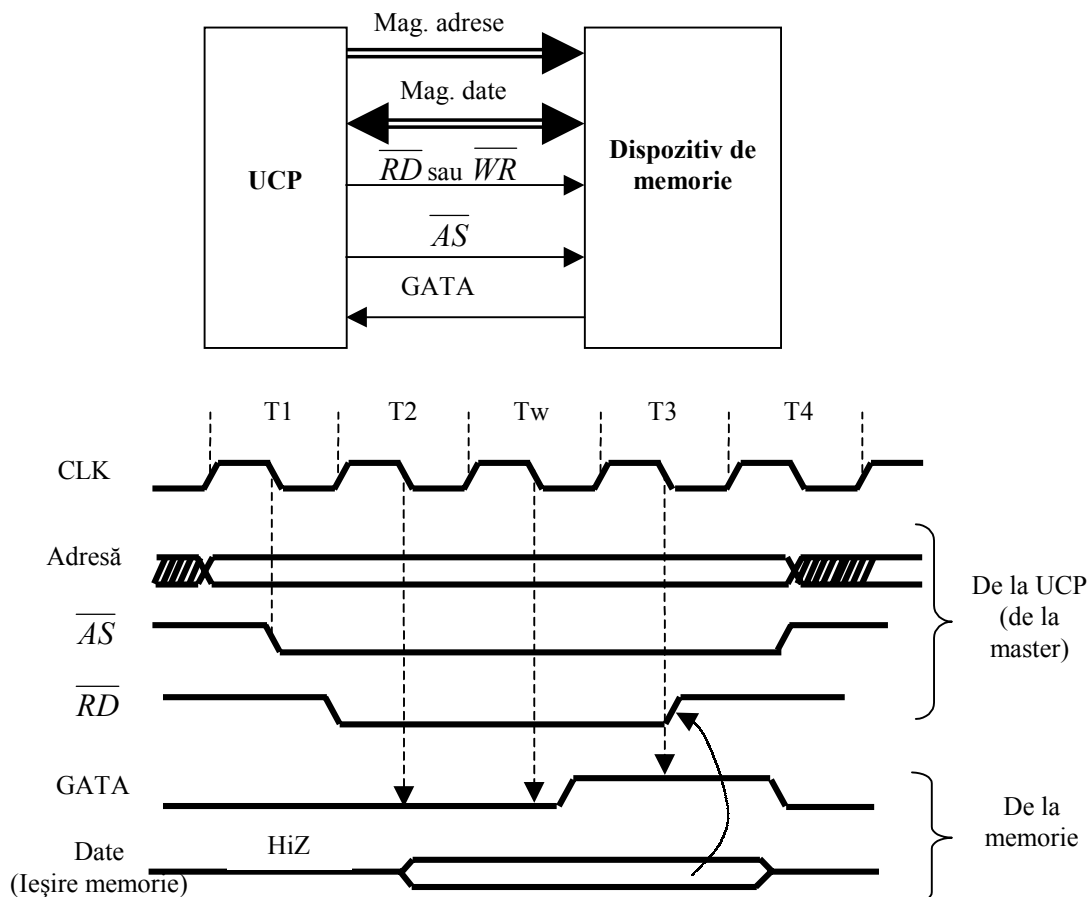


Figura 3.13. exemplificare generală a unui ciclu de citire asincron, care are nevoie de semnale de control de tip handshake. În schema bloc de deasupra s-au reprezentat liniile de semnal la care sunt cuplați cei doi corespondenți, iar în diagrama de timp se prezintă evoluția în timp a semnalelor pe magistrale (notații CLK = impulsuri de ceas, Adresă = magistrală de adrese, Date = ieșirile dispozitivului de memorie către magistrala de date)

Diagrama desfășurării în timp a semnalelor presupune că intrarea GATA este testată de microprocesor începând cu starea T2, pe fiecare front descrescător al impulsului de ceas. Din punctul de vedere al simbolurilor grafice folosite menționăm că pentru semnalele singulare diagrama reprezintă tranzițiile între nivelurile logice “1” (SUS) și “0” (JOS). În cazul reprezentării magistrelor, care conțin mai multe linii de semnal, cele două linii paralele la magistralele de date și adrese reprezintă faptul că informația pe magistrală este stabilă. Intersecția liniilor indică schimbarea informației pe magistrală, iar zonele hașurate arată că informația (stabilă) respectivă este ne-semnificativă pentru ciclul descris de diagramă. Pe ultima linie a diagramei din figura 3.13 s-a reprezentat informația la ieșirile dispozitivului de memorie adresat. Aceste ieșiri sunt cuplate la magistrala de date, dar atunci când dispozitivul nu este selectat ieșirile se găsesc în stare de înaltă impedanță (HiZ) dacă ieșirile sunt de tip TSL. În urma selecției și adresării dispozitivului de memorie, starea la ieșiri se schimbă doar după trecerea timpului de acces la celulele de memorie.

3.6.3.b Semnale de control a cererilor de cedare a controlului magistrelor

Problema cererii / cedării controlului magistrelor se poate pune doar în sisteme de calcul care conțin mai multe dispozitive ce pot controla cele trei tipuri de magistrale. Aceste dispozitive

active pe magistrală (dispozitive master²⁰ care controlează dispozitive slave²¹) pot fi de exemplu: alte procesoare de uz general (UCP), procesoare cu sarcini specifice (de exemplu procesoare de I/O), circuite de tip controller care au și posibilitatea de transfer al datelor direct cu memoria principală (transfer DMA²²). Pentru ca un UCP să poată decide cu privire la cedarea controlului magistralelor, sistemul de arbitraj a controlului magistralelor trebuie să conțină cel puțin două tipuri de semnale:

- un semnal de intrare în UCP care face o *cerere de acces la controlul magistralelor*, semnal pe care-l notăm în continuare cu BR (Bus Request²³).
- un semnal de ieșire din UCP care *confirmă cedarea controlului magistralelor*, notat în continuare BG (Bus Grant). În urma cedării controlului magistralelor, UCP către care s-a lansat cererea BR trece ieșirile sale către magistralele de date, adrese și control (doar o parte dintre semnalele de control) în stare de înaltă impedanță (HiZ). În acest fel se permite controlul (din punct de vedere electric, al nivelurilor logice) liniilor de către cel căruia i s-a cedat.

În tabelul 3.4. se prezintă câteva exemple de semnale echivalente cu semnalele tip pe care le-am numit BR și BG.

Tabelul 3.4.

Tip microprocesor	Echivalent BR	Echivalent BG
I8080, I8085, I8086 în mod minim	HOLD	HLDA
Z80	$\overline{\text{BUSRQ}}$	$\overline{\text{BUSAK}}$
MC680x0	$\overline{\text{BR}}$	$\overline{\text{BG}}$ și $\overline{\text{BGACK}}$

UCP prevăzut cu această pereche de semnale sondează de obicei semnalul de cerere, de tip BR, la sfârșitul fiecărui *ciclu mașină* și cedează controlul magistralelor cât mai curând posibil, cu excepția unor cazuri speciale în care operațiile nu pot fi întrerupte (de exemplu operații prevăzute cu prefixul LOCK la I8086). Întârzierea maximă cu care este servită o cerere de acces la controlul magistralelor este deci un ciclu mașină.

În cazul mai multor cereri de acordare a controlului magistralelor, analiza priorităților se face de obicei cu un circuit de arbitraj centralizată a cererilor de magistrală.

3.6.3.c Semnale de sincronizare cu evenimente externe ce generează cereri de întrerupere

Aceste semnale au rolul sincronizării evenimentelor externe cu UCP. Semnalele de control și sincronizare pentru cereri de întrerupere externă sunt extrem de importante pentru că ele permit ca un dispozitiv periferic să lanseze un semnal de cerere de servicii către UCP (servicii de exemplu care se referă la transferul de date), iar UCP să întrerupă temporar programul rulat la acel moment, să sară la execuția unui program de servire a întreruperii, iar apoi să revină la programul întrerupt. La recepția unei cereri de întrerupere, dacă UCP acceptă întreruperea activității curente, se va informa dispozitivul întreruptor asupra acestui lucru. Ca urmare dispozitivul care a lansat întreruperea va genera (doar pentru întreruperile “vectorizate”) pe magistrala de date un cod de identificare (vector de întrerupere) care să permită dirijarea execuției la subrutina de servire a întreruperii.

²⁰ master (engl.) = stăpân

²¹ slave (engl.) = sclav

²² DMA = Direct Memory Access - Acces direct la memorie.

²³ Bus Request / Grant = Cerere / Acordare de control a magistralelor.

Există două tipuri de cereri de întrerupere hardware:

- întreruperi mascabile, a căror recunoaștere și servire poate fi validată sau invalidată prin setarea unor indicatori de control
- întreruperi nemascabile, al căror efect nu poate fi blocat / mascat, fiind întotdeauna recunoscute.

Un set minim de semnale pentru manevrarea cererilor de întrerupere cuprinde:

- cel puțin un semnal de intrare în UCP, reprezentând o **cerere de întrerupere**, prin care dispozitivul ce a lansat cererea așteaptă servicii de la UCP (notat în continuare cu INT). La unele microprocesoare există mai multe intrări de cerere de întrerupere.
- cel puțin un semnal de ieșire din UCP care reprezintă **confirmarea acceptării întreruperii** (semnal notat în continuare cu INTA) La multe dintre microprocesoare acest semnal are și funcția de control pentru citirea vectorului de întrerupere depus de dispozitivul întreruptor pe magistrala de date.

Câteva exemple de semnale de tip INT și INTA se prezintă în tabelul 3.5.

Tabelul 3.5.

Tip microprocesor	Echivalent INT	Echivalent INTA
I8080 și I80x86	INTR (mascabil), NMI (nemascabil)	INTA
Z80	$\overline{\text{INT}}$ (mascabil), $\overline{\text{NMI}}$ (nemascabil)	$\overline{\text{IORQ}} + \overline{\text{M1}}$
MC680x0	IPL2, IPL1, IPL0	FC2, FC1, FC0

3.6.3.d Semnale indicatoare de stare a UCP

Acestea sunt de obicei semnale, sau combinații de semnale de ieșire care indică starea în care se găsește microprocesorul. Starea automatului este importantă pentru diferite circuite suplimentare conectate la interfața microprocesorului cu exteriorul, aceste circuite având diverse de funcții de control în ceea ce privește magistralele, memoria cache externă și memoria principală. Semnalele de stare sunt extrem de diverse, de la tip la tip de microprocesor dar funcțiile lor pot fi clasificate după informația furnizată. Astfel semnalele de stare pot indica:

- tipul ciclului mașină curent. Informația este utilă circuitelor controller de magistrală, care pe baza stării citite furnizează semnale specifice de control. De exemplu, la Intel 8086 în mod “maxim” semnalele S0-S2 indică tipul ciclului mașină curent. La MC68000 există trei linii de ieșire numite “Function Codes”, FC2 - FC0 care informează exteriorul cu privire la starea procesorului.
- informații despre registrele interne implicate în calculul de adresă. De exemplu la I8086 în mod “maxim” semnalele S3 și S4 indică registrele segment utilizate pentru adresa curentă, iar semnalele QS1 și QS0 dau informații cu privire la starea cozii de instrucțiuni.
- informații de sincronizare cu alte module master de magistrală (de exemplu de tip *lock*)
- starea memoriei tampon (cache, coadă) internă, informație utilă circuitului controller de memorie și cache extern.
- speciale, pentru lucrul cu coprocesoare aritmetice externe. De exemplu cerere și acceptare de transfer operand (PEREQ/PEACK – la Intel 80286), test busy (BUSY la I8086), informare de

eroare ($\overline{\text{BERR}}$ la 68000).

3.6.3.e Semnale utilitare

În această categorie sunt incluse mai multe tipuri de semnale utilitare, dintre care amintim:

- semnale care aduc procesorul într-o stare predeterminată (RESET),
- semnalele de ceas (CLOCK)
- alimentarea cu tensiune.

RESET

Semnalul de tip RESET este prezent la toate microprocesoarele și el aduce conținutul registrelor microprocesorului într-o stare prefixată prin proiectare. În această stare predeterminată contorul de program (PC) se inițializează la o valoare fixă (de obicei la zero), iar întreruperile mascabile sunt invalidate. Scopul principal al acestui semnal de intrare în UCP este ca să se știe cu precizie starea în care se găsește procesorul la alimentarea cu tensiune. Din această cauză la această intrare se conectează circuite R-C de tip Power-on Reset (Resetare la alimentarea circuitului). Chiar dacă pentru majoritatea microprocesoarelor acest semnal este doar intrare pentru UCP, există și excepții. Astfel, la microprocesorul Motorola MC68000 linia de $\overline{\text{Reset}}$ este bidirecțională. Dacă este folosită ca intrare și este forțată la “0” logic din exterior, atunci când intrarea $\overline{\text{Halt}}$ e activă în același timp, se produce inițializarea internă a microprocesorului. Aceeași linie este folosită ca ieșire, pentru inițializarea dispozitivelor de I/O, atunci când microprocesorul execută instrucțiune “Reset” instrucțiune care însă nu modifică și starea internă a microprocesorului.

Semnale de ceas

Semnalele de ceas sunt generate de către un oscilator intern sau extern pe baza unui cristal de cuarț extern procesorului. La unele microprocesoare circuitele de ceas pot efectua și “sincronizarea” unor semnale externe, cu caracteristică asincronă, cum ar fi: RESET și READY. Unele din circuitele oscilator de ceas pot avea și alte funcții. De exemplu, la circuitul I8224 folosit pentru Intel 8080, și la I8284 al lui Intel 8086 se generează și semnale RESET pentru celelalte unități funcționale ale calculatorului. Cristalele cu cuarț au frecvențe de oscilație diferite în funcție de direcția de prelucrare față de rețeaua cristalină și în funcție de grosime. Pot exista cristale echivalente cu circuite rezonante (RLC) serie și paralel.

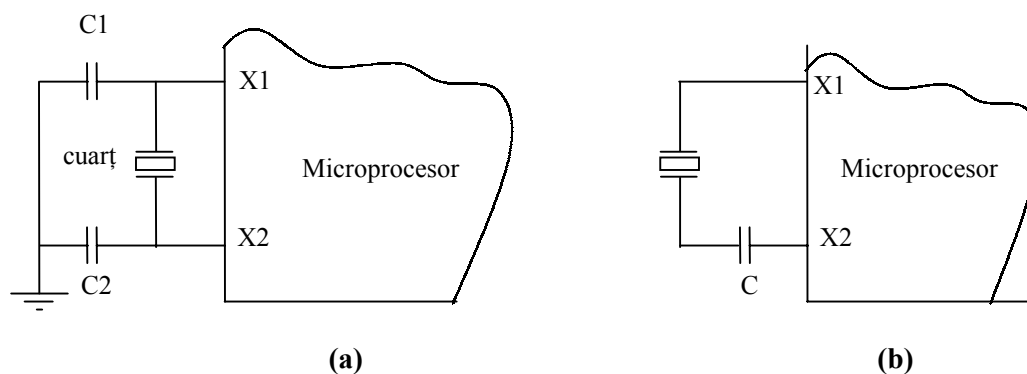


Figura 3.14. Exemplu de conectare a cristalului cu cuarț cu schemă echivalentă paralel (a), respectiv serie (b).

Producătorii specifică dacă cristalele sunt rezonante serie sau paralel. Cristalele cu cuarț sunt construite cu toleranțe (frecvență și stabilitate a frecvenței) ce pot ajunge până la 0,002% - 0,005%. Pentru sistemele de calcul o este acceptabilă o toleranță în jurul valorii de 0,01%. În figura 3.14 se indică două moduri de conectare a cristalelor de curați la capsula microprocesorului. Prima, din figura 3.14 (a) este legarea corespunzătoare microprocesoarelor cu oscilator intern. Tipic fiecare dintre condensatoarele acestei scheme are valoarea de 10 - 30 pF. În figura 3.14 (b) se indică conectarea unui cuarț cu circuit oscilant echivalent de tip serie. Capacitatea introdusă în serie cu cristalul previne polarizarea în curent continuu de către circuitele interne microprocesorului.

3.7. REPREZENTAREA NUMERELOR ÎN CALCULATOR

Modul în care se reprezintă datele numerice într-un calculator are legătură directă cu structura UCP, având influență asupra dimensiunii registrelor de uz general locale, al dimensiunii magistralei interne și al complexității unității de execuție. În funcție de poziția virgulei, reprezentările binare folosite în calculator pot fi:

⇒ reprezentare în virgulă fixă;

⇒ reprezentare în virgulă mobilă.

Virgula (punctul zecimal sau binar, în notația engleza) *nu se reprezintă fizic nicăieri* în registrele calculatorului, dar poziția sa este cunoscută (stabilită) pentru fiecare dintre modurile de reprezentare

Reprezentarea în virgulă fixă este folosită de toate calculatoarele numerice de uz general. Majoritatea procesoarelor de uz general lucrează cu numere întregi în virgulă fixă, datorită avantajelor în ceea ce privește structura hardware a circuitelor pentru operații aritmetice și logice. Reprezentarea în virgulă fixă cu numere subunitare are avantaje din punctul de vedere al operațiilor de înmulțire, pentru că această operație nu va duce la depășirea capacității de reprezentare pentru produs.

Reprezentarea în virgulă mobilă se face pentru numere reale. Nu toate calculatoarele conțin însă suportul hardware necesar pentru aritmetica în virgulă mobilă, dar acestea pot efectua operațiile cu numere în virgulă mobilă prin emulare (interpretare) cu ajutorul unor programe de emulare.

3.7.1. Reprezentarea cu virgulă fixă

Reprezentarea se numește cu virgulă fixă, pentru că poziția acesteia este fixă față de șirul de biți prin care se reprezintă data numerică. Reprezentarea numerelor în virgulă fixă se face, în general, sub formă de numere întregi cu semn sau fără semn. Există de asemenea posibilitatea de a reprezenta datele în virgulă fixă sub forma de numere subunitare. Dacă procesorul calculatorului lucrează în virgulă fixă cu numere subunitare, la introducerea datelor, fiecărui număr i se atașează un factor de scară, care să-l transforme într-un număr subunitar. Acest aliniament este făcut de programul de încărcare în memorie. În compilatoarele scrise pentru virgulă fixă, factorul de scară este tratat de compilator, astfel încât utilizatorul limbajelor de nivel înalt nu trebuie să ia în considerare acest aspect la efectuarea operațiilor.

Pentru **numerele cu semn**, întotdeauna pe prima poziție binară din partea stânga se reprezintă semnul, folosindu-se convenția:

$S = 0$, semn pozitiv ($N \geq 0$)

$S = 1$, semn negativ ($N < 0$)

(3.10)

Pentru numere *întregi cu semn*, reprezentate pe n biți, se va folosi notația:

$$N = b_{n-1}b_{n-2} \dots b_2b_1b_0 \quad (3.11)$$

Bitul b_{n-1} este bitul de semn, bitul b_{n-2} este cel mai semnificativ bit (MSb, cu ponderea 2^{n-2}), iar b_0 este bitul cel mai puțin semnificativ (LSb). Se consideră că virgula are poziția fixă imediat în dreapta LSb (b_0).

Pentru numerele *subunitare cu semn*, reprezentate pe n biți, se va folosi notația:

$$N_s = b_0b_{-1}b_{-2}b_{-3} \dots b_{-(n-1)} \quad (3.12)$$

unde b_0 este bitul de semn, b_{-1} este bitul cel mai semnificativ (MSb, cu ponderea 2^{-1}), iar $b_{-(n-1)}$ este bitul cel mai puțin semnificativ (LSb cu ponderea $2^{-(n-1)}$). Se consideră că virgula se află între bitul de semn și MSb (b_{-1}).

3.7.2. Coduri binare folosite pentru reprezentarea numerelor cu semn

Se folosesc trei tipuri de coduri binare:

- cod *direct* (numit și cod *mărime / modul și semn*, notat pe scurt **MS**)
- cod *invers* (numit și cod *complement față de 1*, notat pe scurt **C1**)
- cod *complementar* (numit și cod *complement față de 2*, notat pe scurt **C2**)

Se va face o descriere sintetică a celor 3 moduri de codificare considerând că numerele sunt reprezentate cu n biți. De asemenea specificăm încă de la început că *pentru numere pozitive toate cele trei moduri de reprezentare sunt identice* (conduc la aceleași numere în binar).

3.7.2.a Reprezentarea numerelor cu semn în cod direct (cod mărime și semn, MS)

Mod de reprezentare: Modulul numărului are aceeași reprezentare atât pentru numere pozitive cât și pentru cele negative, iar semnul ia valoarea 0 pentru numere pozitive, respectiv 1 pentru numere negative.

numere întregi	numere subunitare
$N = b_{n-1} b_{n-2} \dots b_1 b_0$	$N_s = b_0 b_{-1} b_{-2} b_{-3} \dots b_{-(n-1)}$
pentru $N > 0$, $b_{n-1} = 0$ pentru $N < 0$, $b_{n-1} = 1$	pentru $N_s > 0$, $b_0 = 0$ pentru $N_s < 0$, $b_0 = 1$

(3.13)

Domeniul de reprezentare:

numere întregi	numere subunitare
$-(2^{n-1}-1) \leq N \leq 2^{n-1}-1$	$-(1-2^{1-n}) \leq N_s \leq (1-2^{1-n})$

(3.14)

Calculul valorii zecimale pentru numărul reprezentat în binar în MS:

numere întregi	numere subunitare
$N _{10} = (-1)^{b_{n-1}} \sum_{i=0}^{n-2} b_i 2^i$	$N_s _{10} = (-1)^{b_0} \sum_{i=1}^{n-1} b_{-i} 2^{-i} \quad (3.15)$

Avantaje ale codului MS:

Codificarea în MS prezintă avantaje din punctul de vedere al operațiilor de înmulțire și împărțire, algoritmi de calcul și rutinele sau circuitele de implementare cablată rezultând cu o structură simplă. De asemenea codificarea în MS se face foarte simplu pentru numerele negative, doar prin modificarea bitului de semn.

Dezavantaje ale codului MS:

Codul prezintă dezavantaje din punctul de vedere al operațiilor de adunare și scădere. Asta în primul rând pentru că numerele pozitive și negative trebuie tratate diferit în operații și ca urmare rezultă algoritmi de adunare și scădere mai complecși decât în alte moduri de codificare.

Un alt dezavantaj îl reprezintă faptul ca există două reprezentări pentru numărul zero:

000000...0 și 100000...0

Exemplu (pentru n=8):

Pentru n=8, domeniul de reprezentare este:

$|N| \leq 127$ pentru întregi și $|N_s| \leq 1 - 2^{-7}$ pentru subunitare.

zecimal	binar
N = +6 sau N = +6/128	0000 0110 (primul zero este bitul de semn)
N = -6 sau N = -6/128	1000 0110 (primul unu este bitul de semn)

Se observă că numerele binare 00000110, respectiv 10000110 pot fi interpretate și ca +6/128, (pentru alte valori ale lui n valoarea este $+6/2^{(n-1)}$) respectiv -6/128, dacă reprezentarea se face cu numere subunitare.

Calculul valorii zecimale pentru N = 1000 0110:

N = 1000 0110:

$$N|_{10} = (-1)^1 (0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -6$$

sau

$$\begin{aligned} N|_{10} &= (-1)^1 (0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 0 \cdot 2^{-7}) = \\ &= - (1/32 + 1/64) = - 6/128 \end{aligned}$$

3.7.2.b Reprezentarea numerelor cu semn în cod invers (cod complement față de 1, C1)

Mod de reprezentare: Pentru numere pozitive reprezentarea este identică cu cea făcută în MS. Pentru numere negative, bitul de semn este 1, iar biții ce reprezintă valoarea absolută a numărului sunt formați prin complementarea bit cu bit a reprezentării în cod direct.

	<u>numere întregi</u>	<u>numere subunitare</u>
<i>numere pozitive</i>	$N = 0 \ b_{n-2} \dots b_1 \ b_0$	$N_s = 0 \ b_{-1} \ b_{-2} \ b_{-3} \dots b_{-(n-1)}$
<i>numere negative</i>	$N = 1 \ a_{n-2} \dots a_1 \ a_0$	$N_s = 0 \ a_{-1} \ a_{-2} \ a_{-3} \dots a_{-(n-1)}$
	unde $a_i = 1 - b_i$, $(i=0..n-2)$	unde $a_i = 1 - b_i$, $(i=1..n-1)$

(3.16)

Reprezentarea în cod invers a unui număr N , binar negativ, se poate face în două moduri:

- se completează toți biții reprezentării modulului numărului ($|N|$) inclusiv bitul de semn;
- pe baza relațiilor generale:

<u>numere întregi</u>	<u>numere fracționare</u>
$(N)_{C1} = 2^n - N - 1$	$(N_f)_{C1} = 2 - N_s - 1/(2^{(n-1)})$

(3.17)

Domeniul de reprezentare:

<u>numere întregi</u>	<u>numere subunitare</u>
$-(2^{n-1}-1) \leq N \leq 2^{n-1}-1$	$-(1-2^{1-n}) \leq N_s \leq (1-2^{1-n})$

(3.18)

Calculul valorii zecimale pentru numărul reprezentat în binar în C1:

<u>numere întregi</u>	<u>numere subunitare</u>
$N _{10} = -a_{n-1}(2^{n-1} - 1) + \sum_{i=0}^{n-2} a_i 2^i$	$N_s _{10} = -a_0 \left(2^0 - \frac{1}{2^{n-1}} \right) + \sum_{i=1}^{n-1} a_{-i} 2^{-i}$

(3.19)

(relațiile au fost scrise în acest fel pentru a putea fi folosite și pentru numerele pozitive)

Avantaje și dezavantaje ale codului C1:

Reprezentarea este foarte ușor de realizat în hardware, dar algoritmi de înmulțire și împărțire pentru numere în C1 sunt mai complecși decât la MS. Codul prezintă unele avantaje pentru adunare și scădere, dar la aceste operații se preferă codul complementar.

Un alt element dezavantajos este faptul că există două reprezentări pentru zero (plus și minus zero):

000000...0 și 111111...1

Exemplu (pentru n=8):

<u>zecimal</u>				<u>binar</u>
N =	+6	sau	N = +6/128	0000 0110
N =	-6	sau	N = -6/128	1111 1001

$$\begin{array}{rcl}
 (-6)_{C1} = 2^8 - |N| - 1 & = & \begin{array}{r} 1111\ 1111 \\ 0000\ 0110 \\ \hline 1111\ 1001 \end{array}
 \end{array}$$

Calculul valorii zecimale pentru $(N)_{C1} = 1111\ 1001$:

$$\begin{aligned}
 N|_{10} &= (-1) * 2^7 + 1 + (1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0) = -6 \\
 \text{sau} \\
 N|_{10} &= -1 + 1*2^{-7} + (1*2^{-1} + 1*2^{-2} + 1*2^{-3} + 1*2^{-4} + 0*2^{-5} + 0*2^{-6} + 1*2^{-7}) = \\
 &= -1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/128 + 1/128 = -6/128
 \end{aligned}$$

3.7.2.c Reprezentarea numerelor cu semn în cod complementar (cod complement față de 2, C2)

Mod de reprezentare: Pentru numere pozitive reprezentarea este identică cu cea făcută în MS. Pentru reprezentarea în cod complementar a unui număr N , binar negativ, se pot folosi variantele descrise în tabelul de mai jos:

varianta	numere întregi	numere subunitare
a	$(N)_{C2} = (N)_{C1} + 1$ sau prin înlocuirea expresiei pentru C1:	$(N_s)_{C2} = (N_s)_{C1} + 1/(2^{(n-1)})$
b	$(N)_{C2} = 2^n - N $	$(N_s)_{C2} = 2 - N_s $
c	Folosind reprezentarea modulului numărului negativ de codificat, pornind de la dreapta (de la LSb) către stânga se copiază toți biții inclusiv primul bit 1 întâlnit, iar apoi se completează toți ceilalți biți inclusiv bitul de semn.	

(3.20)

(3.21)

Note:

- La operațiile de scădere sau adunare pentru calcularea codului se consideră și transportul sau împrumutul.
- Relațiile de la varianta b indică de ce codul poartă numele complement față de 2.
- Metoda de la varianta c este des folosită în codificarea manuală și de asemenea ușor de implementat cu porți logice.

Domeniul de reprezentare:

numere întregi	numere subunitare
$-2^{n-1} \leq N \leq 2^{n-1}-1$	$-1 \leq N_s \leq 1-2^{1-n}$

(3.22)

Calculul valorii zecimale pentru numărul reprezentat în binar în cod C2:

(3.23)

b)

$$(-6)_{C2} = 2^8 - |6|$$

$$\begin{array}{r} 1\ 0000\ 0000 - \\ 0000\ 0110 \\ \hline 1111\ 1010 \end{array}$$

c)

pe reprezentarea lui +6 =

0000 0110

←

1111 1010

↑

primul 1 copiat, ceilalți complementați

Calculul valorii zecimale pentru $(-6)_{C2} = 1111\ 1010$:

$$N|_{10} = (-1) * 2^7 + (1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0) = -6$$

sau

$$N|_{10} = -1 + (1*2^{-1} + 1*2^{-2} + 1*2^{-3} + 1*2^{-4} + 0*2^{-5} + 1*2^{-6} + 0*2^{-7}) =$$

$$= -1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/64 = -6/128$$

3.8. REPREZENTAREA NUMERELOR ÎN VIRGULĂ MOBILĂ

Pentru stocarea și prelucrarea numerelor reale în calculator se folosește reprezentarea în virgulă mobilă. Denumirea provine de la faptul că virgula nu are o poziție fixă față de șirul de biți ce reprezintă valoarea numărului. Poziția virgulei se poate modifica fie ca să permită efectuarea de operații aritmetice și logice cu numere reale, fie pentru reprezentarea standardizată, în virgulă mobilă, a numerelor reale.

În calculatorul numeric, pentru fiecare număr real N trebuie reprezentate fiecare din componentele următoarei relații:

$$N = M \times B^E \quad (3.24)$$

unde

M = *mantisa*, reprezentată în calculatorul numeric ca un număr binar subunitar cu semn;

E = *exponentul*, reprezentat ca un număr întreg cu semn;

B = *baza*, este 2 sau o putere a lui 2, nu este reprezentată în calculatorul numeric, dar se ține cont de valoarea bazei atunci când se efectuează operații aritmetice. La un anumit calculator numeric valoarea bazei este aceeași pentru toate numerele reale reprezentate.

Reprezentarea numerelor în calculator se face prin cuvinte binare cu lungime egală cu lungimea registrelor de stocare. Precizia de reprezentare a unui număr real este dată în primul rând de numărul de biți folosiți pentru reprezentarea Mantisei. Domeniul maxim de reprezentare este determinat de valoarea adoptată pentru Bază și de numărul de biți folosiți pentru a reprezenta Exponentul.

Spre deosebire de numerele întregi, numerele reale se reprezintă cu o anumită aproximație în calculator. Dacă pentru reprezentarea numerelor reale în calculator se folosesc n biți, se vor putea reprezenta maxim 2^n numere, dintr-un anumit interval continuu $\pm R$ al numerelor reale. Ca urmare, numerele reale sunt reprezentate cu o anumită eroare, determinată de numărul limitat de biți folosit pentru reprezentare. *Eroarea de reprezentare* depinde de distanța dintre două numere succesive

reprezentabile cu cei n biți. Toate numerele reale cuprinse între cele două valori vor trebui aproximate prin una din cele două valori. Dacă baza folosită implicit pentru reprezentare nu este 2 ci 4 sau 16, dezavantajul este că numerele reprezentabile vor fi și mai rar distribuite pe intervalul $\pm R$ deși există avantajul creșterii domeniului de reprezentare cu n biți.

Pentru reprezentarea mantisei numerelor în virgulă mobilă se folosește *forma normalizată* în care, pentru reprezentarea în mărime și semn, prima poziție binară după virgulă este diferită de zero. Normalizarea restrânge mărimea valorii absolute a mantisei binare la domeniul:

$$\frac{1}{2} \leq |M| < 1 \quad (3.25)$$

Normalizarea în binar este ușor de făcut, prin deplasarea mantisei către dreapta sau stânga și incrementarea respectiv decrementarea corespunzătoare a exponentului.

De exemplu, numărul cu mantisa normalizată $0.11011000 \cdot 2^{10}$ poate fi scris:

$$0.11011000 \cdot 2^{10} = 0.011011000 \cdot 2^{11} = 0.0011011000 \cdot 2^{12}$$

Dacă, de exemplu, considerăm $B=2$, exponentul E reprezentat cu semn pe 8 biți, iar mantisa normalizată un număr cu semn reprezentat pe 24 de biți în cod mărime și semn (deci numărul stocat în calculator va avea $8 + 24 = 32$ biți), domeniul maxim de reprezentare va fi cuprins între -2^{127} și $+2^{127}$.

În valoare absolută, cel mai mic număr reprezentabil în calculator este

$$N_{\text{minim_reprezentabil}} = 0.100\dots 0 \cdot 2^{-127}$$

iar cel mai mare număr reprezentabil este

$$N_{\text{maxim_reprezentabil}} = 0.111\dots 1 \cdot 2^{+127}$$

Reprezentând grafic toate numerele pe o scară, la care în partea de sus se reprezintă numerele mari, iar jos cele mai mici, ca în figura 3.16. se pot face câteva observații importante:

- Mantisele se reprezintă pe 24 biți, din care 23 sunt rezervați pentru mărime.
- Toate mantisele reprezentate în cod direct în exemplul din figura 3.16 sunt normalizate (primul bit după virgulă este 1). Dacă în urma efectuării operațiilor aritmetice rezultatul este în afara domeniului $[-N_{\text{maxim_reprezentabil}}, +N_{\text{maxim_reprezentabil}}]$ se spune că s-a produs o depășire a capacității de reprezentare (pentru cei 32 de biți din exemplu). Depășirea poate fi superioară (mai mare decât numărul maxim reprezentabil) sau inferioară. Programul în care se produce depășirea se întrerupe, iar evenimentul este tratat de o rutină (de servire a cazului de excepție "depășire"), care va transmite un mesaj către utilizator. Dacă însă rezultatul se încadrează în intervalul cu numere prea mici pentru a fi reprezentate cu mantise normalizate $[-N_{\text{minim_reprezentabil}}, +N_{\text{minim_reprezentabil}}]$ numărul este aproximat, de obicei, cu zero.
- Reprezentarea lui zero pune unele probleme, pentru că mantisa poate rezulta zero (nenormalizată) dar exponentul său poate avea orice valoare pentru că $0 \times B^E = 0$ pentru oricare valoare a lui E . Din punctul de vedere a circuitelor care testează dacă un număr este egal cu zero, ar fi avantajos ca să avem aceeași reprezentare ca la întregi (succesiune numai de biți 0 în toate câmpurile: semn, exponent, mantisă). Valoarea zero este considerată a fi cea care corespunde la exponentul maxim negativ (-127 în exemplul de mai sus).
- De asemenea, pune probleme de reprezentare faptul că este necesar să se reprezinte două semne pentru fiecare număr (semnul mantisei și semnul exponentului).

O cale de rezolvare a problemei reprezentării numărului zero este folosirea *exponentului deplasat* (sau *caracteristică*, notată în continuare cu C). Dacă în exemplul prezentat mai sus deplasarea se face conform relației:

$$C = E + 127 \quad (3.26)$$

caracteristica poate lua valori între 0 și 254.

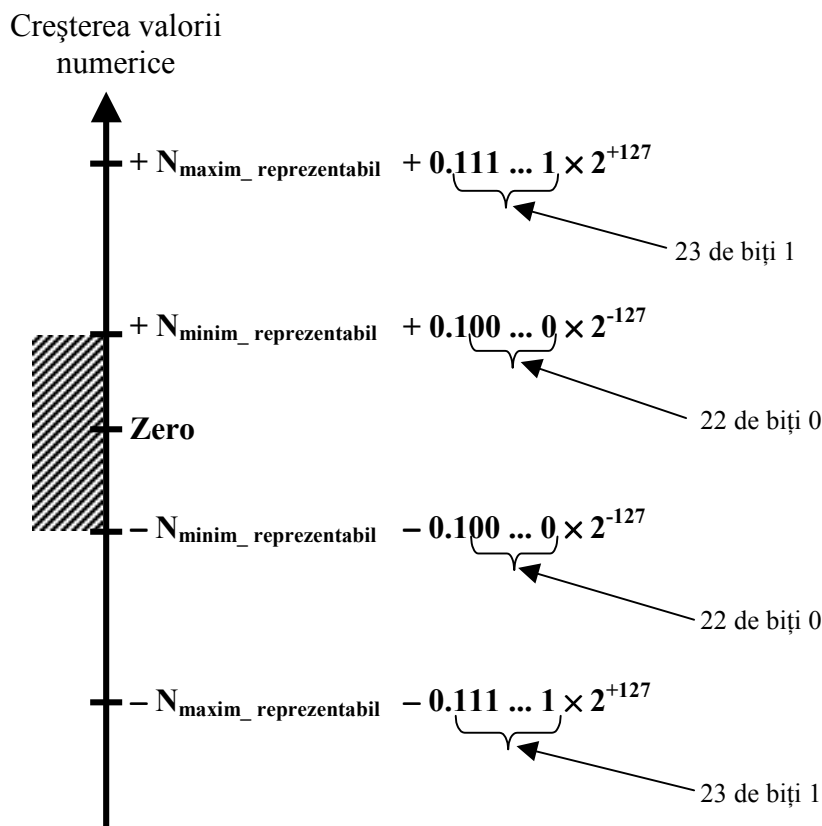


Figura 3.16. Scară a numerelor reale pentru reprezentarea pe 32 de biți, din care pentru mantisă sunt rezervați 24 biți = 23 mărime + 1 semn.

În acest caz, formatul reprezentării pe 32 biți în virgulă mobilă poate fi cel din figura 3.17.

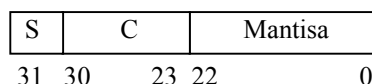


Figura 3.17. Reprezentare în virgulă mobilă pe 32 biți cu exponent deplasat caracteristică (C) și semnul mantisei (S).

În plus, dacă mantisa este reprezentată în cod direct și este normalizată, primul bit al acesteia este 1 pentru toate numerele. Acest prim bit poate să nu mai fie reprezentat (bit ascuns, "hidden bit" - HB). Ca urmare, în formatul pe 32 de biți, mantisa poate fi reprezentată pe 24 de biți (23 biți reprezentați efectiv + 1 HB) precizia de reprezentare crescând de două ori. De asemenea, faptul că bitul 1 este ascuns, permite numerelor pozitive foarte mici ($+0.1 \times 2^{-127}$, reprezentat cu HB) să aibă numai zerouri pe toți cei 32 de biți, deci reprezentarea numărului zero este identică cu cea de la reprezentarea întregilor, putându-se folosi aceleași tipuri de circuite de detecție a rezultatului nul.

Ca urmare, la efectuarea operației de normalizare, virgula mobilă este mutată până când în stânga sa se găsește un bit 1, iar ceilalți biți se găsesc în dreapta virgulei.

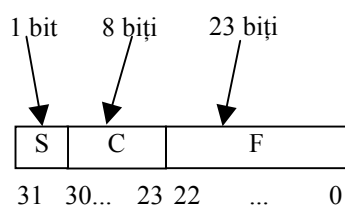
Formatul de reprezentare în virgulă mobilă este standardizat de către standardul IEEE 754, apărut în 1980 și completat ulterior. Până la apariția sa, codurile numerice în virgulă mobilă variaua de la o familie de calculatoare la alta, făcând dificilă portabilitatea programelor între diferite mașini, căci apăreau dificultăți în domenii ca: erori de aproximare, tratarea depășirilor superioare și

inferioare și tratarea altor condiții de excepție. Standardul se referă la reprezentările pe 32 biți și pe 64 biți. În oricare dintre formate standardul impune folosirea *exponentului deplasat*, a *formei normalizate* pentru mantisă, a *bitului ascuns*, iar baza B este considerată a fi egală cu 2. Bitul ascuns nu se reprezintă în formatul binar în virgulă mobilă, dar de valoarea sa se ține cont atunci când se lucrează cu numerele (unitățile aritmetice și logice țin cont de bitul ascuns). Modul standardizat de reprezentare poate fi rezumat astfel:

- *pentru 32 de biți*: 1 bit de semn, 8 biți pentru exponentul deplasat (+127) și 23 de biți pentru mantisă). Implicit se știe ca primul bit (înainte de virgulă) al mantisei nu este reprezentat și el are valoarea 1.
- *pentru 64 de biți*: 1 bit de semn, 11 biți pentru exponentul deplasat (+1023) și 52 de biți pentru mantisă). Implicit se știe ca primul bit 1 al mantisei nu este reprezentat și el are valoarea 1.

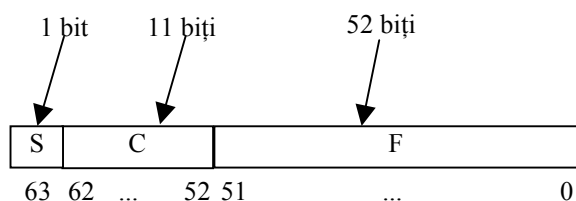
Un bun exemplu de utilizare a standardului IEEE pentru reprezentarea în virgulă mobilă este circuitul coprocesor Intel 8087 al familiei i8086. El folosește un format cu maximum 80 de biți, având registre interne cu această dimensiune. Numerele pot fi reprezentate în formatele 32, 64, sau 80 de biți. În figura 3.18. s-au notat cu S semnul, cu C exponentul deplasat și cu F fracția, prin care înțelegem partea de după virgulă a mantisei, deci fără bitul ascuns:

Formatul real temporar se folosește la familia de coprocesoare Intel x87, dar și pentru emulările software, pentru reprezentarea temporară internă. La acest format temporar nu se mai impune reprezentarea cu bit ascuns.



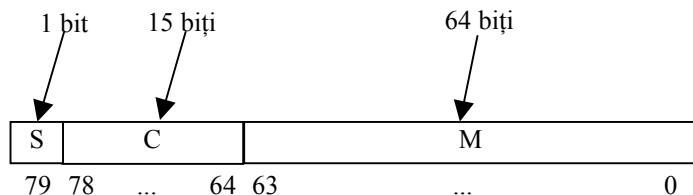
Real scurt:

- Reprezentare pe 32 biți
- Precizie mantisă: 24 biți cu HB
- Caracteristică $C = E + 127$



Real lung:

- Reprezentare pe 64 biți
- Precizie mantisă: 53 biți cu HB
- Caracteristică $C = E + 1023$



Real temporar:

- Reprezentare pe 64 biți
- Precizie mantisă: 64 biți fără HB
- Caracteristică $C = E + 16383$

Figura 3.18. Formatele numerelor reale reprezentate în virgulă mobilă pentru coprocesorul Intel 8087, conform standardului IEEE 754.

În standardul IEEE de reprezentare se indică și modul de *interpretare al rezultatelor*, prin definirea unor situații de excepție. Excepțiile din standardul IEEE (caracteristică egală cu zero sau mai mare de 254 pentru reprezentarea pe 32 de biți, respectiv 2046 pe 64 de biți) sunt definite în așa fel ca să seteze indicatori de condiții ai procesorului gazdă și să genereze ca urmare evenimente ce trebuie tratate ca să nu existe erori de reprezentare sau de precizie. Dacă pentru un număr normalizat, mantisa este zero, iar exponentul are cea mai mică valoare posibilă, este reprezentată valoarea zero. Dacă însă exponentul deplasat are cea mai mică valoare posibilă ($C=0$), iar mantisa are o valoare diferită de zero, mantisa nu este normalizată (este ne-normalizată)²⁴, iar biții stocați ai mantisei dau mărimea acesteia. Acest număr este prea mic pentru a fi reprezentat. Cele mai multe din procesoarele ce lucrează în virgulă mobilă fac, în acest caz, aproximarea la zero a numărului, eroarea fiind proporțională cu mărimea mantisei ne-normalizate.

În figura 3.19 se prezintă modul de tratare al excepțiilor în standardul IEEE 754 pentru numere reale reprezentate pe 32 de biți. Simbolul NAN (*Not-A-Number*, rezultatul nu este un număr) este folosit pentru toate operațiile care conduc la erori, de exemplu împărțire la zero, sau extragere de radical dintr-un număr negativ, pe baza interpretării simultane a fracției diferite de zero $F \neq 0$ (unde Mantisa $M = 1.F$, bitul 1 din fața virgulei fiind bitul ascuns) și a exponentului deplasat (caracteristică) cu valoare $C = 255$ (la formatul pe 32 de biți) respectiv $C = 2047$ (la formatul pe 64 de biți).

Atunci când se produce o depășire a capacității de reprezentare (numărul este prea mare pentru a putea fi reprezentat pe cei n biți) rezultatul este definit ca fiind $+$ sau $-$ infinit ($\pm \infty$) și identificarea se face de asemenea pe baza valorilor fracției și exponentului deplasat ($C = 255$ respectiv $C = 2047$ și $F = 0$).

Caracteristica	Fracție	Interpretare rezultat
$C = 255$	$F \neq 0$, cu HB, (mantisă normalizată)	simbol de eroare NAN (nu este număr)
$C = 255$	$F = 0$, cu HB, (mantisă normalizată)	depășire $N = (-1)^S \infty = \pm \infty$
$0 < C < 255$	Mantisă normalizată, ($M = 1.F$)	Număr real, $N = (-1)^S \times 2^{C-127} \times (1.F)$
$C = 0$	$F = 0$, cu HB, (mantisă normalizată)	Număr = 0
$C = 0$	$F \neq 0$, (mantisă ne-normalizată)	Număr prea mic pentru a fi reprezentat

Figura 3.19. Modul de interpretare al excepțiilor în standardul IEEE 754 pentru numere reale reprezentate pe 32 de biți.

Alt aspect definit de standardul IEEE, cu implicații mari în ceea ce privește proiectarea hardware, sunt regulile de rotunjire. Standardul definește patru moduri de rotunjire:

- rotunjire la valoarea cea mai apropiată (*round to nearest*)
- rotunjire către valoarea mai apropiată de zero (*round toward 0*)

²⁴ Termenul de ne-normalizat (denormalized în limba engleză) se referă la numerele care pot avea orice valoare pentru C , dar mantisa nu poate fi adusă la forma normalizată, cu bit ascuns, pentru că se depășesc limitele de reprezentare pentru caracteristică (ar trebui să rezulte o caracteristică negativă).

- rotunjire către $+\infty$ (*round toward $+\infty$*)
- rotunjire către $-\infty$ (*round toward $-\infty$*)

Modul implicit de rotunjire este rotunjire la valoarea cea mai apropiată. În cazul în care numărul ce trebuie aproximat se găsește exact la jumătatea intervalului dintre două numere succesive reprezentabile, rotunjirea se face la un număr par (*round to nearest even*).

Ca o concluzie, conform standardului IEEE 754 un număr real reprezentat pe 32 de biți, la care se respectă condiția $0 < C < 255$, se poate calcula cu relația, :

$$N = (-1)^S 2^{C-127} (1.F) \quad (3.27)$$

Pentru formatul real lung, pe 64 de biți, dacă se respectă condiția $0 < C < 2047$, numărul poate fi calculat cu relația:

$$N = (-1)^S 2^{C-1023} (1.F) \quad (3.28)$$

CAPITOLUL 4

ARHITECTURA SETULUI DE INSTRUCȚIUNI

Conținut:

- 4.1. Introducere în arhitectura setului de instrucțiuni
- 4.2. Formatul instrucțiunilor
- 4.3. Interdependența set de instrucțiuni - organizare internă a procesorului
- 4.4. Scurtă privire comparativă între arhitecturile RISC și CISC
- 4.5. Tipuri de instrucțiuni
 - 4.5.1. Instrucțiuni aritmetice
 - 4.5.2. Instrucțiuni logice
 - 4.5.3. Instrucțiuni pentru transferul informației
 - 4.5.4. Instrucțiuni pentru deplasarea și rotirea datelor
 - 4.5.5. Instrucțiuni de ramificare (pentru controlul secvenței de program)
 - 4.5.6. Instrucțiuni pentru controlul procesorului
 - 4.5.7. Instrucțiuni pentru lucrul cu șiruri
- 4.6. Moduri de adresare
 - 4.6.1. Adresare imediată
 - 4.6.2. Adresare directă
 - 4.6.3. Adresare indirectă
 - 4.6.4. Adresare relativă
 - 4.6.5. Adresare indexată

Obiective educaționale:

- a. Enumerarea caracteristicilor principale și a componentelor principale ale arhitecturii setului de instrucțiuni (ASI) pentru un microprocesor
- b. clasificarea diferitelor seturi de instrucțiuni
- c. Descrierea formatului instrucțiunilor și a modurilor de adresare
- d. Explicarea importanței alinierii datelor în memoria principală
- e. Clasificarea și analiza comparativă a diferitelor ASI

4.1. INTRODUCERE ÎN ARHITECTURA SETULUI DE INSTRUCȚIUNI

Arhitectura setului de instrucțiuni specifică unui procesor cuprinde:

- ⇒ setul de instrucțiuni (instrucțiuni specifice la nivel mașină) recunoscute de procesor,
- ⇒ tipurile de date care pot fi manipulate cu aceste instrucțiuni și
- ⇒ contextul în care aceste instrucțiuni operează.

Arhitectura setului de instrucțiuni (ASI) este o componentă a mașinii de calcul, vizibilă programatorului la nivel de limbaj de asamblare, componentă care realizează interfața între software și hardware (figura 4.1.).

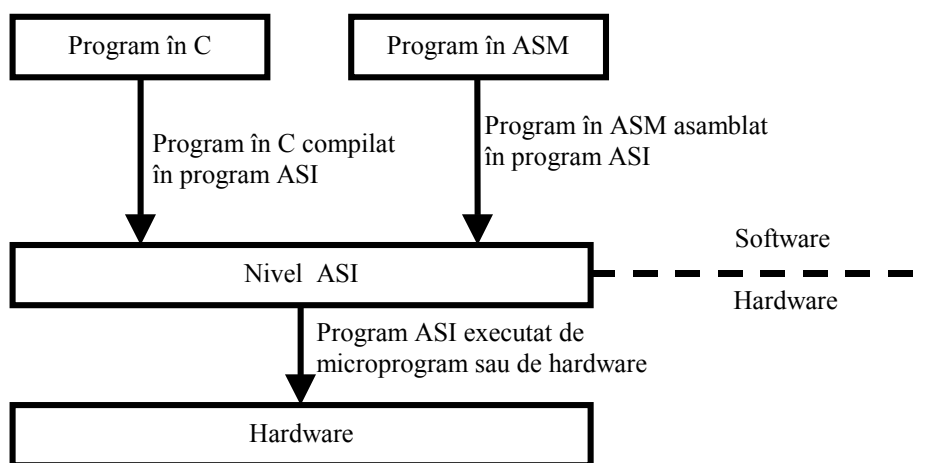


Figura 4.1. Arhitectura setului de instrucțiuni ca interfață între hardware și software (adaptată după [Patterson90])

Pentru a putea comanda componenta hardware a unui calculator trebuie cunoscute instrucțiunile recunoscute de acesta la nivelul mașinii de bază. Instrucțiunile implementate hardware pe o anumită mașină formează setul de instrucțiuni, sau limbajul inclus în ASI. De aceea, în figura 4.1. programele în limbaj de nivel înalt (C în figură) sunt compilate pentru a se genera un program recunoscut de mașină, numit în figură program ASI. Acesta este programul în cod binar recunoscut de unitatea de control (UC) și interpretat de către o mașină cu algoritm de tranziție a stărilor internă UC; algoritmul poate fi implementat ca microprogram sau poate fi cablat în hardware. În mod similar programele scrise în limbaj de asamblare (ASM) trebuie translate cu ajutorul unui program translator numit asamblor în programe executabile, conform ASI specific mașinii.

Setul de instrucțiuni implementat pe un anumit microprocesor poate fi clasificat după mai multe criterii, dintre care menționăm [Patterson90]:

- Numărul de *adrese de operanzi pe instrucțiune*. Numărul de adrese pe instrucțiune afectează în mod direct complexitatea și dimensiunile (în număr de biți) ale unei instrucțiuni. Indiferent de tipul de format ales și de tipul operațiilor numărul maxim de operanzi /adrese) este 3, iar numărul minim este 0.

- *Locul de stocare a operanzilor* pentru instrucțiunile executate de CPU. Operanzii operațiilor aritmetice și logice pot fi imediați, sau se pot găsi în registrele interne UCP, în memoria principală, în spațiul de I/O, sau în memorie de tip stivă. Există arhitecturi care efectuează operații aritmetice și logice doar cu operanzi din registrele interne, iar cu exteriorul permit doar operații de încărcare și stocare a operanzilor (arhitectură load / store¹). Alte tipuri permit operanzi doar în memoria stivă (stack architecture²). Locul de stocare al operanzilor afectează nu numai formatul instrucțiunilor, dar și modurile de adresare ce trebuiesc codificate.
- *Tipul operațiilor* efectuate de setul de instrucțiuni. La implementarea unui set de instrucțiuni se urmărește optimizarea operațiilor cele mai frecvent efectuate pe o anumită mașină. Astfel că tipul instrucțiunilor depinde în special de aplicațiile pentru care este destinat procesorul. De exemplu, un microprocesor destinat aplicațiilor de tip DSP (prelucrare a semnalelor digitale) are instrucțiuni ce optimizează operațiile de: adunare, înmulțire, stocare temporară, shift³-are și manipulare de matrice.
- *Tipul datelor prelucrate* de unitățile aritmetice și logice. Diferitele mașini pot lucra cu numere în virgulă fixă (întregi sau fracționare) și numere în virgulă mobilă (numere reale).

Setul de instrucțiuni al unui procesor poate avea două caracteristici principale:

1. **regularitate (ortogonalitate)** - această caracteristică impune ca fiecare instrucțiune din setul de instrucțiuni să poată opera cu orice tip de date, conținute în oricare registru sau oricare locație de memorie, și să poată fi adresate prin oricare mod de adresare.
2. **completitudine** - această cerință ar impune ca setul de instrucțiuni să cuprindă tot setul de operatori disponibili într-un limbaj. Dar cuprinderea întregului set de operatori, operatori cu frecvențe foarte diferite de utilizare, ar duce la implementări și performanțe departe de cele optime. În consecință se introduce o cenzură pentru operatorii cu frecvență scăzută de apariție în programe, în avantajul operatorilor cu frecvență ridicată. Uneori se introduce regula (cam generală dar practică): “*Execută rapid operațiile frecvente și corect pe cele rare*”. În faza de proiectare a unui procesor, setul de instrucțiuni se croiește anticipând spectrul de aplicații ale procesorului, astfel ca acestea să poată fi ușor exprimate sau programate. Aceasta este faza în care se pot determina operațiile ce vor fi folosite frecvent.

4.2. FORMATUL INSTRUCȚIUNILOR

Formatul instrucțiunilor se reprezintă de obicei în formă grafică sub forma unor dreptunghiuri ce simbolizează câmpurile binare incluse în instrucțiuni. Din punctul de vedere al lungimii instrucțiunilor pot exista două tipuri de ASI:

- arhitecturi la care **lungimea** instrucțiunilor este **fixă**, pentru tot setul recunoscut
- arhitecturi cu **lungime variabilă** a instrucțiunilor recunoscute

La microprocesoarele cu lungime fixă a instrucțiunilor (de exemplu arhitecturile RISC⁴, dar și arhitecturi CISC cum ar fi cea a calculatorului Felix C256) unitatea de control, care interpretează instrucțiunile și lansează comenzile pentru execuția acestora, este mai simplă, dar programele ocupă un spațiu mai mare în memorie. Formatul de lungime variabilă a instrucțiunilor (în general

¹ load / store = încărcă / stochează

² stack architecture = arhitectură pe bază de stivă

³ shift = deplasare a informației dintr-un registru la stânga sau la dreapta cu un anumit număr de poziții binare.

⁴ RISC = Reduced Instruction Set Computer

microprocesoare cu arhitectura CISC⁵) implică o complexitate mai mare a unității de control. Lungimea variabilă a instrucțiunilor (lungime mică pentru operații simple și mare pentru cele complexe) permite programe cu cod mai compact (ocupă în general o zonă mai mică de memorie) și de asemenea permite extinderea ușoară a setului de instrucțiuni, pentru o familie de procesoare cu aceeași ASI, pentru că nu mai există limitarea unui număr fix de octeți.

Instrucțiunile se reprezintă în memoria principală ca octeți succesivi (unul sau mai mulți octeți în funcție de tipul de procesor). Scopul fiecărei instrucțiuni recunoscute de procesor este să specifice operația ce trebuie executată de hardware, operanzii utilizați și locul unde se stochează rezultatul operației. Formatul unei instrucțiuni include două tipuri principale de informații:

- codul operației (Op-Code în figura 4.2.) pe care o comandă instrucțiunea respectivă
- câmpuri de adrese pentru specificarea modului de referire la operanzi și la locul unde se stochează rezultatul. Vom numi aceste câmpuri specificatori de operanzi, pentru că ei indică adresa sau modul de calcul al adresei efective a operanzilor (în figură s-au considerat trei specificatori de adresă de operanzi: OP1, OP2, OP3).

Op-Code	OP1	OP2	OP3
---------	-----	-----	-----

Figura 4.2. Exemplu al formatului general pentru o instrucțiune ASI

Primul câmp binar al instrucțiunii reprezintă codul numeric identificator al instrucțiunii (OpCode = codul operației efectuate de instrucțiune). Lungimea acestuia este în general diferită în funcție de complexitatea operației și eventualele informații suplimentare pe care le cuprinde; aceste informații pot fi de exemplu: modul de adresare al operanzilor sursă, un indicator de condiție testat pentru a lua o decizie la execuția instrucțiunii, un prefix al codului operației care modifică modul de lucru al acelei instrucțiuni. Prefixul poate avea efecte diverse, cum ar fi:

- repetare de un număr de ori a instrucțiunii cu prefix,
- indicația că se lucrează cu operanzi sau adrese extinse (dimensiune dublă față de cea normală)
- identifică o instrucțiune specială destinată coprocesorului aritmetic.

Apoi urmează specificatorii pentru operanzi (modul de găsim al operanzilor se numește *mod de adresare* al operanzilor). Ordinea în care sunt dispuși specificatorii de operanzi în formatul instrucțiunii diferă de la procesor la procesor. Vom considera, doar pentru exemplificare, un format de instrucțiune cu trei operanzi, ca cel din figura 4.2, și la care specificatorul OP1 se referă la destinație (aceasta poate fi un registru intern, o locație de memorie sau porturi de I/O⁶), iar OP2 și OP3 sunt specificatori de operanzi sursă (operanzii putând fi date imediate, sau stocați în registre, memorie, ori porturi de I/O).

Instrucțiunea recunoscută de mașină este scrisă deci în cod binar care codifică o acțiune a procesorului. Specificatorii de operand ce apar *explicit* în formatul instrucțiunii după codul operației se referă de obicei la adrese din memoria principală, adrese de porturi, operanzi imediați (date sau adrese imediate), registre interne de uz general ale UCP. Unele din registrele interne ale UCP au funcții specifice în cazul anumitor instrucțiuni, iar prin decodificarea codului operației acestor

⁵ CISC = Complex Instruction Set Computer

⁶ I/O = intrare / ieșire

instrucțiuni, registrele respective fiind recunoscute *implicit* ca sursă sau destinație de operanzi.

După modul în care se stochează în memoria principală adresele operanzilor imediați din câmpul de specificatori, există două convenții. Convenția "Little Endian" (aluzie la șirul indian care are în față pe cel mai mic) plasează întotdeauna pe prima poziție, la adresa cea mai mică, cuvântul cel mai puțin semnificativ al unei adrese multi - cuvânt. Stocarea în memorie se face în ordine, de la cuvântul cel mai puțin semnificativ (adresa mică), către cuvântul cel mai semnificativ stocat la adresa cea mai mare. În figura 4.3. se prezintă un exemplu de stocare în memorie, în convenția "Little Endian", a unui cuvânt de 32 biți. Memoria din figură este organizată pe octet, având deci alocată câte o adresă fiecărui cuvânt de 8 biți. Valorile numerice au fost scrise în hexazecimal.

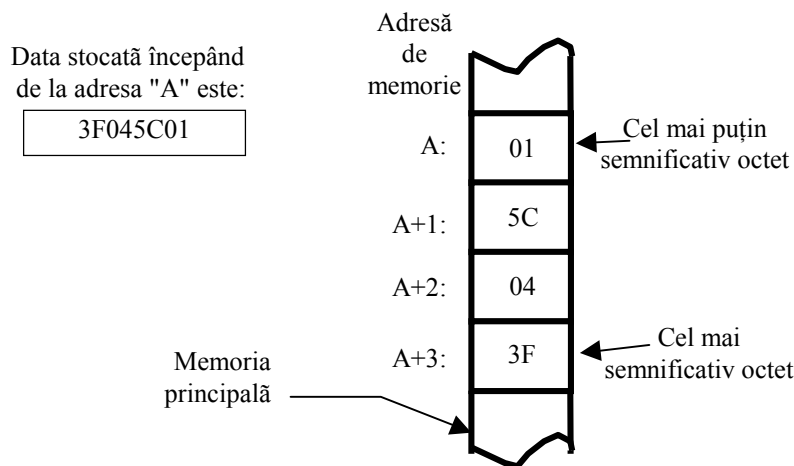


Figura 4.3. Exemplu de stocare a unui cuvânt de 32 de biți, într-o memorie organizată pe octet, convenția utilizată fiind Little Endian. Într-o instrucțiune ce adresează această dată pe 32 de biți se va specifica adresa "A", unde este stocat octetul cel mai puțin semnificativ.

Pentru cealaltă convenție, numită "Big Endian" (marele indian), ordinea este inversă, adică "în față", la adresa cea mai mică se plasează octetul cel mai semnificativ al cuvântului multi-octet.

Legat de convenția de stocare în memorie a informațiilor (instrucțiuni, date, adrese) trebuie menționat aici că la unele microprocesoare accesul la obiecte mai mari decât un octet se poate face doar dacă informația este aliniată. Alinierea se referă la faptul că accesul la un obiect de dimensiunea D octeți se face prin specificarea adresei A primului octet ocupat, unde adresa A respectă ecuația:

$$A \bmod D = 0$$

Considerând o memorie adresabilă la nivel de octet și un microprocesor de 32 biți, alinierea și nealinierea informațiilor pe 32 de biți se exemplifică grafic în figura 4.4.

Nealinierea produce complicații la citirea memoriei și conduce la pierdere de timp. În exemplul din figura 4.4. pentru informația aliniată citirea se poate face într-un singur ciclu de acces la memorie. Pentru informație ne-aliniată trebuiesc cel puțin două cicluri de acces la memorie. Dar situația de ne-aliniere prezentată nu este atât de critică cât ar fi dacă nealinierea s-ar realiza prin stocarea informației pe 32 de biți la adrese succesive, de exemplu în ordinea: 0B, 0C, 0D și 0E hex, când citirea ar trebui făcută pentru fiecare octet în parte.

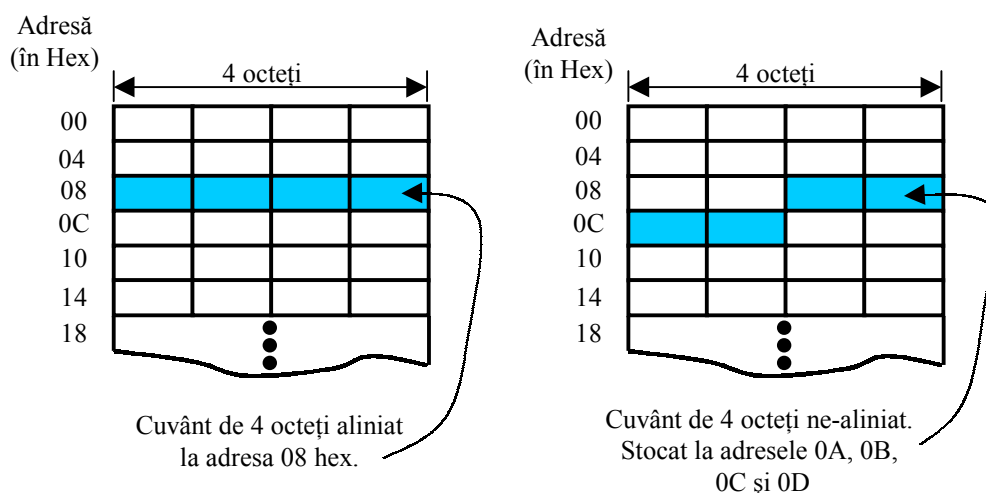


Figura 4.4. Exemplu de adrese de acces la obiecte aliniate și nealiniate. Cuvântul pe 32 de biți nealiniat poate fi citit în cel mai bun caz prin două accesări succesive la memorie pentru citirea a câte 2 octeți.

4.3. INTERDEPENDENȚA SET DE INSTRUCȚIUNI - ORGANIZARE INTERNĂ A PROCESORULUI

O sursă de controverse începând de la prima generație de calculatoare a fost: câte *adrese explicite* pentru operanzi să includă o instrucțiune? Cu cât sunt mai puține adrese cu atât e mai scurtă instrucțiunea. Dar limitând numărul de adrese se limitează domeniul de funcții pe care fiecare instrucțiune le poate realiza. În general, mai puține adrese înseamnă mai multe instrucțiuni primitive și deci programe mai lungi pentru o anumită sarcină de rezolvat.

În timp ce necesitățile de stocare în memorie pentru instrucțiuni mai scurte și programe mai lungi tind să se echilibreze cu instrucțiunile lungi și programe scurte, programe mai lungi cer un timp mai lung de execuție. Pe de altă parte, instrucțiunile lungi, cu adrese multiple, cer de obicei circuite mai complexe pentru decodare și prelucrare. De multe ori procesoarele sunt clasificate după numărul maxim de adrese de memorie principală în câmpurile instrucțiunilor lor. Calculatoarele conținând un număr diferit de adrese au în general instrucțiuni de lungimi diferite. Instrucțiunile cu doi operanzi necesită cel mult trei adrese. De exemplu, pentru o operație de adunare, simbolizată prin mnemonica “add” pot fi implementate instrucțiuni cu minim 0 și maxim 3 operanzi, așa cum se observă în tabelul 4.1.

Numărul de câmpuri de adresă în formatul instrucțiunii unui procesor depinde organizarea internă a registrelor sale, dar în același timp influențează modul în care este proiectat procesorul ca organizare, din punctul de vedere al registrelor interne. Numărul de adrese poate fi diferit, de la instrucțiune la instrucțiune la același procesor. De exemplu instrucțiunea de scădere poate avea trei adrese de operanzi (doi sursă și unul destinație), iar instrucțiunea de complementare poate avea un singur operand (registru sursă este și destinație).

Tabelul 4.1.

Număr adrese	Format instrucțiune	Comentariu
3	add z,x,y	; $z \leftarrow x+y$, unde x,y,z, sunt locații de memorie, nume de registre, sau date imediate
2	add x,y	; $x \leftarrow x+y$ sau $Ac \leftarrow x+y$, în funcție de procesor ; (Ac = registru acumulator)
1	add x	; $Ac \leftarrow Ac+x$
0	add	; adună primele două articole din vârful stivei și ; stochează rezultatul în noul vârf al stivei

Conform legăturii dintre organizarea UCP și numărul de adrese pe instrucțiune, vom putea încadra cele mai multe procesoare într-una din următoarele trei categorii:

1. *organizare pe bază de registru acumulator (Ac)* la care pentru majoritatea operațiilor unul dintre operanzi se găsește implicit în acumulator. De asemenea rezultatul operației se 'acumulează' în acest registru cu funcție specială.
2. *organizare pe bază de registre de utilizare generală (RG)*. La acest tip de organizare operanzii sunt explicit localizați fie în registrele interne (fără restricții), fie în registrele din locațiile memoriei principale.
3. *organizare de tip stivă (ST)* la care operanzii se găsesc implicit în vârful stivei și tot aici se stochează rezultatul. Există unele procesoare de tip stivă care folosesc două stive: una pentru operații asupra datelor și alta pentru operații asupra adreselor.

Tabelul 4.2. Exemple de instrucțiuni

Tip arhitectură	Instrucțiuni	Comentarii
Ac	add x	; $Ac \leftarrow Ac + x$ sau $Ac \leftarrow Ac + @x$; adună la acumulator conținutul registrului x sau al memoriei de la adresa dată de x
RG	add r1,r2,r3 add r1,r2 mov r1,r2 add r1,x	; $r1 \leftarrow r2 + r3$; $r1 \leftarrow r1 + r2$ sau $r2 \leftarrow r1 + r2$; $r1 \leftarrow r2$ sau invers ; $r1 \leftarrow r1 + @x$
ST	push x add	; trimite cuvântul de adresă x în vârful stivei, ; $VS \leftarrow @x$; aduce primele 2 articole din stivă, le adună și ; ; stochează rezultatul în stivă

Fiecare dintre cele trei tipuri de arhitecturi au avantaje și dezavantaje. Astfel arhitectura pe bază de acumulator conduce la instrucțiuni scurte, pentru că acumulatorul este referit ca registru implicit la multe dintre instrucțiunile mașinii. Conduce și la o densitate relativ mare a codului (programelor) stocate în memorie, dar acumulatorul constituie un puternic factor de limitare al vitezei de lucru al procesorului. Mașinile pe bază de registre de uz general prezintă avantaje din punctul de vedere al ușurinței generării codului de către programe compilatoare. Dacă nu ar fi existat aceste registre, foarte multe din rezultatele

intermediare ar fi trebuit stocate în memoria principală ceea ce ar fi micșorat viteza de prelucrare. Dezavantajul arhitecturilor de tip RG este dat de faptul că fiecare registru trebuie adresat de către instrucțiunea ce-l utilizează, astfel că lungimea instrucțiunilor este relativ mare. Ultimul tip de arhitectură permite o evaluare extrem de ușoară a operațiilor pe baza notației inverse (notația “poloneză” - vezi exemplul de mai jos). Aceasta conduce și la instrucțiuni cu lungime foarte redusă. Din păcate o stivă este mai dificil de adresat în mod aleator, iar codul rezultat este relativ inefficient. Stiva introduce și o limitare în viteză și de aceea acest tip de arhitectură este rar implementat la procesoare de uz general.

În tabelul 4.2. s-au prezentat câteva exemple de instrucțiuni pentru cele trei tipuri de arhitecturi. Multe calculatoare pot fi încadrate în cele 3 categorii de mai sus, dar cele mai multe combină caracteristici din mai multe moduri de organizare.

Pentru a ilustra influența numărului de adrese în programele executate de procesor vom evalua egalitatea:

$$x = a \times (b + c) \quad (4.1)$$

utilizând exemple de instrucțiuni cu 0, 1, 2, și 3 adrese. Vom folosi simbolurile (mnemonicele):

- add, mul, - pentru operațiile aritmetice elementare de adunare și înmulțire
- load, store - transfer de la/la memorie spre/dinspre acumulator
- push, pop - înscriere respectiv extragere din stivă.
- r# - registre interne ale UCP, unde # este un indice

Presupunem că operanzii a, b și c din relația (4.1) sunt stocați în memorie, în locațiile de adrese a, b, c, iar rezultatul se va introduce la adresa x în memorie.

a) pentru arhitecturi pe baza de registre de uz general:

- *instrucțiuni cu 3 adrese care pot efectua operații aritmetice sau logice cu operanzi din memorie (CISC):*

add	r1,b,c	;r1 ← @b + @c
mul	x,r1,a	;@x ← r1 * @a

Se observă că acest format poate utiliza fiecare câmp de adresă operand pentru a specifica un operand dintr-un registru sau din memorie. Deși programul este foarte scurt, el conține instrucțiuni cu format de lungime mare și se desfășoară relativ lent. Astfel la prima instrucțiune se fac două accesări la memorie pentru citirea operanzilor b și c. La fel pentru a doua instrucțiune sunt necesare două accesări succesive la memorie (operand și rezultat).

- *instrucțiuni cu 3 adrese (RISC):*

load	r1,a	; r1 ← @a
load	r2,b	; r2 ← @b
load	r3,c	;
add	r2,r2,r3	; r2 ← (r2 + r3)
mul	r1,r1,r2	; r1 ← (r1 * r2)
store	x,r1	; @x ← r1

La acest tip de arhitecturi se folosesc doar instrucțiuni de *load* (*încarcă*) și *store* (*stochează*) când se comunică cu memoria, iar toate celelalte instrucțiuni se referă la operanzi din registrele UCP (deci prelucrare foarte rapidă)

- *instrucțiuni cu 2 adrese (CISC)*

load	r1,b	;r1 ← @b
add	r1,c	;r1 ← r1 + @c
mul	r1,a	;r1 ← r1 * @a
store	x,r1	;@x ← r1

b) pentru arhitecturi pe baza de acumulator (instrucțiuni cu o adresa, acumulatorul fiind adresat implicit):

load	b	; ac ← @b
add	c	; ac ← ac + @c
store	t	; @t ← ac
load	a	; ac ← @a
mul	t	; ac ← ac * @t
store	x	; @x ← ac

unde s-a notat cu "t" adresa unei locații de memorie folosită pentru stocarea temporară a rezultatelor intermediare

c) pentru arhitecturi pe bază de stivă (instrucțiuni cu zero adrese)

push	a	;vs ← @a, fie "adr" adresa curentă a vârfului stivei (vs) ; iar stiva crește spre adrese mici	
push	b	;vs ← @b, la adr-1	
push	c	;vs ← @c, la adr-2	
add		;vs ← (b + c), stocat la adr-1	<i>zero adrese</i>
mul		;vs ← a*(b+c), stocat la adr	<i>zero adrese</i>
pop	x	;@x ← vs, vs pointează la adr+1	

Pentru a evalua expresiile aritmetice într-un calculator stivă, e necesar să se convertească expresia în notația poloneză (inversă). Ideea a pornit de la matematicianul polonez Lukasievics J, care a notat o expresie de forma "a + b" ca "ab+".

După același principiu putem exemplifica notația poloneză și pentru alte ecuații:

$$\begin{aligned}
 x &= a * (b + c) &= & abc+*. \\
 x &= (a+b)*(c+d) &= & ab+cd+* \\
 x &= (a+b)*(c+d)*(e+f) &= & ab+cd+ef+**
 \end{aligned}$$

După cum se vede, notația poloneză are avantajul că nu folosește paranteze. Ba mai mult, pornind de la stânga la dreapta se indică toate operațiile ce trebuie efectuate, în ordine, pe o arhitectură de tip stivă.

4.4. SCURTĂ PRIVIRE COMPARATIVĂ ÎNTRE ARHITECTURILE RISC ȘI CISC

Multe calculatoare au seturi de instrucțiuni ce includ mai mult de 100 - 200 instrucțiuni. Ele folosesc o varietate de tipuri de date și un mare număr de moduri de adresare. Tendința aceasta de a mări numărul de instrucțiuni a fost influențată de mai mulți factori, dintre care amintim:

- perfecționarea unor modele de procesoare existente anterior, pentru a pune la dispoziția utilizatorilor (programelor utilizator) cât mai multe funcții cablate
- adăugarea de instrucțiuni care să faciliteze translatarea din limbajele de nivel înalt în programe cod executabil (limbaj mașină)
- tendința de a deplasa cât mai multe funcții de la implementarea programată (software) către cea cablată (hardware), în scopul obținerii unor performanțe de viteză cât mai mari.

Ideea simplificării setului de instrucțiuni, în scopul măririi performanțelor procesorului, provine atât din proiectul calculatorului CDC 6600, cât și din proiectele realizate la universitățile americane din Berkeley (RISC I, RISC II și SOAR) și Stanford (proiectul MIPS). Proiectele RISC (Reduced Instruction Set Computer - Calculator cu set redus de instrucțiuni) au urmărit ca instrucțiunile procesorului să fie de aceeași lungime, instrucțiunile să se execute într-o singură perioadă de ceas (cu ajutorul tehnicii de tip conductă - pipeline), iar unitatea de control să fie implementată cablat, pentru a reduce complexitatea chip-ului și pentru a crește viteza. La RISC se urmărește de asemenea ca accesările la memorie (consumatoare de timp) să se efectueze doar pentru operațiile de încărcare și stocare (arhitectura fiind numită în consecință: "load/store"), iar celelalte operații să se efectueze cu operanzi stocați în registrele interne ale UCP. Unele din proiectele de arhitecturi RISC folosesc un set mare de ferestre de registre (numit și *stack cache* - memorie stivă de registre) pentru a accelera operațiile de apel al subrutinelor. Cu toate aceste specificații, denumirea RISC nu conduce și la niște criterii stricte de proiectare, ea fiind doar o descriere a cerințelor generale impuse procesorului. De aceea multe din mașinile anterioare (numite de atunci arhitecturi CISC - Complex Instruction Set Computer - calculator cu set complex de instrucțiuni) au "împrumutat" aceste idei de proiectare (pipeline, execuția unei instrucțiuni pe ciclu de ceas, ferestre de registre, etc.) dar ele nu pot fi numite RISC-uri (MC68040, 80486, Pentium, H16, etc.). Concepția RISC pornește de la faptul că un circuit mai simplu pentru procesor, poate lucra la frecvențe de ceas mai mari, iar setul mai simplu de instrucțiuni este mai potrivit pentru optimizări ale codului de către un compilator.

Arhitectura numită CISC, se referă la un set complex de instrucțiuni; aceasta însă, doar prin comparație cu RISC, căci nu există un set de caracteristici generale, așa cum s-au definit la RISC. Ideea care s-a manifestat în dezvoltarea acestor mașini, de a introduce structuri hardware suplimentare pentru creșterea vitezei, nu este greșită ca principiu. Proiectarea unui set de instrucțiuni pentru un procesor trebuie să țină seama nu numai de construcția limbajului mașină ci și de cerințele impuse de utilizarea limbajelor de programare de nivel înalt. Translatarea se face cu un program compilator. Deci scopul final ar fi pentru arhitectura CISC să permită folosirea unei singure instrucțiuni mașină pentru fiecare instrucțiune din programul scris în limbaj de nivel înalt. Aceasta ar conduce la simplificarea compilatorului, căci sarcina acestuia este mult ușurată dacă există o instrucțiune mașină care implementează direct o instrucțiune de limbaj înalt. Dar asta presupune realizarea câte unei arhitecturi pentru fiecare tip de limbaj de programare. Dacă nu este așa, de cele mai multe ori, instrucțiunile complexe sunt rar utilizate de către compilatoare. Alt dezavantaj al CISC constă în faptul că este dificil de mărit frecvența de ceas pentru un

circuit complex.

Rezumând, putem enumera câteva din elementele caracteristice pentru fiecare din cele doua arhitecturi.

Rezumat al caracteristicilor definitorii pentru mașinile RISC:

1. Acces la memorie limitat, doar prin instrucțiuni de *încărcare (load)* și *stocare (store)*;
2. Format de *lungime fixă* pentru instrucțiuni, deci ușor de decodificat; caracteristică care contribuie la simplificarea structurii unității de control;
3. O structură *simplă a unității de control* implementată sub formă cablată, deci cu viteză mare de funcționare;
4. Relativ *puține tipuri de instrucțiuni* (tipic sub 100 de instrucțiuni) și puține moduri de adresare (din nou această caracteristică contribuie și la simplificarea structurii unității de control);
5. Execuția instrucțiunilor într-un singur ciclu prin tehnici pipeline⁷; se face prin suprapunerea diferitelor faze de execuție (fetch, decodificare, aducere operanzi etc.) ale mai multor instrucțiuni. Tehnica de tip conductă (pipeline) este utilizată și la arhitecturile CISC, dar la RISC tehnica este mai eficientă și mai ușor de implementat, datorită lungimii constante a instrucțiunilor;
6. Un număr relativ mare de registre în interiorul UCP;
7. Utilizarea compilatoarelor optimizatoare - pentru a optimiza performanțele codului obiect.

La CISC câteva din caracteristicile care le deosebesc de RISC-uri sunt:

1. Multe instrucțiuni care prelucrează operanzi din memorie;
2. Format de *lungime variabilă* pentru instrucțiuni;
3. Unitate de control microprogramată (micro-codată), avantajoasă din punctul de vedere al flexibilității implementării, dar lentă;
4. Set complex (extins) de instrucțiuni și o mare varietate de moduri de adresare;
5. Un număr relativ mic de registre în interiorul UCP.
6. Utilizarea compilatoarelor optimizatoare - pentru a optimiza performanțele codului obiect;
7. Exista instrucțiuni complexe de mare eficiență, dar uneori acestea sunt rar utilizate de către compilatoare.

Asa cum s-a arătat mai sus, arhitecturile RISC restricționează numărul de instrucțiuni care accesează direct memoria principală. Cele mai multe instrucțiuni ale RISC presupun doar operații între registrele interne UCP. Pentru că instrucțiunile complexe nu există în setul de instrucțiuni, dacă este nevoie de ele, acestea se implementează prin rutine cu ajutorul instrucțiunilor existente. În final, într-un program executabil vor fi mai multe instrucțiuni decât la CISC, dar execuția pe ansamblu va fi mai rapidă. Totuși, dacă într-un program, frecvența operațiilor complexe este mare, este foarte posibil ca performanțele unei mașini CISC să fie mai bune (execuție mai rapidă).

4.5. TIPURI DE INSTRUCȚIUNI

Prezentarea tipurilor de instrucțiuni se va face cu exemplificări din setul de instrucțiuni specifice procesoarelor Intel 80x86. Pentru toate instrucțiunile ce se vor da ca exemplu, care cuprind doi specificatori de operand, întotdeauna primul specificator se referă la destinație (D - rezultat) și implicit la una din sursele de operand, iar al doilea la sursă (S).

⁷ pipeline = conductă, bandă de montaj. În continuare vom folosi termenul de conductă.

4.5.1. Instrucțiuni aritmetice

Asigură prelucrarea unor operanzi aflați în memorie sau în registrele generale interne UCP, având la bază operații de o complexitate foarte diferită la diferitele microprocesoare. Un set tipic de instrucțiuni aritmetice este rezumat în continuare:

Tip instrucțiuni aritmetice	Exemple I8086
• adunare	ADD D,S
• adunare cu considerarea transportului	ADC D,S
• scădere	SUB D,S
• scădere cu considerarea împrumutului	SBB D,S
• incrementare	INC D
• decrementare	DEC D
• înmulțire (fără semn)	MUL S *
• înmulțire (cu semn)	IMUL S *
• împărțire (fără semn)	DIV S *
• împărțire (cu semn)	IDIV S *
• realizare cod complementar (C2)	NEG D
• comparare	CMP D,S
• ajustare zecimală	DAA (După adunare în NBCD) DAS (După scădere în NBCD)

Notă: * = deîmpărțitul sau deînmulțitul se găsesc implicit în registrele acumulator și eventual DX. Din această cauză la I8086 există un singur operand. La procesoare Intel pe 32 de biți (de la I80386), operația de înmulțire cu semn a întregilor poate avea și forma cu doi sau trei operanzi, caz în care se pot folosi și alte registre generale, decât AX și DX.

În urma efectuării operațiilor aritmetice este influențată valoarea indicatorilor de condiții, în funcție de rezultat. Instrucțiunile de comparație (CMP la I8086) ar putea fi asociate instrucțiunilor logice dacă comparația s-ar face prin instrucțiuni logice; comparația este inclusă însă la instrucțiuni aritmetice pentru că se efectuează prin scădere și testarea rezultatului (de fapt testarea indicatorilor de condiții). La compararea prin scădere conținutul registrelor sursă nu este afectat, iar operația se face fără memorarea rezultatului, doar prin setarea indicatorilor de condiții.

Tot în cadrul instrucțiunilor aritmetice pot fi incluse și operații aici pot fi incluse instrucțiuni de conversie a dimensiunii (de exemplu la I 8086 din 8 în 16 biți (CBW), din 16 în 32 (CWD) la I80386: CWDE (16→32) și CDQ (32→64).

4.5.2. Instrucțiuni logice

Acestea efectuează operații logice cu operanzii specificați de instrucțiune. Deși, în principiu, ar fi suficiente operațiile operatorilor compleți NAND⁸ și / sau NOR, setul de instrucțiuni al unui microprocesor include ca instrucțiuni logice cel puțin operațiile logice de tip ȘI, SAU, NU, SAU-

⁸ NAND / NOR = ȘI-NU / SAU-NU

Exclusiv. Aceste operații se efectuează bit cu bit între operanzi, cu excepția complementării (NU) care are un singur operand pentru care se complementează fiecare bit.

În urma efectuării operațiilor logice este influențată valoarea indicatorilor de condiții, în funcție de rezultat. Un set tipic de instrucțiuni logice este rezumat în continuare:

Tip instrucțiuni logice	Exemple I8086
• ȘI logic	AND D,S
• SAU logic	OR D,S
• SAU-Exclusiv	XOR D,S
• complement (NU)	NOT D
• test logic	TEST D,S *

Nota: * = se efectuează ȘI logic între sursă și destinație, rezultatul nu se stochează nici-unde, dar se poziționează indicatorii de condiții.

Tot în cadrul instrucțiunilor logice sunt incluse și instrucțiunile de prelucrare și test logic a biților unui cuvânt. De exemplu, la I80386 există instrucțiunile de examinare (scan) a biților unui octet / multi-octet (BSF - începând cu LSbit, BFR - începând cu MSbit), sau de testare și prelucrare a biților unui octet / multi-octet, cu poziționarea indicatorului CF (BT, BTC, BTR, BTS).

Operațiile de prelucrare pe bit sunt importante în special pentru modificarea registrelor de control, sau la microcontrollere integrate pentru scrierea selectivă la anumite ieșiri ale portului paralel.

4.5.3. Instrucțiuni pentru transferul informației

Aceste instrucțiuni se folosesc pentru transferul (*copierea*) informației între registre, între registre și memorie, sau între registre și porturi de I/O. În general, sunt operații de copiere a informației din sursă în destinație. Aceste instrucțiuni nu afectează de obicei valorile indicatorilor de condiții.

Pot fi clasificate în:

- Instrucțiuni de transfer de uz general
- Instrucțiuni de transfer cu stivă
- Instrucțiuni pentru transferul adreselor (pointeri)
- Instrucțiuni de transfer I/O.

Pentru microprocesoarele care au porturile de I/O mapate în spațiul de adrese de memorie nu există instrucțiuni specializate de transfer de I/O. De obicei, pentru transferul I/O există multe instrucțiuni de I/O specifice fiecărui microprocesor, care se referă la transferul indirect în registre dedicate, transferul repetat de la o adresă de port etc.

Exemplificare pentru instrucțiunile de transfer:

Tip instrucțiuni de transfer	Exemple I8086
• transfer de uz general	MOV D,S ;D←S XCHG D,S ;D↔S

- transfer cu stiva

PUSH S	; înscriere în stivă
POP D	; extragere din stivă
PUSHF	; scriere indicatori în stivă
POPF	; extragere indicatori din stivă
- transferul adreselor

LEA r16,adr	; (r16) ← adr
LDS r16,adr	; r16Low ← [DS:adr]
	; r16High ← [DS:(adr+1)]
	; DSLow ← [DS:(adr+2)]
	; DSHigh ← [DS:(adr+3)]
- transfer de I/O

IN AL,adr	; AL ← adr (port)
IN AX,adr	; AL ← adr (port), 16 biți
IN AL,DX	; AL ← (DX)
OUT adr,AL	
OUT DX,AL	

4.5.4. Instrucțiuni pentru deplasarea și rotirea datelor

Instrucțiunile propriu-zise de deplasare pot fi asociate instrucțiunilor logice, în cazul în care deplasarea se efectuează în scopul izolării unor biți dintr-un cuvânt (pentru a testa sau prelucra acei biți), sau pot fi asociate instrucțiunilor aritmetice (înmulțire / împărțire cu puteri ale lui 2). În cazul deplasărilor aritmetice se conservă bitul de semn al operandului deplasat.

Operațiile de deplasare pot fi:

- ⇒ deplasări închise (rotații stânga / dreapta)
- ⇒ deplasări deschise (stânga / dreapta)

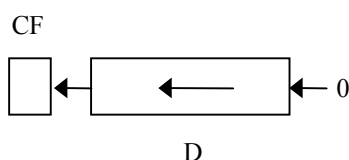
La toate operațiile de deplasare, se influențează și valoarea indicatorului de transport (Carry), care se încarcă cu ultima valoare de bit ieșită din registrul folosit pentru deplasarea operandului.

Deplasările închise (rotirile) sunt doar de tip logic. Rotațiile se pot efectua pe n biți (n fiind dimensiunea registrelor ce conțin datele) sau pe $n+1$ biți (rotație prin bitul de Carry). Deplasările deschise pot fi de tip logic sau de tip aritmetic. Pentru toate exemplele care se dau mai jos (i8086), al doilea operand poate fi 1 sau CL; dacă al doilea operand este 1 se indică deplasare cu o poziție binară, iar dacă acest operand este CL, rezultă o deplasare cu p poziții binare, p fiind valoarea conținută în registrul CL.

Exemple (Intel x86):

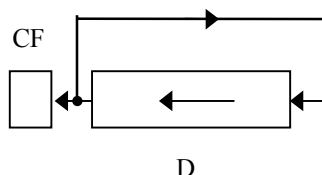
- deplasare aritmetică / logică la stânga

SAL D,1
SHL D,1



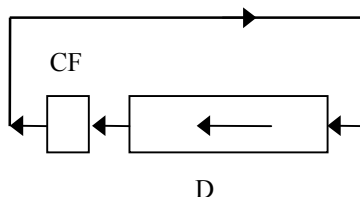
- rotire la stânga

ROL D,1



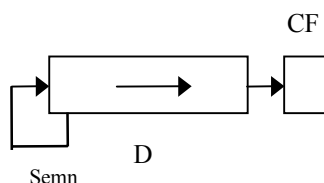
- rotire la stânga prin CF (rotire pe $n+1$ biți, dacă D are n biți)

RCL D,1



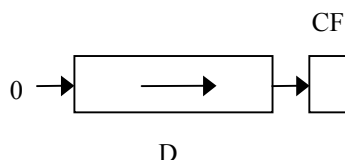
- deplasare aritmetică la dreapta (se păstrează valoarea bitului de semn)

SAR D,1



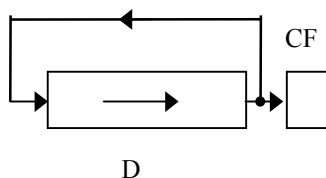
- deplasare logică dreapta

SHR D,1



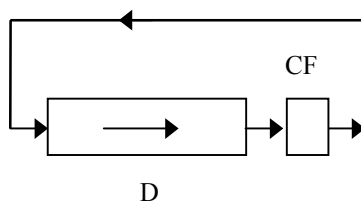
- rotire dreapta

ROR D,1



- rotire dreapta prin CF

RCR D,1



Începând cu microprocesorul I80386 au apărut instrucțiuni de deplasare suplimentare (de exemplu SHLD, SHRD), la care se folosesc 3 operanzi, primul fiind destinația, următorul sursa, iar ultimul contorul care indică numărul de biți de deplasare.

4.5.5. Instrucțiuni de ramificare (pentru controlul secvenței de program)

Aceste instrucțiuni determină modificarea secvenței normale, liniare, de execuție a unui program, în funcție de condițiile în care s-a terminat execuția unei instrucțiuni precedente. Se asigură astfel ramificațiile în program și implementarea diferitelor structuri de control specificate de un anumit algoritm.

Aceste instrucțiuni nu afectează starea indicatorilor de condiții, dar unele din instrucțiunile de salt sunt influențate de valoarea conținută în indicatori.

Instrucțiunile de ramificare (salt) pot fi:

- necondiționate. La execuția acestor instrucțiuni saltul se efectuează întotdeauna. Exemple de salturi necondiționate sunt instrucțiunile de apel de procedură (CALL op), salturi absolute (JMP op), revenire din procedură / return (RET), instrucțiuni de apel a unor funcții sistem (INT #).
- condiționate. Execuția acestor instrucțiuni produce salt doar dacă este îndeplinită o anumită condiție. Condiția se poate referi la un indicator de condiții sau la conținutul unor registre. Tipice sunt aici salturile condiționale (Jcc - Jump if condition = TRUE), apeluri condiționale de proceduri (CALL if condition = TRUE) sau “repetă până când condition = TRUE”.

Exemplificări pentru I8086:

JMP op

Este operația de salt necondiționat, prin care IP este înlocuit de offset-ul etichetei țintă în toate salturile inter-segment; același lucru și pentru salturile indirecte în cadrul aceluiași segment.

CALL op

Instrucțiunile de chemare a procedurilor lucrează similar cu JMP (salt intersegment sau extra-segment), cu excepția faptului că la execuția instrucțiunii CALL se salvează automat în stivă adresa de revenire din procedură.

RET

Instrucțiunile de tip RET produc revenirea dintr-o procedură, cu refacerea conținutului contorului de program și eventuala incrementare suplimentară (RET n) a indicatorului de stivă (registrul SP).

Jcc op

Instrucțiunile de salt condițional se referă la instrucțiuni ce se execută doar dacă o anumită condiție este îndeplinită. Îndeplinirea acestei condiții se verifică prin testarea indicatorilor de condiții. Vom încerca o prezentare sistematizată a salturilor condiționale pe baza tabelului 4.3. Fiecare mnemonică se referă la inițialele cuvintelor următoare, ce indică condiția în limba engleză: Above (peste, mai mare), Below (sub, mai mic), Equal (egal), Not (nu), Greater (mai mare), Less (mai mic), Carry (transport), Zero, Overflow (depășire de capacitate), Parity (PEven - paritate pară, POdd - paritate impară), Sign (semn). Din tabel se observă că se folosesc condiții diferite și litere diferite pentru a exprima relația mai mare, sau mai mic, pentru numerele cu semn, respectiv pentru cele fără semn.

(Exemplu de citire: JNBE = jump if not below nor equal, salt (J) dacă nu (N) e mai mic (B) sau egal (E)).

Tabelul 4.3. indică mnemonicele instrucțiunilor pentru testarea condiției din prima coloană, la scăderea a două numere A - B.

Tabelul 4.3. Explicativă la salturile condiționate pentru I8086

	Condiția de salt pentru:	
	numere fără semn	numere cu semn
A > B	JA/JNBE (CF = 0 și ZF = 0)	JG/JNLE (ZF = 0 și SF = OF)
A ≥ B	JAE/JNB/JNC (CF = 0)	JGE/JNL (SF = OF)
A = B	JE/JZ (ZF = 1)	
A ≤ B	JBE/JNA (CF = 1 sau ZF = 1)	JLE/JNG (dacă SF != OF sau ZF = 1)
A < B	JB/JNAE/JC (CF = 1)	JL/JNGE (SF != OF)

Acolo unde în tabel apar două tipuri de mnemonice separate prin semnul “/”, formele sunt echivalente, oricare formă conducând la aceeași instrucțiune.

În afara de salturile descrise în tabel, mai există instrucțiunile: JCXZ (salt dacă CX=0), JNZ/JNE, JO, JNO, JP/JPE, JNP/JPO, JS, JNS (ultimele se referă la testarea indicatorilor de condiții, condiția fiind valoarea 1 a indicatorului, sau valoarea zero dacă în fața numelui de indicator apare N(ot)).

4.5.6. Instrucțiuni pentru controlul procesorului

Sunt instrucțiuni care controlează anumite funcții ale procesorului, ce acționează fie prin intermediul unor indicatori de control (sau registre de control), fie prin introducerea unor stări sau semnale necesare pentru sincronizarea cu evenimentele externe.

Exemple (I8086):

CMC	;complementarea valorii indicatorului CF
CLC	;poziționarea pe 0 a indicatorului CF
STC	;poziționarea pe 1 a indicatorului CF
NOP	;Nici o operație, dar consumă 3 perioade de ceas.
CLD	;poziționarea pe 0 a indicatorului DF
STD	;poziționarea pe 1 a indicatorului DF
CLI	;poziționarea pe 0 a indicatorului IF, dezactivare întreruperi mascabile
STI	;poziționarea pe 1 a indicatorului IF
HLT	;Oprire microprocesor până la RESET, NMI, sau INT (dacă sunt activate)
WAIT	;așteptare până când vine semnalul exterior test=0
ESC	;operație destinată coprocesorului
LOCK	;prefix care activează semnalul extern /lock, astfel că microprocesorul anunță ;că nu va răspunde la o cerere de cedare a controlului magistralelor.

4.5.7. Instrucțiuni pentru lucrul cu șiruri

În afară de tipurile de bază amintite mai sus, există și posibilitatea efectuării unor operații de transfer, sau operații aritmetice și logice cu șiruri de date (cu informații aflate în zone continue de memorie). Operațiile pe șiruri pot fi efectuate individual, pentru fiecare cuvânt din șir, sau automat - cu repetare, numărul de repetări al instrucțiunii fiind dictat de conținutul unui registru contor. Operațiile tipic efectuate sunt:

- transferul unui șir din zonă sursă în zonă destinație
- comparare între două șiruri
- căutarea unei valori într-un șir
- încărcarea acumulatorului cu elementele unui șir.
- citirea unui șir de la un port de intrare
- scrierea unui șir la un port de ieșire

Exemple (I8086):

MOVSB(W)	;transfer pe 8 (16) biți [DS:SI]→[ES:DI] ;SI←SI+1; DI←DI+1 (decrementare pentru DF=1)
CMPSB(W)	;comparare pe 8(16) biți [DS:SI] cu [ES:DI] ;SI←SI+1; DI←DI+1 (decrementare pentru DF=1)

SCASB(W)	;comparare pe 8(16) biți între AL(AX) și [ES:DI] ;DI←DI+1 (decrementare pentru DF=1)
LODSB(W)	;se încarcă AL(AX) de la [DS:SI] ;SI←SI+1 (decrementare pentru DF=1)
STOSB(W)	;se stochează AL(AX) la [ES:DI] ;DI←DI+1 (decrementare pentru DF=1)
REP	;prefix de repetare. Contorul este CX

4.6. MODURI DE ADRESARE

Pe măsura evoluției arhitecturii calculatoarelor, s-a dezvoltat și o mare varietate de moduri de adresare a instrucțiunilor și datelor, scopurile principale fiind:

- optimizarea spațiului de memorie ocupat de instrucțiuni;
- facilități sporite privind prelucrarea unor structuri de date de tip vector (matrice):
- facilități privind implementarea în programe a unor structuri de control complexe.

Noțiunea de *mod de adresare* se referă la modul în care în formatul instrucțiunii se specifică adresa unui operand de prelucrat, adresa de stocare a rezultatului operației, sau adresa următoarei instrucțiuni. Prin decodificarea instrucțiunii procesorul va recunoaște modurile de adresare folosite (pentru operanzi și rezultat) și va putea calcula locul în care se găsesc operanzii ceruți de execuția instrucțiunii.

Operanzii pot fi imediați, sau se pot găsi în registre interne UCP, în memorie, sau în registre port de I/O. În cazul unui program executabil, valoarea operanzilor este calculată în momentul compilării / asamblării programului sursă pentru operanzii imediați, în momentul încărcării în memoria principală pentru adresarea directă la memorie și respectiv, în momentul execuției pentru operanzii adresați în registre sau adresați indirect.

În general instrucțiunile care specifică numai operanzi în registre sunt compacte și se execută rapid, pentru că registrele sunt incluse în UCP și nu mai sunt necesare transferuri pe magistrala externă. Registrele pot conține operanzi sursă și/sau destinație).

Operanzii imediați sunt date constante de 8, 16, 32, sau 64 biți conținute în instrucțiune. La microprocesoarele care au memorie tampon internă pentru instrucțiuni și date, (de exemplu coada de instrucțiuni a lui 8086, sau memorie cache internă) operanzii imediați pot fi adresați la fel de rapid ca și cei din registre, la momentul execuției instrucțiunii, ne-fiind necesare operații suplimentare pentru extragerea lor. Limitările operanzilor imediați se datorează faptului că ei au doar valori constante și pot servi numai ca operanzi sursă.

Pentru fiecare mod de adresare descris în continuare, automatul de adresare al UCP calculează o *adresă efectivă (AE)* a operandului. Obținerea AE se face în funcție de modul de adresare selectat prin cuvântul de cod al instrucțiunii. AE este adresa care va fi folosită de instrucțiune pentru a găsi operandul. La microprocesoarele cu memoria segmentată, AE rezultă ca un întreg fără semn ce reprezintă deplasamentul (adresa relativă a operandului) fata de începutul segmentului în care se află.

În descrierea modurilor de adresare se vor folosi notațiile:

- **r** - registru (eventual r8, r16 etc pentru a indica dimensiunea), litera referindu-se la conținutul acestui registru.
- **d** - deplasament (adresă relativă). Este o valoare care participă la calculul AE, fiind tratat de obicei ca un număr cu semn în cod complement față de doi. Adunarea deplasamentului poate genera o adresa mai mică sau mai mare decât adresa de bază (pentru reprezentarea lui d cu 8 biți în C2, variația față de adresa de bază se face în intervalul -128 ... +127. Restul componentelor care intră în calculul AE sunt tratate ca numere fără semn.

- **x** - adresă de memorie; se referă la conținutul locației de memorie de adresă x.
- **@x** - adresa de la adresa x (adresa conținută în locația de memorie cu adresa x; la adresa x se găsește nu o simplă dată, ci un pointer către altă adresă). De asemenea prin **@r** se indică că operandul este conținutul locației de memorie cu adresa conținută în registrul r. Adesea însă, atunci când poate fi vorba fie de locații de memorie fie de registre interne în același format de instrucțiune, se va folosi doar litera x.

Nu se va presupune un anumit tip de microprocesor; deci cuvintele pentru codul operației, registrele implicate, valorile numerice numite deplasări și operanzi, pot avea 8, 16 sau mai mulți biți, în funcție de tipul instrucțiunii și de tipul microprocesorului. În documentațiile de firmă apar de obicei mult mai multe moduri de adresare decât folosim în clasificarea de mai jos, pentru că se fac particularizări ale fiecărui mod de adresare. Motivele pot fi și comerciale, dar toate modurile de adresare pot fi încadrate în clasificarea de mai jos.

Pentru a simplifica explicația, se va presupune în continuare că memoria este organizată pe octeți, că se folosește convenția de memorare Little Endian și că este adresată liniar (și nu segmentat ca la procesoarele I80x86).

Dacă ne referim la segmentarea memoriei, la microprocesorul I8086, memoria este segmentată în segmente logice de câte 64 KB. Pentru I8086 pot exista 4 segmente active simultan în memorie, segmente a căror adresă de bază se găsește în registrele segment CS, DS, SS, ES. Prin modurile de adresare specifice lui I8086 se calculează o adresă efectivă de 16 biți (care este un deplasament în cadrul segmentului curent).

Adresa fizică (AF) se obține din adresa efectivă (AE) la care se adaugă adresa de segment (AS) înmulțită cu 16, ca în ecuația:

$$AF = AS \times 2^4 + AE \quad (4.2)$$

Această sumare nu se face de către ALU ci de către un sumator separat de 20 biți și este transparentă pentru programator.

Standardul IEEE pentru limbajul de asamblare al microcalculatoarelor amintește cel puțin 19 moduri diferite de calcul al adreselor efective atât pentru operanzii sursă cât și pentru cei destinație. Se va face clasificarea modurilor de adresare în 5 tipuri [Gorsline86]:

1. adresarea imediată
2. adresarea directă
3. adresarea indirectă
4. adresarea relativă
5. adresarea indexată

4.6.1. Adresare imediată

Corpul (formatul) instrucțiunii înglobează și operandul care apare *imediat* după codul operației, așa cum se indică în reprezentarea grafică a modului de adresare din figura 4.5.

Operandul este format dintr-unul sau mai mulți octeți succesivi. Modul imediat de adresare este folosit numai pentru adresarea datelor constante ale programului. La microprocesoarele de 8 biți sunt necesare 2 sau mai multe citiri din memorie (cod + data). La microprocesoarele de la 16 biți în sus, cu tampon local (de tip FIFO sau cache) de date și instrucțiuni, aceste instrucțiuni sunt executate rapid, pentru că operandul se afla în tamponul microprocesorului.

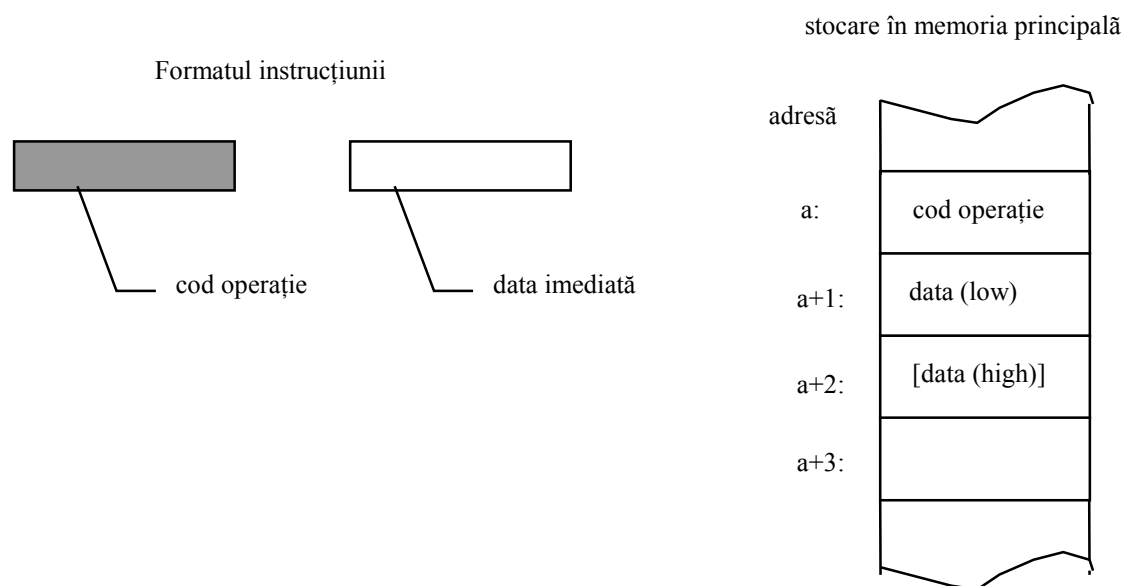


Figura 4.5. Explicativă la modul de adresare imediată

Exemple (I8086):

```
and al,0FFh      ; adresare imediată la data 0FF
mov ax,3A40h     ; adresare imediată la data 3A40hex
                  ; în memorie se scrie b8 40 3a , unde b8 este codul operației iar data
                  ; multi-octet se stochează în convenția little endian
mov al,5         ; în hexa, în memorie= b0 05
mov ax,5         ; în hexa, în memorie = b8 05 00
```

4.6.2. Adresare directă

În instrucțiune apare un câmp ce specifică o adresă de date (fixă - nu mai poate fi modificată în timpul rulării). Adresa apare după codul operației instrucțiunii, ca și la adresarea imediată, dar de data aceasta ea nu reprezintă un operand - dată imediată, ci adresa unde se găsește operandul. Această adresă poate fi identificatorul (nume) al unui fanion de condiții, adresa unui registru intern UCP (operandul este conținut de registru), sau o adresa din memoria principală (adresa la care se găsește operandul). Din punctul de vedere al locului specificat de adresă există variantele acestui mod de adresare, numite:

- **adresare directă la registru (implicită),**
- **adresare directă la memorie (absolută sau extinsă),**
- **adresare directă la fanion/indicator** (pentru testarea acestuia); este cazul adresărilor directe ale instrucțiunilor, atunci când se face un salt condiționat sau o chemare de procedură condiționată.

În cazul *adresării directe la registru*, ($AE = r$), operandul este conținut într-un registru. Informația pentru AE este de multe ori inclusă în codul operației, prin codul ce indică adresa registrului intern UCP implicat în instrucțiune. Avantajele acestui mod de adresare se referă în primul rând la codificarea ușoară a adresei (numărul de registre interne nu este foarte mare) și execuția rapidă, pentru că nu este necesar un acces suplimentar la memoria principală, exterioară UCP. Atunci când adresa registrelor este inclusă implicit în codul operației, acea instrucțiune

lucrează doar cu acele registre, iar adresarea este numită **adresare implicită**. În cazul arhitecturilor RISC, adresele registrelor interne apar întotdeauna explicit în formatul binar al instrucțiunilor.

Pentru unele microprocesoare, termenul "*implicit*" are o semnificație mai aparte: informația se află implicit (sau rezultatul se va transfera implicit) într-un registru dedicat numit acumulator (deci nu mai exista biți de adresare pentru acel registru).

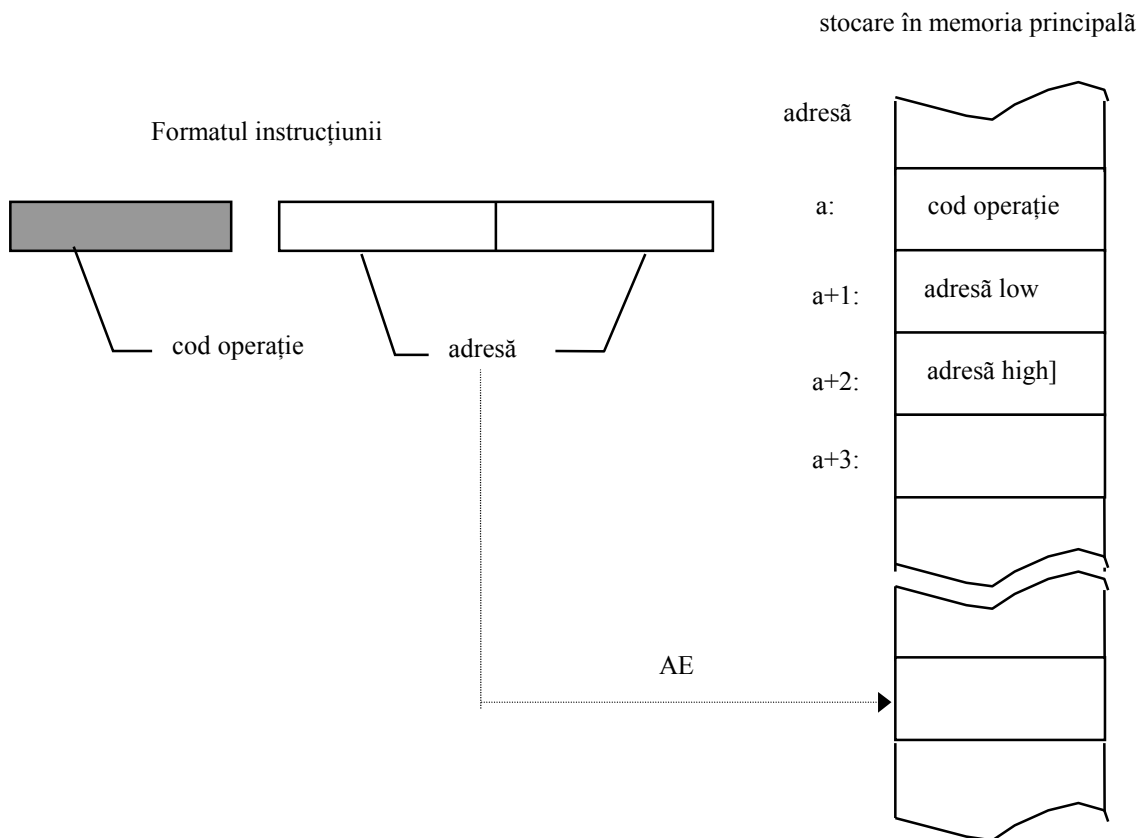


Figura 4.6. Explicativă la modul de adresare directă la memorie

Adresarea directă la memorie mai este numită adresare absolută, sau adresare extinsă; iar adresa efectivă se poate scrie ca fiind $AE = x$. Instrucțiunile cu acest mod de adresare conțin chiar adresa efectivă a operandului. AE apare în corpul instrucțiunii, după codul operației. Adresarea directă la memorie se folosește mai ales la instrucțiunile de transfer între registrele interne și memorie, precum și la instrucțiunile de salt absolut.

Numărul de octeți din câmpul operand depinde de tipul microprocesorului. De exemplu la microprocesoarele de 8 biți, adresa este de obicei pe 16 biți (spațiu maxim de adresare de 64 KB). La cele de 16 biți, cu adresa segmentată, cum sunt procesoarele Intel, există două situații de referire la o adresă de memorie:

- adresă în interiorul aceluiași segment (AE = adresă pe 16 biți)
- adresă în afara segmentului curent, care necesită AE pe 32 de biți.

La microprocesoarele care au spații de adresare separate pentru memorie și pentru dispozitivele de I/O există o variantă a modului de adresare directă, numită *adresare directă la port*. De obicei în aceste cazuri adresa portului are dimensiunea de 8 biți, deci pot fi adresate maximum 256 porturi

Exemplu pentru I8086:

out adr_p8,al ;Se transferă conținutul registrului de 8 biți numit "al" la adresa de

```

;port adr_p8
in ax,adr_p8 ;Se citesc 16 biți de la portul de adresă adr_p8 și se transferă în registrul
;de 16 biți numit "ax"

```

4.6.3. Adresare indirectă

La acest mod de adresare în câmpul operand al instrucțiunii se specifică un registru sau o adresă de memorie principală din care se va citi adresa operandului. Există astfel adresare indirectă prin registru (AE = @r) și adresare indirectă prin memorie (AE = @x).

De exemplu, se poate scrie:

```

mov ax,@bx ;adresa de memorie se găsește în registrul bx. Se transferă în
;registrul ax informația din memorie de la adresa dată de
;conținutul registrului bx
mov ax,@there ;locația de memorie de adresa there conține adresa datelor.
;Se transferă în registrul ax informația din memorie de la adresa
;dată de conținutul locației cu adresa simbolică there

```

La *adresarea indirectă prin registru*, adresa operandului este conținută în registrul specificat de codul instrucțiunii. Acest mod de adresare permite o flexibilitate sporită în localizarea informațiilor în memorie. În formatul instrucțiunii se adresează un registru intern UCP, (sau o pereche de registre), în care se află AE a operandului. Este un mod util de adresare atunci când adresa efectivă a operandului este variabilă. Adresa registrului de adresare se poate specifica *explicit* (la procesoarele cu un mare număr de registre generale, unde oricare registru poate fi folosit la adresare indirectă) sau *implicit*, (adresa binară a registrului nu apare explicit în vectorul binar al instrucțiunii) în cazul în care la acest mod de adresare se poate folosi doar un registru (dublu).

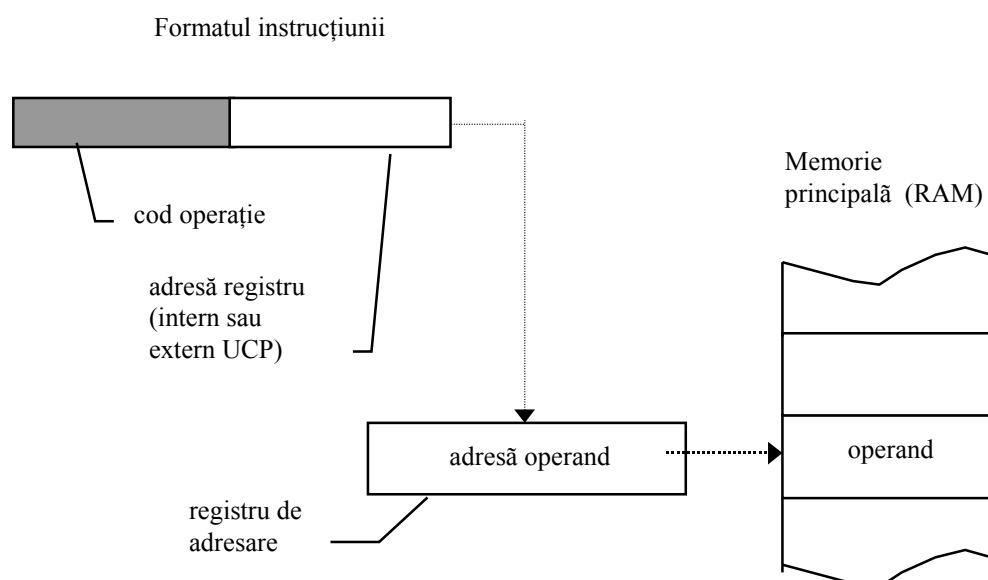


Figura 4.7. Explicativă la modul de adresare indirectă la memorie

Adresarea indirectă prin registru (figura 4.7) poate fi folosită și la instrucțiunile de apel al procedurilor, adresa efectivă găsindu-se indirect, prin citirea conținutului unui registru de adresare.

De exemplu la Intel 8086 se poate scrie:

call bx ;bx → ip, care face o adresare *directă* la registrul bx

sau

call [bx] ;@bx → IP_L și @(bx+1) → IP_H, acesta fiind o adresare indirectă prin
;registrul bx. Adresa de început a procedurii se găsește în memorie
;la adresa dată de registrul bx.

La microprocesoarele care au spații de adresare separate pentru memorie și pentru dispozitivele de I/O există o variantă interesantă a *adresării indirecte prin registru a porturilor*. Această adresare este similară cu adresarea indirectă prin registre a memoriei. Numărul porturilor posibil de adresat este dat de dimensiunea registrului folosit în adresare. De exemplu, la I8086, adresa portului este stocată în registrul “dx” de 16 biți, deci numărul maxim de porturi adresabile este 65536. La acest microprocesor, registrul dx este singurul ce poate fi folosit la adresare indirectă a porturilor și de aceea registrul apare implicit în codul operației. De exemplu codurile (în hexazecimal) ale instrucțiunilor pentru I8086 sunt:

in ax,dx ;cod instrucțiune = ED hex. Aduce (*input*) în registrul intern ax
;data de la portul specificat prin adresa din registrul dx

out dx,al ;cod instrucțiune = EE hex. Transmite (*output*) din registrul intern al
;data de la portul specificat prin adresa din registrul dx

out dx,ax ;cod instrucțiune = EF hex. Transmite (*output*) din registrul intern ax
;data de la portul specificat prin adresa din registrul dx

Din ultimele două exemple se observă că există un singur bit care specifică transferul pe 8 biți (în registrul al) sau pe 16 biți (în registrul ax).

Adresarea *indirectă prin memorie*, $AE = @x$, este dezavantajoasă din punctul de vedere al vitezei, pentru că, pentru a găsi operandul sunt necesare două citiri succesive ale memoriei. Acest mod de adresare este folosit de multe microprocesoare în cazul servirii cererilor de întrerupere, adresarea indirectă făcându-se prin intrările tabelului vectorilor de întrerupere.

Unele calculatoare permit incrementarea sau decrementarea automată a registrului folosit pentru adresare, fie înainte fie după utilizare. Rezultă astfel variante de adresare pentru adresarea indirectă prin registru:

a) *indirectă prin registru cu auto-post-incrementare*: se face acces la o locație de memorie a cărei adresă este conținută într-un registru, iar după fiecare acces de acest fel registrul este incrementat automat cu un număr egal cu dimensiunea datei.

De exemplu, dacă registrele r1 și r2 sunt de 16 biți, iar memoria este adresabilă pe octet, se poate scrie:

mov r1,@r2+ ;registrul r2 conține adresa de memorie unde se găsesc datele,
;să zicem adresa 61h. După execuția instrucțiunii (copiere
;din memorie în registrul r1, conținutul lui r2 va fi 63h pentru
;că s-au citit doi octeți, de la adresele 61h și 62h

b) *indirectă prin registru cu auto-post-decrementare*: se face acces la o locație de memorie a cărei adresă este conținută într-un registru, iar după fiecare acces de acest fel registrul este decrementat automat cu un număr egal cu dimensiunea datei.

De exemplu:

mov r1,@r2- ;registrul r2 conține adresa de memorie unde se găsesc datele,
;să zicem adresa 61h. După execuția instrucțiunii (copiere
;din memorie în registrul r1, conținutul lui r2 va fi 5Fh (61_{hex}-2)
;pentru că s-au citit doi octeți, de la adresele 61h și 62h

b) indirectă prin registru cu auto-pre-incrementare: se face acces la o locație de memorie a cărei adresă este conținută într-un registru și al cărui conținut a fost incrementat automat înainte de utilizare, cu un număr egal cu dimensiunea datei.

De exemplu:

```
mov r1, +@r2 ;registrul r2 conține adresa de memorie unde se găsesc datele,
              ;să zicem adresa 61h. Citirea datelor și copierea în r1 începe
              ;de la adresa 63h.
```

b) indirectă prin registru cu auto-pre-decrementare: se face acces la o locație de memorie a cărei adresă este conținută într-un registru și al cărui conținut a fost decrementat automat înainte de utilizare, cu un număr egal cu dimensiunea datei.

De exemplu:

```
mov r1, -@r2 ;registrul r2 conține adresa de memorie unde se găsesc datele,
              ;să zicem adresa 61h. Citirea datelor și copierea în r1 începe
              ;de la adresa 5Fh.
```

În acest grup de adresări indirecte prin registru, cu actualizarea automată a registrului de adresare este inclusă și tehnica de adresare a memoriei stivă. La memoria stivă data este conținută în locația de memorie din vârful curent al stivei, locație a cărei adresă este totdeauna conținută în registrul indicator de stivă (SP).

Adresarea stivei se face indirect prin registrul SP, cu incrementarea sau decrementarea automată a conținutului lui SP, în funcție de operația efectuată cu stiva. Pentru introducerea informației în stiva ("push" - adăugarea unui element în stivă se consideră conform convenției Intel - adăugare către adrese mici) se decrementează mai întâi SP (operație de pre-decrementare), iar apoi informația se stochează în memorie în locația indicată de adresa efectivă conținută în SP. Pentru operația de extragere din stivă ("pop") se extrage din stivă conținutul locației adresate de SP și apoi se incrementează SP (post-incrementare).

Operația de scriere în stivă ("*push sursa*") poate fi scrisă

```
mov -@SP, sursa
      care presupune succesiunea de operații:
          SP ← SP - N          ; N indică numărul de octeți înscriși
                               ; în stivă în cadrul operației curente.
          @SP ← sursa          ; sursa este conținutul unui registru
                               ; sau locație de memorie
```

Operația de extragere din stivă ("*pop dest*") într-un registru destinație poate fi scrisă

```
mov dest, @SP+
      care presupune succesiunea de operații:
          dest ← @SP
          SP ← SP + N
```

4.6.4. Adresare relativă

La acest mod de adresare în corpul instrucțiunii se găsește un operand (un deplasament, notat cu d) care este adunat la adresa de start a unei secțiuni de memorie, adresă conținută într-un registru desemnat explicit sau implicit în instrucțiune.

Putem deosebi următoarele variante de adresare relativă:

a) *adresare relativă bazată* (numită și *indirectă prin registru bază, cu deplasare*);

$$AE = @(r + d)$$

De exemplu:

mov ax,[bx+8] ;daca în bx se gaseste adresa 07h citirea operandului ce se va copia în ax se face de la adresa $7+8 = 15 = 0fh$. Pentru că registrul este pe 16 biți iar memoria este adresată pe octet, se citesc locațiile 0fh și 10h

Adresa efectivă (AE) este suma dintre conținutul; registrului bază și deplasarea. Modul de adresare este folosit pentru adresarea unor structuri de tip tablou. Prin registru de bază se înțelege un registru utilizat pentru calculul de adresă efectivă. Registrul de bază păstrează adresa de început a unei zone de memorie, unde sunt încărcate programe aflate în execuție sau blocuri de date.

Este un mod de adresare util atunci când se urmărește accesul la un operand aflat într-o tabelă ce poate ocupa diferite locuri în memorie. Pentru calculul AE la I8086, se adună conținutul registrului bp sau bx la un deplasament furnizat în cadrul instrucțiunii. Pentru calculul adresei fizice se folosește registrul ds sau ss, după cum calculul adresei efective folosește registrul bx respectiv bp.

$$\text{Adresa_fizică} = \text{ds} * 16 + \text{bx} + \text{d} \quad \text{sau}$$

$$\text{Adresa_fizică} = \text{ss} * 16 + \text{bp} + \text{d}$$

b) adresare relativă la contorul de program (PC)

În mod obișnuit la execuția de secvențe succesive de instrucțiuni din memorie avem de-a face cu o adresare indirectă prin registrul PC cu auto-post-incrementare. Exemplul următor, poate deci fi trecut ca exemplu și la adresarea indirectă: $\text{AE} = @\text{PC} +$

Adesea acest mod de adresare relativă este numit *adresare secvențială*. În cazul unor ramificații însă, la conținutul PC se adună deplasamentul cu semn din câmpul operand.

De exemplu:

```
IT:  mov dx,there      ; instrucțiune cu lungime de 3 octeți
      mov ax,1732h     ; lungime de 3 octeți
      mov bx,cx        ; lungime de 2 octeți
      jmp *-9          ; lungime de 2 octeți. La primul octet se decodifică salt relativ
                        ; se face salt la instrucțiunea cu eticheta IT.
```

4.6.5. Adresare indexată

Adresarea indexată este asemănătoare ca principiu, cu adresarea relativă bazată, dar spre deosebire de aceasta ea este folosită doar în adresarea datelor. Prin acest mod de adresare, adresa datelor din memoria principală este modificată cu ajutorul conținutului unui registru index. Adresarea indexată se utilizează pentru a referi elementele unui vector, poziția unui element din vector fiind determinată printr-un indice, iar indicele se poate modifica prin modificarea valorii conținute în registrul index.

Numărul de registre index dintr-un procesor indică numărul de tablouri potențiale care se pot adresa în memorie, iar mărimea deplasamentului determină dimensiunea maximă a tabloului. Numărul registrelor index este un criteriu de performanță pentru microprocesoare.

Indexarea se combină de obicei cu alte moduri de adresare putând exista variantele: adresare indexată directă la memorie, adresare pre-indexată indirectă la registru, adresare post-indexată indirectă la registru, adresare pre-indexată indirectă la memorie, adresare post-indexată indirectă la memorie, adresare indexată relativă la adresa de bază (ultima fiind adesea numită și adresare bazată și indexată). De observat că modurile combinate dintre indexare și adresările indirecte pot specifica că indexarea se va realiza înainte sau după ce indirectarea este efectuată. Astfel, similar cu cele expuse la adresarea indirectă prin registre:

$$\text{AE} = @(\text{ri} + \text{d}), \text{ sau}$$

$$\text{AE} = @(\text{ri} + \text{x}), \text{ sau}$$

$$AE = @(ri + r)$$

De exemplu, la microprocesorul I8086 se pot utiliza adresări indexate de genul:

a) Adresare indexată-directă la memorie

mov ax,adr1[si] ;fie adresa lui adr1 = 29h. Adresa datelor este egală cu adr1+ si,
;iar conținutul registrului index si se consideră a fi 2. Ca urmare
;copierea în registrul ax se face de la adresele 2Bh și 2Ch
; pentru că (29+02=2B hex)

b) Adresare indexată indirectă prin registru bază (adresare bazată și indexată)

mov ax,[bp][si] ;dacă conținuturile bazei și indexului sunt bp=34h, si=04h,
;datele se citesc începând cu adresa 38h

CAPITOLUL 5

UNITATEA DE CONTROL A UCP

Conținut:

- 5.1. Funcțiile principale ale unității de control a UCP
- 5.2. Control cablat
 - 5.2.1. Metoda tabelului de tranziție a stărilor
 - 5.2.2. Metoda sincronizării prin numărătoare de secvențe
- 5.3. Controlul microprogramat
 - 5.3.1. Metode de optimizare a UC microprogramate
- 5.4. Paralelism în execuția instrucțiunilor prin tehnici pipeline
- 5.5. Întreruperi și excepții
- 5.6. Rolul memoriei stivă
- 5.7. Utilizarea ferestrelor de registre

Obiective educaționale:

- a. Analiza funcțiilor principale ale unității de control și a modului de interpretare și execuție a instrucțiunilor mașină
- b. Explicarea metodelor de proiectare și implementare a unităților de control
- c. Analiza comparativă a avantajelor și dezavantajelor modurilor diverse de implementare
- d. Explicarea fundamentelor privind implementarea paralelismului la nivelul instrucțiunii prin tehnicile de tip pipeline
- e. Descrierea și clasificarea sistemului de întreruperi al microprocesoarelor
- f. Analiza modului de implementare și utilizare a memoriei stivă

5.1. FUNCȚIILE PRINCIPALE ALE UNITĂȚII DE CONTROL A UCP

Structura calculatorului, cu componentele de bază descrise de von Neumann, rămâne, în general, valabilă și astăzi. Partea cea mai importantă a calculatorului este Unitatea Centrală de Procesare (UCP), sau procesorul, în a cărei structură am deosebit în capitolul 1 unitățile de prelucrare a datelor și de control, astfel că într-un calculator, procesorul conține totul cu excepția memoriei și a sub-sistemului de intrare - ieșire. Funcțional, putem spune că UCP este formată dintr-o parte de calcul (unitate de prelucrare a datelor, sau *cale de date*) și o parte de control (*cale de control*). Călea de date este un ansamblu de unități funcționale capabile să regăsească datele și să le prelucereze. Ea cuprinde: unitatea / unitățile aritmetice și logice, registrele de uz general și căile de comunicare (magistrale interne UCP) dintre acestea. Rolul unității de control a unui procesor ce recunoaște un set de instrucțiuni, este să aducă instrucțiunile codificate binar din memoria principală (externă procesorului) și să transmită semnale de control către unitatea de prelucrare (calea de date), și de asemenea către memorie și sub-sistemul de I/O, în scopul executării instrucțiunilor. Semnalele de control selectează funcțiile ce trebuiesc executate în calea de date la momente discrete de timp și conduc datele de la / la unitățile funcționale potrivite. Momentele de timp individuale sunt definite de către impulsurile ce provin de la circuitul de ceas al UCP. Conform semnalelor emise, unitatea de control (UC) face, periodic, o *reconfigurare* din punct de vedere logic a unității de prelucrare, astfel încât aceasta să execute un set impus de (micro)operații. Pentru execuția corectă a unui program UC are două funcții principale: secvențierea instrucțiunilor și interpretarea acestora.

(a) *Secvențierea instrucțiunilor* se referă la maniera în care controlul procesorului este transferat de la o instrucțiune la alta, prin ordinea de generare a adreselor către memoria principală a calculatorului. Instrucțiunile sunt selectate pentru execuție într-o anumită ordine, dictată de UC prin interpretarea informației codificate în cadrul instrucțiunilor și pe baza unor semnale recepționate de UC de la calea de date sau din exteriorul procesorului. Semnalele din exteriorul procesorului sunt asincrone cu programul executat.

În principiu, fiecare instrucțiune ar putea conține adresa următoarei instrucțiuni ce trebuie adusă din memorie. Cele mai multe instrucțiuni dintr-un program au un succesor unic; astfel că dacă o instrucțiune I_i este stocată în memorie într-o locație de adresă A, iar I are ca unic succesor instrucțiunea I_{i+1} , atunci este natural să se stocheze I_{i+1} în locația de adresă A+1 ce urmează imediat după A. Adresa instrucțiunii I_i este conținută într-un registru de adresare a instrucțiunilor, iar adresa instrucțiunii I_{i+1} poate fi obținută pentru cazul de mai sus prin incrementare. Din această cauză vom numi registrul de adresare al instrucțiunilor "*contor de program*", notat pe scurt PC. Contorul de program este un indicator al adresei instrucțiunii curente. Adresa instrucțiunii I_{i+1} se calculează prin operația:

$$PC \leftarrow PC + c \quad (5.1)$$

unde c reprezintă numărul de cuvinte de memorie pe care se întinde instrucțiunea I_i .

În programe există de asemenea instrucțiuni de ramificare, sau se pot produce situații independente de program, care impun UC să facă trecerea către instrucțiuni aflate la adrese ne-consecutive. UC trebuie să controleze modul în care se fac aceste salturi, care pot fi necondiționate, condiționate (prin testarea unor indicatori de condiții), sau pot fi salturi pentru transferul controlului către un alt program sau subprogram (proceduri, întreruperi, excepții). În cazul unei ramificări, registrul de adresare PC se încarcă direct cu adresa la care se face saltul. După aceasta funcționarea PC se face conform relației (5.1) până la următoarea instrucțiune de ramificare.

(b) *Interpretarea instrucțiunilor* se referă la modul în care UC decodifică codul fiecărei instrucțiuni și la modalitatea de generare a semnalelor de control către calea de date, pentru a comanda execuția instrucțiunilor. Comportarea UC din punctul de vedere al semnalelor de control generate (ca funcție și ca succesiune în timp) pentru comanda operațiilor se poate descrie pe baza unor tabele de tranziție a stărilor, organigrame, limbaje de descriere, dar și pe baza unor combinații ale metodelor amintite. Fiecărei instrucțiuni interpretate de UC, îi corespunde o succesiune de funcții logice de ieșire din UC, conform algoritmului dorit de proiectantul UC.

Tipurile de semnale de intrare și ieșire dintr-o unitate de control tipică pot fi descrise cu ajutorul figurii 5.1.

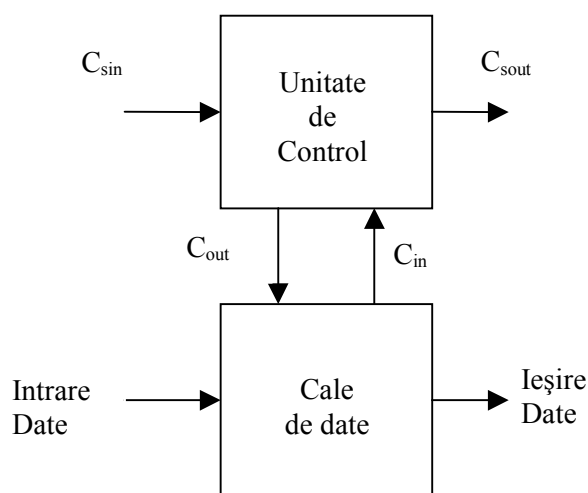


Figura 5.1. Semnale de intrare și ieșire al unității de control la interfața cu calea de date și cu alte sub-sisteme

Cele 4 grupuri de semnale de control indicate în figura 5.1. au următoarele funcții:

1. C_{out} - sunt semnale ce controlează direct funcționarea căii de date. Generarea acestor semnale reprezintă principală sarcină a UC.
2. C_{in} - sunt semnale ce permit căii de date să influențeze funcționarea unității de control în sensul modificării secvenței de comenzi C_{out} generată. De exemplu, semnalele C_{in} pot indica prezența unor condiții neobișnuite în unitatea de prelucrare a datelor, cum ar fi depășirea capacității de reprezentare ca urmare a unei operații aritmetice.
3. C_{sout} - aceste semnale sunt transmise la alte unități de control, (ale altor procesoare de uz general sau specific) și pot indica condiții de stare, și semnale de sincronizare, cum ar fi "busy / ocupat", cerere de cedare, sau acceptare a cedării controlului magistralelor sistemului, sau semnale de informare că o operație cerută anterior s-a terminat.
4. C_{sin} - Semnale primite de la alte unități de control, de exemplu de la un controller supervisor. De obicei includ semnale de *start* și *stop* și informații de sincronizare, funcțiile fiind similare cu cele ale C_{sout} . Grupurile de semnale C_{sin} și C_{sout} sunt în primul rând folosite pentru a sincroniza unitatea de control cu operațiile altor unități de control.

Organizarea internă aleasă pentru UC și aspectele specifice de implementare folosite la proiectarea unității, influențează în mod direct viteza cu care lucrează UC, costurile de proiectare și construcție și suprafața ocupată pe circuitul integrat de către aceasta unitate. Acest ultim aspect

indică faptul că la microprocesoare, metodele de proiectare ale UC influențează întreaga arhitectură a UCP prin constrângerile impuse datorită suprafeței limitate a chip-ului de siliciu.

Orice proiect de unitate de control pornește de la descrierea setului de instrucțiuni recunoscute și de la descrierea proiectului căii de date.

Există două metode de proiectare și implementare a UC: cablat și microprogramat.

1. *Unitatea de control cablată* este un automat secvențial proiectat și construit pentru generarea unui set specific și într-o secvență fixă de semnale de control. Odată ce UC a fost construită, algoritmul de control nu mai poate fi modificat decât prin reproiectarea întregii structuri. La controlul cablat, scopurile principale urmărite la proiectarea automatului se referă la minimizarea numărului de componente utilizate și maximizarea vitezei de operare.
2. La *unitatea de control microprogramată* semnalele de control sunt incluse în cuvinte binare succesive (microinstrucțiuni) stocate într-o memorie de mare viteză, inclusă în UC, numită *memorie de control*. Implementarea algoritmului de control se face prin programarea conținutului memoriei de control. Fiecărei instrucțiuni recunoscute de procesor îi corespunde o secvență de microoperații generate prin citirea unei secvențe de microinstrucțiuni din memoria de control (secvență de microinstrucțiuni numită *microprogram*). Microprogramele conținute în memoria de control formează un interpretor al setului de instrucțiuni recunoscut de procesor.

Tehnica cablată are câteva avantaje:

- dimensiuni mici ale UC
- viteză mare de operare

Tehnica microprogramată are și ea câteva avantaje:

- un set de instrucțiuni recunoscut poate fi ușor schimbat prin schimbarea microprogramelor și nu prin reproiectarea întregii UC
- permite o metoda sistematică de proiectare a UC
- dacă este nevoie, permite modificarea dinamică (în timpul rulării programelor) a setului de instrucțiuni recunoscut de UC prin utilizarea unei memorii de control cu scriere-citire

5.2. CONTROL CABLAT

Unitatea de control este un automat secvențial proiectat special pentru a recunoaște un anumit set de instrucțiuni, respectiv pentru a genera set corespunzător (pre-fixat) de semnale de control. Cu cât setul de instrucțiuni este mai complex, cu atât proiectarea și optimizarea proiectului este mai dificilă. De aceea acest tip de control este folosit pentru procesoare care recunosc un set regulat și relativ redus de instrucțiuni (de obicei instrucțiuni cu format de lungime fixă).

Unitatea de control trebuie să lanseze semnalele de comandă pentru aducerea instrucțiunilor din memoria principală, să decodifice codul fiecărei instrucțiuni și ca urmare să genereze toate semnalele de control către calea de date pentru execuția completă a instrucțiunii. Cu cât unitatea de control a procesorului recunoaște un set mai mare și mai eterogen (lungimi diferite, câmpuri diferite în funcție de modurile de adresare etc.) de instrucțiuni, cu atât complexitatea unității de control crește.

Ca schemă de principiu, descrisă în figura 5.2., unitatea de control cablată este un automat ce cuprinde un registru de memorare a stării curente, registrul de instrucțiuni, care memorează codul operației și logică de control (de obicei combinațională). În schema simplificată din figură nu s-a ținut cont de faptul că unitatea de control cuprinde și registre de adresare a memoriei principale și nu s-au desenat semnalele de control și sincronizare pentru registrele interne ale unității de control. Registrul de instrucțiuni păstrează codul operației instrucțiunii curente pe toată durata execuției instrucțiunii. Execuția poate fi descrisă prin mai multe stări ale automatului, stare următoare fiind generată de logica de control conform algoritmului cablat și conform reacțiilor de la calea de date. Starea următoare se încarcă în registrul de stare sincron cu impulsul de ceas. Complexitatea logicii

de control este proporțională cu numărul de intrări (biți pentru cod instrucțiune, reacție cale de date și cod al stării curente) și de numărul maxim de semnale de control de ieșire.

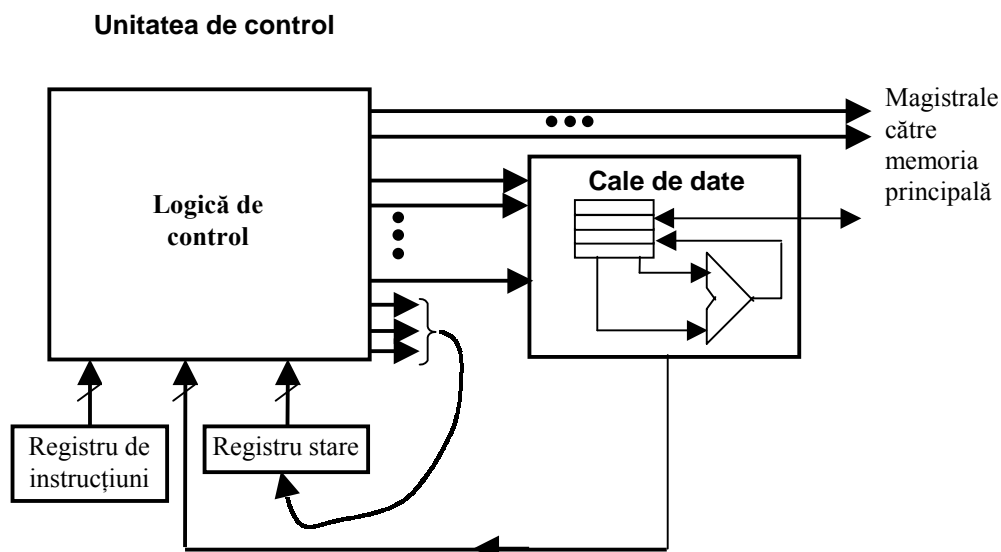


Figura 5.2. Unitate de control cablată, cu o structură simplificată și relația cu calea de date și memoria principală.

Pentru o structură cablată, în [Patterson96] se consideră că numărul de linii de intrare este 21 iar numărul de ieșiri 40 și se calculează dimensiunea logicii de control, dacă ar fi implementată cu ajutorul unui circuit ROM, deci *fără minimizare*. Dimensiunea rezultă $2^{21} \times 40$ biți = $2M \times 5 \times 8$ biți, adică 10 MB de ROM. Rezultatul indică obligativitatea minimizării logicii de control, din cauza dimensiunii exagerate a unității de control, deși în exemplu s-a considerat un număr relativ mic de intrări și ieșiri.

Metodele de proiectare ale unităților de control cablate sunt metodele folosite pentru implementarea cablată a oricărui automat secvențial. Metodele moderne de proiectare asistată de calculator includ limbajele de descriere hardware, care permit automatizarea diferitelor faze ale proiectării [Toacse96].

Pentru înțelegerea metodelor de proiectare a acestor unități de control de dimensiuni relativ mici vom descrie pe scurt două metode.

5.2.1. Metoda tabelului de tranziție a stărilor

Este o metodă standard de descriere și proiectare a automatelor secvențiale și poate încorpora tehnici sistematice de minimizare a porților și bistabililor. Tabelul de tranziție a stărilor cuprinde toate stările automatului de control și corespondența cu stările următoare și cu semnalele de control generate, în funcție de combinația semnalelor de intrare în automat. În figura 5.3. se prezintă un exemplu general de tabel, în care s-au notat cu C_{in} variabilele de intrare ale UC. Liniile tabelului corespund setului de stări interne $\{S_i\}$ ale mașinii. O stare internă este determinată de informația stocată în unitate în momente discrete de timp (perioade de ceas / clock). Informația ce corespunde liniei stării S_i și coloanei I_j (combinația I_j a semnalelor de intrare C_{in} în UC) are forma S_{ij}, C_{ij} , unde S_{ij} reprezintă starea următoare a unității de control, iar C_{ij} reprezintă setul de

semnale de ieșire corespunzător (C_{out}), semnale activate prin aplicarea setului I_j la intrările unității de control când aceasta este în starea S_j .

Stări interne	Combinații de intrare C_{in}			
	I_1	I_2	...	I_m
S_1	S_{11}, C_{11}	S_{12}, C_{12}	...	S_{1m}, C_{1m}
S_2	S_{21}, C_{21}	S_{22}, C_{22}	...	S_{2m}, C_{2m}
...
S_n	S_{n1}, C_{n1}	S_{n2}, C_{n2}	...	S_{nm}, C_{nm}

Figura 5.3. Exemplu general de tranziție al stărilor. Pentru fiecare stare S_i a automatului la aplicarea unui set de intrări I_j (pe coloană) se indică combinația (S_{ij}, C_{ij}) între stare următoare în care va trece automatul și setul de semnale de control generate în starea actuală.

Această metodă, deși sugestivă, este destul de greoaie în cazul în care UC are multe stări și multe combinații ale semnalelor de intrare. La fel ca și la următoarea metodă amintită aici, la descrierea prin tabel de tranziție a stărilor modul de codificare al fiecărei stări este deosebit de important din punctul de vedere al minimizării funcțiilor rezultate. Pentru aceasta se folosesc programe de proiectare asistată, care ajută la o codificare optimă a stărilor automatului și apoi la minimizarea circuitelor unității de control.

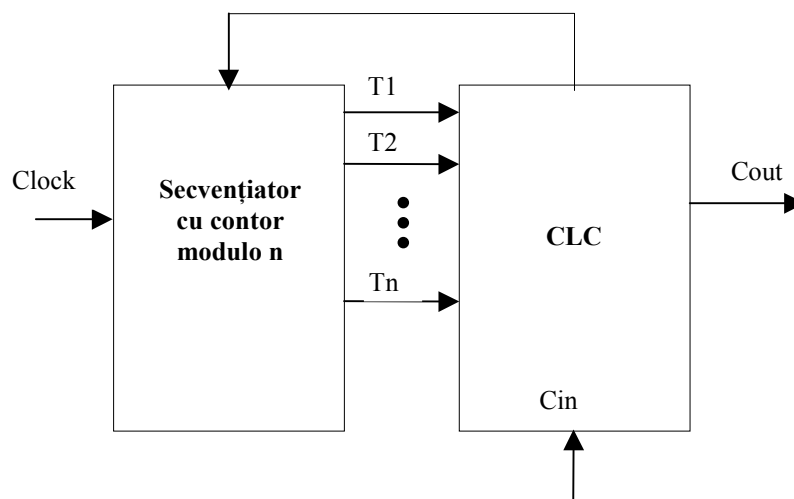


Figura 5.4. Exemplu de implementare cablată a unei unități de control cu ajutorul unui secvențiator modulo n și a logicii combinaționale. Modul de defazăre în timp a semnalelor T_i se poate vedea în figura 5.5b.

5.2.2. Metoda sincronizării prin numărătoare de secvențe

Această metodă utilizează circuite de generare a unor impulsuri succesive (faze) de sincronizare, circuite realizate cu numărătoare și circuite de decodificare. Fiecărei faze îi corespunde un set de semnale de control, numărul de faze fiind egal cu numărul maxim de stări ale UC. Vom numi aceste circuite *secvențiatoare*. Un secvențiator modulo n poate conține de exemplu un numărător modulo n a cărui ieșire este conectată la un decodificator 1:n sincronizat cu numărătorul.

Semnalele de control de la ieșirea UC vor fi activate pe baza acestor semnale de sincronizare (decalate în timp) cu ajutorul unei logici combinaționale (figura 5.4.).

Structura internă a unui circuit secvențiator ar putea arăta ca în figura 5.5. Pentru un semnal de ceas (Clock) continuu, numărătorul ciclează continuu prin cele n stări. Asta presupune că numărătorul are cel puțin m celule, iar n este mai mic sau egal cu 2^m . Cele m ieșiri ale numărătorului sunt apoi decodificate de către decodificatorul $1:n$ care este validat prin intrarea sa de *Validare* doar pe palierul pozitiv al impulsurilor de ceas. Decodificatorul generează semnale de impuls $\{T_i\}$ la liniile de ieșire ($i = 1, n$). Succesiunea $\{T_i\}$ împarte timpul cerut pentru un ciclu complet de numărare în n părți egale, semnalele T_i fiind numite *semnale de fază*.

Posibilitatea utilizării secvențiatoarelor cu contoare modulo n , de tipul amintit mai sus, rezultă și din faptul că multe circuite digitale sunt proiectate să realizeze un număr relativ mic de acțiuni, în mod repetat, funcționare ce poate fi descrisă ca o organigramă constând dintr-o singură buclă conținând n pași.

Dacă cei n pași succesivi corespund operațiilor elementare ce se efectuează la execuția unei instrucțiuni într-un procesor, este util ca unitatea care controlează aceste operații să fie capabilă să modifice operațiile realizate într-un pas oarecare i , în funcție de semnalele primite pe intrare de către UC și în funcție de starea i în care se găsește UC. De aceea în figura 5.4 s-au adăugat și semnalele de intrare C_{in} .

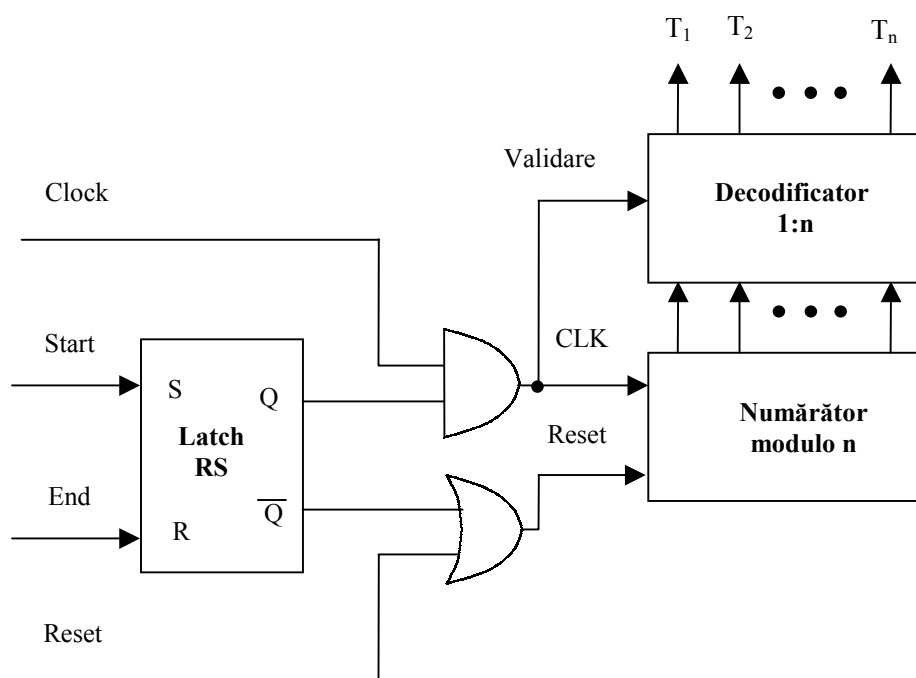


Figura 5.5a. Exemplu de structură de secvențiator construit cu pe baza unui numărător modulo n ale cărui ieșiri sunt decodificate sincron cu impulsul de ceas (clock)

Implementarea logicii combinaționale a automatelor UC se poate face în jurul unor celule de circuit tip PLA sau cu circuite proiectate complet pentru o anumită structură UC. În mașinile reale, circuitele UC nu pot fi implementate cu un singur PLA, pentru că dimensiunea acestuia rezultă foarte mare. De aceea un tabel mare este împărțit în mai multe PLA-uri ale căror ieșiri sunt multiplexate pentru a alege controlul corect. De asemenea, numărul de stări în diagrama cu stări finite poate avea influență mare asupra dimensiunii PLA. Ideea este să se dea numere similare pentru stări ce realizează operații similare. Diferența între modelul biților pentru astfel de stări se face de exemplu numai printr-un singur bit. Există programe CAD care ajută rezolvarea acestei probleme de atribuire a valorilor stărilor.

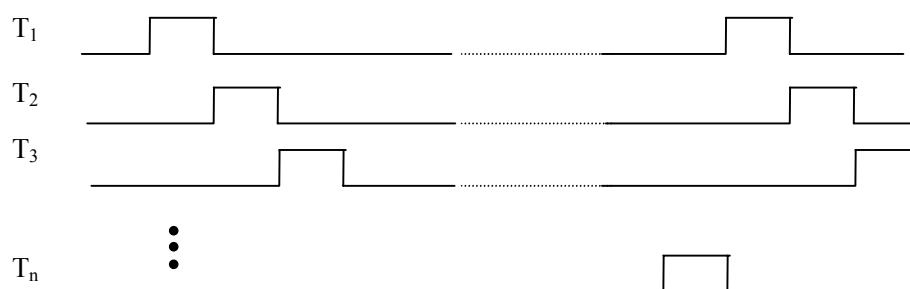


Figura 5.5b. Diagrama semnalelor de fază T_i generate de secvențiatorul modulo n din figura 5.5a.

5.3. CONTROLUL MICROPROGRAMAT

Acest mod de construcție a Unității de Control a fost enunțat în 1951 de către Maurice Wilkes [Hayes88]. Soluția sa a fost să transforme UC într-un calculator în miniatură, ce are două tabele în memoria de control. Primul dintre tabele specifică modul de control a căii de date, iar al doilea determină ordinea de generare (secvențierea) semnalelor de control pentru operațiile elementare. Soluția a fost numita de Wilkes "microprogramare" și de aici alte denumiri derivate: microinstrucțiune, microcod, microprogram, etc.

Microinstrucțiunile specifică toate semnalele de control pentru calea de date plus posibilitatea de a decide condițional care microinstrucțiune va fi următoarea executată (figura 5.6). Așa cum sugerează și numele "microprogramare", o dată ce calea de date și memoria pentru microinstrucțiuni, sunt proiectate (și construite), controlul devine în special o problemă de programare care constă în scrierea unui interpretor pentru setul de instrucțiuni. Microprogramarea permite astfel schimbarea unui set de instrucțiuni prin schimbarea conținutului memoriei de control (în care se stochează "tabelor" amintite de Wilkes), fără ca, de cele mai multe ori, să fie necesară și schimbarea structurii hardware. Structura microprogramului este asemănătoare cu diagrama de stări, în care pentru fiecare stare din diagramă corespunde o microinstrucțiune.

Fiecare bit dintr-o microinstrucțiune reprezintă o comandă elementară care fie merge direct la o resursă comandată (registru, sau CLC), fie este prelucrată împreună cu alte informații (pentru decodificare și sincronizare) pentru ca apoi să se comande resursa fizică.

Fiecare cod de instrucțiune citit și decodificat, va trebui să producă saltul la o adresă din memoria de control unde se găsește microprogramul asociat execuției acelei instrucțiuni. O posibilitate de efectuare a acestui salt, este chiar folosirea informației din op-code (codul operației) ca index într-un tabel cu adrese de start de microprograme. O altă posibilitate, care crește eficiența utilizării memoriei de control, este convertirea codului instrucțiunii curente într-un alt cod care se constituie în index de adresă.

La realizarea unei UC sub forma microprogramată, principalul scop este reducerea costurilor generale de proiectare și construcție a UC, de reducere a dimensiunilor microprogramelor și a dimensiunii microinstrucțiunilor. În mod normal, instrucțiunile care cer cel mai mare timp pentru execuție sunt optimizate ca viteză, iar celelalte instrucțiuni sunt optimizate ca spațiu ocupat în memoria de control.

Avantajele microprogramării, identificate de Wilkes în 1953, rămân valabile și astăzi:

- microprogramarea permite schimbarea ușoară a algoritmului de control. Această calitate este utilă în faza de dezvoltare când simpla schimbare a unui 0 în 1 în memoria de control, salvează uneori re-proiectarea componentelor hardware.

- prin emularea altui set de instrucțiuni, în microcod, se simplifică compatibilitatea software.
- reduce costurile în cazul adăugării unei instrucțiuni mai complexe la o microarhitectura standard.
- flexibilitate - construcția hardware poate începe înainte ca setul de instrucțiuni și microcodul să fie complet scris, pentru că specificarea controlului este doar o problema de software.

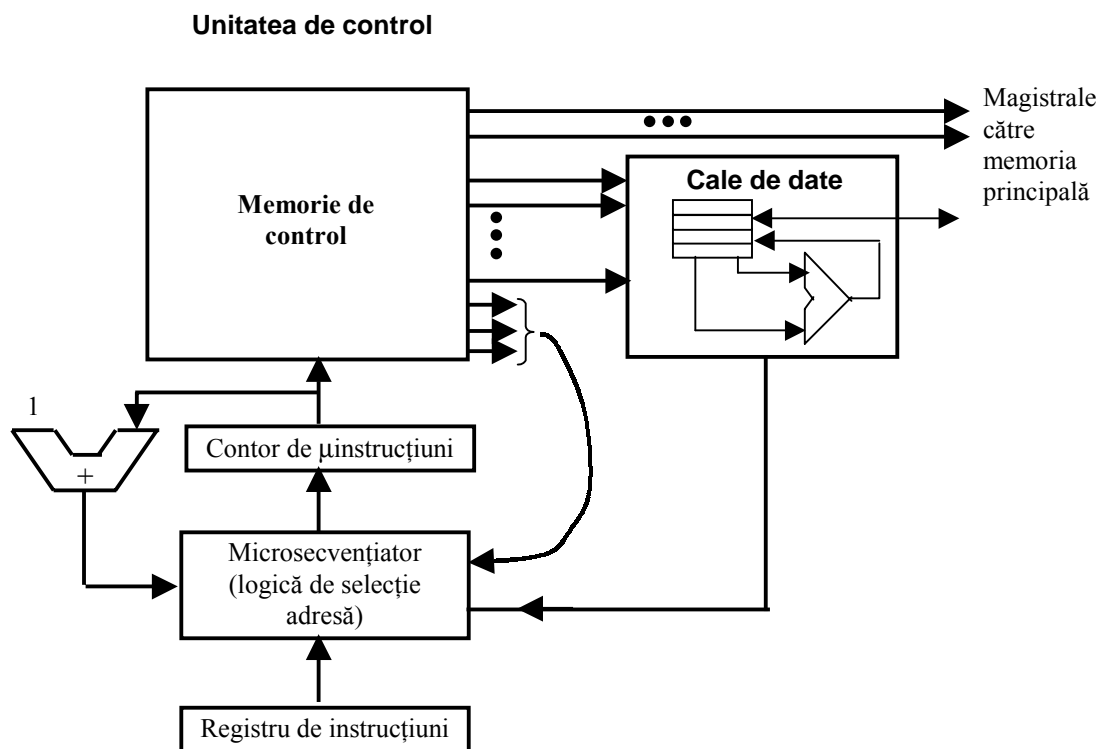


Figura 5.6. O unitate de control microprogramată simplificată. Deosebiri față de figura 5.2, cu UC cablată, se referă în primul rând la modul în care se calculează intrările în logica de control (microadresa pentru microinstrucțiune). În plus logica de control este implementată cu ajutorul unei memorii de control, de obicei de tip ROM, care specifică semnalele de control și semnalele de secvențiere.

Dezavantajul microcodului este viteza redusă de lucru, pentru că microprogramarea este dependentă de tehnologia de realizare a memoriilor.

Din punct de vedere istoric conceptul microprogramării (1953) a fost cu un pas înaintea tehnologiei timpului și a rămas aproximativ 10 ani doar o posibilitate virtuală de implementare. Wilkes a construit totuși unitatea de control a calculatorului numit EDSAC 2 utilizând controlul microprogramat, în 1958, cu o memorie de control ROM construită din miezuri magnetice. De abia în 1964 firma IBM a folosit microprogramarea la familia IBM 360. S-a realizat chiar emularea lui unei mașini anterioare IBM 7090 cu ajutorul noii mașini IBM 360, care a demonstrat că mașina hardware diferită poate rula aceleași programe uneori chiar mai rapid decât pe mașina originală. Pentru operația de emulare un calculator microprogramabil C1 poate fi utilizat pentru a executa programe scrise în limbajul mașină L2 al unui alt calculator, C2, prin plasarea unui emulator pentru L2 în memoria de control a lui C1. C1 este deci capabil să emuleze pe C2. Mașina pe care se face emularea este denumită mașină gazdă (host-machine), iar mașina care este simulată, mașină țintă (target machine). Prin emulare se poate face astfel o testare pe mașina gazdă a unui procesor înainte ca acesta să fie realizat fizic.

Cele mai multe din calculatoarele incluse astăzi în categoria CISC folosesc implementarea microprogramată a algoritmului de control al UC.

Principiul structurii de control microprogramate poate fi descris prin schema bloc din figura 5.7. Așa cum s-a descris în primul paragraf al acestui capitol unitatea de procesare a datelor este comandată pentru a executa funcții asupra unei structuri de date și ca urmare pot fi generate semnale ce caracterizează operația efectuată (indicatori de condiții). Comanda funcțiilor și interpretarea efectului produs se face aici cu un automat construit ca o structură de control microprogramată. Orice operație apelată printr-o instrucțiune (recunoscută prin câmpul cod al operației) a mașinii, presupune o secvență de pași ce poate cuprinde transferuri între registre interne, sau registre interne și registre de memorie. Fiecare din acești pași, realizează o microoperație, care se efectuează prin activarea unor semnale asociate registrelor respective. Execuția unei instrucțiuni mașină este deci compusă dintr-o secvență de microoperații comandată de un microprogram. Microprogramul este memorat de obicei într-un ROM de control.

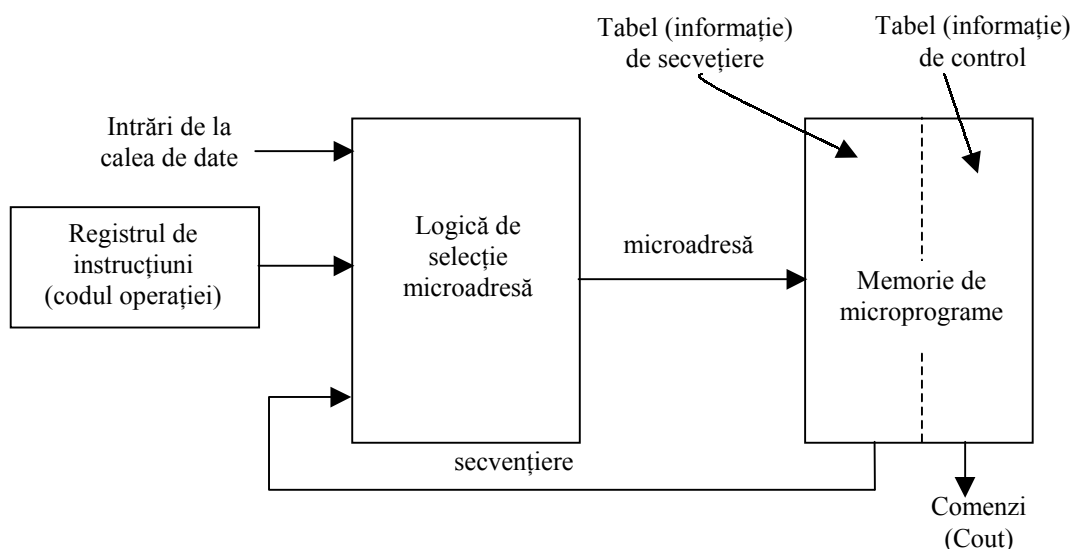


Figura 5.7. Schema bloc de principiu a unei structuri de control microprogramate. Informațiile de secvențiere și control incluse în microinstrucțiunile din memoria de microprograme au fost denumite "tabele" ca în enunțul structurii microprogramate a lui Wilkes.

Logica de selecție a adresei (microadresei) memoriei de control (sau de microprograme), calculează adresa următoarei microinstrucțiuni pe baza informației din registrul de instrucțiuni (în care se stochează codul operației instrucțiunii), pe baza unor informații externe ce provin de la calea de date, sau pe baza informației de secvențiere (de reacție) conținută în corpul microinstrucțiunii curente.

Fiecare locație adresabilă a memoriei de control este privită ca o microinstrucțiune. Atunci când locația este adresată ea va activa un anumit set de linii de control și linii de adresă de secvențiere. Un exemplu de microinstrucțiune, cu exemplificarea unor câmpuri specifice se prezintă în figura 5.8.

Destinație	Operație ALU	Adr. sursa 1	Adr. sursa 2	Divers	Cond	Adresa următoare
------------	--------------	--------------	--------------	--------	------	------------------

Figura 5.8. Exemplu de microinstrucțiune cu mai multe câmpuri de control și secvențiere. În afara operației comandate pentru ALU se specifică codul adresă pentru registrele sursă și destinație, eventuali biți pentru testarea de condiții (Cond.) și informații diverse (Divers) de control. Câmpul de secvențiere (adresa următoare) se dă de obicei ca adresă relativă față de adresa microinstrucțiunii curente sau la secvențe liniare indică doar necesitatea incrementării micro-contorului de program.

Cea mai simplă secvență de comandă a unei căi de date este o secvență liniară, în care starea următoare a acestui automat simplu ar putea generată de un numărator cu incrementare. În mod obișnuit adresa generată pe baza codului operației instrucțiunii este o adresă de start a unui microprogram din memoria de control. Microprogramul poate fi rulat secvențial, dar unitatea de control trebuie să aibă și capacitatea de a răspunde la semnale externe, sau condiții, prin efectuarea de salturi, deci de întrerupere a secvenței liniare de microinstrucțiuni. Salturile sunt efectuate pe baza testării semnalelor de la calea de date și pe baza informației de secvențiere. Circuitul de generare a adresei următoare din microprogram este numit microsecvențiator. Astfel că adresarea unei microinstrucțiuni se poate face:

- secvențial, similar cu adresarea la nivel macro a instrucțiunilor prin contorul de program (PC), micro-adresa fiind generată de un micro-contor de program (μ PC);
- salturi condiționate, prin testarea de către microinstrucțiuni a reacțiilor de la calea de date;
- salturi absolute prin specificatori de micro-adrese următoare(secvențiere), saltul fiind de obicei relativ la adresa microinstrucțiunii curente.

5.3.1. Metode de optimizare a UC microprogramate

În scopul creșterii vitezei de lucru a UC microprogramate și de reducere a dimensiunilor memoriei de control principalele metode folosite sunt:

- a. reducerea lățimii microinstrucțiunilor
- b. utilizarea structurilor de tip pipeline la nivelul microprogramului
- c. utilizarea mai multor niveluri ierarhice de memorii de control

a) Reducerea lățimii microinstrucțiunilor

Adesea, microinstrucțiunile sunt clasificate în microinstrucțiuni *orizontale* și *verticale*, în funcție de numărul de resurse hardware ale mașinii controlate simultan de o microinstrucțiune.

Microinstrucțiunile orizontale controlează foarte multe resurse hardware care funcționează în paralel. O singură microinstrucțiune orizontală ar putea să controleze funcționarea simultană și independentă a uneia sau mai multor unități ALU, accesul la memoria principală, generarea condiționată a adresei următoare, accesul la diverse registre de lucru, bistabili de stare etc. O asemenea microinstrucțiune are însă un număr mare de biți (tipic peste 64 de biți).

Microinstrucțiunile verticale controlează simultan un număr mic de resurse hardware și comandă operații simple, de tipul încărcare, adunare, memorare, salt etc. Acest tip de microinstrucțiune se aseamănă în principiu cu instrucțiunile în limbaj mașină, care conțin un cod al operației și unul sau mai mulți operanzi. De aceea microprogramarea verticală mai este numită microprogramare software, iar prin comparație microprogramarea cu instrucțiuni orizontale este numită microprogramare hardware.

Aceeași mașină poate fi controlată cu microprograme lungi și lente, scrise cu microinstrucțiuni verticale, sau cu microprograme scurte și rapide, scrise cu microinstrucțiuni orizontale.

Lungimea unei microinstrucțiuni depinde de gradul de codificare al informației conținute. La o microinstrucțiune fără codificare, fiecare bit controlează o singură resursă sau micro-operație. La o microinstrucțiune cu codificare (de exemplu în figura 5.9), câmpuri binare controlează resurse mutual exclusive. Prin codificare, un câmp al microinstrucțiunii va putea activa în același timp numai un singur semnal de control. De aceea două semnale de control sunt *compatibile* pentru codificare și pot fi incluse în același câmp de control numai dacă sunt mutual exclusive (nu este necesar să fie activate simultan de o microinstrucțiune). Pot exista însă mai mult decât două semnale compatibile pentru fiecare câmp ce folosește codarea controlului.

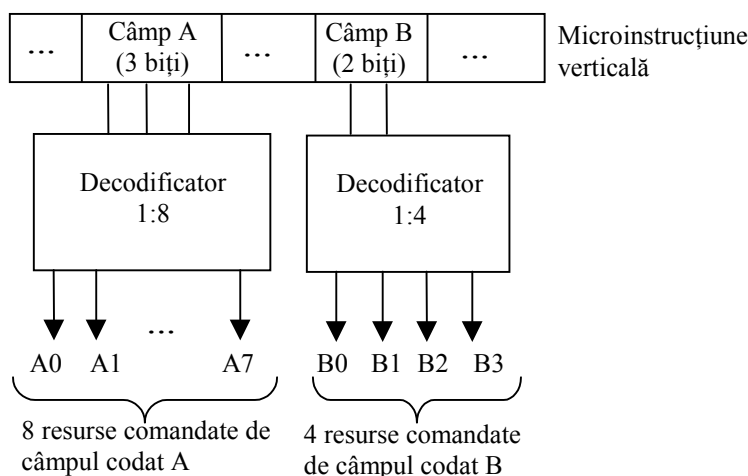


Figura 5.9. Exemplu de microinstrucțiune verticală ce utilizează codarea informației de control în două câmpuri binare. Câmpul A poate comanda simultan una singură din resursele A0 - A7 prin semnalul de la ieșirea decodificatorului.

Concluzionând putem spune că microinstrucțiunile orizontale au caracteristicile: format lung, capacitate de a exprima un grad mare de paralelism (multe comenzi simultane), codare redusă a informației de control. Microinstrucțiunile verticale au ca principale caracteristici; format scurt, capacitate mică de control al microoperațiilor paralele, codare puternică a informației de control.

b) Utilizarea structurilor pipeline, pentru operații în paralel

Deși execuția microinstrucțiunii se desfășoară conform unei secvențe ce cuprinde fazele de citire, decodificare și execuție, detaliile de implementare pot fi diferite de la o soluție de implementare la alta. Gradul de suprapunere al operațiilor de execuție pentru microinstrucțiunea curentă respectiv de acces (găsire și citire) a microinstrucțiunii următoare este dependent de structura hardware a UC.

Dacă metoda folosită pentru execuția microprogramelor este serială, citirea microinstrucțiunii următoare nu începe până când nu se termină execuția microinstrucțiunii curente. Durata unui ciclu de microinstrucțiune (CMI) va fi suma dintre timpul de acces, TA, la memoria de control și timpul de execuție, TE, al microoperațiilor ce controlează resursele hardware ale calculatorului:

$$\text{CMI} = \text{TA} + \text{TE} \quad (5.2)$$

În acest fel, dacă se consideră valori medii pentru TA și TE se poate calcula durata medie de execuție a unui microprogram cu M microinstrucțiuni ca $M \cdot (\text{TA} + \text{TE})$.

În relația de mai sus timpul de acces (TA) se referă la timpul necesar din momentul lansării micro-adresei până când microinstrucțiunea poate fi citită la ieșirea memoriei de control. Timpul de execuție (TE) este timpul minim cât semnalele de control trebuie să rămână active pentru ca operația de control să se desfășoare corect.

Avantajul modului serial de lucru al UC (figura 5.10a și b) este dat de simplitatea hardware a unității și ca urmare de costul redus. În plus UC microprogramată serială nu trebuie să controleze simultan execuția și citirea și nu pune probleme în cazul apariției microinstrucțiunilor de salt

condiționat.

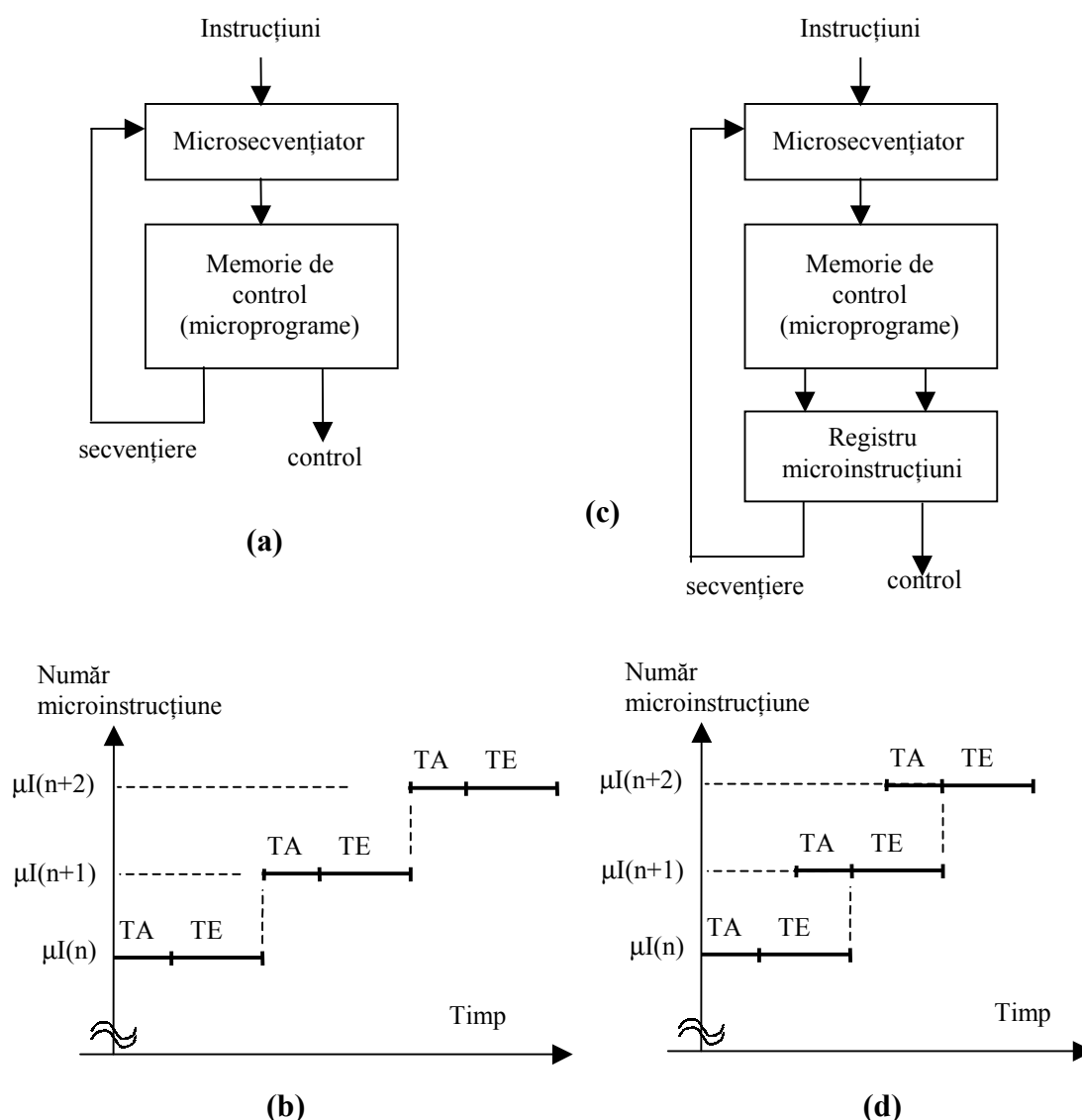


Figura 5.10. Structura bloc a unității de control și distribuirea în timp a execuției microinstrucțiunilor. (a) structura bloc cu funcționare serială, (b) sincronizarea în timp a execuției a trei microinstrucțiuni succesive, (c) structura bloc cu registru pipeline de microinstrucțiuni, (d) sincronizarea în timp a execuției a trei microinstrucțiuni succesive.

Prin adăugarea unui etaj suplimentar în UC (etaj constituit dintr-un registru de microinstrucțiuni) citirea microinstrucțiunii următoare este realizată în paralel cu execuția microinstrucțiunii curente. Această structură de tip pipeline (bandă de operații, sau conductă de operații) are avantajul unei viteze de lucru mai mari, pentru că execuția microinstrucțiunii curente începe imediat după terminarea execuției celei precedente. În această implementare valoarea CMI va fi dată de maximul dintre timpul de acces la memoria de control și timpul de execuție: $CMI = \max(TA, TE)$. Durata minimă a unui microprogram de M microinstrucțiuni este în acest caz $TA + M \cdot TE$. Ca dezavantaj, în afara de circuitele suplimentare, reprezentate în principal de registrul pentru memorarea microinstrucțiunii curente (registru pipeline), la UC microprogramată din figura 5.10 c și d se pune și problema salturilor condiționate. În acest caz este nevoie de un mecanism de anticipare a adresei următoarei microinstrucțiuni. Dacă anticiparea este corectă, în majoritatea

cazurilor, nu se va pierde din viteză. În cazul unei anticipări greșite se va pierde însă cel puțin un ciclu pentru citirea corectă a microinstrucțiunii ce urmează.

c) *Organizarea pe mai multe niveluri a memoriei de control*

Memoria de control poate fi organizată pe unul sau pe două niveluri. Memoria structurată pe două niveluri folosește noțiunile de microinstrucțiune și microprogram pentru nivelul superior și nanoinstrucțiune, respectiv nanoprogram pentru nivelul inferior. Așa cum o instrucțiune este executată prin rularea unui microprogram (la un singur nivel de memorie de control), fiecare microinstrucțiune este executată prin rularea unui nanoprogram asociat. Tehnica nanoprogramării este echivalentă conceptual cu microprogramarea, iar structurarea memoriei de control pe două niveluri (vezi figura 5.11) oferă o flexibilitate mai mare în proiectarea microinstrucțiunilor de la nivelul superior.

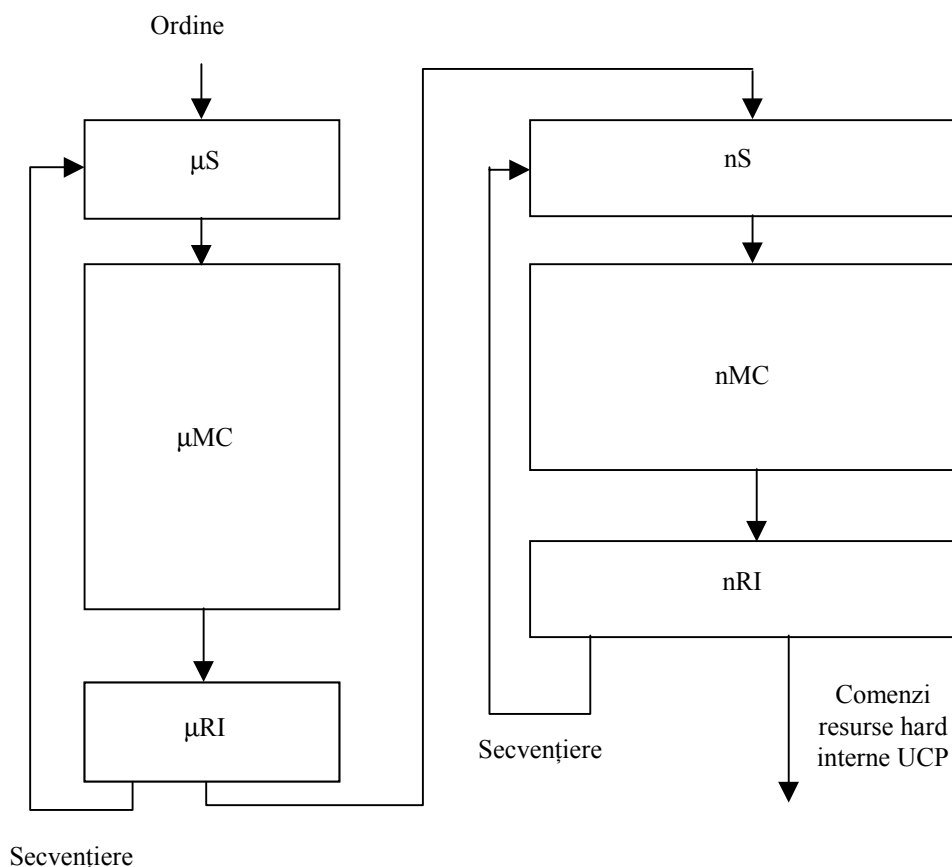


Figura 5.11. Organizarea memoriei de control pe două niveluri. Notății folosite: μS / nS = microsecvențiator / nanosecvențiator; μRI / nRI = micro / nano registru de instrucțiuni, μMC / nMC = memorie de microprogram / nanoprogram.

În calculatoarele microprogramate convenționale (de exemplu seria Intel 80x86), fiecare instrucțiune adusă din memoria principală (MP) este interpretată de un microprogram stocat într-o singură memorie de control. În unele mașini însă, microinstrucțiunile nu emit direct semnalele ce

controlează resursele hardware. Semnalele de ieșire sunt folosite pentru a accesa, printr-un nanosecvențiator o memorie suplimentară de control numită memorie de nanocontrol. Nanoinstrucțiunile conținute în această din urmă memorie controlează direct resursele hardware. Memoria de microcontrol (μ MC) conține microprograme cu *microinstrucțiuni verticale*, interpretate de *nanoinstrucțiuni orizontale* ce se găsesc în memoria de nanocontrol (nMC). Mai multe microinstrucțiuni (ce fac parte din microprograme diferite) pot fi interpretate de același nanoprogram. Rezultă astfel o dimensiune mai mică, pe ansamblu, a memoriei de control. Un alt avantaj este flexibilitatea mai mare a operației de proiectare a arhitecturii, ce rezultă din slăbirea legăturilor dintre instrucțiuni și hardware, prin două niveluri intermediare de control, în loc de unul. Dezavantajul principal al nanoprogramării este reducerea vitezei datorită accesului suplimentar la memoria nMC și a structurii ceva mai complexe a organizării unității de control.

Posibilitatea optimizării mărimii totale a memoriei de control prin nanoprogramare a fost folosită la seria de microprocesoare Motorola 680x0. De exemplu MC 68000 conține o memorie de microprogram de 640 x 10 biți respectiv o memorie de nanocontrol de 280 x 70 biți, adică în total circa 26 kbiți. O proiectare microprogramată clasică ar necesita peste 45 kbiți.

5.4. PARALELISM ÎN EXECUȚIA INSTRUCȚIUNILOR PRIN TEHNICI PIPELINE¹

Tehnica pipeline este o tehnică de introducere a paralelismului în execuția instrucțiunilor. Aceasta se face prin împărțirea ciclului de execuție al unei instrucțiuni în mai multe faze de prelucrare, iar execuția paralelă a mai multor instrucțiuni indică că fiecare dintre aceste instrucțiuni se află în altă fază de prelucrare. Tehnica de tip pipeline poate fi aplicată atât pentru execuția instrucțiunilor cât și pentru circuitele aritmetice de prelucrare. Aici ne vom referi doar la execuția paralelă a instrucțiunilor prin tehnica pipeline.

Vom folosi în continuare termenul de conductă, care este sugestiv și de asemenea scurt, pentru a exprima cuvântul din limba engleză “pipeline” (vezi și nota din josul paginii). O conductă este similară unei linii de asamblare / montaj din industrie. Fiecare pas din conductă completează o parte din operațiile necesare în execuția instrucțiunii. La fel ca într-o linie de asamblare pentru autovehicule, munca care trebuie făcută într-o instrucțiune este spartă în părți mai mici, fiecare din aceste părți necesitând o fracțiune din timpul necesar pentru a termina complet execuția instrucțiunii. Fiecare din acești pași este numit *un etaj* al conductei sau *segment* al conductei. Etajele sunt conectate unul după celălalt pentru a realiza o conductă (bandă) de instrucțiuni ce intră la un capăt și ies prelucrate la celălalt capăt. Pentru că toate segmentele conductei sunt legate între ele, toate segmentele trebuie să fie gata pentru a avansa (pentru a furniza informația către următorul etaj) în același timp.

Timpul necesar pentru deplasarea unei instrucțiuni cu un pas în interiorul conductei constituie un *ciclu mașină al conductei*. Lungimea acestui ciclu este determinată de timpul cerut pentru prelucrarea în cel mai lent segment din conductă, pentru că toate segmentele avansează în același moment. Un ciclu mașină al conductei este egal cu unul sau mai multe cicluri de ceas, dar obiectivul oricărei mașini ce implementează această tehnică este să se ajungă la o medie de o instrucțiune executată pe ciclu de ceas. Tehnica de tip pipeline conduce la o reducere a timpului de execuție mediu pentru o instrucțiune.

Cea mai simplă cale de a înțelege principiul conductei este să imaginăm fiecare etaj ca fiind format dintr-un registru urmat de un circuit combinațional de prelucrare. (vezi figura 5.12 cu reprezentarea bloc a unei conducte cu 4 etaje / faze).

¹ Pipeline = bandă de execuție/ conductă de execuție / linie de asamblare

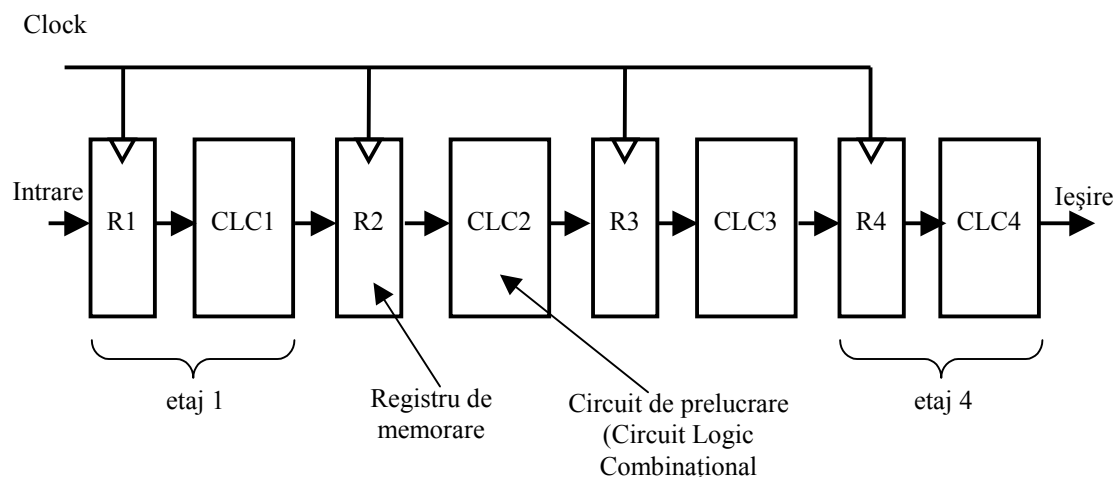


Figura 5.12. Structură de tip conductă (pipeline) cu patru etaje.

Registrele stochează datele de intrare pentru combinaționale pe durata necesară ca sub-operația corespunzătoare segmentului să fie terminată și rezultatele să apară corect la ieșirea circuitului combinațional. Aceste ieșiri ale etajului se aplică la intrările unui registru dintr-un alt etaj similar și așa mai departe. Dacă încărcarea registrelor cu datele de la etajul anterior se face sincron cu un impuls de ceas, perioada ceasului trebuie să fie suficient de mare astfel ca cel mai lent dintre etaje să termine prelucrarea.

Revenind la conducta pentru instrucțiuni, vom presupune că este formată din patru etaje, a căror notație și funcție este:

- Etaj 1 IF (Instruction Fetch) - aducere instrucțiune din memoria principală
- Etaj 2 ID (Instruction Decode) - decodificare instrucțiune
- Etaj 3 OF (Operand Fetch) - aducere operanzi
- Etaj 4 EX (Execution) - execuție operație.

Funcționarea unei conduite este explicată adeseori cu ajutorul unei diagrame timp-spațiu cum este cea din figura 5.13, corespunzătoare unei conduite cu 4 etaje cu notațiile propuse mai sus. Axa orizontală reprezintă timpul în multiplii de impulsuri de ceas iar axa verticală dă numărul segmentului.

		Impulsuri de ceas								
		1	2	3	4	5	6	7	8	9
Etaj:	IF	I1	I2	I3	I4	I5	I6			
	ID		I1	I2	I3	I4	I5	I6		
	OF			I1	I2	I3	I4	I5	I6	
	EX				I1	I2	I3	I4	I5	I6

Figura 5.13 Diagrama timp-spațiu pentru o conductă cu 4 etaje (IF, ID, OF, EX) corespunzător unui ciclu instrucțiune. S-a presupus introducerea în conducta de execuție a 6 instrucțiuni, notate de la I1 la I6.

Dacă se consideră o conductă cu k etaje și n instrucțiuni de executat, vom încerca să deducem creșterea în viteză pentru structura de tip pipeline. Dacă timpul de ciclu al conductei este notat cu T_{CLK} va fi nevoie de un timp egal cu $k \times T_{CLK}$ ca să umplem conducta (sau altfel spus ca prima instrucțiune să treacă prin toate etajele) și de timpul suplimentar de execuție a celorlalte $n-1$ instrucțiuni, adică $(n-1) \times T_{CLK}$. Timpul total pentru execuția celor n instrucțiuni într-o conductă cu k etaje va fi:

$$T_P = (k + n - 1) \times T_{CLK} \quad (5.3)$$

Considerăm de asemenea cazul unei unități de execuție a instrucțiunilor fără folosirea tehnicii pipeline. Fie un timp notat T_{mediu} necesar pentru execuția fiecărei instrucțiuni.

Pentru un program cu n instrucțiuni, fără pipeline (suprapunere) timpul mediu total de execuție va fi egal cu:

$$T = n \times T_{mediu} \quad (5.4)$$

Creșterea în viteză datorită folosirii conductei poate fi calculată ca raport între cele două valori de timp:

$$S = \frac{n \cdot T_{mediu}}{(k + n - 1) \cdot T_{CLK}} \quad (5.5)$$

Pe măsură ce numărul instrucțiunilor crește este valabilă relația $n \gg k - 1$, iar creșterea în viteză tinde către o valoare maximă egală cu:

$$S = \frac{T_{mediu}}{T_{CLK}} \quad (5.6)$$

Dacă presupunem că timpul mediu de execuție al unei instrucțiuni (T_{mediu}) este același cu timpul mediu de execuție al unei instrucțiuni în unitatea construită ca pipeline (adică $k \cdot T_{CLK}$), creșterea în viteză tinde către k , numărul de etaje al conductei. Aceasta este doar o limită teoretică, pentru că pe măsura creșterii numărului de etaje diferențele între T_{mediu} și $k \cdot T_{CLK}$ se măresc. Mai mult, cu cât crește numărul de etaje cu atât mai greu este să echilibreze timpii de execuție pe fiecare etaj. În plus apar probleme de care nu s-a ținut cont în relația calculului de creștere în viteză: salturile în program și dependența de date și resurse hardware inter-instrucțiuni. Aceste din urmă elemente creează ceea ce se numește hazardul structurii de tip conductă.

În general există trei probleme mari care pot face ca o conductă de instrucțiuni să nu funcționeze normal:

1. *Conflicte de resurse* produse de accesul la aceleași resurse din două etaje diferite ale conductei. De exemplu accesul simultan al două etaje la memoria principală, unul din accese pentru aducerea codului unei instrucțiuni, iar celălalt pentru accesarea operanzilor. Aceste conflicte se rezolvă prin mărirea numărului de unități funcționale (de exemplu ALU multiple) și prin utilizarea de spații separate de memorie pentru date și instrucțiuni. (hazard structural)
2. *Dependența datelor* se referă la instrucțiuni care folosesc ca operanzi rezultate ale instrucțiunilor anterioare din conductă, dar acestea încă nu au fost executate, iar rezultatul nu este disponibil. (hazard de date). Hazardul de date se înlătură fie prin metode hardware (bypass) fie prin reordonarea instrucțiunilor din program.

3. *Ramificările* produc probleme pentru că dacă se face saltul stiva trebuie golită și început procesul de execuție al instrucțiunilor de la adresa de salt (hazard de control). Problemele hazardului de control se rezolvă prin predicția salturilor, sau prin instrucțiuni de salt întârziat, la care se execută întotdeauna câteva dintre instrucțiunile ce urmează după instrucțiunea de salt condiționat, în așa fel conducta să rămână plină până la execuția instrucțiunii de salt și aflarea adresei următoare de instrucțiune.

5.5. ÎNTRERUPERI ȘI EXCEPȚII

În timpul rulării programelor pot apărea unele evenimente neobișnuite (de excepție). Aceste evenimente pot conduce la suspendarea temporară a programului aflat curent în execuție; în acest interval de *întrerupere* a rulării programului se va executa, de obicei, o rutină de tratare a evenimentelor ce au întrerupt programul. Toate aceste evenimente produse de condiții neobișnuite / neașteptate pentru programul curent în execuție sunt numite la modul general "*întreruperi*" pentru că ele produc întreruperea programului și devierea / saltul către o rutină specială de tratare a evenimentului "întreruptor". Putem clasifica aceste evenimente "întreruptoare" în:

1. *cereri de întrerupere*: evenimente generate din exteriorul UCP, *asincrone* cu programul rulat, care cer tratare. Aceste cereri de întrerupere pot proveni:
 - 1.a de la echipamente periferice care cer servicii de la UCP (cerere de transfer de date, sau informări cu privire la starea perifericului), sau
 - 1.b de la circuite specializate pentru supravegherea funcționării normale a componentelor hardware (eroare de paritate la citirea memoriei, eroare de paritate pe magistrală, căderea iminentă a tensiunii de alimentare etc.);
 Evenimentele ce produc acest tip de întrerupere a programelor sunt numite și *întreruperi hardware*.
2. *excepții*: evenimente neobișnuite *sincrone* cu programul (produse la rularea programului), care cer tratare. De obicei indică situații de excepție cum ar fi:
 - 2.a întâlnirea unor instrucțiuni de control ilegale în programul utilizator,
 - 2.b încercarea de violentare a sistemului de protecție a informațiilor din memorie,
 - 2.c condiții aritmetice speciale (depășiri, împărțire la zero),
 - 2.d accesul la segmente (sau pagini virtuale) de memorie nerezidente în memoria principală.
 Excepțiile produc *devieri*, ("traps") sau *întreruperi software*.

Efectul general al unei întreruperi, atât pentru întreruperea hardware cât și pentru întreruperea software este o deviere către o rutină de tratare a întreruperii (rutină de servire a întreruperii). Indiferent de sursa ce a generat întreruperea cele două tipuri sunt servite în mod asemănător, prin saltul la o rutină de tratare a evenimentului întreruptor sau de excepție. În literatura actuală, termenii de "întrerupere" și "cerere de întrerupere" sunt folosiți, în general, pentru evenimente ce produc cereri asincrone cu programul curent rulat. Excepțiile - sincrone cu programul rulat - produc devieri, care la rândul lor pot fi clasificate în mai multe tipuri.

De exemplu, după modul de manifestare și tratare a excepțiilor, Intel împarte aceste excepții în următoarele categorii:

- *devieri ('traps')*. O deviere poate fi recunoscută doar după terminarea ciclului instrucțiune curent (recunoscută doar înainte de începerea ciclului de execuție al unei instrucțiuni ce

urmează imediat după instrucțiunea în care s-a detectat excepția). După tratare *devierile* produc reluarea programului întrerupt începând cu instrucțiunea următoare.

- *erori ('faults')* . Sunt excepții ce sunt detectate fie înainte de începutul execuției instrucțiunii, fie în timpul execuției acestora. Dacă erorile se detectează în timpul execuției unei instrucțiuni, după tratarea erorii reluarea programului se va face începând cu instrucțiunea întreruptă.
- *esecuri - terminari anormale ('abort')*. Produc abandonarea procesului, iar rutina de tratare afișează mesaje “stresante” pentru utilizator. Eșecurile sunt utilizate pentru a trata erori severe cum sunt valori ilegale si / sau inconsecvente în tabelele sistemului sau erori de hardware.

Din cele expuse de mai sus se observă că indiferent dacă evenimentul ce produce întreruperea provine de la circuitele calculatorului (hardware), sau de la rularea instrucțiunilor programului (software), tratarea evenimentului se face în mod asemănător:

- când UCP recunoaște o eroare sau o condiție neobișnuită, el oprește obligatoriu rularea programului aflat în execuție, și produce saltul la o rutină care va trata corespunzător condiția apărută.
- când UCP recepționează o cerere de întrerupere, el oprește rularea programului aflat în execuție (presupunem că sunt permise și întreruperile mascabile) și produce saltul la o rutină de tratare corespunzătoare a cererii de întrerupere

Conform clasificării de mai sus, cererile de întrerupere (asincrone) sunt întreruperile generate hardware (prin semnal electric). O întrerupere generată hardware se produce ca răspuns la activarea unui pin de intrare în UCP. După modul de servire al cererilor de întrerupere, exista două tipuri de intrări în microprocesor (vezi și figura 5.14):

- ⇒ cerere de întrerupere este *mascabilă* (notată INTR în figura 5.14), la care se poate bloca (“masca”) acțiunea de recunoaștere a acesteia de către UCP. Ca urmare a mascării programul rulat de UCP nu va fi întrerupt, iar semnalul de cerere de întrerupere va fi neglijat;
- ⇒ cerere de întrerupere *nemascabilă* (notată NMI în figura 5.14). Acest tip de întrerupere este obligatoriu, va fi recunoscut necondiționat de UCP și ca urmare va produce deviere către o rutină de tratare.

Figura 5.14. prezintă schematic logica de intrare în microprocesor, pentru întreruperile mascabile și nemascabile.

La intrarea de întreruperi mascabile se conectează de obicei linii de semnal de la dispozitivele de I/O. Electric acest lucru este posibil pentru că la majoritatea microprocesoarelor intrarea este activă pe nivel logic JOS fiind destinată ieșirilor cu colector în gol de la dispozitive. Intrarea INTR permite dispozitivelor externe să întrerupă execuția unui program. Pentru a asigura recunoașterea cererii, intrarea INTR trebuie să fie menținută la nivel activ până când procesorul confirmă întreruperea prin realizarea unei secvențe de confirmare a acceptării întreruperii. Intrarea INTR este eșantionată la sfârșitul execuției fiecărei instrucțiuni. În plus, întreruperile mascabile trebuie să fie validate prin software (instrucțiuni ce produc setarea indicatorului de control a întreruperilor mascabile - reprezentat ca un bistabil în figură) pentru a putea fi recunoscute. Semnalul INTR poate fi furnizat direct de un dispozitiv periferic, sau este furnizat de circuite specializate pentru controlul și arbitrarea mai multor semnale externe de cerere de întrerupere (controller programabil de întreruperi).

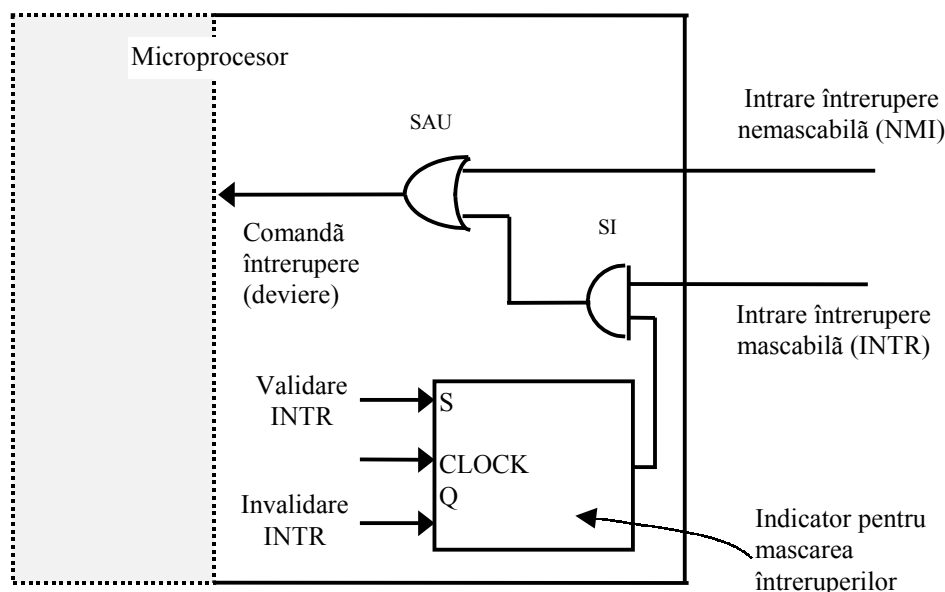


Figura 5.14. Reprezentare a modului de recunoaștere pentru cererile de întrerupere mascabile și nemascabile

Intrarea NMI (pentru cerere de întrerupere ne-mascabilă) este folosită de obicei de circuitele externe UCP care supraveghează buna funcționare a sistemului hardware. Ele pot semnaliza de exemplu erori de paritate la transferul datelor, erori de memorie, creșterea peste limite a temperaturii interne, căderea iminentă a tensiunii de alimentare etc. Activarea intrării NMI face ca procesorul să întrerupă programul la sfârșitul ciclului instrucțiune curent și să execute automat o rutină de servire care corespunde, de obicei, unei locații fixe din memoria principală (întrerupere ne-vectorizată). Procesorul nu va servi cereri succesive NMI până când tratarea cererii curente nu este terminată.

Dacă mecanismul hardware de recunoaștere a unei întreruperi nu este complicat, în cazul excepțiilor produse de rularea instrucțiunilor lucrurile sunt un pic mai complexe. Cauzele excepțiilor sunt diverse și ele se manifestă în diferite moduri. De exemplu, cauzele aritmetice, cum ar fi depășirea domeniului de reprezentare de rezultatul unei adunări, pot fi tratate și rezolvate relativ ușor. Dar dacă cuvântul binar adus ca și cod al instrucțiunii nu reprezintă nici un cod recunoscut de procesor evenimentul va produce sigur abandonarea programului. Codul nerecunoscut este tratat ca "instrucțiune ilegală" și el poate apărea fie din cauza erorilor umane fie datorită erorilor hardware ale mașinii. Tratarea acestei excepții trebuie făcută nu numai pentru a informa utilizatorul despre evenimentul produs, dar și pentru a închide execuția acelui program. Cea mai defavorabilă soluție este bineînțeles blocarea calculatorului. Pentru cazul depășirilor, unele calculatoare au instrucțiuni speciale de test pentru overflow (ex. BOV Branch if OVerflow). Este însă complet neconvenabil ca fiecare instrucțiune aritmetică să fie urmată de un test. În cazul instrucțiunilor ilegale nu este posibil să se testeze condiția, pentru că trebuie luate imediat măsuri. Calculatoarele timpurii, care nu aveau un sub-sistem de întreruperi, s-ar fi oprit în cazul unor asemenea condiții, dar asta n-ar fi făcut decât să împingă problema mai jos cu un nivel, către operatorul de serviciu, care trebuia să rezolve problema. La mașinile actuale, de mare viteză, este dorit să se acționeze imediat pentru a se elimina timpul pierdut de calculator. Se presupune că de obicei condiția ce a produs eroarea și deci întreruperea este înțeleasă, tratată și apoi se va continua cu sarcina întreruptă dacă este posibil. Dacă eroarea este fatală sarcina curentă este abandonată și se va trece la execuția unei alte sarcini de lucru. Descriem în continuare câteva din condițiile care pot

deveni surse de deviere a programului rulat.

Condiții aritmetice. Aceste excepții sunt date de condiții aritmetice neobișnuite cum sunt depășirile (overflow și underflow), încercare de împărțire la zero etc. Deși utilizatorul poate testa aceste condiții cu instrucțiuni de test, este preferabil să existe o monitorizare continuă, hardware - prin circuite specializate, a acestor condiții. Acest procedeu este mai eficient din punctul de vedere al timpului de execuție pentru că nu trebuie executate instrucțiuni suplimentare decât în cazul când se produc condițiile testate. De multe ori se permite utilizatorului (uneori în faza de compilare) să specifice ce trebuie făcut, separat, pentru fiecare din tipurile de evenimente provenite din condiții aritmetice. Dacă nu se indică nici o specificație, acțiunea implicită este de obicei abandonarea sarcinii (abort).

Instrucțiuni ilegale. O deviere datorată unei instrucțiuni ilegale (instrucțiune de control nepermisă utilizatorului, sau cod ne-recunoscut) este de obicei fatală, adică termină execuția programului utilizator pentru că se presupune că există o eroare în program, sau o eroare severă de hardware. La abandonare utilizatorul este informat despre eroare și despre starea programului utilizator (conținutul registrelor și a memoriei).

Protecție. Cele mai multe sisteme de calcul au în memorie la un moment dat mai mult decât un program pentru ca există mai mult decât un proces ce se execută. (de exemplu mai multe programe lansate în execuție pe un calculator uni-procesor utilizat prin partajarea timpului²). De aceea cele mai multe sisteme prevăd un anumit sistem de protecție. Protecția memoriei se referă la capacitatea calculatorului de a marca acele zone de memorie care sunt disponibile unui program particular și să detecteze referințele interzise la acele zone. O zonă de memorie nedisponibilă pentru programul aflat în rulare este numită *regiune protejată* a memoriei. O încercare a programului utilizator de a scrie într-o regiune de memorie protejată este în mod clar o eroare de programare (intenționată sau nu) și produce o deviere (trap). În mod normal acest eveniment de excepție este fatal, la fel ca și devierile pentru instrucțiuni ilegale.

Erori ale mașinii. În mod ideal, calculatorul nu face niciodată erori, dar în practică, pot fi acceptate două feluri de erori hardware. O eroare catastrofală se produce când eroarea de hardware este atât de serioasă încât oprește funcționarea unui subsistem major. Asemenea erori sunt evidente imediat și sunt relativ ușor de rezolvat pentru tehnicianul de service. Totuși, cea mai frecventă formă de erori se produce ocazional, iar erorile pot să nu fie repetitive, chiar în aceleași condiții. Aceste erori nu se produc foarte des ci probabil doar la intervale mari, de ore sau zile. De aceea sunt dificil de localizat, deci dificil de depanat. Totuși unele erori sunt detectate automat de hardware; iar detecția unei erori produce o deviere (trap) și astfel sistemul de operare poate decide ce trebuie făcut. Rutina de monitorizare a sistemului de operare poate repeta acțiunea pentru a vedea dacă se produce fără eroare a doua oară. Dacă repetarea se face cu succes, controlul poate fi returnat programului utilizator și nu trebuie să ne mai facem probleme cu privire la acea eroare. Dacă eroarea se produce repetat, programul utilizator va fi abandonat (aborted) și se va afișa un mesaj informativ adecvat pentru utilizator. De obicei dispozitivele de memorie au unele echipamente de detecție a erorilor, tipic pentru generarea și detecția bitului de paritate. Paritatea permite doar detectarea erorilor izolate, dar erorile multiple în același grup de biți pot să treacă nedetectate.

Unitatea centrală de procesare răspunde la o cerere de întrerupere (hardware sau software) și tratează evenimentul ce a produs întreruperea în pași succesivi care pot fi rezumați astfel:

1. UCP identifică sursa ce a produs întreruperea. Această identificare se poate face fie printr-o informație furnizată de evenimentul întreruptor (informație numită de obicei "vector de întrerupere") fie prin testarea conținutului unor registre interne pre-definite sau a unor zone pre-definite din memoria principală. Uneori, în cazul cererilor de întrerupere hardware există intrări separate de întrerupere la care se conectează dispozitive de I/O, identificarea fiind făcută automat prin identificarea intrării de cerere

² time sharing = partajarea timpului procesor pentru execuția mai multor programe. Fiecărui program i se alocă un interval de timp conform unui mecanism de planificare implementat de sistemul de operare.

- de întrerupere activată.
2. Pe baza informației de identificare UCP calculează adresa unde se găsește rutina de tratare a întreruperii.
3. Se salvează informația de stare a programului ce va fi întrerupt (în primul rând conținutul contorului de program) într-o memorie de tip stivă. Se face saltul la rutina de tratare prin încărcarea PC cu adresa de început a rutinei de tratare.
4. Se execută rutina de tratare a întreruperii.
5. Se face revenire la programul întrerupt prin restaurarea informației de stare.

De observat că după ce rutina de tratare a fost executată, ea va face revenire la programul întrerupt (printr-o instrucțiune specială de tip "return from interrupt³") și se va continua cu execuția acestuia. Este important ca rutina de tratare să salveze conținutul registrelor interne pe care urmează să le modifice, ca să poată reface conținutul acestora înainte de revenirea la programul întrerupt. Dacă rutina de tratare nu este scrisă în acest mod este foarte posibil ca programul întrerupt să nu mai funcționeze corect. Pentru a se realiza operația de revenire la programul întrerupt este necesar să se știe locația ultimei instrucțiuni executate din programul întrerupt. De aceea în momentul saltului la o rutină de tratare, sistemul de întreruperi al UCP salvează automat adresa următoarei instrucțiuni (adresa de întoarcere). Salvarea și restaurarea contorului de program (PC) cu această adresă sunt automate și nu cad în sarcina programului întrerupt. Vom numi "vector de stare" informația salvată în momentul devierii; ea cuprinde de obicei nu numai adresa locației următoarei instrucțiuni, dar și conținutul registrelor de stare și control ai UCP. Informația vectorului de stare, împreună cu conținutul spațiului de memorie alocat programului utilizator, furnizează o descriere completă a stării procesului la momentul devierii, numita context de lucru al procesului. Procesul poate fi repornit prin restaurarea acestei informații în registrele potrivite și în locațiile de memorie.

Cu privire la identificarea evenimentului ce a produs întreruperea, unele procesoare produc saltul către o adresă diferită pentru fiecare tip diferit de condiție / întrerupere (*întreruperi vectorizate*). Altele produc saltul la aceeași adresă și stochează informația pentru identificarea și tratarea întreruperii la anumite adrese prestabilite din registrele procesorului sau din memorie (*întreruperi nevectorizate*). În oricare din cazuri, efectul este același, prin saltul la un program diferit (program care deviază cursul normal al programului întrerupt și care este numit "rutină de tratare"), cunoscându-se care este condiția care a produs aceasta deviere. Existența mecanismului de deviere, implementat în așa numitul "subsistem de întreruperi" prezent în toate tipurile de procesoare, face sarcinile de programare ale utilizatorului mai simple.

În cazul întreruperilor vectorizate identificarea se face pe baza unui vector de întrerupere. Cu ajutorul acestuia UCP calculează o adresă pointer într-un tabel al vectorilor de întrerupere unde UCP găsește adresa de început a rutinei de tratare a întreruperii (RTI) respective. Modul de calcul al adresei pointer și modul de construcție al tabelii vectorilor de întrerupere este diferit de la microprocesor la microprocesor deși principiul este asemănător. Ca exemplificare al modului de determinare al adresei RTI se propune exemplul din figura 5.15. Se presupune pentru simplificare că la acest sistem pot exista doar patru dispozitive ce pot lansa cerere de întrerupere, deci vectorul de întrerupere (notat VI) poate fi codificat cu doi biți. Memoria este adresabilă pe octet, iar adresele au lărgimea de 16 biți. La recunoașterea unei cereri de întrerupere UCP citește VI de 2 biți și îl salvează în PC pe pozițiile binare cu ponderile 2^3 și 2^2 iar în același timp ceilalți 14 biți ai PC sunt poziționați la zero.

³ return from interrupt = întoarcere din rutina de tratare a întreruperii

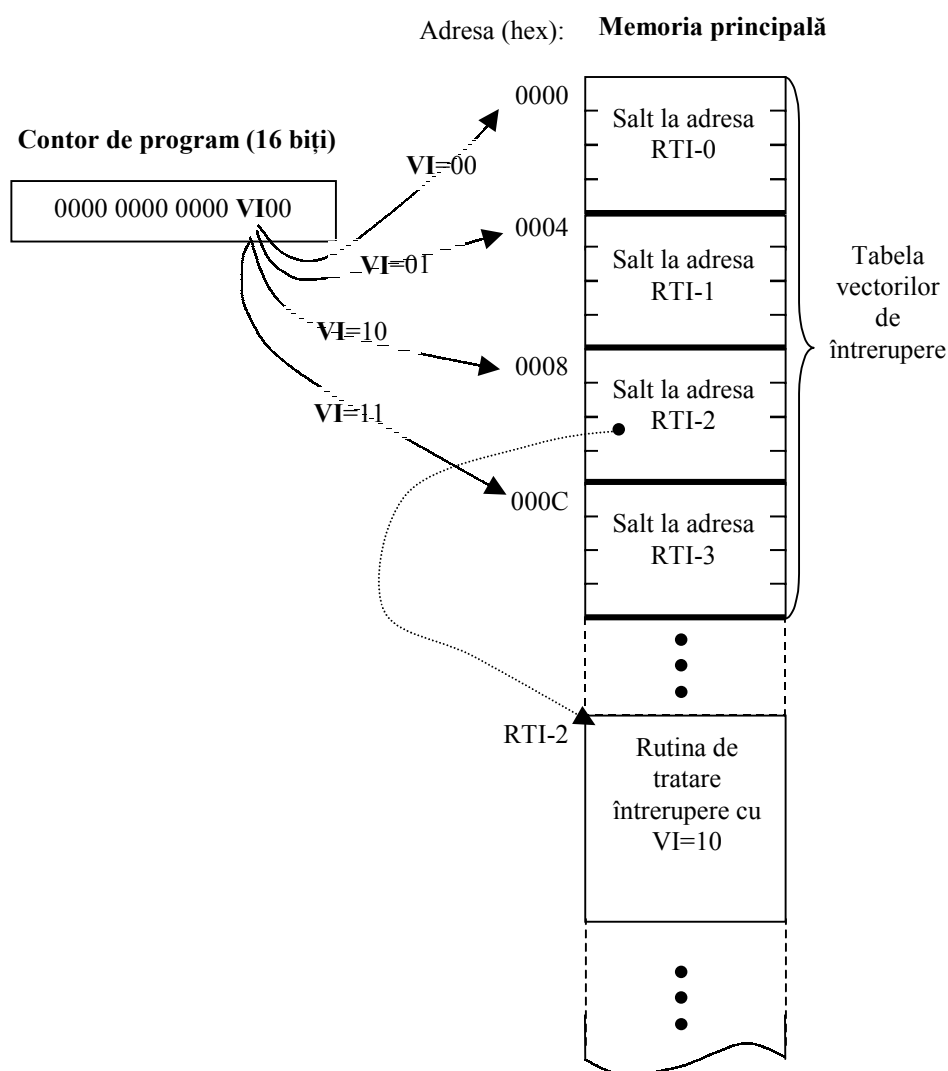


Figura 5.15. Exemplificare privind calculul adresei rutinei de tratare a întreruperilor (RTI) pentru un sistem cu doar patru cereri de întrerupere posibile. Cu VI s-a notat vectorul de întrerupere furnizat de dispozitivul întrerupător.

În acest fel fiecare intrare în tabela vectorilor de întrerupere din memoria principală corespunde la 4 octeți unde se poate înscrie o instrucțiune de salt către RTI specifică. În figură s-a indicat prin linie punctată saltul la rutina de tratare a cererii de întrerupere cu vector de întrerupere egal cu “10”. La unele microprocesoare între adresele pointer de intrare în tabela vectorilor de întrerupere poate exista o distanță de 8 adrese în așa fel încât o eventuală RTI foarte scurtă să poată fi introdusă aici. Mai mult tabela poate fi stocată oriunde în memoria principală și nu doar începând cu adresa zero, ca în exemplificarea din figură.

De observat că mecanismul de salt la RTI face o adresare indirectă prin locație de memorie. Avantajul determinării adresei rutinei de tratare, prin acest mod indirect (relativ lent) este dat de faptul că rutinele de tratare pot fi stocate la orice adresă în memoria principală. De observat că o soluție “rigidă” care ar stoca rutinele de tratare la adrese pre-fixate din memoria principală, pe lângă faptul că ar impune niște restricții în alocarea memoriei, ar fi foarte puțin flexibilă, fiind aproape imposibil de rescris o anumită rutină de tratare, dacă aceasta are lungime mai mare decât spațiul alocat inițial.

Exemplul 1 pentru întreruperi vectorizate

În cazul microprocesorului Z80 (mod 2 de lucru al sistemului de întreruperi), dacă procesorul recunoaște cererea de întrerupere, lansează pe magistrala de control semnale de acceptare a cererii de întrerupere (M1+IORQ în figura 5.16).

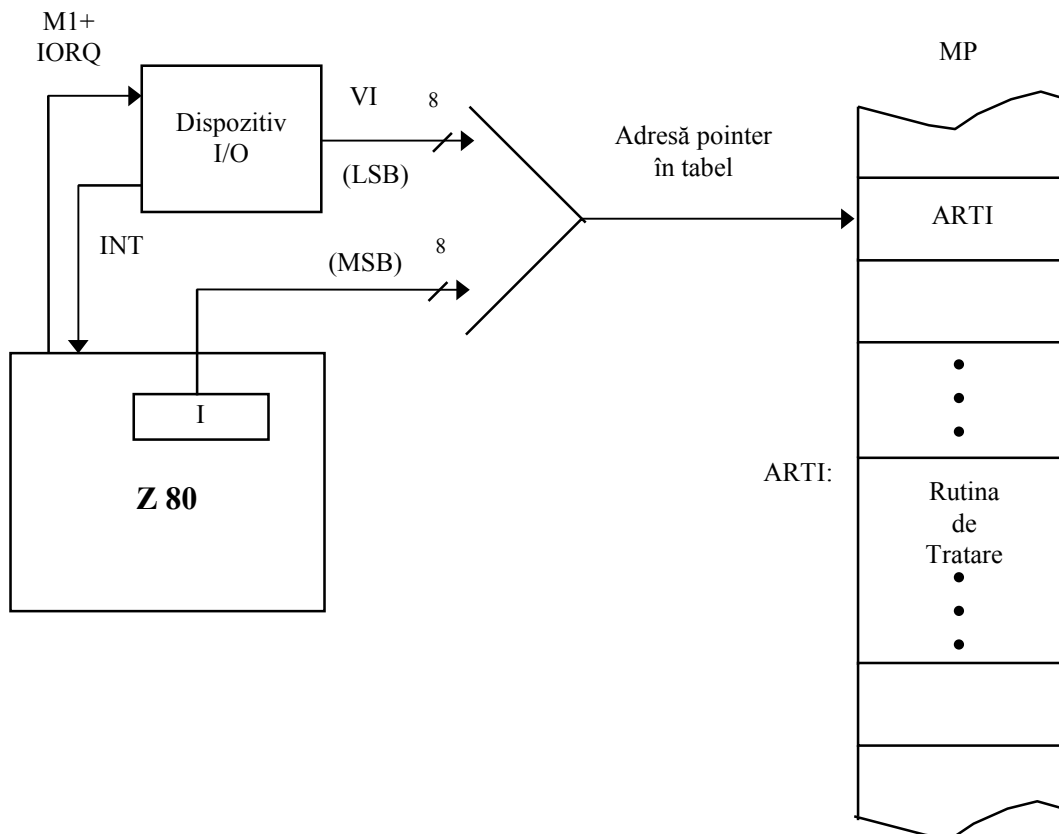


Figura 5.16. Exemplu simplificat pentru salt la rutina de tratare a unei întreruperii, în cazul procesorului Z80, în modul 2 de întreruperi. *I* = registru de întreruperi (păstrează octetul mai semnificativ - MSB - al adreselor din tabelul vectorilor de întrerupere); *VI*=vector de întrerupere, furnizat de dispozitivul de intrare/ieșire (I/O) căruia i s-a acceptat cererea de întrerupere; *ARTI*=adresa rutinei de tratare a întreruperii (este o adresă pe 16 biți, iar memoria Z80 e organizată pe octet; de aceea pentru a citi *ARTI* se citesc doi octeți începând de la adresa pointer construită).

Ca urmare a acestei acceptări, dispozitivul întreruptor va pune pe magistrala de date o informație de identificare de 8 biți pe care o vom numi vector de întrerupere (*VI*). Vectorul de întrerupere formează octetul mai puțin semnificativ al unei adrese pointer de 16 biți. Octetul mai semnificativ al adresei pointer se citește dintr-un registru special utilizat pentru întreruperi (*I*), intern microprocesorului. Adresa pointer este adresa din memoria principală (din tabelul vectorilor de întrerupere) unde se găsește adresa de salt către rutina de tratare a întreruperii. Pot exista maximum 255 de intrări în tabela vectorilor de întrerupere.

Exemplul 2 pentru întreruperi vectorizate

La procesorul I8086 (și seria I80x86 în mod real) toate adresele rutinelor de tratare a întreruperilor se păstrează într-o tabelă a vectorilor de întrerupere (*TVI*) care în modul ne-protejat de lucru se găsește începând cu adresa 0 din memoria principală. La procesoarele Intel memoria este segmentată, așa încât adresa de 20 de biți se

formează din două componente: adresa de bază a segmentului de cod (conținută într-un registru segment numit CS) și adresa în cadrul segmentului (conținută pentru programe în registrul IP). Fiecare dispozitiv întreruptor sau eveniment de excepție, va furniza un vector de întrerupere pe 8 biți (Intel numește "tip" acest vector) cu ajutorul căruia se calculează adresa pointer (P) de intrare în tabela TVI. În tabelă se păstrează adresele rutinelor de tratare în felul următor: începând cu adresa cea mai mică dintr-un grup de 4 octeți se găsesc valorile ce se vor înscrie în registrele de 16 biți IP (mai întâi octetul LSB și apoi MSB) și respectiv CS (LSB, iar la adresa cea mai mare MSB).

Relațiile de calcul de adresare în TVI, respectiv transferurile ce se efectuează sunt:

$P = 4 * (VI)$ - pointer care va conține adresa de intrare în TVI

$IP \leftarrow TVI(P+1, P)$

$CS \leftarrow TVI(P+3, P+2)$

În acest fel întreruperea 0 începe de la adresa 0, întreruperea 1 de la adresa 4, iar întreruperea 255 începând de la adresa 1020 (în zecimal).

Pentru toate evenimentele care pot produce întrerupere, este deosebit de importantă viteza cu care sistemul răspunde la cerere. *Timpul* care se scurge înainte ca o întrerupere să fie tratată depinde de mai mulți factori. Sursa de întrerupere trebuie să ia în considerare aceasta întârziere. Toți factorii următori pot afecta timpul scurs până la recunoașterea și tratarea unei întreruperi:

- dacă întreruperile mascabile sunt invalidate, o cerere INTR nu va fi recunoscută până când întreruperile nu sunt re-validate (prin indicatorul de întreruperi).
- dacă se servește o întrerupere nemascabilă, o nouă cerere de întrerupere nemascabilă nu va fi recunoscută până când procesorul nu execută instrucțiunea de revenire din rutina de tratare a întreruperii (return from interrupt).
- salvarea registrelor de indicatori și a altor registre consumă timp.
- de obicei instrucțiunile sunt neîntreruptibile. O cerere de întrerupere poate fi recunoscută doar la sfârșitul ciclului instrucțiune curent. Există și excepții de la această regulă, care se referă la întreruperi software de tip "faults" (erori) și instrucțiuni cu prefix de repetare a execuției. Acestea din urmă pot fi de obicei întrerupte după fiecare execuție.
- dacă mai mult de o întrerupere sau excepție așteaptă terminarea unei instrucțiuni, procesorul le servește pe rând, în ordinea priorității. În general, cea mai mare prioritate din listă o au excepțiile, sincrone cu programul (nu se încadrează aici excepțiile (trap) folosite pentru a depana un program). Urmează apoi cererile de întrerupere nemascabile. Cea mai mică prioritate o au întreruperile mascabile.

5.6. ROLUL MEMORIEI STIVĂ

Memoria stivă ("stack memory") este o memorie specială ce lucrează pe principiul LIFO (Last In First Out = *ultimul intrat - primul ieșit*). Așa cum s-a văzut anterior, subsistemul de întreruperi, este responsabil pentru salvarea și restaurarea automată din stivă a informațiilor vectorului de stare pentru programul întrerupt.

Memoria stivă este utilizată ca memorie de stocare temporară nu numai de sistemul de întreruperi, ci și de către tehnicile de apelare a procedurilor (subrutinelor), sau poate fi folosită de programe pentru stocarea temporară și regăsirea rapidă a datelor. De obicei prin stivă se transmit și alte informații, cum ar fi transmiterea parametrilor către proceduri.

O primă metodă de implementare a memoriei stivă (avantajoasă din punctul de vedere al

vitezei, dar problematică din punctul de vedere al capacității de memorare), constă în folosirea unor registre interne ale UCP, (registre cu posibilitatea de deplasare stânga / dreapta a informației). Acest tip de memorie stivă este numită *stiva "construită" hardware*. A doua metodă utilizată la majoritatea microprocesoarelor este implementarea memoriei stivă în cadrul memoriei principale, într-o zonă a cărei dimensiune poate fi satisfăcătoare pentru orice aplicație (*stiva software*).

Numele de "stivă" sugerează modul de funcționare al acestei zone de memorare. Astfel fiecare cuvânt nou introdus în stivă se așează peste anterioarele cuvinte, iar atunci când se dorește extragerea articolelor din stivă se începe de la "*vârful stivei*" (ultimul articol introdus).

Stiva hardware, reprezentată schematic în figura 5.17, este implementată de obicei cu ajutorul unui set de registre de deplasare (R1 - RN), interne UCP. Vârful stivei (VS) se va găsi întotdeauna în celulele registrelor de pe primul rang binar (c1i, unde $i = \overline{1, N}$), celule cuplate la intrările seriale (I1 - IN). Pentru scriere în stivă (PUSH), se face deplasare "sus-jos" a conținutului registrelor (informația din vârful stivei trece pe următoarea poziție binară din registre (c2i), iar în vârful stivei apare ultimul articol înscris. La extragere din stivă (POP) informația din vârful stivei este citită la ieșirile Y1-YN și se face deplasare în sens "jos-sus" a informației, pentru fiecare din registrele din figură. Capacitatea de memorare a stivei este limitată la un număr de cuvinte egal cu numărul de celule inclus în fiecare registru component (M în figura 5.17). Numărul de registre folosite determină dimensiunea în biți a fiecărui cuvânt. De obicei stiva hardware are o adâncime maximă de 4 cuvinte, iar la microprocesoarele unde este implementată oferă o viteză mare de salvare / restaurare a informațiilor stocate la apelul procedurilor, sau la saltul către rutine de tratare a întreruperilor.

Ultimul rând de celule ce pot stoca informație în stivă formează Baza Stivei (BS). Dacă în stiva hardware se introduc mai multe cuvinte decât capacitatea acesteia informația cea mai veche introdusă în stivă se pierde.

Cele mai multe dintre microprocesoarele ce au implementat mecanismul pentru stivă hardware pot administra și o stivă software.

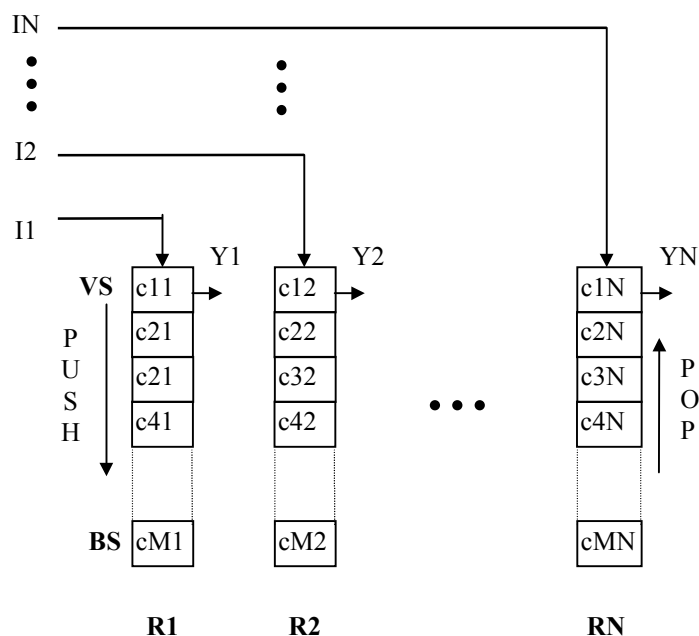


Figura 5.17. Implementarea hardware a stivei, cu ajutorul unui set de registre cu deplasare serială bidirecțională.

În cazul implementării stivei în memoria principală, informația nu mai este deplasată sus sau jos, iar gestiunea informațiilor din stivă este realizată cu ajutorul unui registru ce memorează

întotdeauna informația de adresare a vârfului stivei. Acest registru este numit *registru Indicator de Stivă (SP - Stack Pointer)*. Stiva construită software, în memoria principală, nu are dimensiuni fixe, acestea putând fi stabilite de programator. Chiar dacă dimensiunea mare este un avantaj, faptul că scrierea sau citirea din stivă înseamnă acces la memoria principală, externă UCP, conduce la o viteză de lucru mai mică decât la stiva hardware.

La înscrierea unui articol în stivă se spune că stiva crește. Creșterea stivei înseamnă și reactualizarea conținutului registrului SP care va conține întotdeauna adresa vârfului stivei, deci a ultimului cuvânt introdus în stivă, sau adresa primului articol ce a rămas în vârful stivei după o extragere din stivă.

În lumea microprocesoarelor există două convenții pentru creșterea stivei: creștere către adrese mici (convenția Intel) și creștere către adrese mari (convenția Motorola). Aici vom considera creșterea stivei doar către adrese mici, ceea ce înseamnă că ori de câte ori se introduce un nou articol în stivă acesta va avea adresă mai mică decât anteriorul cuvânt înscris și că la scriere registrul SP se decrementează. Dacă memoria este organizată pe octet (fiecare octet are o adresă specifică), la scriere, micșorarea valorii de adresă conținută în SP constă în scăderea din valoarea inițială a lui SP a unui număr egal cu numărul de octeți înșcriși în stivă. În cazul extragerii cuvintelor din stivă operațiile sunt inverse, conținutul registrului SP fiind incrementat

Valoarea inițială a conținutului lui SP se numește **adresa de bază a stivei** (baza stivei) și aceasta se fixează la lansarea programului în execuție conform directivelor introduse de utilizator în programul sursă.

Transferul între UCP și stiva construită în memoria principală se poate face în două moduri:

- **automat** (de către unitatea de control) la întreruperea unui program prin încărcarea în VS a adresei de revenire și a altor informații de stare. Întreruperea poate să constea într-un salt la o rutină de tratare de întrerupere, sau într-un salt la o procedură.
- **prin instrucțiuni speciale** pentru transfer cu stiva (numite PUSH și POP ca în figura 5.17, sau MOV - mută în stivă). La fiecare încărcare sau extragere din stivă se actualizează SP.

După cum s-a menționat la începutul paragrafului memoria stivă este folosită pentru salvare și restaurare automată nu numai de către sistemul de întreruperi dar și de apelurile de proceduri. O procedură este o secvență de instrucțiuni apelată într-un punct al alteia, executată, după care are loc revenirea în secvența apelantă, la adresa imediat următoare celei în care s-a făcut apelarea. Rutinele de tratare a întreruperilor produc o secvență similară de acțiuni, dar apelarea nu se face prin instrucțiuni de tip CALL, ci ca urmare a recunoașterii unui eveniment extern, numit cerere de întrerupere. La unele microprocesoare există în plus instrucțiuni numite "întreruperi software", care de asemenea folosesc stiva și care se apelează prin furnizarea codului instrucțiunii și a unei valori ("tip") de identificare. De exemplu la Intel instrucțiunea se scrie "*INT tip*"

Atunci când se face apel la o procedură, unitatea de control trebuie să facă salvarea valorii curente (adresa de revenire) conținută în registrului contor de program. Pentru prezentarea lucrului cu memoria stivă de mai jos este deocamdată neimportant dacă informația salvată provine dintr-un registru numit contor de program (PC), sau dacă în afară de registrul contor (IP - Instruction Pointer la Intel), în stivă se mai salvează și conținutul unui registru de adresă de segment, cum se întâmplă la Intel 8086 pentru proceduri far. După salvare urmează încărcarea PC cu adresa la care se face saltul (adresa unde începe procedura). Procedura se termină cu o instrucțiune de salt de tip RETURN prin care se produce încărcarea automată a PC cu adresa de revenire la programul întrerupt. Alte salvări /restaurări ale variabilelor contextului întrerupt se pot efectua în cadrul procedurii, prin intermediul unor instrucțiuni de înscriere (push) / extragere (pop) din stivă.

Pentru a se urmări conținutul memoriei stivă la apelul procedurilor în figura 5.18a se prezintă un exemplu de apel de proceduri, P1 din programul principal PP și P2 din procedura P1. În cadrul fiecăreia dintre rutine s-au notat cu IN instrucțiuni oarecare ale programelor (cu excepția instrucțiunilor de tip call, ret, int, push, pop etc., care afectează și conținutul stivei); cu litere mici și două puncte s-au notat adresele, <t> reprezintă momente de timp succesive (ordinea corespunde cu indicele folosit).

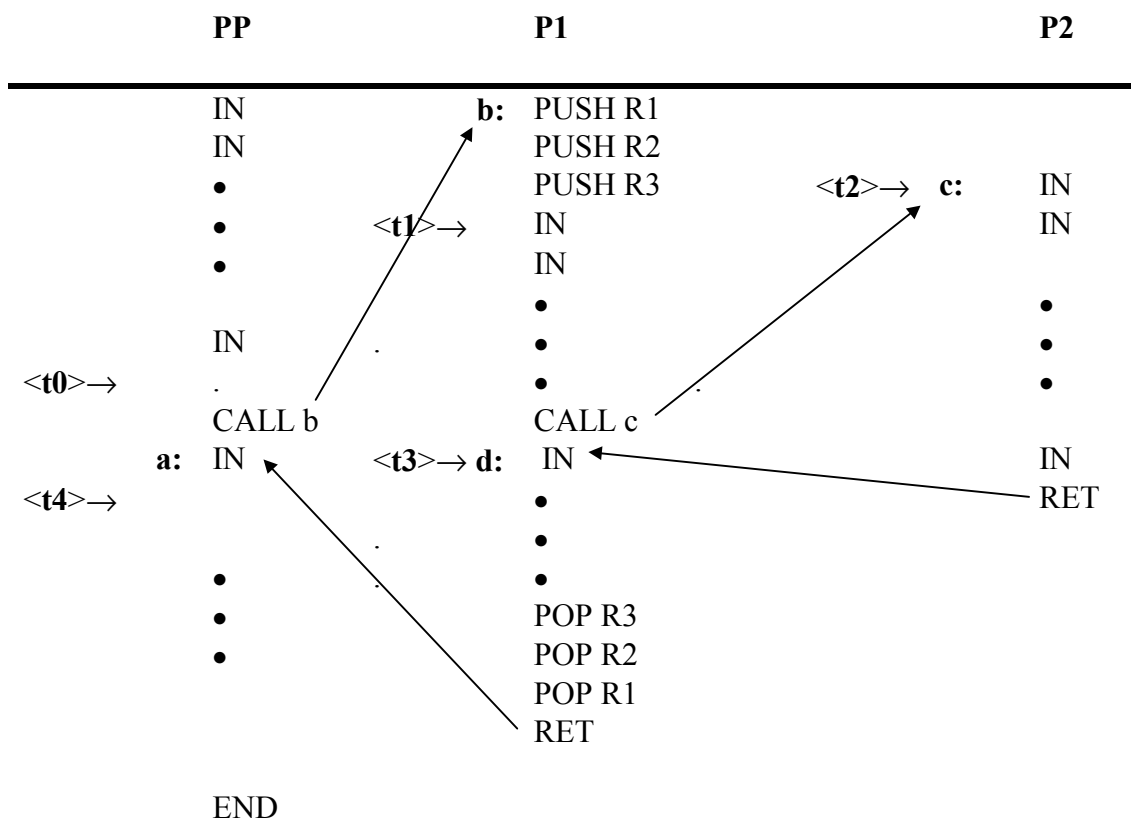
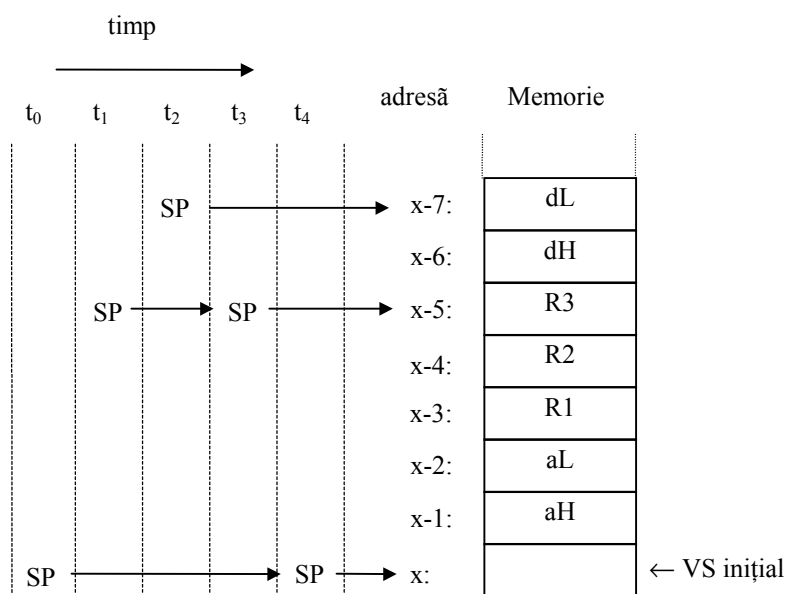


Figura 5.18a. Exemplu de apel a două proceduri notate cu P1 și P2. Pe figură s-au notat momentele de timp la care în figura 5.18b se analizează conținutul memoriei stivă.

În figura 5.18b s-a prezentat poziția vârfului stivei la momente de timp succesive. Se observă că s-a notat baza stivei (adresa inițială a vârfului stivei cu x). De asemenea s-au notat cu aH și aL, octeții mai semnificativ (High), respectiv mai puțin semnificativ (Low) ai adresei a. În procedura P1 s-au făcut și salvări ale conținutului unor registre (presupuse cu dimensiune jumătate față de adresă, în număr de biți). Restaurarea conținutului acestora trebuie făcută în ordine inversă salvării în stivă, datorită modului de lucru al acesteia.



5. 18b. Conținutul stivei și indicarea adresei vârfului stivei pentru exemplul din figura 5.18a.

5.7. UTILIZAREA FERESTRELOR DE REGISTRE

Dacă în paragrafele anterioare s-a considerat accepțiunea clasică a salvării și restaurării informației la o schimbare de context, pentru procesoarele RISC au apărut noi metode de salvare a vectorului de stare al unui program întrerupt și de asemenea metode noi pentru transferul parametrilor către proceduri. Ideea de bază este să se înlăture accesul repetat la o memorie stivă externă pentru salvarea / restaurarea informațiilor.

Schimbarea de context presupusă de un apel de procedură duce la o mare risipă de timp, pentru că presupune salvări sau restaurări din memoria stivă construită în memoria principală. La fiecare apelare de procedură variabilele locale programului apelant trebuie salvate din registre în memorie pentru ca registrele să poată fi utilizate la programul apelat. În plus, la apelarea procedurilor trebuie transferați și parametrii către procedură. La revenire trebuie restaurate variabilele programului părinte (încărcate din nou în registre) iar rezultatele trebuie trecute către programul părinte. Din această cauză soluția este să se introducă un set mare de registre interne procesorului, în care să se poată păstra informația necesară schimbării contextului, inclusiv pentru transmiterea de parametrii și returnarea de valori din proceduri. Mai mult, o bancă de registre locală permite adresarea cu un număr mai mic de biți, decât în cazul stivei externe, pentru că dimensiunea acestei bănci este în general mică. Astfel strategia este să se păstreze informația cât mai mult în registre locale și să se minimizeze operațiile de transfer registre - memorie.

Există două abordări pentru rezolvarea acestei probleme. Prima este o abordare software și se bazează pe un compilator care maximizează utilizarea registrelor în operațiile necesare de transfer. Compilatorul va încerca să aloce registre pentru acele variabile care sunt utilizate în cea mai mare perioadă a timpului de execuție. Această abordare cere utilizarea unor algoritmi sofisticati de analiză a programului. Abordarea hardware se referă pur și simplu la implementarea mai multor registre interne astfel ca să se poată menține cât mai multe variabile în memoria locală. Abordarea hardware a fost propusă pentru prima oară pentru procesorul RISC1 proiectat și implementat în anii '70 ai secolului trecut la University of California at Berkeley.

La acest procesor s-a utilizat un set mare de registre interne UCP. Setul conține 138 de registre de 32 de biți ce au fost aranjate în 8 ferestre a câte 22 de registre cu suprapunere parțială a

câte 6 registre vecine. Se folosesc de asemenea 10 registre globale, pentru toate procedurile active. Fiecare fereastră a fost împărțită astfel încât 6 registre pot fi utilizate pentru transmiterea parametrilor la apelarea procedurilor. Pentru selecția grupului de 6 registre se modifică doar un registru pointer. Pentru efectuarea unei apelări ("call") sau reveniri ("return") se modifică doar registrul pointer. Numărul mare de registre este necesar pentru a minimiza numărul de transferuri cu exteriorul procesorului. Prin această tehnică a ferestrelor de registre, cu suprapunere parțială apelarea procedurilor poate fi efectuată foarte rapid. Viteza este avantajoasă în special la aplicații în timp real unde este necesar un răspuns foarte rapid.

Totuși există și dezavantaje, pentru că dacă apelul procedurilor cere mai mult de șase variabile unul dintre registre trebuie să adreseze un vector stocat în memoria externă. Aceste date trebuie încărcate înainte de orice prelucrare iar tehnica ferestrelor de registre pierde mult din performanța propusă. Dacă se utilizează toate ferestrele suprapuse sistemul poate rezolva situația prin urmărirea utilizării ferestrelor și salvarea în memoria externă fie a unei ferestre fie a întregului set de registre. Acest eveniment poate face ca avantajul ferestrelor de registre locale să fie anulat, mai ales dacă la schimbarea de context trebuie salvate toate cele 138 de registre, pentru că crește timpul de răspuns.

Principiul ferestrelor de registre presupune că fiecare fereastră are același număr de registre, pentru fiecare procedură apelantă alocându-se o fereastră. Un apel de procedură va face doar o comutare a setului de registre utilizat de procesor în loc de a salva registrele în memorie. Ferestrele procedurilor adiacente se suprapun pentru a permite transferarea parametrilor. Conceptul este explicat în figura 5.19.

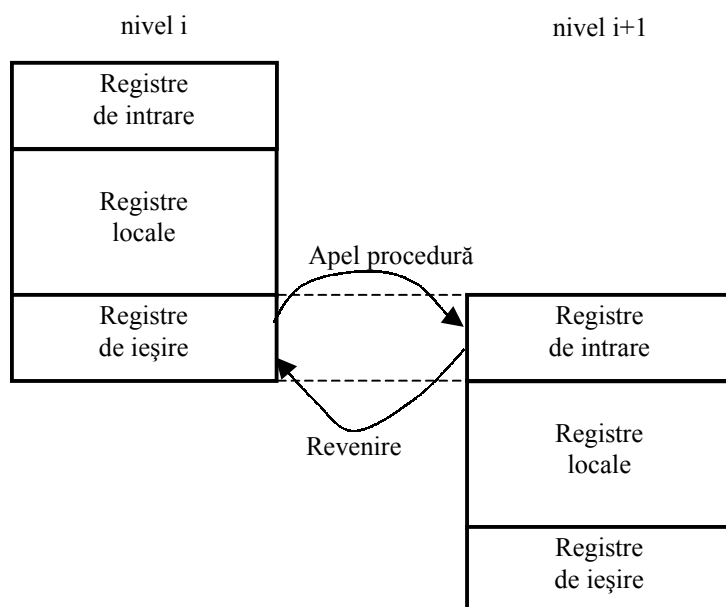


Figura 5.19. Explicativă la suprapunerea ferestrelor de registre

La fiecare moment de timp doar o fereastră de registre este vizibilă și adresabilă, ca și cum nu ar exista decât acel număr limitat de registre. Fereastra este împărțită în trei zone de lungime fixă. Registrele de intrare păstrează parametrii primiți de la programul (părinte) ce a apelat procedura și de asemenea păstrează rezultatele ce vor fi returnate părintelui. Registrele locale sunt utilizate pentru variabile locale, așa cum sunt atribuite de compilator. Registrele de ieșire sunt utilizate pentru a schimba parametrii și rezultate cu următorul nivel inferior (procedură apelată din procedura curentă). Registrele de ieșire de pe un nivel sunt *fizic aceleași* cu registrele de intrare de

pe nivelul inferior. Această suprapunere permite transmiterea parametrilor fără un transfer suplimentar de date între programele apelant și apelat.

Pentru a permite oricât de multe apelări de proceduri numărul de ferestre ar trebui să fie nelimitat. Această situație este rezolvată prin organizarea ferestrelor ca un tampon circular, ultima fereastră fiind suprapusă cu prima, ferestrele de registre păstrând doar cele mai recente proceduri apelate. Cele apelate anterior și care nu au loc în registrele locale sunt salvate în memoria principală și apoi restaurate de acolo. Pentru exemplificare vom presupune un set de patru ferestre de registre organizate ca un tampon circular în figura 5.20. În exemplu s-a presupus că procedura P1 (ca părinte) a apelat P2 (ca fiu), astfel că registrul pointer al ferestrei curente (PFC) a ajuns la fereastră 2, care este fereastră procedurii active la momentul actual. Se pot face în continuare apelări de genul P2 apelează (ca părinte) P3 (fiu) și P3 apelează P4, cu actualizarea registrului PFC. Adresa registrelor în corpul instrucțiunilor procesorului cu ferestre de registre constituie deplasament față de acest pointer (PFC), astfel că numărul de biți alocați adresării registrelor este determinat doar de dimensiunea fiecărei ferestre.

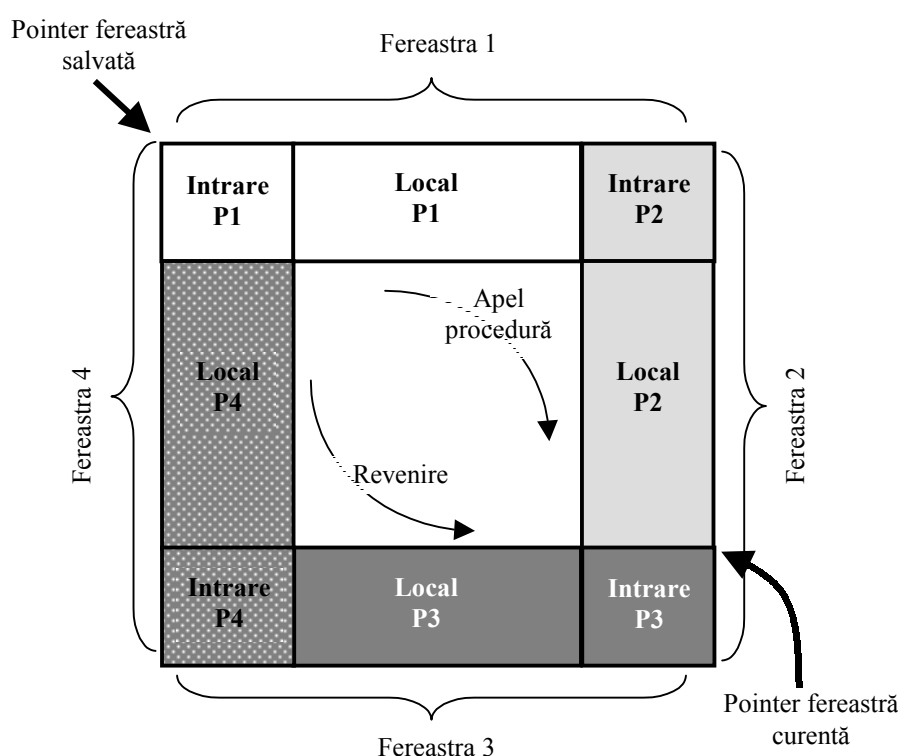


Figura 5.20. Ferestre suprapuse de registre cu organizare circulară și indicarea sensului de apel al procedurilor respectiv revenire din procedură.

Pointerul pentru fereastră salvată identifică cea mai recent fereastră salvată în memoria principală. Astfel dacă procedura P4 ar apela o altă procedură, de exemplu P5, registrul de ieșire al ferestrei 4 (identic cu registrul de intrare al ferestrei 1) ar încerca să scrie parametrii către procedura P5 peste registrele de intrare alocate procedurii P1. Astfel că atunci când PFC este incrementat (modulo 4 pentru acest exemplu) el devine egal cu pointerul ferestrei salvate ceea ce va produce o întrerupere (excepție) iar fereastră procedurii P1 este salvată în memoria externă. La revenire, dacă prin depășirea capacității numărului de ferestre a fost necesară salvarea în memoria principală, apare o nouă întrerupere, care va restaura informația corespunzătoare procedurii P1. Ori de câte ori se face

apelare, registrul PFC se incrementează, iar la revenire (return) se decrementează.

Un bun exemplu al acestei abordări este procesorul SPARC. Procesorul SPARC⁴ are o arhitectură RISC pe 32 de biți dezvoltată de firma Sun Microsystems. Arhitectura de bază este conform modelului RISC Berkeley și utilizează ferestre de registre pentru a îmbunătăți comutarea de context și transmiterea parametrilor.

Se utilizează o bancă de 120 de registre de 32 de biți împărțită în 7 ferestre de registre plus un set de opt registre disponibile global (16 registre \times 7 ferestre = 112 registre plus cele 8 globale). Fiecare fereastră conține 24 de registre împărțite în trei secțiuni pentru a furniza 8 registre de intrare, 8 registre locale și 8 registre de ieșire. Cele 8 registre de ieșire furnizează cele 8 intrări pentru următoarea fereastră. Dacă se selectează o nouă fereastră în timpul unei schimbări de context sau ca urmare a apelării unei proceduri, datele sunt copiate în registrele de ieșire, care constituie registre de intrare pentru procedura noului context de lucru. Ferestrele sunt suprapuse formând un lanț continuu (circular) ultima fereastră având registre comune cu prima fereastră.

La revenire, informația returnată la rutina apelantă este pusă în registrele de intrare și se execută return.

⁴ SPARC = Scalable Processor ARChitecture

CAPITOLUL 6

SISTEMUL DE MEMORIE

Conținut:

- 6.1. Organizarea ierarhică a sistemului de memorie
- 6.2. Caracteristici principale ale dispozitivelor de memorie
- 6.3. Structura și organizarea memoriilor statice cu acces aleator (SRAM)
- 6.4. Structura și funcționarea memoriilor dinamice cu acces aleator (DRAM)
 - 6.4.1. Structura și modul de lucru al unei DRAM convențional
 - 6.4.2. Moduri de lucru de mare viteză ale DRAM
 - 6.4.3. Moduri de reîmprospătare
 - 6.4.4. Exemple de circuite DRAM cu interfață asincronă
 - 6.4.5. Circuite DRAM sincrone (SDRAM)
- 6.5. Memoria cache
- 6.6. Tehnici de adresare și alocare a memoriei
- 6.7. Tehnici de translatăre a adreselor
 - 6.7.1. Maparea adreselor folosind pagini
 - 6.7.2. Mapare prin segmentare
 - 6.7.3. Mapare segmentat-paginată
 - 6.7.4. Memoria virtuală
 - 6.7.5. Noțiuni privind protecția memoriei

Obiective educaționale:

- a. Clasificarea tipurilor de memorie folosite în calculator și identificarea principalelor caracteristici
- b. Analiza modurilor specifice de funcționare ale dispozitivelor de memorie statice și dinamice
- c. Demonstrarea principalelor metode de alocare a spațiului de memorie, pentru asigurarea conceptelor memoriei virtuale și a protecției informației

6.1. ORGANIZAREA IERARHICĂ A SISTEMULUI DE MEMORIE

Dispozitivele de memorare din cadrul unui sistem de calcul au rolul de a stoca informația numerică (programe, date inițiale, rezultate intermediare și finale) cu posibilitatea ca informația numerică să poată fi accesată printr-o adresă, pentru a fi citită sau scrisă.

Dispozitivele de stocare a informației plus algoritmi de control ai memoriei (implementați fie în software fie în hardware) constituie **sistemul de memorie al calculatorului**.

Pentru un sistem de calcul unul din scopurile principale este prelucrarea unei cantități cât mai mari de date în unitatea de timp (pentru a defini această performanță globală de viteză a sistemului se folosește termenul "throughput" din limba engleză). Pentru creșterea performanțelor sistemului de calcul se urmărește nu numai creșterea vitezei de lucru a procesorului (UCP), dar și creșterea vitezei de transfer a datelor cu sistemul de memorie, și cu celelalte echipamente de intrare-ieșire (I/O). De obicei în cadrul calculatorului, sistemul de memorie are "onoarea" de a fi frânarul. Cantitatea maximă de date ce se pot transfera cu memoria în unitatea de timp e numită lărgime de bandă a memoriei (în engleză "bandwidth").

Din punctul de vedere al procesorului, ideal ar fi ca să existe acces la o memorie unică, rapidă și de dimensiuni suficient de mari pentru aplicațiile rulate. Practic însă, memoriile ce lucrează la viteze comparabile cu cea a procesorului sunt scumpe. Nu este posibil (cu excepția calculatoarelor foarte mici) să se folosească o singură memorie, ce utilizează doar un singur tip de tehnologie. De aceea, în proiectarea și construcția unui sistem de memorie, se face un compromis, al cărui obiectiv este obținerea unei viteze de acces medii cât mai mari în condițiile unui cost total minim, sau rezonabil, pentru sistemul de memorie. Datorită restricțiilor impuse de costuri, gabarit și consum de putere, în condițiile în care se doresc performanțe cât mai bune de la sistemul de memorie se folosesc următoarele căi de realizare ale acestui sistem:

- informația stocată este *distribuită* pe o varietate de dispozitive de memorie, dispozitive cu raport cost / performanță și caracteristici fizice foarte diferite. De aceea ele sunt astfel organizate pentru a asigura o *viteză medie cât mai mare*, la un *cost specific mediu cât mai mic*. Dispozitivele de memorie formează o ierarhie de dispozitive de stocare (vezi figura 6.1).
- *transferul de informații* între diferitele niveluri ale memoriei trebuie să se desfășoare *transparent* pentru utilizator (intră în sarcina sistemului de operare). De asemenea se urmărește realizarea unor metode de *alocare dinamică* a spațiului disponibil de memorie, pentru o utilizare cât mai eficientă.
- realizarea conceptelor *memoriei virtuale* (bazate pe metodele de alocare dinamică a spațiului de memorie principală) pentru a elibera programele utilizator de funcția administrării spațiului de memorie și pentru a face programele relativ independente de configurația dată a memoriei fizice.
- creșterea *lărgimii de bandă* a sistemului de memorie și asigurarea unor mecanisme de *protecție*, pentru a preveni ca programele utilizator să altereze zonele de memorie ale altor programe utilizator sau ale sistemului de operare.

În ceea ce privește organizarea ierarhică a dispozitivelor de memorie, pomenită mai sus, fiecare nivel de stocare are utilizare, viteză și dimensiuni specifice. Nivelul cel mai apropiat de UCP are capacitatea de stocare cea mai mică dar viteza cea mai mare. Vom numi acest nivel "*memoria locală a procesorului*". Ea este constituită dintr-un set de registre, integrate în UCP și este o memorie de mare viteză folosită pentru stocarea temporară a instrucțiunilor și datelor. UCP are *acces direct* la această memorie (nivelul 1 în figura 6.1). Noțiunile privind modul de lucru al registrelor interne s-au descris în capitole anterioare și deci nu vor fi tratate în continuare.

Următorul nivel are o capacitate relativ mare de stocare și o viteză mare, dar cu valoare mai mică decât viteza memoriei locale a UCP. Memoria de pe acest nivel este numită "*memorie*

principală" (dar și memorie internă, sau memorie operativă). Memoria principală este folosită pentru stocarea programelor și datelor necesare la un anumit moment de timp. Programele utilizator se încarcă (copiază) în memoria principală doar atunci când se lansează în execuție. Memoria principală este caracterizată de faptul că locațiile sale pot fi accesate direct și rapid de setul de instrucțiuni al UCP. Memoria principală este realizată în tehnologie semiconductoare (circuite integrate).

Ultimul nivel, cel mai îndepărtat de UCP este numit "memorie secundară" (memorie auxiliara, sau memorie externă) și are o capacitate foarte mare de stocare și o viteză mult mai redusă decât memoria principală. Este folosită pentru stocarea programelor de sistem, a fișierelor de program și date și altele de acest gen, care nu sunt cerute permanent de UCP. Informația din memoria auxiliara este accesată *indirect* de către UCP, prin intermediul sistemului de intrare - ieșire (I/O), cu ajutorul căruia informația cerută de UCP este adusă mai întâi în memoria principală. Tehnologii reprezentative pentru realizarea memoriei auxiliare sunt: magnetică (discuri și benzi), optică, semiconductoare.

Utilizarea unei ierarhii de dispozitive de memorie este determinată de asigurarea unui compromis între cerințele impuse de programele rulate (capacitatea de stocare, viteză medie de transfer cu memoria, protecția informației) și costurile sistemului de memorie (tehnologie disponibilă, preț, gabarit, algoritmi de transfer între diferitele niveluri ale ierarhiei de memorie).

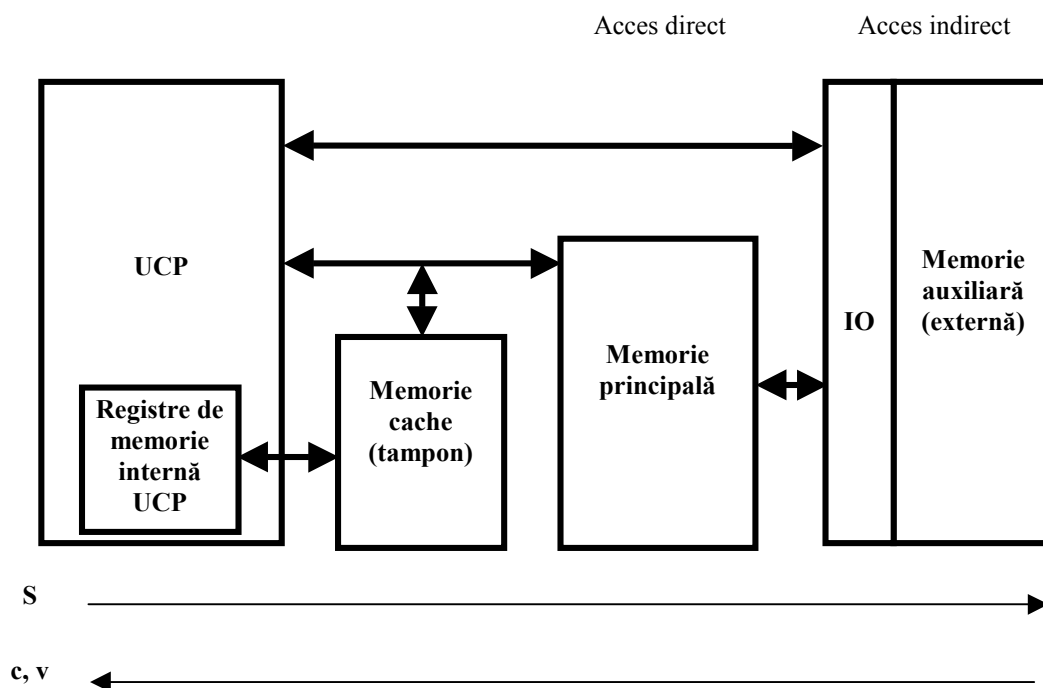


Figura 6.1 Evidențierea organizării ierarhizate a memoriei într-un sistem de calcul: Nivelul 1 este nivelul de memorie internă UCP, iar pe nivelurile 2 și 3 se găsesc memoria principală, respectiv memoria auxiliara (externă), privită ca un echipament periferic de I/O pentru stocarea datelor. Parametrii viteza (v), cost specific (c) și capacitate de stocare (S), cresc în direcțiile indicate de săgețile de sub figura. Memoria cache este o memorie tampon, intermediară nivelurilor 1 și 2.

Datorită faptului că viteza de lucru a procesoarelor de uz general a crescut continuu în ultimii ani (în mod curent peste 1GHz), iar viteza dispozitivelor de memorie folosite în cadrul memoriei principale nu a crescut în aceeași măsură, în multe din mașini se utilizează un nivel intermediar de memorie (intermediar ca poziție față de UCP, dar și ca viteză și capacitate de

memorare) numită memorie "cache". Această memorie are doar rol de memorie temporară, fiind un tampon de viteză mare între registrele interne ale procesorului și registrele ce constituie locațiile memoriei principale. Memoria cache conține în permanență doar copii din memoria principală ale zonelor de program și date active la un moment dat. Spre deosebire de cele trei niveluri de organizare a memoriei, memoria cache este transparentă pentru programator / utilizator (nu este văzută efectiv, pentru că nu există adrese separate pentru cache și pentru memoria principală. S-a mers chiar mai departe, încât la multe din procesoarele moderne se folosesc două niveluri intermediare de cache, cel mai apropiat de UCP fiind mai mic și mai rapid și încastrat în același circuit integrat cu UCP. Conform aceluiași principiu de compensare a diferențelor de viteză dintre niveluri, între memoria principală și memoria auxiliară pot apărea niveluri de memorie tampon, de asemenea transparente utilizatorului. Asemenea memorii tampon sunt specifice fiecărui dispozitiv periferic folosit ca memorie externă și se găsesc de obicei în cadrul circuitelor de control ale acestor dispozitive.

Este important de observat (figura 6.1) că datorită vitezei mari de lucru a memoriei cache, este de preferat ca toate operațiile procesorului să se desfășoare cu memoria cache, care permite de obicei citiri / scrieri într-o singură perioadă de ceas. Pentru ca acest lucru să fie posibil, circuitul care controlează memoria cache (controller de cache) trebuie să facă o anticipare a zonei din memoria principală în care se găsesc informațiile ce vor fi cerute de programul rulat de UCP. Controllerul este responsabil de copierea acestor informații din memoria principală în memoria cache și de scrierea în memoria principală a zonelor de date ce se elimină din cache și care au fost modificate în cache.

Memoria principală are un timp de acces moderat ca valoare (zeci de nanosecunde) și o capacitate de stocare mai mare de câțiva zeci de mega-octeți. Transferul direct între UCP și memoria principală se face numai în cazul când informația cerută nu a fost găsită în cache (evenimentul este numit *ratare la acces cache*¹).

Memoria auxiliară poate fi realizată în diferite tehnologii, având capacități de stocare mari și foarte mari (peste ordinul giga-byte în prezent), iar timpul de acces depinde puternic de tipul memoriei auxiliare și de modul cum se face transferul de informație între memoria auxiliară și cea principală. Legătura directă dintre cele două niveluri de memorie, simbolizată în figura 6.1, indică un transfer prin acces direct la memorie (DMA²) între perifericul memorie auxiliară și memoria principală. Transferul de informație între memoriile principală și auxiliară poate fi făcut însă și prin intermediul UCP, când UCP citește sau scrie prin program memoria auxiliară.

Schimbul de informații între diversele niveluri ale memoriei, trebuie să aibă un caracter transparent pentru utilizator. Aceasta se realizează prin folosirea unor resurse hardware și a unor tehnici software pentru administrarea și alocarea spațiului memoriei (principale) cu acces direct.

De obicei, la rularea unui program se folosește proprietatea numită "*localizarea referințelor la memorie*". Pe baza acestei proprietăți sistemul de memorie păstrează cele mai recent accesate articole, pe cât posibil, în memoria cea mai rapidă (care are și cea mai mică capacitate de stocare). Localizarea referințelor la memorie, observabilă în majoritatea programelor, se exprimă sub două forme:

- localizare *temporală* a referințelor: dacă programul face acces la o celulă de memorie la momentul t , este foarte probabil ca programul să facă din nou acces la aceeași celulă la momentul $t + \Delta$
- localizare *spațială* a referințelor: dacă la momentul t programul face acces la o celulă de memorie de adresă X , este foarte probabil ca la momentul $t + \Delta$ programul să facă acces la celula de adresă $X + \epsilon$.

¹ cache mis

² DMA = Direct Memory Access

Pentru a folosi proprietatea referințelor localizate în timp, datele cele mai recent accesate trebuie păstrate cât mai aproape de procesor. Pentru a folosi proprietatea referințelor localizate în spațiu la transferul între niveluri trebuie mutate blocuri continue de date și nu cuvinte individuale.

Dar organizarea ierarhică și folosirea proprietății localizării referințelor la memorie, face ca pe diferite niveluri ierarhice să existe informații copii ale aceluiași bloc de date, date ce pot fi modificate doar pe nivelul cel mai apropiat de UCP, sau și pe nivelurile inferioare. De aceea sistemele de memorie multi-nivel, cu posibilitatea de scriere-citire trebuie să satisfacă două proprietăți privind informația stocată:

- proprietatea de *incluziune*
- proprietatea de *coerență*

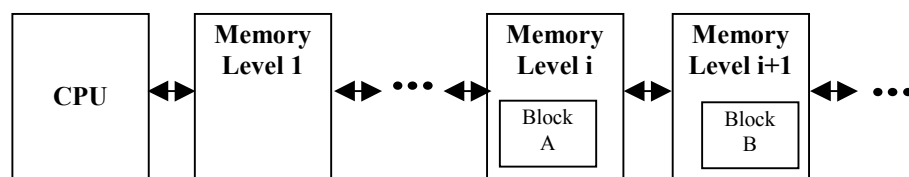


Figura 6.2. Sistemul de memorie indicat ca o succesiune de niveluri de memorie. Creșterea distanței față de UCP reprezintă, în general, o creștere a timpului de acces, dar și o creștere a capacității de stocare.

Pentru explicarea celor două proprietăți se desenează ierarhia de memorie ca o succesiune de niveluri de memorie (M_1, M_2, \dots, M_n) ca în figura 6.2.

Proprietatea de *incluziune* exprimă faptul că totdeauna informațiile care se găsesc pe un anumit nivel de memorie se vor găsi și pe nivelul de memorie inferior ($M_1 \subset M_2 \subset \dots \subset M_n$). Este posibil ca o informație existentă pe nivelul M_{i+1} să nu se găsească pe nivelul M_i , dar să fie accesată (cerută) pe nivelul i de memorie. În acest caz (evenimentul este numit "word miss") cuvântul necesar a fi accesat trebuie adus pe un nivel de memorie superior (de pe nivelul $i+1$, pe nivelul i).

Proprietatea de *coerență* exprimă faptul că informația existentă la o anumită adresă în spațiul de memorie trebuie să fie aceeași, indiferent de nivelul de memorie pe care se află. Dacă un cuvânt este modificat pe nivelul i , atunci va trebui modificat și pe nivelul $i+1$ și pe toate nivelurile inferioare. Coerența nivelurilor de memorie se poate obține fie prin propagarea valorii modificate spre toate nivelele inferioare ("write-through"), sau prin actualizarea nivelurilor inferioare în momentul înlocuirii informației de pe nivelul curent ("write-back").

6.2. CARACTERISTICI PRINCIPALE ALE DISPOZITIVELOR DE MEMORIE

Deși există o mare varietate de dispozitive de memorie, toate acestea au la bază un număr relativ mic de fenomene fizice și folosesc un număr relativ mic de principii de organizare. În continuare se vor descrie câteva caracteristici principale, comune atât pentru dispozitivele de memorie ce compun memoria principală cât și pentru cele din componența memoriei auxiliare.

Capacitatea de stocare a memoriei (S) reprezintă numărul total de biți ce pot fi stocați într-un dispozitiv sau circuit de memorie. Exprimarea parametrului S se face destul de rar în biți,

folosindu-se multiplii de octeți (vom nota bit cu b iar Byte (octet) cu B):

1 octet (Byte) are 8 biți

2^{10} biți = 1 kilo-bit (kb)

2^{10} Bytes = 1 kilo-Byte (kB)

2^{10} kB = 2^{20} B = 1024 kB = 1 MB

2^{10} MB = 1024 Mb = 1 GB

1024 GB = 1 TB

Pentru capacitatea de stocare exprimarea în multiplii de biți sau de octeți nu ține seama de modul de organizare internă și adresare a cuvintelor dispozitivului de memorie.

Timpul de acces la dispozitivul de memorie exprimă un timp mediu necesar pentru a scrie sau a citi o cantitate fixă de informație din / în memorie (un cuvânt de memorie). Se calculează din momentul când dispozitivul de memorie primește o cerere de citire (scriere) până în momentul când informația este disponibilă la terminalele de ieșire (sau este înscrisă în locația de memorie adresată). Acest timp depinde de caracteristicile fizice ale mediului de stocare și de tipul (modul) de acces folosit. Timpul de acces pentru memoriile semiconductoare moderne se exprimă în nanosecunde. Dintre tehnologiile semiconductoare de realizare, cele mai rapide sunt memoriile bipolare, dar pe de alta parte ele au un grad mai mic de integrare decât cele unipolare (CMOS sau celule dinamice) și consumă putere mai mare de la sursa de alimentare.

Inversul timpului de acces (t_A) este numit viteză de acces (b_A) la dispozitivul de memorie:

$$b_A = \frac{1}{t_A} \quad (6.1)$$

Timpul mediu de acces este o caracteristică a dispozitivului de memorie indiferent de viteza cu care lucrează procesorul.

Modul de acces (sau tipul de acces) spune modul în care se accesează informația dintr-un anumit dispozitiv de memorie. Modul de acces depinde în primul rând de tehnologia de realizare a dispozitivului de memorie și de principiul constructiv adoptat. Accesul la celulele individuale de stocare se poate face fie într-o anumită ordine (*acces pozițional, sau serial*), fie aleator (*acces aleator*). Accesul aleator este independent de poziția informației în cadrul dispozitivului de memorie. La memoriile cu acces serial (bandă magnetică, CD-ROM) scrierea sau citirea locațiilor de stocare se face în intervale diferite de timp, timpul de acces depinzând de poziția în care se găsește informația pe suportul fizic. Accesarea unei anumite locații presupune mișcarea mecanică a unui cap de citire / scriere și / sau a suportului de stocare.

Memoriile cu acces aleator sunt memoriile semiconductoare, la care accesarea oricărei locații de memorie (de la oricare adresa) se face în același interval de timp. Scrierea sau citirea informației se face direct la locația adresată fără parcurgerea unei secvențe de adrese.

Timp de ciclu și viteză de transfer a datelor. La memoriile cu citire distructivă (prin operația de citire informația stocată se pierde și trebuie refăcută) și la memoriile dinamice, nu este posibil să se înceapă un nou ciclu de acces la memorie până când nu s-a efectuat o restaurare a informației sau o reîmprospătare a acesteia. Ca urmare, intervalul minim de timp ce se scurge între inițierea a două accesări pentru transferul succesiv la / de la același dispozitiv de memorie, este mai mare decât timpul de acces; vom numi acest interval, *timp de ciclu* (t_M) al dispozitivului de memorie. Inversul acestui interval de timp exprimă cantitatea maximă de informație ce poate fi transferată la sau de la memorie în fiecare secundă:

$$b_M = \frac{1}{t_M} \text{ [cuvinte/sec]} \quad (6.2)$$

unde b_M este numită *viteză de transfer a datelor*, sau *lărgime de bandă* ("bandwidth"). Un factor

care limitează viteza de transfer a datelor este lăţimea magistralei de date pentru lucrul cu memoria, notată w . Valoarea lui w , indică numărul de biţi ce pot fi transferaţi în paralel pe magistrala de date / instrucţiuni. Această valoare, numită uneori cuvânt calculator, este specifică fiecărui procesor şi este, în general, un multiplu de octet. Ca urmare viteza de transfer poate fi exprimată şi ca:

$$b_M = \frac{w}{t_M} \text{ [biţi/sec]} \quad (6.3)$$

Atunci când t_M este diferit de t_A se folosesc amindouă mărimile caracteristice pentru a măsura viteza memoriei. Din punctul de vedere al cuplării memoriei la UCP timpul de acces (t_A) este deosebit de important pentru că arată timpul cât procesorul trebuie să aştepte după iniţierea unui acces la memorie.

Timpul de ciclu al memoriei este definit ca timpul minim ce trebuie alocat între iniţierea a doua operaţii succesive cu acelaşi dispozitiv de memorie (scriere sau citire). Timpul de ciclu pentru un dispozitiv de memorie, poate să fie diferit, în funcţie de operaţiile de scriere sau citire. Pentru memoriile semiconductoare statice (bipolare sau CMOS) cele două intervale de timp sunt foarte apropiate. Cel mai des însă $t_M > t_A$ pentru alte tipuri de memorie, cum ar fi cele semiconductoare de tip RAM dinamic (DRAM), din cauza naturii distructive a operaţiei de citire şi a necesităţii reîmprospătării ulterioare a informaţiei stocate. Desfăşurarea în timp a operaţiilor unui ciclu de lucru cu memoria poate fi descrisă prin figura 6.3.

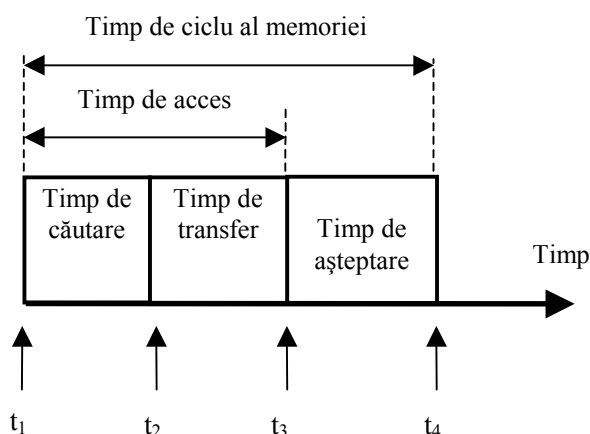


Figura 6.3. Explicativă la desfășurarea în timp a operaţiilor de acces la un dispozitiv de memorie.

În figura 6.3. s-au notat momentele de timp semnificative ale unui ciclu de acces la memorie. Astfel la momentul t_1 se face referirea la memorie (prin adresă şi semnale de control - selecţie). La momentul t_2 locaţia adresată a fost găsită. La momentul t_3 informaţia adresată este complet transferată (scriere sau citire), iar la momentul t_4 este permisă o nouă referire la acelaşi dispozitiv (circuit) de memorie.

Costul dispozitivului de memorie nu cuprinde de obicei doar costul celulelor de stocare ci şi pe cel al echipamentelor periferice sau circuitelor de acces necesare în funcţionarea memoriei. Se exprimă de obicei global, pe niveluri de memorie, sau pentru întreaga memorie a unui calculator, în unităţi specifice, (cost/bit). Valoarea specifică se obţine prin împărţirea costului total al blocului, sau sistemului de memorie la capacitatea totală de stocare. Există o contradicţie între viteza memoriei şi costul pe bit: costul specific al memoriei scade pe măsură ce creşte timpul de acces (scade viteza de acces) la memorie şi, de fapt, această relaţie constituie fundamentul tehnico-economic al realizării memoriei sub formă de mai multe niveluri ierarhice.

Puterea consumată se exprimă de obicei ca putere consumată raportată la un bit de informație, obținându-se prin împărțirea puterii totale consumate de memorie la capacitatea de stocare.

$$p = \frac{P_{tot}}{S} \quad [\text{W/bit}] \quad (6.4)$$

Geometria sau modul de organizare al memoriei este exprimată sub forma $M \times N$, unde M reprezintă numărul de cuvinte posibil de memorat, iar N dimensiunea cuvântului, în biți. Caracteristica numită geometrie este folosită doar pentru dispozitive de memorie din memoria principală. Este o caracteristică importantă din punctul de vedere al accesării informației din memoria principală, pentru că indică cantitatea minimă de informație adresabilă. Dacă memoria este organizată pe octet, atunci $N = 8$ și fiecărui octet i se alocă o adresă specifică, chiar dacă magistrala de date este de 32 sau de 64 de biți.

Alterabilitate. Este caracteristica care se referă la posibilitatea modificării, de către procesor (on-line) a informației înscrise într-un dispozitiv de memorie. La circuitele semiconductoare ROM, metoda utilizată pentru scrierea informației în memorie este ireversibilă; nu mai poate fi modificată în timpul funcționării memoriei. La PROM-urile semiconductoare schimbarea, (PROM și EPROM) schimbarea conținutului se poate face off-line prin programarea memoriilor cu ajutorul unui circuit programator. Toate aceste circuite sunt incluse în categoria celor ne-alterabile. Dispozitivele de memorie alterabile sunt cele la care conținutul poate fi modificat de către UCP în timpul rulării programelor. Ca exemple putem specifica dispozitivele RAM statice și dinamice, discul magnetic, banda magnetică etc.

Permanenta stocării. Procesul fizic folosit pentru stocarea informației este uneori instabil, astfel că informația s-ar pierde după un timp dacă nu se iau măsuri de restaurare. Există trei evenimente care pot duce la pierderea informației stocate:

- citire distructivă
- stocare dinamică
- volatilitatea

Citirea distructivă se referă la faptul că dacă o celulă de memorie este citită, informația din celulă este alterată și ea trebuie restaurată imediat după citire. Citirea distructivă se datorează în primul rând fenomenului fizic folosit pentru stocarea informației și apoi structurii hardware a celulei. Exemple de celule la care citirea este distructivă: celule dinamice cu un singur tranzistor unipolar de acces și celule pe bază de ferită. Citirea trebuie urmată obligatoriu de operația de scriere, pentru restaurarea informației. Operația este efectuată de către circuite specifice dispozitivului de memorie și nu intră de obicei în sarcina UCP.

Stocarea dinamică a informației se referă la stocarea în celule ce folosesc capacitatoarele MOS. Aceste capacitatoarele se descarcă în timp, și dacă informația conținută nu este reîmprospătată dinamic (condensatoarele re-încărcate), la intervale pre-stabilite informația se pierde.

Memoriile volatile sunt cele care pierd informația la pierderea tensiunii de alimentare. Încadrăm aici toate memoriile RAM semiconductoare, indiferent că sunt dinamice sau statice. Ca exemple de memorii nevolatile: ROM, PROM, EPROM, EEPROM, disc, bandă etc.

6.3. STRUCTURA SI ORGANIZAREA MEMORIILOR STATICE CU ACCES ALEATOR (SRAM)

Memoriile cu acces aleator sunt caracterizate prin faptul că fiecare locație poate fi accesată independent, timpul de acces pentru oricare locație fiind același - independent de poziția locației. Fiecare celulă conține linii de date (citire sau scriere - numite linii de bit la majoritatea celulelor

bipolare sau unipolare) și linii de selecție.

În acest paragraf ne ocupăm de memoriile cu acces aleator de tip citește / scrie, notate cu sigla RAM = Random-Access Memory, la care celulele de memorie sunt bistabile ce păstrează informația pe toată durata alimentării cu energie.

Dispozitivele interne RAM necesare pentru decodificarea cuvântului de adresă, pentru comanda semnalelor de selecție a celulelor, pentru circuitele de citire sau scriere a informației și circuitele de control intern sunt numite *circuite de acces* la unitatea de memorie.

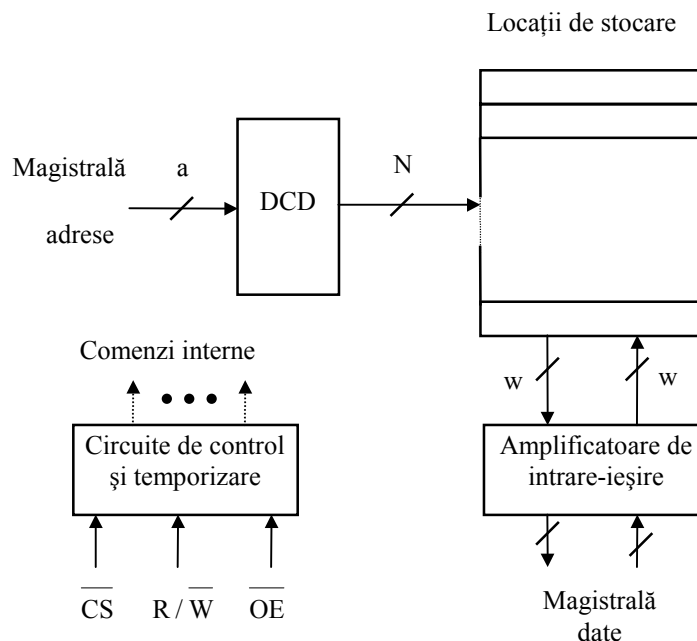


Figura 6.4. Schemă bloc pentru memorie RAM statică cu adresare uni-dimensională a locațiilor de memorie

Modul de organizare al circuitelor de acces au un efect important asupra costului total al circuitului de memorie. Dacă un singur circuit decodificator (DCD) selectează fiecare locație de memorie (în care se găsește o celulă de un bit sau un registru de w biți) avem de-a face cu o schemă de adresare unidimensională ($d=1$) a locațiilor de memorie. (figura 6.4).

Pentru fiecare adresă cu lungimea de a biți la intrarea circuitului de memorie, decodificatorul activează una din cele N ($= 2^a$) linii de ieșire care selectează o locație (celulă sau cuvânt de w biți) de memorie, pentru citire sau scriere. Semnalele de control din figură au semnificațiile următoare:

- \overline{CS} = chip select - selecție circuit. Semnalul este activ în nivel jos (0 logic)
- R/\overline{W} = read/write, scrierea făcându-se pe nivel 0 logic, iar citirea pe nivel 1
- \overline{OE} = output enable - validarea ieșirilor dispozitivului de memorie. Dacă $\overline{OE}=0$ atunci amplificatoarele de ieșire sunt deschise. Altfel amplificatoarele de ieșire ele sunt în stare de înaltă impedanță (HiZ), pentru a permite cuplarea mai multor ieșiri la aceeași magistrală comună de date. La multe din circuitele integrate SRAM intrarea de control \overline{OE} lipsește funcția sa fiind preluată de \overline{CS} care pe lângă funcția de selecție a circuitului are și sarcina controlului circuitelor de ieșire, eventual în combinație cu semnalul de citire / scriere.

De exemplu pentru o memorie cu organizarea $8K \times 8$ biți cu o astfel de organizare a circuitelor de selecție, $a=13$, $N = 2^{13}$, $w = 8$.

Întotdeauna este însă mai avantajoasă o organizare bidimensională ($d=2$) a selecției circuitului de memorie, prin împărțirea câmpului de adrese în două părți (a_x biți pentru linii și a_y linii pentru coloane) și implicit împărțirea decodicatorului în două blocuri: decodicator de linii (decodicator X) și decodicator de coloane (decodicator Y), ca în figura 6.5.

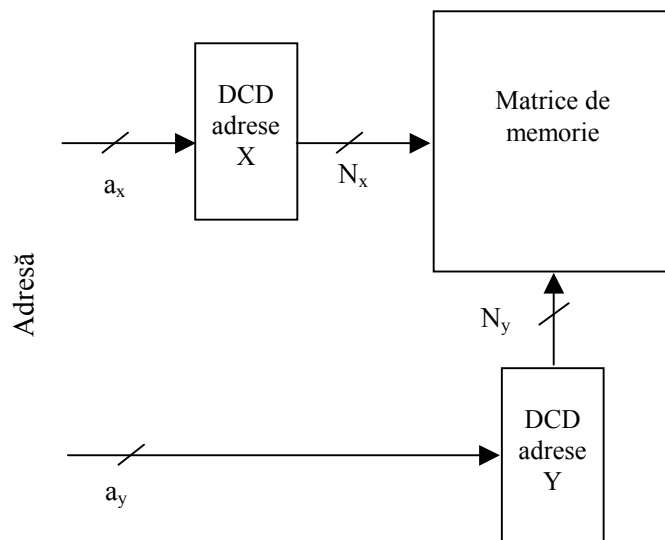


Figura 6.5. Schemă bloc care indică selecția bidimensională a matricei de celule de memorie

Celulele de memorie din figura 6.5 sunt selectabile individual, prin adresa de linii și adresa de coloane. Numărul total de celule este $N = N_x \times N_y$, unde de obicei $N_x = 2^{a_x}$ iar $N_y = 2^{a_y}$. Acest mod de selecție necesită circuite de acces mult mai simple (de unde rezultă și costuri mai reduse) decât în cazul $d=1$. De exemplu, pentru a compara cele două moduri de selecție, dacă $N_x = N_y = \sqrt{N}$, numărul de dispozitive ce comandă direct liniile de selecție ale matricei de memorie este $2\sqrt{N}$ în loc de N ieșiri ca la $d=1$.

Pentru $N \gg 4$ diferența este semnificativă. În plus organizarea bidimensională este adecvată și cu structura bidimensională a suprafeței chip-ului de siliciu folosit de tehnologia de realizare a circuitului integrat (CI). Aceste avantaje se diminuează, sau chiar dispar pentru dimensiuni d mai mari decât 2. De aceea organizări cu $d > 2$ sunt rar utilizate.

Bineînțeles că în figura 6.5, celulele adresate pot fi înlocuite cu registre de w biți selectate de aceleași semnale. Rezultă astfel un SRAM organizat pe cuvânt. Considerând același exemplu numeric ca și la selecția unidimensională (8K cuvinte de 8 biți) structura de principiu a memoriei ar fi cea din figura 6.6, în care se observă introducerea a 8 matrice de memorie selectabile la nivel de celulă (bit) extinse în "adâncime". Aceleași decodificatoare fac selecția pentru toate cele 8 planuri, dar sunt necesare 8 amplificatoare de citire / scriere (nefigurate).

Structura bloc din figura 6.6. pune însă probleme de implementare, din cauza planurilor suprapuse de celule de memorie. De aceea se încearcă construirea unui astfel de circuit de memorie în așa fel încât matricea de memorie să aibă aproximativ același număr de linii cu numărul de coloane ("matrice pătrată"), iar celulele să fie construite în același plan al circuitului integrat. Varianta corectă din punctul de vedere al implementării, pentru exemplul memoriei cu 8k cuvinte fiecare de câte 8 biți se indică în figura 6.7.

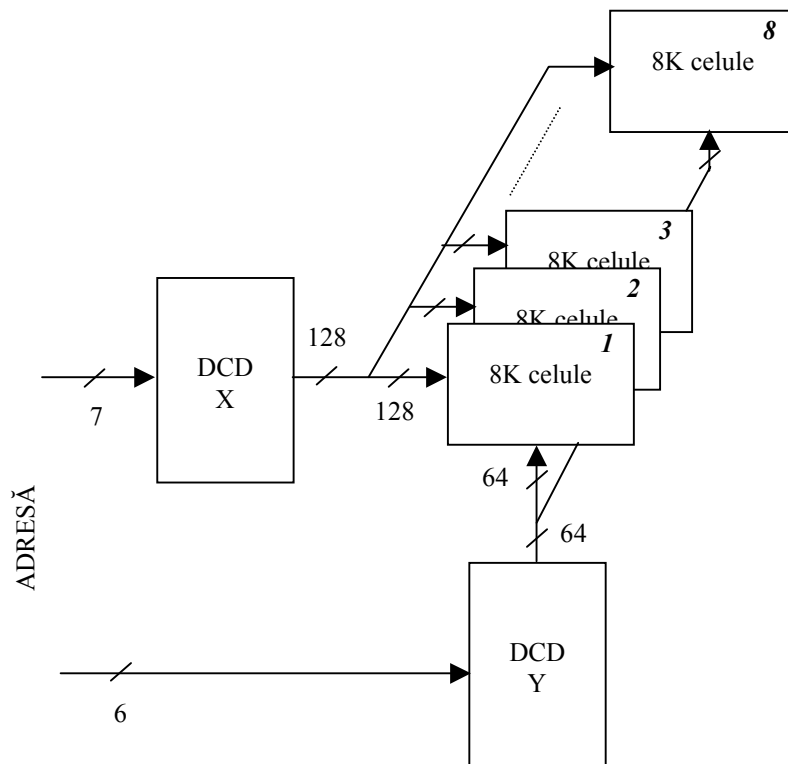


Figura 6.6. Schemă bloc de principiu pentru o memorie SRAM cu capacitatea $8k \times 8$ biți și selecție bidimensională.

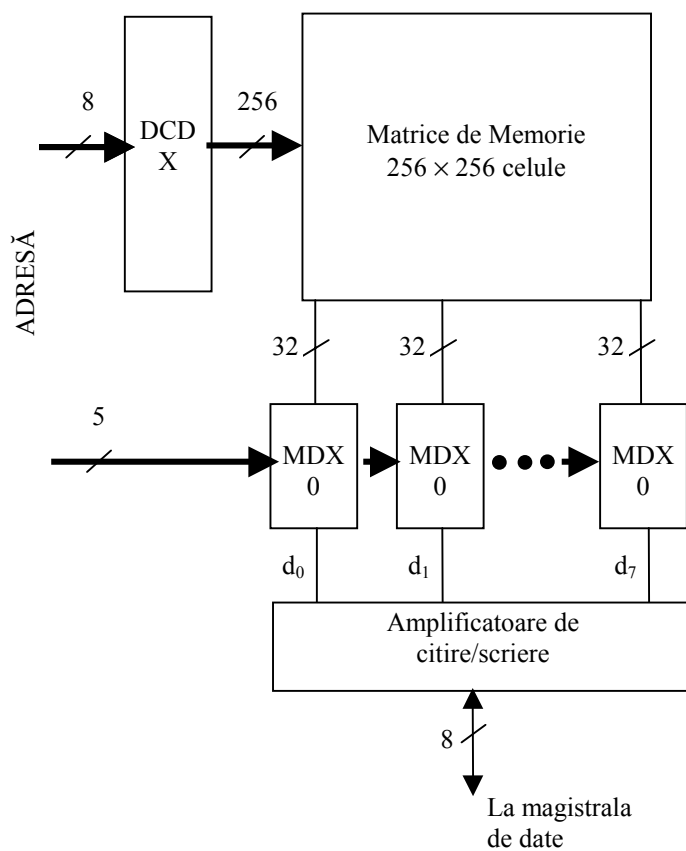


Figura 6.7. Exemplu de memorie RAM statică cu dimensiunea de $8k \times 8$ și selecție bidimensională a matricei de memorie.

Selecția este bidimensională, iar în locul decodicatorului de coloane s-au introdus circuite multiplexor / demultiplexor (MDX). Structura unui asemenea circuit poate fi găsită în [Stefan93]. Circuitul MDX conține un decodicator, iar la ieșirea sa se comandă porți de transfer de tip MOS. Atunci când se face citire funcționarea circuitului este cea specifică demultiplexorului, iar când se face scriere funcționarea este de multiplexor.

Conform caracteristicilor prezentate în paragraful anterior, fiecărui dispozitiv de memorie care lucrează cu procesorul i se impun anumite *condiții și restricții de timp*. Acestea sunt descrise de obicei prin diagrame de semnale în timp, pe baza unor cicluri caracteristice de funcționare ale dispozitivelor de memorie.

Pentru o memorie statică cu acces aleator ciclurile caracteristice de funcționare pot fi doar cele de citire și de scriere. În figura 6.8.a se prezintă un ciclu tipic de citire, iar în figura 6.8.b un ciclu tipic de scriere.

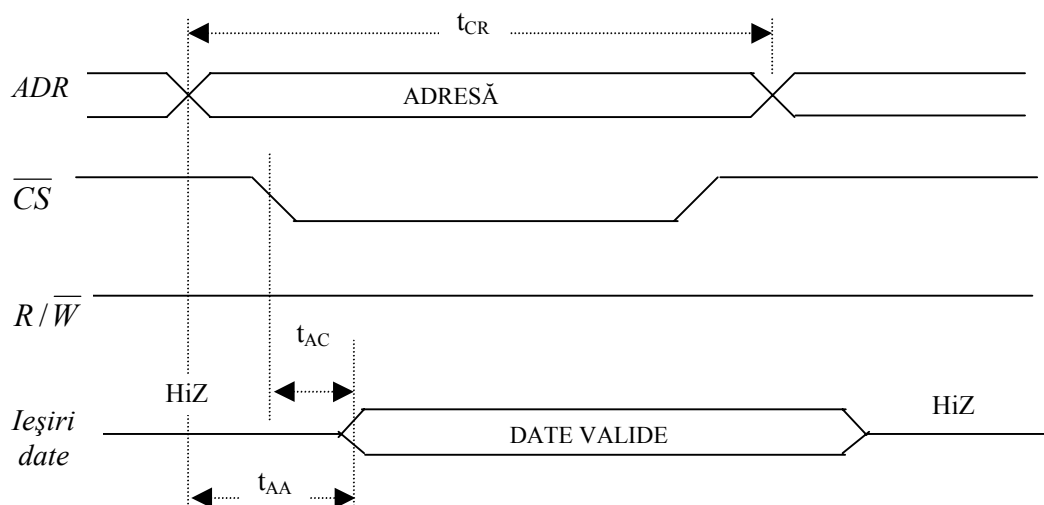


Figura 6.8.a. Exemplu de ciclu de citire la SRAM

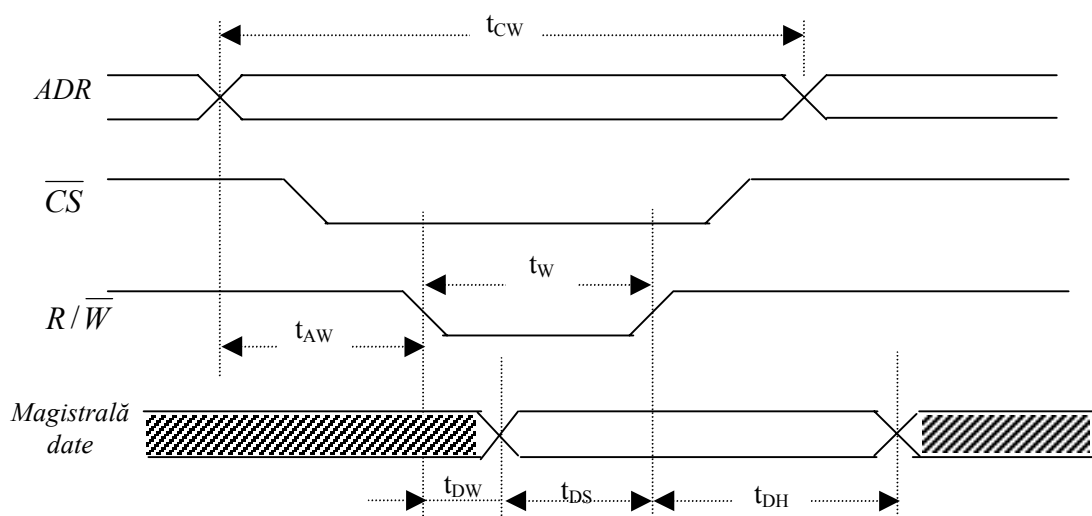


Figura 6.8.b. Exemplu de ciclu de scriere la SRAM

În cele două figuri s-au notat intervalele de timp importante, cu următoarea semnificație:

t_{CR} = durata ciclului de citire (se impune ca timp minim pentru funcționarea corectă a memoriei)

t_{AA} = timp de acces de la activarea adreselor (se garantează ca o valoare maximă ce se poate obține)

t_{AC} = timp de acces de la activarea semnalului \overline{CS} (se garantează ca o valoare maximă ce se poate obține)

t_{HD} = timp de menținere a datelor (valoare minimă impusă)

t_{CW} = durata ciclului de scriere (se impune ca timp minim pentru funcționarea corectă a memoriei)

t_W = durata semnalului de comandă de scriere (write - este valoare minimă în catalog)

t_{DS} = timp de prestabilire date (valoare minimă impusă)

t_{DH} = timp de menținere a datelor (valoare minimă impusă)

t_{DW} = întârzierea validării datelor la intrare

t_{AW} = întârziere între stabilirea adreselor și semnal de control de scriere

Toate restricțiile de timp se indică în foile de catalog ale dispozitivelor de memorie și ele trebuie privite ca limitări în viteza cu care se desfășoară comunicația dintre UCP și respectivul dispozitiv. În general valorile de catalog ale acestor timpi trebuie considerate ca valori minime, sub care nu este bine să se scadă. Acestea limitează frecvența maximă a operațiilor efectuate de UCP cu memoria. De asemenea în cazul valorilor timpilor de acces este bine să se considere valorile maxime date în catalog ca valori după care UCP este sigur că datele au fost citite (de pe magistrala de date) sau înscrise în memorie. De aici rezultă uneori necesitatea unor mecanisme de sincronizare (de tip READY) ale UCP cu dispozitivele de memorie.

În figura 6.8.a s-au reprezentat terminalele de ieșire ale dispozitivului de memorie, terminale ce se cuplează la magistrala de date. Ele se găsesc în stare de înaltă impedanță până în momentul când s-a scurs timpul de acces și atunci când dispozitivul de memorie este neselectat. Pentru ciclul de scriere din figura 6.8.b zona hașurată a magistralei de date reprezintă valori neimportante pentru ciclul de scriere. Porțiunea ne-hașurată reprezintă datele stabile puse de microprocesor pe magistrala de date, pentru a putea fi scrise corect în dispozitivul de memorie.

6.4. STRUCTURA ȘI FUNCȚIONAREA MEMORIILOR DINAMICE CU ACCES ALEATOR (DRAM)

Acest tip de memorii se numesc "dinamice" datorită principiului de lucru al celulelor interne. În fiecare celulă se reține informația sub formă de sarcină electrică într-un capacitor MOS³. Indiferent de tehnologia de realizare sarcina electrică dintr-un capacitor se descarcă lent. În cazul memoriei, pierderea sarcinii, înseamnă pierderea informației - circuitul "uită" informația stocată. Pentru a preveni acest fenomen, din timp în timp, informația trebuie reîmprospătată, adică condensatoarele trebuie reîncărcate cu informația stocată anterior. Aceasta înseamnă că informația poate fi păstrată doar printr-o *funcționare dinamică*, ce presupune încărcarea periodică a condensatoarelor interne.

Dispozitivele DRAM⁴ sunt în prezent cele folosite dispozitive pentru construirea memoriei principale a calculatoarelor de uz general, pentru că ele prezintă cea mai mare densitate de integrare

³ MOS = Metal Oxid Semiconductor, tehnologie de realizare a dispozitivelor semiconductoare unipolare

⁴ DRAM = Dynamic Random Access Memory

dintre tehnologiile adecvate memoriei principale și totodată prezintă cel mai mic cost pe bit.

6.4.1. Structura și modul de lucru al unei DRAM convențional

Dispozitivele DRAM convenționale au următoarele caracteristici structurale și funcționale:

- *Matricea* de memorie este *pătrată* ($m \times m$; iar $m = 2^n$)
- Se folosește o schemă de adresare bidimensională în care sarcinile de selecție ale matricei de memorie sunt împărțite între un decodificator de linii și unul de coloane.
- Se folosește *multiplexarea* biților pentru adresele de linie și adresele de coloană. Adresa furnizată dispozitivului de memorie se face printr-un număr de linii egal cu jumătate din numărul total de biți de adresă (de exemplu pentru 2^{22} celule = 4 Mbit cu matrice de 2048 x 2048 celule, numărul de pini de adresă este 11 și nu 22). Principalul avantaj îl reprezintă un număr mai mic de pini la capsulă, deci un gabarit mai redus al circuitului. Gabaritul mai mic se referă nu doar la numărul de terminale ci și la circuitele tampon - interfață cu exteriorul necesare pentru fiecare pin.
- au o *interfață asincronă* cu exteriorul (în particular cu procesorul) operațiile fiind comandate, nu de un semnal de ceas ci, prin semnale la care se impun durate și întârzieri minime între fronturi, în așa fel încât operațiile să se desfășoare corect.

Structura bloc a unui circuit DRAM convențional este prezentată în figura 6.9. Amplificatoarele senzor din figură sunt circuite care generează nivel logic 0 sau 1 pornind de la tensiunea de pe condensatorul - celulă selectat. De câte ori se selectează o linie, același circuit amplificator face și reînscriserea tuturor condensatoarelor de pe linia respectivă.

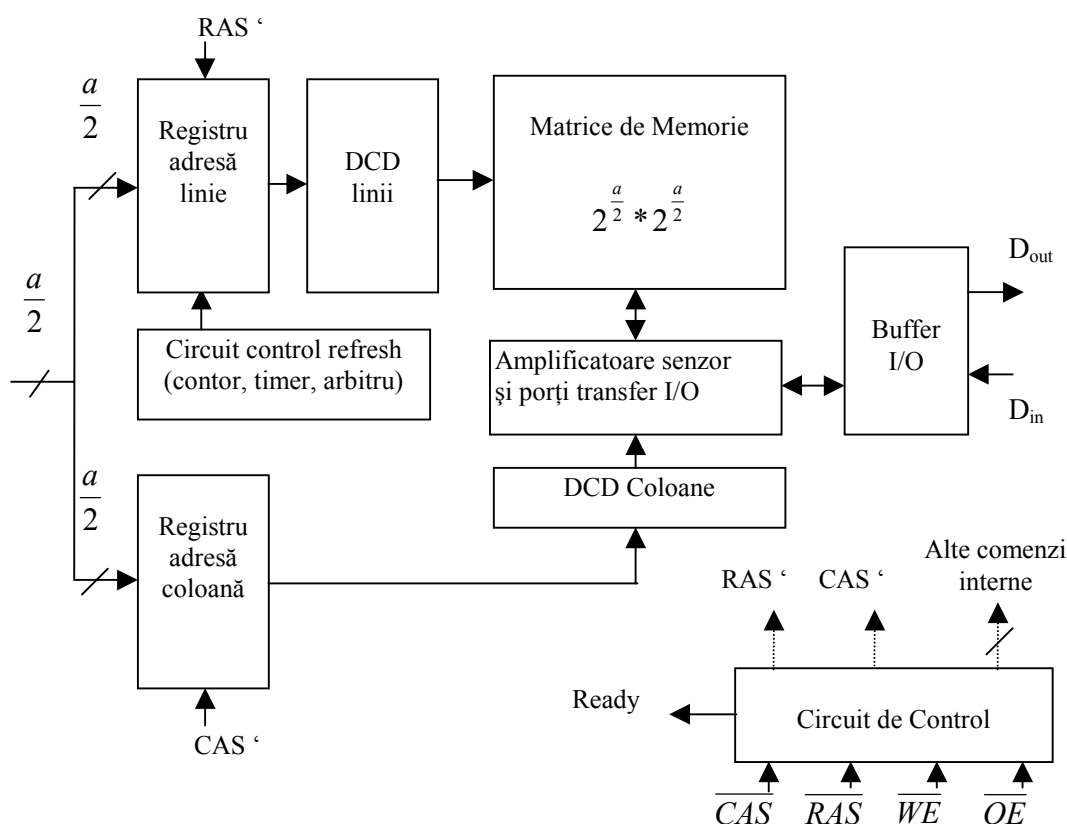


Figura 6.9. Structura bloc a unui circuit DRAM convențional și semnalele tipice de interfațare cu procesorul

Structura formată din decodificatorul (DCD) de coloane și porțile de transfer de I/O funcționează ca o structură de demultiplexare / multiplexare similară blocului MDX pomenit la memoriile SRAM.

Pe structura din figura 6.9. s-a presupus că circuitul de control al operației de re-împrospătare se găsește pe același circuit cu memoria dinamică, deși la primele memorii DRAM aceste circuite de control erau externe.

Atunci când se dorește accesul la o locație de memorie, adresele se introduc în ordinea:

1. *adresa de linie*, validată de semnalul \overline{RAS} (row address strobe), care produce memorarea acestei adrese în registrul de adresă linie, intern circuitului de memorie, iar apoi
2. *adresa de coloană*, validată de semnalul \overline{CAS} (column address strobe), care produce memorarea acestei adrese în registrul de adrese coloane. Semnalul \overline{CAS} mai este folosit de logica de control internă circuitului pentru a controla tampoanele de ieșire ale memoriei. În cazul în care \overline{CAS} nu a trecut în zero pentru a valida adresa de coloană, tampoanele de ieșire rămân în stare de înaltă impedanță (HiZ). Lungimea semnalelor \overline{RAS} și \overline{CAS} ca și întârzierea dintre ele trebuie să satisfacă cerințele de catalog pentru fiecare tip de circuit DRAM. Celula adresată furnizează data stocată prin amplificatorul senzor (care realizează niveluri logice de tensiune la ieșirea sa) către tamponul de ieșire date, via o poartă de I/O. Dacă se dorește scriere, se activează \overline{WE} iar datele se furnizează pe la intrările D_{in} . Informația este preluată de poarta I/O, amplificată de amplificatoarele senzor și înscrisă în celula adresată.

Circuitele de control ale memoriei principale construite cu DRAM au trei sarcini principale:

- separarea adreselor de la UCP în adrese de linii și adrese de coloană;
- activarea corectă a semnalelor \overline{RAS} , \overline{CAS} , \overline{WE}
- transmisia și recepția datelor citite sau scrise
- generarea semnalelor de control pentru reîmprospătare (la unele memorii trebuie generate din exterior și adresele de reîmprospătare)

Noile concepte ale DRAM moderne cum sunt modul de lucru prin *întrețesere* și modul *pagină*, creează sarcini suplimentare pentru circuitul care controlează DRAM. De aceea controllerul extern de memorie reprezintă o parte extrem de importantă a calculatorului. În ceea ce privește semnalele furnizate DRAM (figura 6.10), adresele de linii și coloane trebuie să fie stabile înainte (timp de pre-stabilire⁵) și după apariția frontului descrescător al \overline{RAS} respectiv \overline{CAS} (timp de menținere⁶).

În mod normal selecția celulei pentru citire sau scriere nu mai are loc dacă succesiunea \overline{RAS} - \overline{CAS} nu s-a produs (Există însă situații când această ordine este încălcată, la memoriile moderne unde reîmprospătarea se face cu activare \overline{CAS} înainte \overline{RAS}). Mai specificăm pe scurt câteva caracteristici pentru semnalele de control și temporizările la DRAM convenționale:

- $\overline{WE} = 0$ comandă scriere, iar $\overline{WE} = 1$ citire. Dacă \overline{WE} trece în zero înaintea lui \overline{CAS} ("ciclu de scriere în avans") ieșirile de date rămân în HiZ de-a lungul întregului ciclu de scriere. Dacă însă \overline{WE} trece în zero după \overline{CAS} ("ciclu de scriere - citire") după operația de scriere datele apar și la ieșirile de date ale circuitului. La scriere, ultimul dintre semnalele \overline{CAS} și \overline{WE} (fronturi căzătoare) captează datele pentru scriere în celulă.

⁵ setup time = timp de prestabilire

⁶ hold time = timp de menținere

- Semnalul \overline{RAS} mai contribuie la activarea decodificatorului de linii și a circuitelor de reîmprospatare.
- \overline{CAS} face selecția circuitului (activează decodificatorul de coloană și tamponalele de I/O). Când timp $\overline{CAS} = 1$ ieșirile de date sunt în HiZ. În cazul în care memoria are și semnal de intrare de tip \overline{OE} (validare ieșire - Output Enable) funcțiile semnalului \overline{CAS} asupra circuitelor tampon de ieșire sunt preluate de semnalul \overline{OE} suplimentar.
- Intervalele de timp trecute în figura 6.10 se referă la:
 - t_{RAC} = timp de acces față de RAS; timpul minim de la activarea (cădere în 0) a liniei RAS până la apariția datelor valide la ieșire. Adesea este valoarea înscrisă pe dispozitivul DRAM (de exemplu $t_{RAC} = 60$ ns)
 - t_{RC} = timp de ciclu la operația de citire; timp minim între începerea unui acces la o linie și începutul următorului ciclu de acces.
 - t_{CAC} = timp de acces față de CAS; timpul minim de la activarea (cădere în 0) a liniei CAS până la apariția datelor valide la ieșire. Pentru o memorie cu $t_{RAC} = 60$ ns t_{CAC} este de aproximativ 15 ns.
 - t_{PC} = intervalul de timp minim între două accesări succesive la adrese de coloană. Pentru o memorie cu $t_{RAC} = 60$ ns t_{PC} este de aproximativ 35 ns.

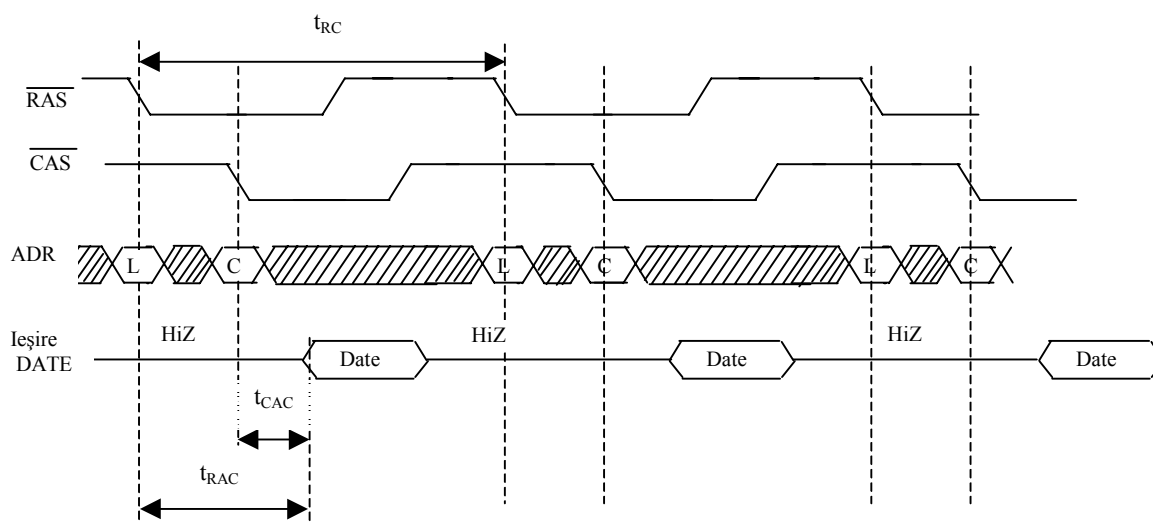


Figura 6.10. Cicluri de acces pentru citire la DRAM convențional. Pe formele de undă ce simbolizează magistrala de adrese s-a notat cu L adresa de Linie și cu C adresa de Coloană.

6.4.2. Moduri de lucru de mare viteză ale DRAM

În plus față de modul convențional de funcționare, descris mai sus, la care adresele de linii și coloane sunt furnizate pentru fiecare ciclu de acces s-au realizat și alte moduri de acces la coloane, pentru a reduce timpul mediu de acces la memoria DRAM.

Mod Pagină

Principiul de lucru în modul pagină consideră că pagina corespunde unei adrese de linie și toate adresele de coloană, cu aceeași adresă de linie se găsesc în aceeași pagină. În mod pagină

controllerul de memorie schimbă în mod repetat doar adresa de coloană. De aceea o "pagină" este echivalentă unei linii din matricea de celule de memorie.

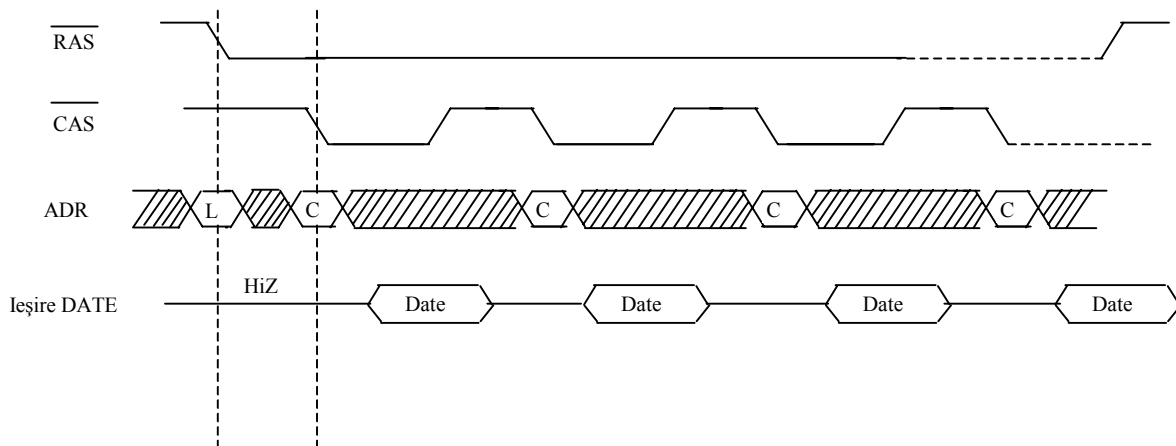


Figura 6.11. Cicluri de acces la DRAM în mod pagină

De exemplu, pentru un DRAM 4 Mbit (cu matrice 1024 linii x 4096 coloane; 1M x 4) o pagină are 1024 locații (a câte 4 biți).

La citire, sau scriere, \overline{RAS} rămâne activat, iar \overline{CAS} se activează de mai multe ori, pentru a valida adrese din aceeași pagină. Toate tranzistoarele de acces ale liniei selectate (prin adresa de linie unică paginii) de către decodificatorul de linii, rămân deschise. Toate datele citite de la liniile de bit sunt menținute într-o stare stabilă de către amplificatoarele senzor. Decodificatorul de coloane va decodifica noua adresă de coloană și va deschide porțile de transfer corespunzătoare. Se observă că în modul pagină, timpul necesar pentru încărcarea adresei de linie și timpul de transfer și decodificare a adresei de linie nu se mai adaugă la timpul de acces, începând cu cel de-al doilea acces pe aceeași linie. După inițializarea adresei de linie (pagină) se transferă și se decodifică doar adresa de coloană. Astfel că, în comparație cu modul standard de acces, timpul de acces este redus cu aproximativ 50% iar timpul de ciclu cu aproximativ 70%.

Durata activă a lui \overline{RAS} nu poate fi însă foarte mare din motive de stabilitate. Tipic, se pot executa aproximativ 200 de accesări succesive într-o pagină înaintea necesității ca dispozitivul de control a memoriei să dezactiveze semnalul \overline{RAS} pentru un interval de un ciclu.

Procedurile de scriere și citire pot fi mixate în mod pagină, fără a fi nevoie să se părăsească acest mod când se comută de la scriere la citire sau invers.

Mod coloană statică ("static column mode")

Este asemănător modului pagină (vezi figura 6.12). La modul coloană statică semnalul \overline{CAS} nu mai comută pentru fiecare adresă nouă de coloană, ci rămâne stabil la nivel jos. După un scurt timp de răspuns circuitul controller DRAM intern recunoaște că adresa de coloană furnizată s-a modificat. Spre deosebire de modul pagină se economisește timpul suplimentar pentru comutarea \overline{CAS} și timpul de răspuns, deci acest mod este mai rapid decât anteriorul.

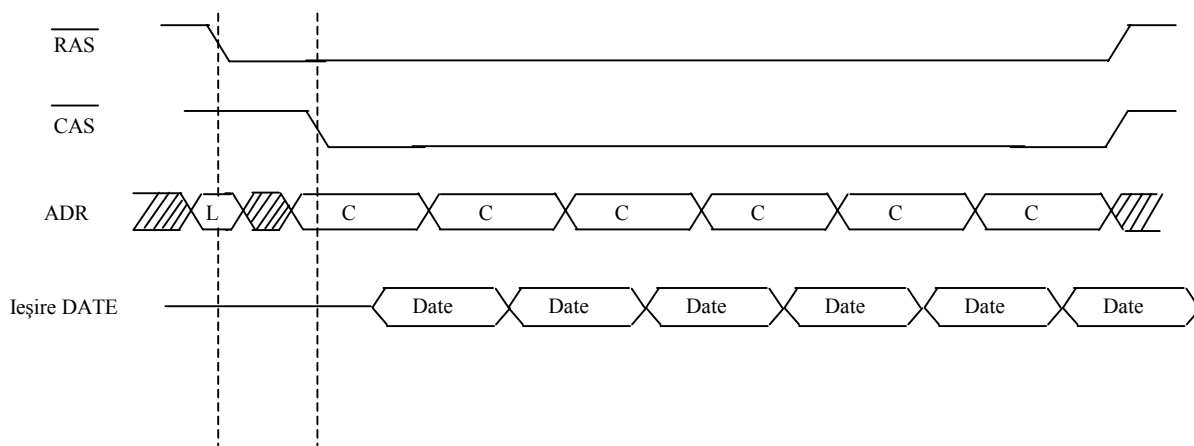


Figura 6.12. Cicluri de acces la DRAM în mod coloană statică

Nici la acest mod de lucru nu este permis ca \overline{RAS} și \overline{CAS} să rămână în stare activă (nivel jos) pentru o perioadă foarte lungă de timp. Prin logică internă pe chip se face deschiderea (către tamponul I/O) doar a porților ce corespund noii adrese de coloană.

Mod tetradă ("nibble mode, rafale mici, serial4)

Este un mod rafală simplu, prin care 4 biți de date dintr-o linie și de la patru adrese succesive de coloană, sunt transmiși secvențial spre exterior (un nibble este echivalent cu jumătate de octet) prin comutarea lui \overline{CAS} care e repetat de 4 ori. Prima dată accesată este determinată de adresa de coloană furnizată circuitului, iar următoarele trei sunt de la adresele succesive de coloană.

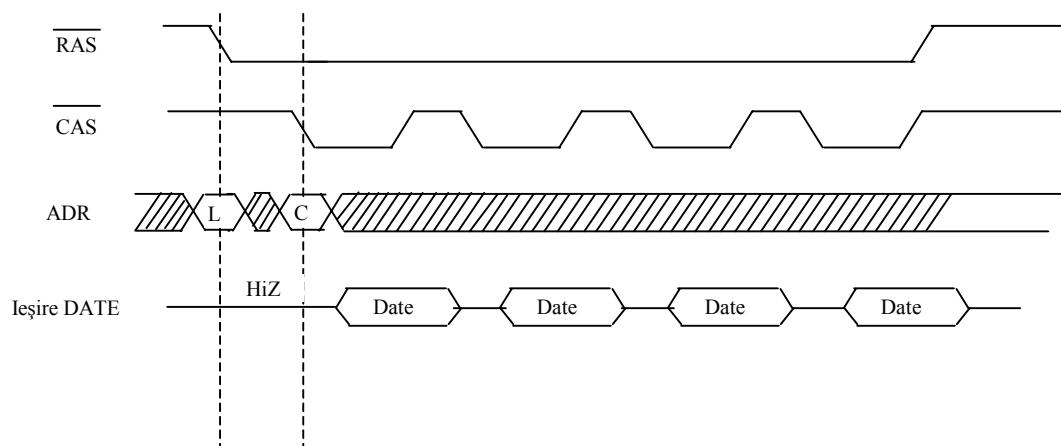


Figura 6.13. Cicluri de acces la DRAM în mod nibble

În mod nibble DRAM are intern (pe chip) un tampon intermediar de 4 biți, care recepționează simultan cei 4 biți succesivi de date și îi scrie apoi unul după altul la tamponul de ieșire, sincronizat cu semnalul de \overline{CAS} . Operația de transfer se efectuează foarte rapid, pentru că toți cei 4 biți adresați (unul explicit și 3 implicit) au fost deja transferați către tamponul (buffer-ul) intermediar.

Modul rafală - burst (sau mod serial)

Poate fi privit ca o extindere (o versiune extinsă) a modului nibble. Și aici, biții de pe o linie adresată sunt furnizați sincronizat cu \overline{CAS} . Diferența față de modul nibble este că numărul de comutări a lui \overline{CAS} nu mai este limitat la 4. În principiu, se poate accesa pe chip o întreagă linie de date succesive - în rafală. În acest caz organizarea internă a chip-ului joacă un rol foarte important, pentru că la un chip de 4 Mbit linia poate avea de exemplu 1024, 2048 sau 4096 coloane.

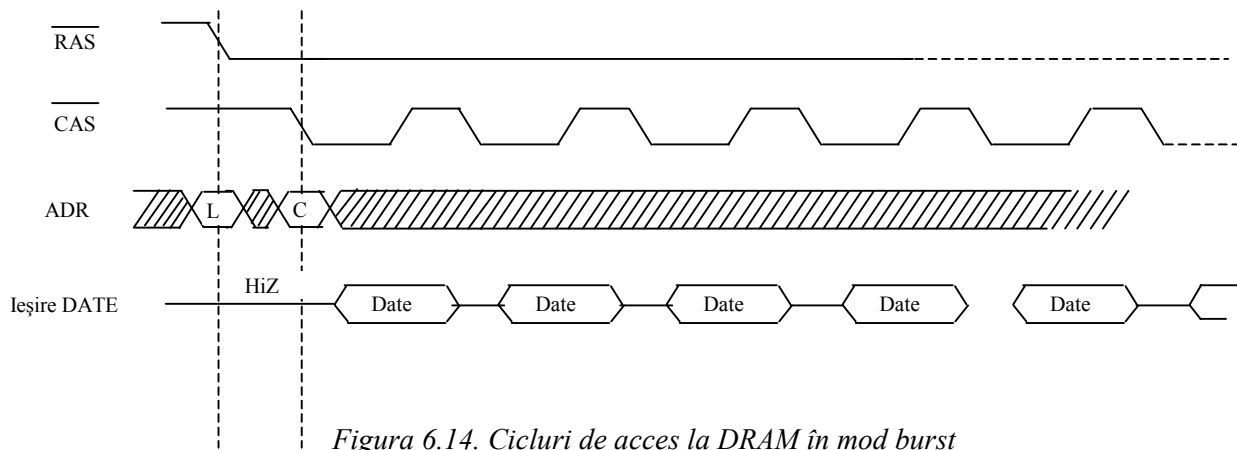


Figura 6.14. Cicluri de acces la DRAM în mod burst

Adresele de linie și coloană furnizate identifică doar adresa de start a accesului. Circuitele interne DRAM incrementează adresa de coloană independent. Modul rafală (serial) este avantajos pentru citirea memoriei video sau pentru umplerea unei linii de cache, pentru că accesările de citire pentru un controller CRT sau pentru un controller de cache sunt în mod natural seriale pe număr mare de biți.

6.4.3. Moduri de reîmprospătare

Nu există capacitor perfect, nici chiar în chip-ul DRAM. Descărcarea se face în timp, prin tranzistorul de acces sau prin dielectricul condensatorului. În funcție de tipul constructiv dispozitivele DRAM obișnuite trebuie să fie reîmprospătate la fiecare 4-16 ms. Formele de reîmprospătare folosite sunt:

1. reîmprospătare numai cu RAS ("ras only refresh")
2. reîmprospătare cu CAS înaintea lui RAS ("CAS-before-RAS refresh")
3. reîmprospătare ascunsă ("hidden refresh")

Reîmprospătare numai cu RAS

Este modul clasic de reîmprospătare. Pentru efectuarea operației se execută un ciclu fictiv de citire, iar odată cu semnalul \overline{RAS} se furnizează adresa de linie de reîmprospătare. \overline{CAS} rămâne inactiv așa cum se indică în figura 6.15. Dezavantajul acestui mod simplu de reîmprospătare este dat de circuitele externe suplimentare necesare pentru generarea adreselor și semnalelor de reîmprospătare. De exemplu, la PC-urile cu DRAM canalul 0 al chip-ului DMA din PC este activat de contorul 1 al chip-ului timer 8253/8254 pentru a se emite un ciclu fictiv de transfer.

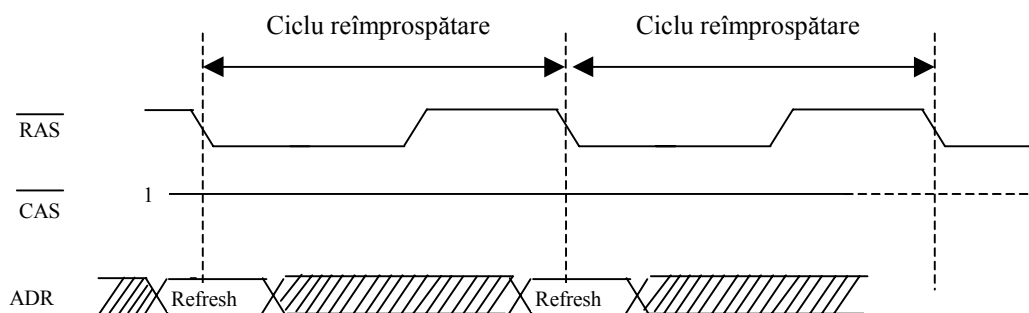


Figura 6.15. Reîmprospătare prin ciclu fictiv de citire (numai cu RAS)

Reîmprospătare cu CAS înainte de RAS

Cele mai multe chip-uri DRAM moderne au acces la unul sau mai multe moduri de reîmprospătare interne. Cea mai importantă este așa numita reîmprospătare cu CAS înainte de RAS (figura 6.16). Pentru asta pe chip-ul DRAM se implementează o logică de reîmprospătare individuală ce are și contor de adrese. În cazul folosirii acestui mod de reîmprospătare, controllerul extern de memorie menține semnalul CAS la nivel jos pentru o perioadă predeterminată de timp înainte ca RAS să treacă la nivel jos. Logica internă de reîmprospătare se activează la sesizarea acestei tranziții și produce o reîmprospătare internă automată. Adresa de reîmprospătare este produsă intern de un contor de adresă din logica de reîmprospătare. După fiecare ciclu CAS-before-RAS, contorul intern de adresa este incrementat. În acest mod de reîmprospătare declanșarea procesului de reîmprospătare se face de către controllerul extern de memorie, dar reîmprospătarea se produce intern în chip-ul DRAM.

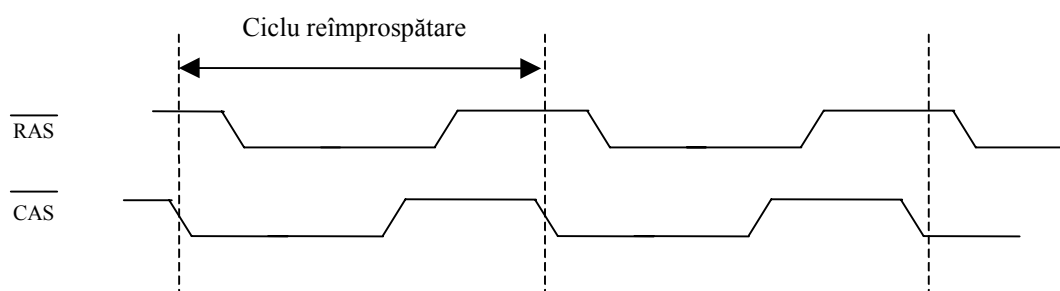


Figura 6.16. Reîmprospătare prin activarea CAS înainte de RAS

Reîmprospătare ascunsă

Este de asemenea o metoda elegantă, care economisește timpul de lucru cu DRAM (figura 6.17). Aici ciclul de reîmprospătare se produce ascuns în spatele unui ciclu de acces de citire normal. Când se face această reîmprospătare, semnalul \overline{CAS} e menținut jos, iar \overline{RAS} este comutat. Datele citite în timpul ciclului vor fi prezente în continuare la ieșirea circuitului, chiar și în timpul reîmprospătării. Ca durată, un ciclu de reîmprospătare este mai scurt decât un ciclu de citire și de aceea acest mod de reîmprospătare economisește timpul. Și aici există un contor de adrese de reîmprospătare intern în DRAM. Dacă semnalul \overline{CAS} este menținut jos un timp suficient de lung, pot fi efectuate mai multe cicluri de reîmprospătare, unul după altul, prin comutarea lui \overline{RAS} .

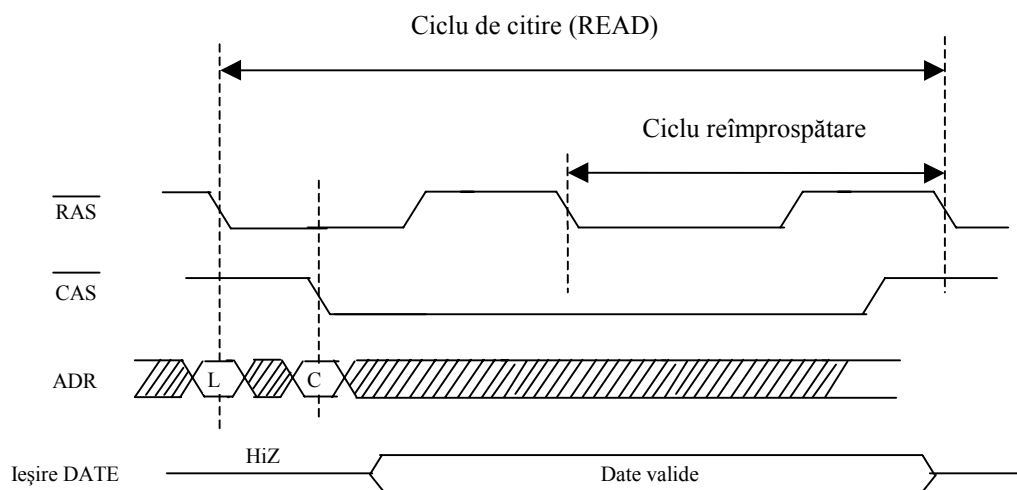


Figura 6.17. Reîmprospătare ascunsă în cadrul unui ciclu de citire

6.4.4. Exemple de circuite DRAM cu interfață asincronă

Primele apărute au fost circuitele care lucrau în mod pagină (*FPM* - "fast page mode"). Funcționează după modul pagină descris mai sus. Au în plus față de circuitele DRAM convenționale o intrare de validare a tamponelor de ieșire (\overline{OE}). La un ciclu de citire, tamponele de ieșire a datelor se blochează (trec în HiZ) după frontul crescător al \overline{CAS} , chiar dacă \overline{OE} rămâne pe nivel jos.

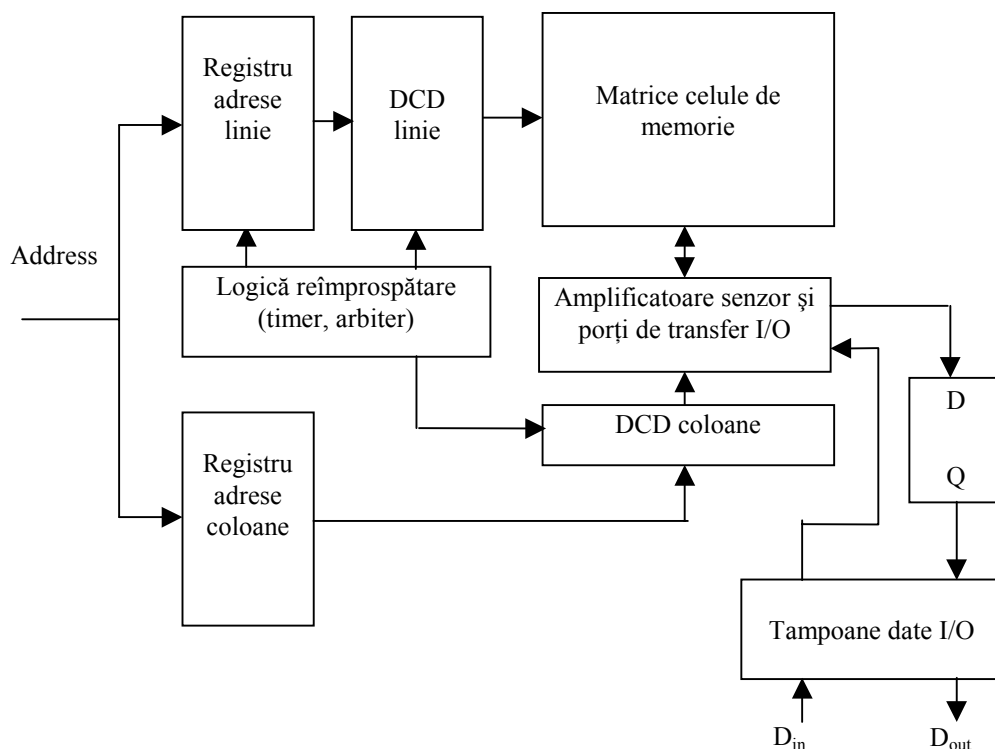


Figura 6.18. Diagramă bloc Extended Data Out (EDO). Se adaugă un latch de memorare la ieșire care permite comutarea mai rapidă decât la FPM a CAS

La circuitele numite *EDO* (Extended data-out DRAM - circuite DRAM cu extindere pe ieșirea datelor) la structura similară cu FPM s-a adăugat un latch la ieșirea amplificatoarelor senzor (figura 6.18). Acest latch permite ca semnalul \overline{CAS} să treacă pe nivel “sus” în timp ce se așteaptă ca datele valide să treacă la ieșire; controlul ieșirii datelor făcându-se de \overline{OE} . EDO pot să lucreze la viteze mari, viteza maximă fiind determinată de timpul de acces față de adresă (tipic 30 nsec). De aceea transferurile în rafală se pot produce cu 30 % mai rapid decât la FPM.

Burst EDO (*BEDO*, EDO în rafală) a fost creat prin înlocuirea latch-ului de ieșire cu un registru (adică un etaj suplimentar de latch-uri) și adăugând un latch de adresă și un contor de 2 biți. Din cauza registrului de ieșire datele nu ajung la ieșiri după primul ciclu CAS. Totuși, etajul acestei structuri de tip pipeline (conductă) interne permite datelor să apară într-un timp mai scurt față de frontul CAS de activare din al doilea ciclu (adică t_{CAC} mai scurt). Valoarea t_{PC} pentru un BEDO de 52 ns este de 15 ns. BEDO sunt DRAM-uri cu acces-rafală în care toate ciclurile de citire sau scriere se produc în rafale de 4. Pinul CAS produce incrementarea contorului intern de rafală.

Pe măsură ce viteza procesoarelor și a magistralelor de memorie a continuat să crească, viteza DRAM standard a devenit tot mai inadecvată. Pentru a crește performanțele generale ale sistemului funcționarea DRAM a fost sincronizată cu clock-ul sistemului, apărând ceea ce se numește *SDRAM* (synchronous DRAM).

6.4.5. Circuite DRAM sincrone (SDRAM)

Spre deosebire de DRAM la SDRAM există în plus: semnal de intrare de ceas (clock), semnal de validare a ceasului, controlul prin RAS și CAS pe *tranziția* semnalului de ceas și operație simultană cu două bănci de memorie adresate întrețesut (prin biții cei mai puțini semnificativi ai adresei). Pentru a ușura interfața între sistemul de calcul și SDRAM toate comenzile SDRAM sunt referite față de semnalul de ceas al sistemului.

Performanțele de viteză obținute cu SDRAM sunt net superioare, față de vechile DRAM care aveau o interfață asinconă. Una din cauze este că SDRAM poate face transfer în mod rafală de la o serie de adrese (1, 2, 4, 8, sau 256) pe baza furnizării unei adrese de început. Lungimea rafalei poate fi programată de către sistem. Spre deosebire de DRAM-urile convenționale, care cer ca fiecare acces să fie inițiat printr-o adresă de coloană trimisă chip-ului, la SDRAM se pot accesa intern până la 256 de adrese succesive, înainte ca o altă adresă de coloană să fie trimisă circuitului. De asemenea SDRAM folosește o magistrală internă de date mai largă pentru a obține viteze mai mari de transfer.

La SDRAM întârzierea dintre validarea coloanei și obținerea datelor este pre-programată de procesor și este numită *întârziere la citire* (“*read latency*”). Întârzierea la citire poate fi programată ca să fie de la 1 la 3 impulsuri de ceas. Această întârziere depinde, la un anumit sistem, de frecvența de ceas a sistemului. Rezumând, principalele diferențe funcționale dintre SDRAM și DRAM sunt:

DRAM

- Nu au intrare de ceas (sistem)
- Control RAS pe nivel
- operare cu o singură bancă de memorie
- un transfer pentru fiecare adresă de coloană (sau impuls de selecție coloană) furnizată
- Întârzierea la citire (read latency) este neprogramabilă

SDRAM

- lucrează cu frecvența ceasului de sistem
- control RAS prin impuls (tranziție)
- două bănci de memorie pe chip cu întrețeserea adreselor
- transfer rafală (burst) cu 1, 2, 4, 8, sau 256 transferuri pentru o singură adresă de coloană furnizată
- Întârzierea la citire este programabilă

6.5. MEMORIA CACHE

Performanțele sistemelor moderne de calcul pot fi serios afectate de viteza de lucru a dispozitivelor ce compun memoria principală, viteză în general mai mică decât viteza de lucru a procesoarelor. La ce bun un procesor ce funcționează cu o frecvență de ceas mai mare de 1GHz, dacă memoria principală este lentă și se introduc multe stări de așteptare. Circuitele integrate de memorie dinamică tradiționale (care constituie și nucleul memoriilor sincrone ce pot fi citite la frecvențe de ordinul sutelor de MHz) au timpi de acces standard de aproximativ 60 ns, dar timpii medii de acces, prin diferite organizări și tehnici de sincronizare pot, în prezent, să ajungă sub 10 ns. Se consideră că un sistem funcționează ineficient, cu multe stări de așteptare pentru procesor, dacă viteza procesorului este mai mare de cel puțin 3 ori față de viteza cu care se transmit datele pe magistrala comună cu memoria; excepție face cazul când memoria cache este substanțial mărită.

Câteva din soluțiile folosite pentru a compensa aceasta incompatibilitate de viteză între procesor și memoria principală (MP) sunt:

- introducerea unei memorii intermediare între procesor și memoria principală, numită memorie tampon (cache) care lucrează cu mare viteză și are dimensiuni medii / mici;
- suprapunerea operațiilor de adresare la memoria principală cu alte activități efectuate de procesor, prin realizarea unei unități de interfață cu magistrala care face o aducere (fetch) prealabilă a instrucțiunilor din memoria principală și le stochează într-o coadă locală a UCP;
- organizarea memoriei principale sub forma de blocuri, cu porturi distincte de acces și cu adrese întreșesute, ceea ce va asigura o viteză medie de transfer mai mare decât inversul timpului de acces la un bloc.

Soluțiile menționate mai sus nu sunt complet independente. Crucial pentru performanțele sistemului sunt creșterea lărgimii de bandă a memoriei și scăderea latenței (întârzierii răspunsului în timp) accesului la memorie.

Memoriile cache (tampon) sunt folosite pentru stocarea unor fragmente de program, date și rezultate intermediare, frecvent folosite de către procesor. Memoria cache este amplasată între procesor și memoria principală.

Utilizarea unei memorii cache, de mare viteză, în care se pot stoca atât instrucțiuni, cât și date, frecvent folosite, are ca scop principal obținerea unei valori mici pentru timpul mediu de acces al UCP la memorie. Având un timp mic de acces, memoria cache permite ca o operație de citire /scriere să se efectueze într-o singură perioadă de tact, ceea ce face ca procesorul să nu între în stări de așteptare. Prezența memoriei cache va fi vizibilă pentru utilizator numai prin sporirea vitezei de lucru la execuția programelor.

Transferul unor zone de cod sau date între memoria principală și memoria cache se realizează automat, prin hardware, fără intervenția utilizatorului. Circuitele de control ale memoriei cache pot fi construite pe același chip ca și UCP sau pot fi externe UCP. Există și variante mixte, cu două niveluri de memorie cache, un nivel (L1) intern UCP și celălalt (L2) extern, sau chiar ambele niveluri integrate în aceeași capsulă cu UCP. Memoriile cache interne au dimensiuni de ordinul sutelor de KBytes, iar memoriile cache externe au valori de ordinul MBytes. Indiferent de varianta de procesor, dimensiunea memoriei cache se încadrează ca și capacitate între a zecea și a mia parte din capacitatea memoriei principale.

Ideea de la care s-a plecat în introducerea memoriei cache este bazată pe *proprietatea programelor de a folosi referințe localizate*. Proprietatea de localizare a referințelor se bazează pe structura secvențială a programelor, cu numeroase cicluri, apeluri de subrutine etc. Astfel încât, pe baza acestei proprietăți, pentru intervale scurte de timp, adresele generate de un program tipic se referă în mod repetat la câteva zone restrânse ca dimensiune de memorie, în timp ce restul memoriei este accesat relativ rar. Dacă porțiunile active ale programelor și datelor sunt plasate într-o memorie

rapidă, timpul mediu de acces la memorie poate fi redus, asta reducând timpul de execuție al programelor. Conform proprietății referințelor localizate dacă un întreg bloc de instrucțiuni este copiat din memoria principală în cache, există o mare probabilitate ca un timp toate accesările să se facă la cache.

Atunci când UCP lansează o adresă pentru a face acces la memoria principală se examinează mai întâi conținutul memoriei cache. Dacă cuvântul este găsit, este citit din cache, iar dacă nu este găsit, se accesează memoria principală pentru citirea cuvântului. Deoarece accesul la cache se efectuează foarte rapid, de obicei într-o perioadă de tact, procesorul va controla direct aceste operații, în timp ce accesul la memoria principală, mai lentă, se va efectua sub controlul unității de comandă a memoriei cache. Transferul între memoria principală și cache nu se face însă la nivel de cuvânt ci la nivel de bloc de cuvinte, bloc care cuprinde și cuvântul adresat de UCP și negăsit în cache. Dacă un cuvânt căutat se găsește în cache se spune ca s-a produs un *acces reușit la cache* (sau pe scurt o *reușită*, "*cache hit*" în limba engleză). Dacă cuvântul căutat nu este în cache s-a produs o *ratăre* ("*cache miss*"). Performanța memoriei cache este adesea măsurată cantitativ prin așa-numitul *raport de reușită* ("*hit ratio*", notat în continuare cu *hr*). Valoarea acestuia se calculează ca raport între numărul total de reușite și numărul total de accesări la memorie (ratări plus reușite). Acest raport se măsoară de obicei prin rularea unor programe de test reprezentative pentru clase de programe. Valoarea lui *hr* este subunitară dar mai mare decât 0,9. Timpul mediu de acces la o memorie se îmbunătățește considerabil dacă se utilizează memorie cache. De exemplu dacă timpul de acces la memoria cache este de 10 ns, iar la memoria principală de 100 ns, iar *hr* = 0,9 rezultă un timp de acces mediu de 19 ns. (din 10 accesări nouă sunt reușite).

$$N = N_x \times N_y$$

Un algoritm special de anticipare aduce în memoria cache secvențele de program care urmează logic unei instrucțiuni deja executate. Procesorul execută programele fără să simtă practic că memoria de lucru este mai lentă. Factorul de succes al memoriei cache este legat de dimensiunea sa și de eficiența algoritmului de înlocuire a informației din cache.

Din punctul de vedere al structurii și organizării memoriei cache există trei tehnici principale (tehnici de mapare):

1. mapare asociativă
2. mapare directă
3. mapare asociativă pe seturi.

Mapare Asociativă

Organizarea memoriei cache ca o memorie asociativă (adresabilă prin conținut) conduce la viteza și flexibilitatea cea mai bună. Dar pentru că o memorie asociativă este scumpă și are un grad mic de integrare, metoda este destul de rar utilizată și doar pentru implementarea unor memorii de mici dimensiuni.

În cadrul memoriei asociative ce funcționează ca memorie cache se stochează atât adresele cât și conținutul (datele) unor locații din memoria principală (figura 6.19). Asta face ca orice locație din cache să poată memora oricare cuvânt din memoria principală. Dacă se produce o ratăre, se face acces la memoria principală pentru aducerea unui bloc de date ce poate fi stocat oriunde în memoria cache asociativă. Dacă memoria cache este plină, trebuie ales un mod prin care să se elimine informația ce nu mai este necesară din cache. Pentru decizie se folosesc *algoritmi de înlocuire*. O procedură simplă este de tip FIFO (algoritm pe bază de "vârsta în cache" = primul intrat primul ieșit). Alți algoritmi de înlocuire folosiți sunt LRU (Least Recently Used), înlocuire aleatoare, LFU (Least Frequently Used).

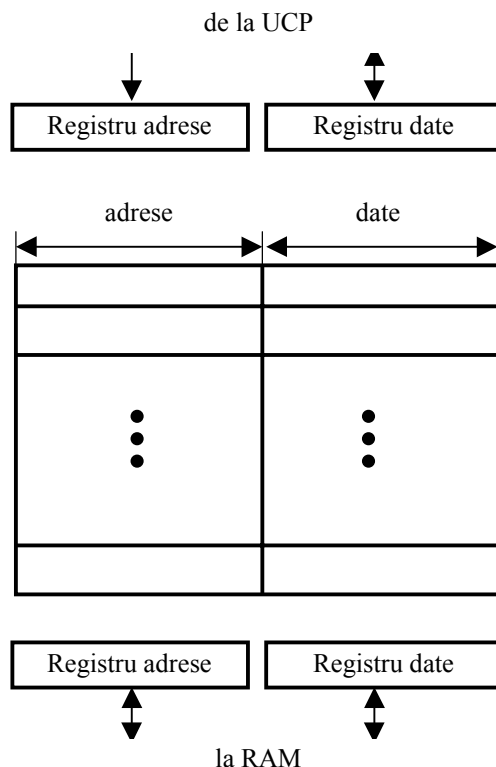


Figura 6.19. Structură de memorie cache asociativă

La lansarea unei adrese de către UCP aceasta se compară cu toate adresele memorate în matricea de adrese a memoriei cache asociative. Dacă compararea combinațională conduce la o reușită data corespunzătoare apare la ieșire și se înscrie în registrul de date (pentru operația de citire).

Mapare directă

Memoriile asociative sunt scumpe și de aceea s-au căutat metode de organizare cu ajutorul unor memorii RAM statice. Exemplificarea acestei mapări este prezentată în figura 6.20. Adresa de a biți (în exemplul din figură $a = 32$) este logic împărțită în două câmpuri, numite *index* (cel mai puțin semnificativ, care din punct de vedere logic poate fi privit ca fiind format din alte două câmpuri: *bloc și cuvânt*) și respectiv *etichetă*. Indexul are un număr k de biți, număr determinat de dimensiunea memoriei cache (2^k cuvinte adresabile prin index), valoarea sa constituind adresa pentru memoria cache. În cache se stochează nu numai data corespunzătoare ci și eticheta (din formată din $a-k$ biți) asociată datei. De asemenea, lângă etichetă mai pot fi înscrise și alte informații cum ar fi:

- bit ce indică dacă datele sunt valide sau locația e privită ca fiind goală,
- bit de protecție la scriere a locației etc. Protecția la scriere este utilă atunci când datele se referă la rutine ale sistemului de operare.

La citire câmpul etichetă al adresei de memorie de la UCP este comparat cu eticheta cuvântului găsit în cache și având aceeași adresă de index cu adresa de memorie. Dacă se potrivesc, s-a produs un eveniment de reușită. Pot exista 2^{a-k} cuvinte cu același index deci stocabile în aceeași linie de memorie cache. Dacă se accesează repetat două sau mai multe adrese cu același index (distanța între două asemenea adrese succesive este 2^k) factorul de reușită scade mult.

Dar atunci când se produce o rată nu se aduce un singur cuvânt ci un întreg bloc de

cuvinte, cu aceeași adresă de bloc. Blocurile sunt alese cu dimensiunea putere a lui doi. Mai mult, în cache se stochează o singură dată eticheta unui bloc, pentru că ea este aceeași pentru toate cuvintele blocului. În figura 6.20 s-a presupus că se folosește o adresă de memorie de 32 de biți, dintre care 8 biți sunt folosiți pentru etichetă, 20 de biți pentru adresarea blocurilor de memorie și 4 biți pentru adresarea cuvintelor în blocuri (deci 16 cuvinte pe bloc de cache).

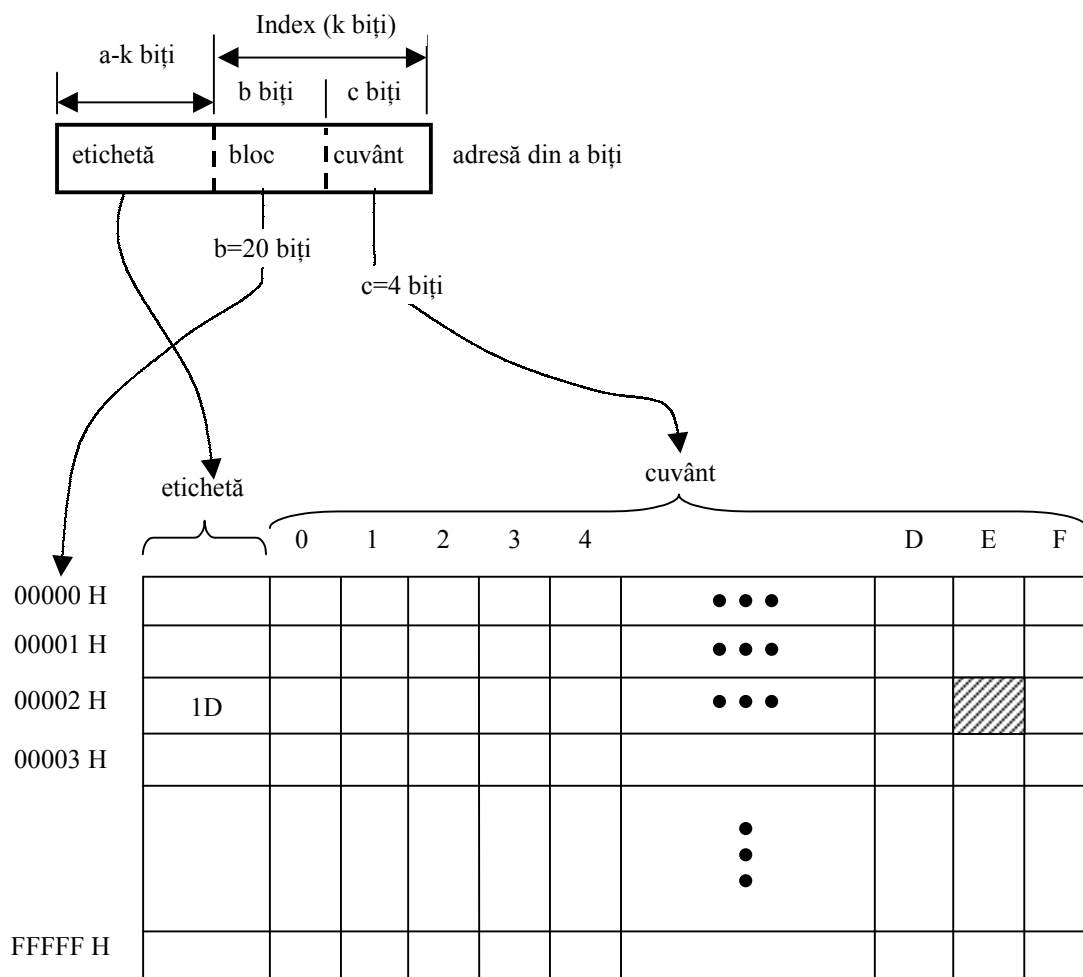


Figura 6.20. Exemplu de mapare directă la memoria cache. S-a hașurat cuvântul găsit în cache la adresa de memorie: 1D00002E hex

Maparea asociativă pe seturi:

Înlătura dezavantajul care face ca la maparea directă două cuvinte cu același index (dar etichetă diferită) nu pot să fie simultan în cache. Ca structură, maparea asociativă pe două seturi ar putea considera două structuri asemănătoare cu cea de la maparea directă, pentru fiecare adresă de bloc putându-se stoca în stivă două blocuri cu etichete diferite (figura 6.21).

Altă deosebire față de maparea directă este că atunci când se lansează o adresă, compararea câmpului etichetă al adresei cu etichetele blocurilor stocate în cache se face prin metode combinatoriale (deci prin *asociere* cu conținutul câmpurilor de etichetă), de unde și denumirea de mapare asociativă. De obicei partea de etichetă și comparare combinatorială se păstrează în circuitul controller de cache.

Se pot organiza memorii cache asociative cu două sau mai multe seturi. Ca urmare pentru fiecare adresă de bloc se pot stoca două sau mai multe blocuri, cu etichete diferite.

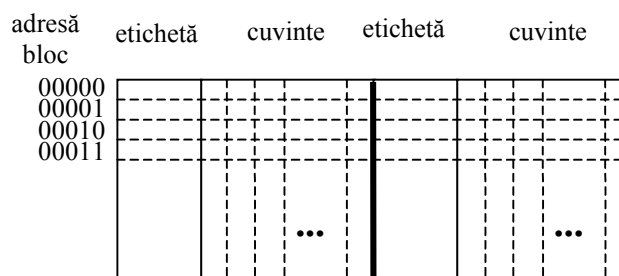


Figura 6.21. Structură bloc simplificată pentru mapare asociativă pe seturi.

Pentru scrierea în memoria cache se folosesc două strategii principale, pe care le-am amintit și la organizarea pe niveluri ierarhice a memoriei:

- scriere cu propagare către nivelurile inferioare de memorie ("write-through"), notată pe scurt WT
- scriere prin actualizarea nivelurilor inferioare în momentul înlocuirii informației din cache ("write-back"), notată pe scurt WB.

La pornire, pentru scrierea în cache, se forțează ratări pentru toate liniile memoriei cache, până când memoria cache se umple cu date valide.

6.6. Tehnici de adresare și alocare a memoriei

De regulă la microprocesoarele de 8 biți, adresa generată și depusă pe magistrala externă de adrese este folosită direct în adresarea unei locații de memorie sau a unui dispozitiv de I/O. Începând cu microprocesoarele de 16 biți apar noțiunile de adresă logică și adresă fizică. Adresele generate de un program sunt considerate *adrese logice* și totalitatea acestora formează *spațiul adreselor logice*. Totalitatea adreselor ce corespund memoriei (și/sau dispozitivelor de I/O) formează *spațiul adreselor fizice*. Cele 2 spații, al adreselor logice și fizice, pot să fie egale sau inegale. Ca urmare, trebuie să existe un mecanism de conversie a adreselor, de translatăre din adrese logice în adrese fizice. La microprocesoarele Intel din seria x86 mecanismul de translatăre este inclus pe același substrat de siliciu cu microprocesorul. La unele din microprocesoare (ca de exemplu Motorola MC68000) mecanismul de translatăre din adresele logice în cele fizice este preluat de un circuit extern microprocesorului denumit "unitate de administrare a memoriei" (MMU - Memory Management Unit).

În general, tehnicile folosite pentru lucrul cu sistemul de memorie într-un calculator, fie că este vorba de selecție, adresare, alocare de spațiu în memoria principală, folosesc un ansamblu de resurse hardware și software, operațiile executate de acestea fiind incluse în noțiunea generală de *management al memoriei*. Administrarea întregii memorii (de pe toate nivelurile ierarhice) a calculatorului, incluzând și modalitățile de partiționare logică și adresare în spațiul adreselor logice (L) și în spațiul de memorare (M , adrese fizice) se face cu ajutorul unei unități hardware pe care o notăm în continuare MMU (Figura 6.22).

Când un program trebuie rulat (lansat în execuție), el este mai întâi adus din memoria auxiliară în memoria principală (în întregime sau parțial).

Pentru MMU se impun următoarele cerințe de bază:

1. să realizeze translatărea adreselor și să susțină alocarea dinamică a memoriei

Înainte ca un program să fie executat, este necesar ca acestuia să i se aloce un spațiu în memoria principală (spațiu de adrese în memoria fizic existentă în calculator). Dacă această atribuire de spațiu (adrese) se face doar la încărcarea programului în memoria principală, procedeul este

numit *alocare statică* și prezintă dezavantajul că adresele de încărcare sunt fixe. *Alocarea dinamică* a memoriei se caracterizează prin atribuirea spațiului necesar programelor, sau proceselor, în timpul execuției acestora. Adresa la care se încarcă programul în memoria principală nu mai este fixă, ea putând fi modificată, la momente de timp diferite.

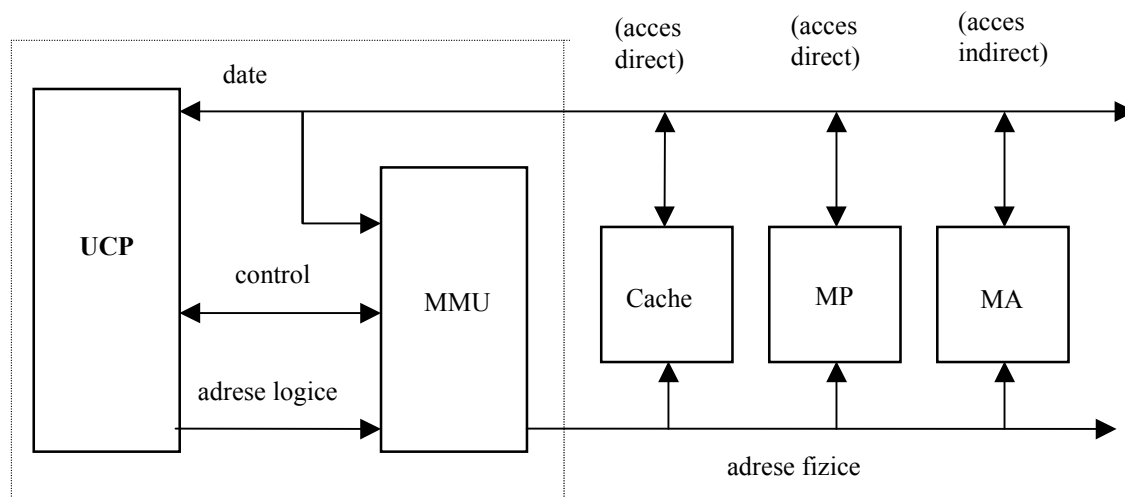


Figura 6.22.: Organizarea ierarhica a memoriei, împreună cu MMU

Se folosesc trei procedee de traducere a adreselor logice în adrese fizice toate bazându-se pe o divizare a spațiului adreselor logice în blocuri de adrese logice continue, după cum urmează:

- *segmente* de lungime arbitrară;
- *pagini* (în spațiul adreselor logice) și *cadre-pagină / blocuri* (în cadrul adreselor fizice), de lungime fixă;
- combinații de adresare *segmentat-paginată*.

Mecanismul de traducere, indiferent că se folosesc pagini sau segmente, se produce în faza de execuție a programului și include printre altele, un *tabel de traducere* prin care se face corespondența între adrese logice și adrese fizice (principiul de lucru al mecanismului este prezentat în figura 6.23).

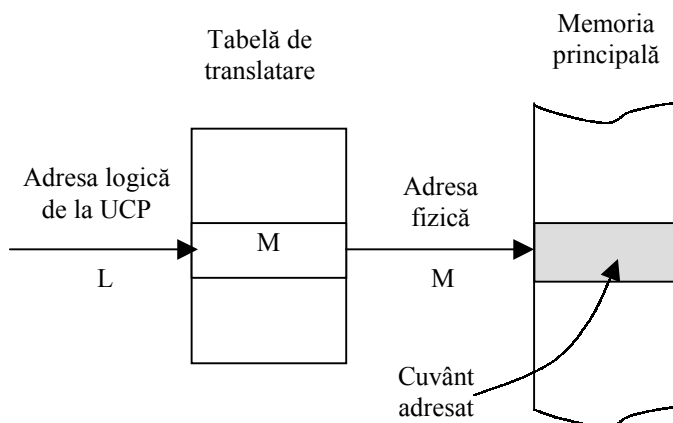


Figura 6.23. Principiul tehnicilor de traducere a adreselor

De la microprocesor se recepționează adresa logică L. Dacă elementul adresat nu se găsește în memoria principală, atunci registrul (de adresă) L va conține adresa din memoria externă unde

se găsește elementul respectiv și el va fi adus de rutine ale sistemului de operare în memoria principală, la o adresă fizică a cărei valoare M se înscrie în registrul de adresă L din tabela de traducere. Dacă elementul adresat este în memoria principală rezultă că adresa fizică este M . Acest mecanism este totuși inefficient, căci tabela de traducere are dimensiunile egale cu ale memoriei principale. Pentru micșorarea dimensiunii tabelului se folosește divizarea spațiului logic în segmente / pagini, iar referirile se fac doar la nivelul de adresă logică de bloc. Trebuie menționat că dacă sunt lansate două procese în execuție, fiecare cu spațiul său de adrese logice, există două tabele de traducere spre memoria principală. Sistemul de operare va reactualiza conținutul tabelului de traducere ori de câte ori va avea loc o relocare (schimbare de poziție în spațiul adreselor fizice) a programelor în memorie.

2. *să susțină mecanismele de implementare ale memoriei virtuale*

Mecanismul memoriei virtuale permite programelor să fie executate chiar dacă numai o parte din instrucțiunile lor se încarcă în memoria principală, iar restul se găsește în memoria auxiliară de tip disc magnetic. Adresele logice sunt numite adrese virtuale, în cazul implementării memoriei virtuale, pentru că nu există fizic în memoria principală ci ele se găsește în memoria auxiliară.

3. *să poată furniza protecție și securitate pentru memorie*

Cerința se referă la protecția informației, la alocarea de drepturi de acces la zone din memoria principală și la zone din spațiul adreselor logice, (protecție la citire, scriere, ștergere, copiere, execuție). Se protejează memoria disponibilă alocată programelor de sistem și programelor utilizator, asigurându-se securitatea informației (acces limitat la informații), prin controlul accesului la resursele sistemului.

Așa cum s-a pomenit și mai sus există două strategii folosite pentru implementarea MMU:

- MMU se află pe chip-ul procesorului; se aplică la microprocesoarele cu spațiul segmentat al adreselor logice. De exemplu: Intel x86, Pentium, Zilog Z800, Z8000)
- MMU este construită separat de UCP, variantă aleasă la microprocesoarele cu spațiu liniar (organizare liniară a) al adreselor logice. De exemplu: MC 680x0, Z8001).

În cazul în care organizarea logică a memoriei este liniară, adresele încep în mod obișnuit de la 0 și avansează liniar. Memoria poate fi apoi structurată prin software, la nivelul translației de adrese. În cazul în care organizarea memoriei este segmentată, programele nu sunt scrise ca secvențe liniare de instrucțiuni și date, ci ca segmente separate de cod, date, sau stivă. Spațiul adreselor logice este spart în mai multe spații cu adresare liniară, fiecare cu lungime diferită, definită de compilator sau de programator. O adresă logică efectivă este calculată ca o combinație între numărul segmentului (care indică adresa de bază a unui bloc de memorie) și un deplasament (offset) în cadrul segmentului.

În general schemele de adresare liniară sunt mai potrivite pentru manipularea structurilor mari de date, în timp ce segmentarea facilitează programarea, permițând programatorului să structureze programele în module segment. În plus adresarea segmentată simplifică protecția și relocarea obiectelor în memorie. Segmentarea facilitează gestiunea memoriei în sistemele de calcul multiuser, multitasking.

6.7. TEHNICI DE TRANSLATARE A ADRESELOR

Indiferent de schema de organizare a memoriei (liniară sau segmentată) procesorul trebuie să aibă un mecanism de traducere a adreselor, util în implementarea memoriei virtuale. Acest mecanism este de asemenea util pentru protejarea informației din memorie. Traducerea de adrese este un proces atribuire și organizare ("mapare") a adreselor logice în adrese fizice de memorie.

Mecanismul de traducere împarte memoria principală în *blocuri (cadre pagină)*. Așa cum am amintit mai sus se folosesc 3 scheme de traducere:

- (1) traducere prin paginare
- (2) traducere prin segmentare
- (3) traducere segmentat-paginată

În *sistemele paginate*, memoria principală este împărțită în blocuri de *lungime fixă* în timp ce în *sistemele segmentate* blocurile sunt de *lungime variabilă*.

Paginile au în general lungimi de ordinul 256 - 4096 cuvinte, în timp ce segmentele cu lungimea definită de compilator sau programator au 64 K-cuvinte, sau mai mult. În sistemele cu multiprogramare și time-sharing⁷, mai mulți utilizatori folosesc aceleași programe cum sunt editoare, compilatoare, programe utilitare, biblioteci de programe etc. Atât sistemele cu paginare cât și cele cu segmentare permit mecanisme de partajare, între procesele utilizator, a paginilor, respectiv a segmentelor. Aceste mecanisme se bazează pe intrări în tabele de mapare de pagină (sau segment) în care intrările diferitelor procese indică către același bloc din memoria principală.

Combinația între segmente și pagini presupune că un segment conține una sau mai multe pagini virtuale. Mecanismul de segmentare administrează spațiul virtual, împărțind programele în segmente, în timp ce paginarea este destinată administrării memoriei fizice care este împărțită în cadre pagină (blocuri).

Vom descrie în continuare mecanismele folosite pentru maparea segmentelor și paginilor.

6.7.1. Maparea adreselor folosind pagini

În cazul paginării, implementarea tabelului pentru maparea adreselor este simplă pentru că informația din spațiile de adresare și memorare este divizată în blocuri de dimensiune fixă. Memoria fizică este împărțită logic în blocuri de aceeași dimensiune (64-4096 cuvinte fiecare). Termenul *pagină* se referă la blocuri, *de aceeași dimensiune*, de adrese din spațiul de adresare. De exemplu, presupunem un calculator care are 20 de biți de adresă și folosește doar 32KB în memoria principală (memorie implementată fizic prin circuite de memorie); dacă o pagină, respectiv un bloc, au dimensiuni egale cu 1 kB, atunci spațiul de adrese e divizat în 1024 pagini, iar spațiul de memorie e divizat în 32 de blocuri.

Se consideră că și programele sunt împărțite în pagini. Porțiuni din programe sunt mutate din memoria auxiliară în memoria principală în înregistrări egale cu mărimea paginii. În loc de bloc de memorie e folosit uneori termenul de cadru pagină ("page frame").

În cazul mapării prin paginare adresa virtuală este reprezentată printr-un singur cuvânt de adresă împărțit într-un câmp corespunzător numărului paginii (adresa paginii) și un câmp pentru deplasament. La maparea prin segmentarea un singur cuvânt de adresă nu mai este suficient; dimensiunea variabilă a segmentelor conduce la existența a două cuvinte de adresare, în care primul indică numărul (adresa) segmentului, iar cel de-al doilea deplasamentul în cadrul segmentului.

Într-un calculator cu 2^p cuvinte pe pagină, p biți sunt folosiți pentru a specifica o adresă de linie iar cei mai semnificativi biți rămași în adresa virtuală specifică numărul de pagină. Fie de exemplu un sistem simplu cu o adresă virtuală cu dimensiunea de 16 biți și pagini cu 4096 cuvinte. Pentru că o pagină are 2^{12} cuvinte, cei patru biți mai semnificativi vor specifica una din cele 16 pagini, iar cei 12 biți mai puțin semnificativi indică adresa liniei în cadrul paginii. Ca urmare maparea trebuie făcută doar de la un număr de pagină la un număr de bloc din memoria principală,

⁷ time-sharing = partajarea timpului UCP

pentru că adresa liniei e aceeași pentru ambele spații. Organizarea de principiu a tabelului de mapare a memoriei într-un sistem paginat poate arăta ca în figura 6.24.

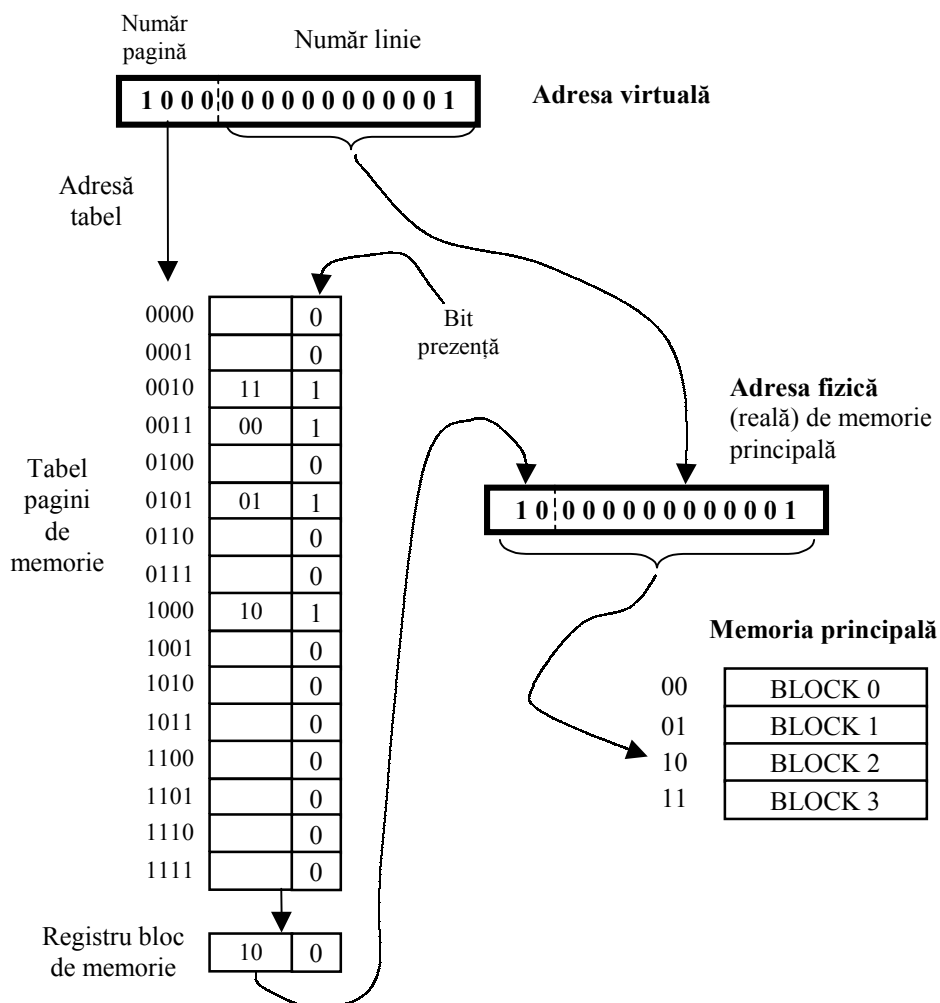


Figura 6.24: Schemă de principiu, explicativă la operația de paginare (modalitatea de traducere din figură nu se aplică din cauza dimensiunii mari a tabelului de traducere a paginilor de memorie)

În tabelul paginii de memorie, adresa conduce la numărul paginii, iar conținutul indică numărul blocului unde pagina este stocată în memoria principală. Se observă că paginile virtuale 2, 3, 5 și 8 se află în memoria principală, în blocurile 3, 0, 1 și respectiv 2. Un bit de prezență adăugat fiecărei locații a tabelului indică dacă pagina respectivă a fost mapată în memoria principală (i s-a alocat spațiu și a fost transferată din memoria auxiliară în memoria principală). Valoarea 0 a bitului de prezență indică că pagina nu este în memoria principală. În exemplul dat, cei 4 biți ai adresei virtuale specifică atât numărul paginii cât și adresa în tabelul paginilor de memorie. Conținutul tabelului este citit în registrul ce indică blocul de memorie. Dacă bitul de prezență este 1, numărul de bloc e transferat în registrul ce păstrează adresa fizică din memoria principală. Dacă bitul de prezență este 0 rezultă că adresa virtuală se referă la un articol ce nu se găsește în memoria principală. În acest caz, se generează o cerere (în forma unei întreruperi software, de tip “eroare de pagină⁸”) către sistemul de operare (SO) pentru a aduce pagina cerută din memoria auxiliară în memoria principală, înainte de reluarea programului care a accesat adresa din pagina respectivă.

⁸ page fault = eroare de pagină

O astfel de organizare în memoria principală a tabelului de traducere este însă inefficientă, căci tabelul are multe locații goale. Dacă presupunem un calculator cu spațiul de adrese de 4 GB (2^{32}), cu pagini de memorie de 32 KB (2^{15}), iar spațiul memoriei fizice este 32 MB (2^{25}) rezultă 128 K-pagini virtuale de memorie și 1024 blocuri (1 K-blocuri). Conform principiului din figura 6.24 tabelul de traducere ar avea 131071 ($2^{17} = 128 \text{ K}$) linii și doar 1024 locații ar fi ocupate (cu bit de prezență 1), restul fiind goale, neutilizate. O soluție a acestei probleme o constituie folosirea unei memorii asociative, ca în figura 6.25, (*tabel asociativ de pagini de memorie*) cu număr de locații egal cu numărul de blocuri din memoria principală.

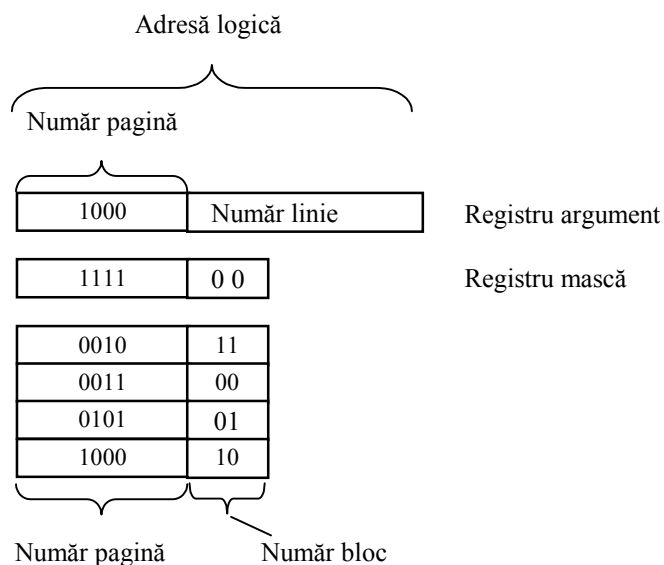


Figura 6.25. Exemplificare a tabelului pentru traducerea adresei de pagină construit cu memorie asociativă (adresabilă prin conținut)

În fiecare locație de memorie se stochează numărul paginii virtuale și numărul corespunzător al blocului de memorie principală alocat. Adresa logică se compară combinațional cu conținutul tabelului asociativ, doar pentru pozițiile binare ce corespund la 1 logic în registrul de mascare, deci în figura 6.25, doar pentru câmpul de adresă al numărului de pagină virtuală. Dacă la apelarea unei pagini virtuale, aceasta nu este mapată în memoria principală, operația de adresare este o ratăre (miss) și evenimentul este rezolvat printr-o rutină de servire a sistemului de operare care aduce pagina respectivă din memoria auxiliară. Și acest tabel asociativ de traducere poate avea dimensiuni mari uneori. De aceea se folosesc tabele asociative de mare viteză la care numărul liniilor este mai mic decât numărul de blocuri (cadre pagină) și în care se salvează numai o parte din mapările realizate. Acest registru asociativ este numit TLB (Translation lookaside buffer) și este descris mai jos.

6.7.2. Mapare prin segmentare

La acest tip de mapare spațiul adreselor logice este împărțit în *segmente*, care spre deosebire de pagini, *nu au o dimensiune fixă*. Segmentele sunt spații de adrese continue (liniare), care includ programe sau porțiuni din programe cu atribute comune. Mecanismul de traducere este prezentat în schema de principiu din figura 6.26.

Spre deosebire de operația de paginare, la segmentare nu se mai face concatenarea (alăturarea) adresei de deplasament (din adresa logică) la adresa fizică. Aici adresa de deplasament este adunată cu adresa fizică a segmentului (adresa sa de bază) citită din tabelul de traducere al segmentelor, specific unui anumit proces.

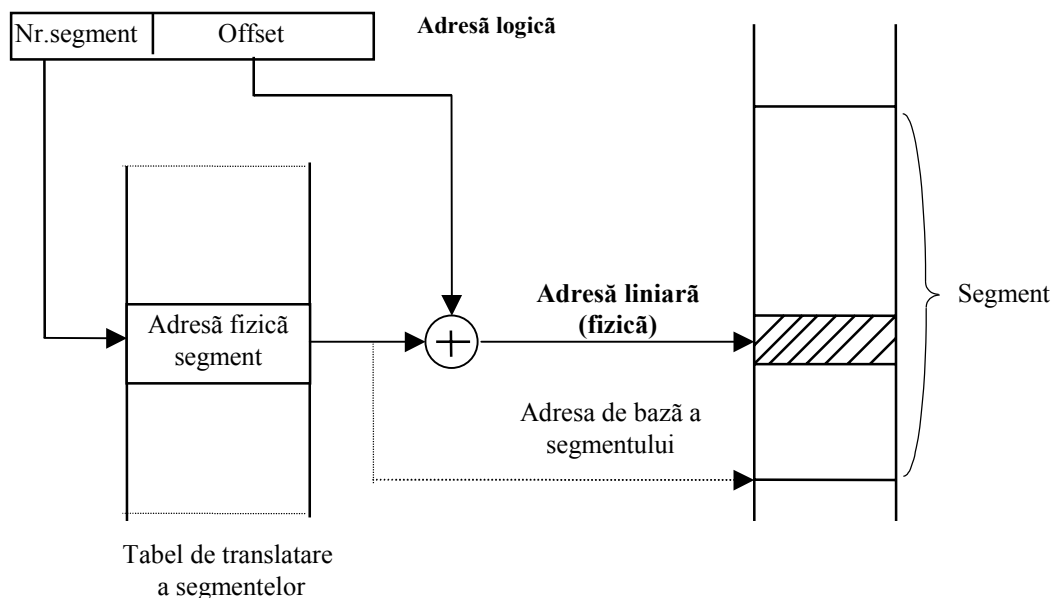


Figura 6.26: Schemă de principiu, explicativă la operația de segmentare

6.7.3. Mapare segmentat-paginată

Maparea combinată paginat-segmentată presupune de obicei că un segment include mai multe pagini. Operația de traducere începe cu segmentarea și apoi continuă cu paginarea. Ca urmare doar traducerea de pagină produce transformarea adresei liniare ce rezultă din segmentare într-o adresă fizică. Ca regulă generală la metoda combinată cu segmentare-paginare, segmentele cu lungime variabilă conțin una sau mai multe pagini cu lungime constantă. Acest mod de împărțire conduce la algoritmi mai simpli pentru administrarea și protecția memoriei. Există și excepții de la această regulă. De exemplu, la procesoarele Intel din seria x86, de la '386 în sus, poate să nu existe o relație directă între pagini și structura logică a unui program (un cadru pagină la 386 are 4KB). La aceste procesoare, segmentele pot avea dimensiunea mai mare sau mai mică decât o pagină. Arhitectura nu impune nici o relație între marginile paginilor și segmentelor. Astfel un segment poate conține sfârșitul unei pagini și începutul alteia și similar pentru relația pagină-segment. Chiar dacă procesorul permite această libertate în relația dintre pagini și segmente, se recomandă ca un segment să conțină una sau mai multe pagini, iar paginile să fie incluse complet într-un segment, deci să nu aparțină simultan la două segmente.

Dacă fiecare segment va conține una sau mai multe pagini de memorie (de lungime fixă), lungimea segmentului se poate specifica atât în multiplu de octeți (Bytes) dar și în multiplu de pagini de memorie. Adresa logică va fi împărțită în trei câmpuri: primul indică numărul segmentului (ca intrare într-un tabel de traducere pentru segmentele procesului), al doilea un număr de pagină, (care indică o adresă relativă într-un tabel de traducere paginată în cadrul segmentului), iar al treilea câmp este deplasamentul. Dacă numărul de biți ai câmpului pagină este p , rezultă maxim 2^p pagini în segment.

Valoarea citită din tabelul segment este un pointer către baza tabelului de pagină. Suma

dintre baza tabelului de pagină și numărul paginii (din adresa logică) este pointer către o intrare în tabelul de pagină, așa cum se indică în figura 6.27.

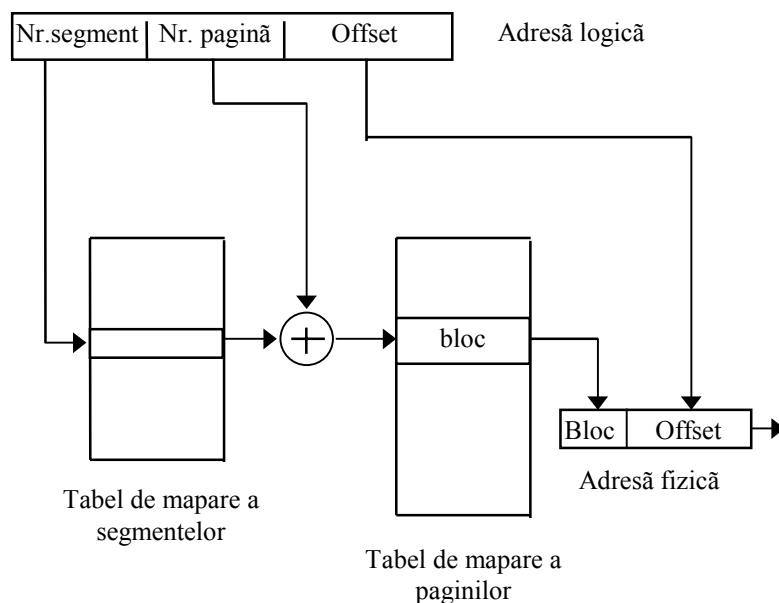


Figura 6.27. Principiul operației de traducere segmentat-paginată

Valoarea găsită în tabelul de pagină furnizează numărul de bloc din memoria fizică. Cele două tabele pot fi construite în memoria principală, dar de asemenea pot fi construite separat de memoria principală, eventual într-o memorie locală procesorului. Ultima variantă este mai avantajoasă ca viteză, pentru că organizarea în memoria principală conduce la trei accesări succesive, deci viteză mică. Se folosește de aceea o memorie asociativă pentru a stoca în tabele intrările cele mai recent referite. Aceasta memorie asociativă este numită "*Translation Lookaside Buffer*" = tabel pentru memorarea funcțiilor de traducere (TLB), și are structura din figura 6.28.

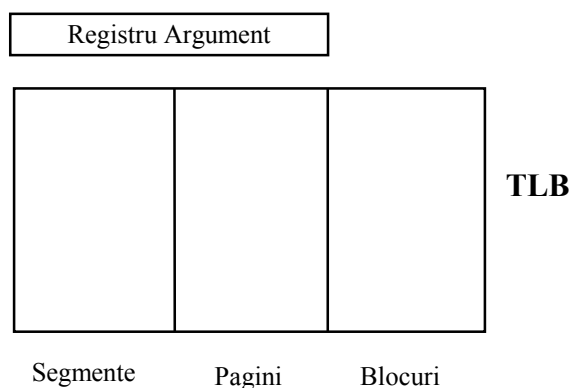


Figura 6.28. Structura de principiu a unui TLB (tablou asociativ pentru memorarea funcțiilor de traducere)

Prima oară când un anumit bloc e referit, valoarea sa împreună cu numerele corespunzătoare de segment și pagină intră în TLB. Dacă, la adresare, adresa cerută nu este mapată în TLB se folosește maparea mai lentă, cu tabele în memorie, descrisă mai sus, iar rezultatul e înscris în memoria asociativă (TLB) pentru referirile ulterioare.

6.7.4. Memoria virtuală

Într-un sistem ierarhic de memorie, programele și datele sunt mai întâi stocate în memoria auxiliară. Porțiuni din programe sau date sunt aduse în memoria principală pe măsură ce ele sunt necesare unității centrale de procesare. Memoria virtuală reprezintă un ansamblu de tehnici (software) și mecanisme (hardware) folosit pentru a mări capacitatea aparentă a memoriei principale RAM. Astfel, numărul de biți ai adresei ce rezultă prin execuția unei instrucțiuni, depășește ca posibilități de adresare, capacitatea memoriei principale, fizic instalată. În aceste condiții, din programul stocat în memoria externă vor fi aduse în memoria principală numai zonele active la un moment dat (figura 6.29). Procesul acesta de transfer între memoria externă (disc magnetic) și memoria principală este transparent pentru utilizator, el desfășurându-se automat, sub controlul sistemului de operare. Adresele logice furnizate de UCP și MMU explorează un spațiu de adresare ce corespunde ca dimensiune, capacității memoriei virtuale, mult mai mare decât dimensiunile memoriei principale.

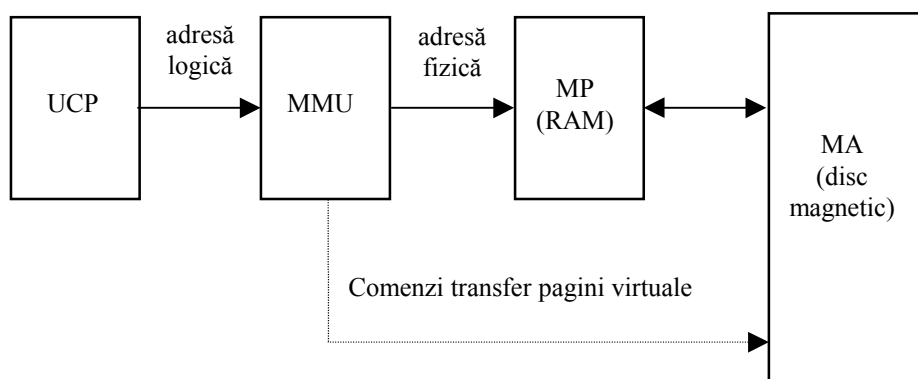


Figura 6.29. Schema bloc de principiu a sistemului de memorie virtuală (MMU = unitate de control a memoriei, MP = memoria principală, MA = memoria auxiliară)

Din punct de vedere *fizic*, memoria virtuală va fi organizată în *memoria auxiliară* (disc magnetic). Se mărește deci capacitatea aparentă a memoriei cu "acces direct". Numărul de adrese accesibile în mod direct aflate la dispoziția programatorului este substanțial mai mare decât numărul de locații din memoria principală fizic existentă; spațiul de adresare corespunde capacității memoriei virtuale. Memoria virtuală este organizată, din punct de vedere *logic*, sub formă de pagini, astfel că o adresă furnizată de o instrucțiune a procesorului conține două câmpuri: numărul paginii și respectiv, adresa în pagină. Dimensiunile unei pagini sunt mult mai mici decât dimensiunea memoriei principale. În acest mod, în memoria principală pot fi aduse, din memoria secundară, mai multe pagini active, ale unui program sau ale mai multor programe. Un sistem cu memorie virtuală furnizează un mecanism pentru traducerea adreselor generate de program în adrese fizice ale locațiilor memoriei principale. Operația de traducere se face dinamic, în timp ce programele se execută în UCP.

Se va considera, pentru exemplificare, organizarea unei memorii virtuale cu capacitatea de 64 MB (2^{26}), structurată pe cuvinte de 32 de biți: 16 Mega-cuvinte x 32 de biți/cuvânt ($2^{24} \times 32$). Într-un cuvânt - instrucțiune de 32 de biți, 24 de biți sunt alocați pentru adresarea memoriei virtuale. Primii 12 biți, mai semnificativi, vor specifica numărul paginii, iar ultimii 12 biți, mai puțin semnificativi, vor constitui adresa în pagină. În figura 6.30. se prezintă organizarea logică a memoriei virtuale în 2^{12} pagini, fiecare pagină având capacitatea de 2^{12} cuvinte de câte 32 de biți. Organizarea fizică a memoriei virtuale în cadrul memoriei auxiliare este prezentată de asemenea în figura 6.30. Programul utilizatorului este constituit dintr-un număr oarecare de pagini, organizate

într-un segment. Prima pagină, din cadrul fiecărui segment, conține, pe lângă un identificator de pagină, un identificator de segment și numărul de pagini din segmentul dat. Celelalte pagini vor fi constituite dintr-un antet de identificare pagină și din pagina propriu-zisă. În cazul de față se consideră că sistemul de operare se ocupă cu gestiunea segmentelor/fișierelor, în memoria externă.

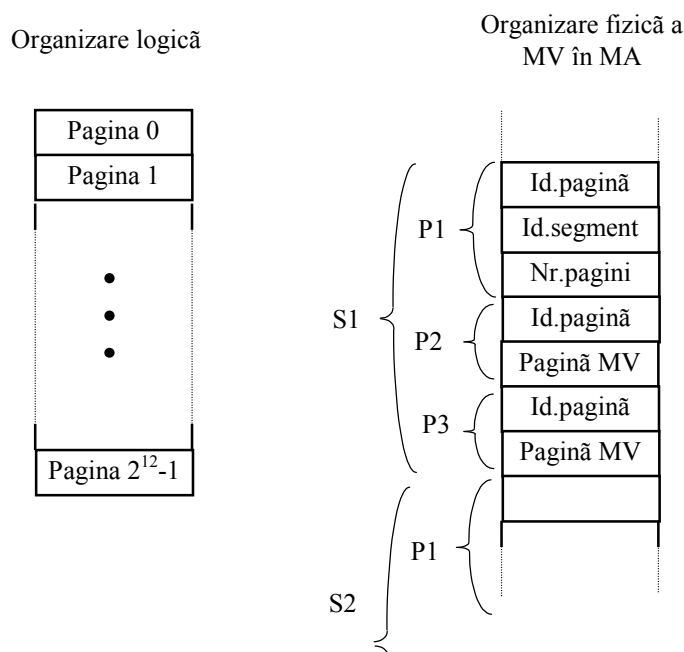


Figura 6.30. Organizarea logică și fizică a memoriei virtuale. Cu P_i s-a notat structura paginii fizice i din memoria virtuală, iar cu S_i s-a notat structura segmentului i . (MV = memorie virtuală, MA = memorie auxiliară)

Fie de exemplu o memorie principală cu capacitatea de $256 \text{ k} \times 32$ de biți. Se pot stoca în cadrul memoriei principale maximum $2^6 = 64$ pagini, având fiecare o capacitate de $2^{12} = 4096$ cuvinte. Nucleul sistemului de operare va fi rezident în memoria principală și va ocupa, de exemplu, primele 8 pagini (32 k-cuvinte). Sistemul de operare va gestiona o serie de tabele de traducere a adreselor logice / virtuale în adrese fizice. Schimbarea paginilor între memoria principală și memoria auxiliară se face după algoritmi de înlocuire asemănători cu cei de la memoria cache. Pentru exemplul dat anterior în figura 6.31 se prezintă modalitatea de traducere a adreselor virtuale, folosind următoarele notații:

- TA - Tablou Asociativ în celulele căruia sunt stocate numerele paginilor virtuale aduse în memoria principală. Conține 64 de cuvinte de câte 12 biți.
- CU - Contoare de Utilizare a paginilor prezente în memoria principală. Se folosesc în mecanismul de înlocuire. La fiecare acces contoarele ce corespund intrărilor neselectate se decrementează, până la zero, iar contorul liniei selectate se setează la valoarea maximă. Dacă este nevoie de spațiu pentru noi intrări vor fi înlocuite liniile care au contoarele cu conținut 0;
- BLOC - Memorie cu conținut permanent (doar cu citire), adresată asociativ, care transformă numărul paginii virtuale în număr de pagină fizică din memoria principală RAM;
- AM - Registrul de adrese pentru memoria principală (adresa fizică), de 18 biți
- RI - Registrul de Instrucțiuni, cu 32 de biți
- PC - Contorul de Program cu lungimea de 24 de biți;
- MP - Memoria principală RAM ($2^{18} \times 32$ biți);

Dacă o pagină oarecare din memoria virtuală a fost adusă în memoria principală, ea poate fi localizată, ca număr de pagină fizică, prin referirea la o memorie asociativă cu conținut fix pe care o notăm BLOC. În cazul coincidenței între numărul paginii virtuale, și conținutul unei celule din Tabloul Asociativ, celula corespunzătoare, situată pe aceeași linie, în BLOC, va furniza primii 6 biți mai semnificativi ai adresei fizice. Aceștia vor fi concatenați în registrul AM, cu ultimii 12 biți din RI sau PC, pentru a forma adresa fizică de acces la memoria principală. Primele opt celule din BLOC au conținuturi fixe, corespunzând numerelor primelor opt pagini din memoria principală în care se află nucleul sistemului de operare. Primele opt celule din TA vor conține numerele paginilor virtuale corespunzătoare nucleului sistemului de operare. Conținutul acestora va fi fix. De asemenea, conținuturile primelor opt contoare de utilizare nu vor suferi decrementări. Gestiunea memoriei virtuale începe prin transferul unui segment sau al unui număr de pagini, dintr-un segment, corespunzător unui program din memoria secundară în memoria principală, sub controlul sistemului de operare, și lansarea în execuție.

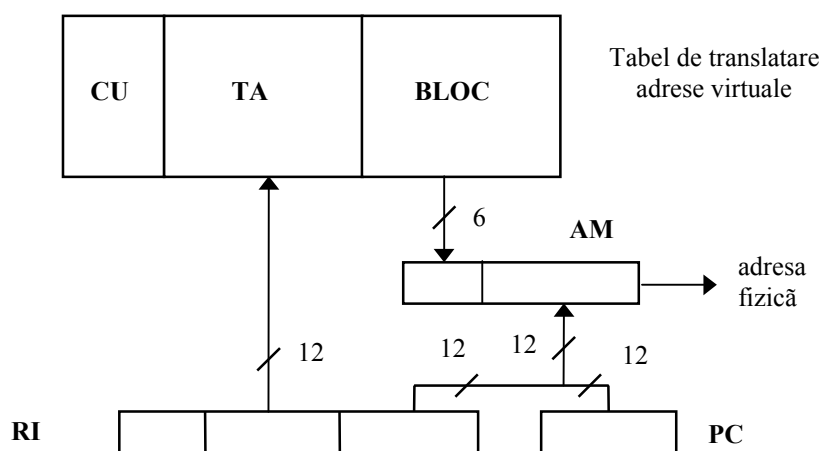


Figura 6.31 Mecanismul asociativ care asigură suportul pentru traducerea adreselor virtuale în adrese fizice.

Dacă se constată lipsa unei pagini, se trece la aducerea paginii virtuale în memoria principală. În figura 6.32, operațiile din blocurile 1 și 4 se execută prin software, în timp ce operațiile din blocurile 2 și 3 se execută prin hardware. Operațiile din cadrul blocului 2 sunt operații obișnuite de citire-interpretare și execuție a instrucțiunilor programului. După generarea întreruperii de tip lipsă-pagină în memoria principală întreaga responsabilitate pentru transferul paginilor revine sistemului de operare. Într-o memorie auxiliară pe disc magnetic organizată cu scopul de a facilita implementarea memoriei virtuale, pagina de memorie auxiliară va înlocui înregistrarea, drept cea mai mică unitate de date adresabilă pe disc. Sistemul de operare va gestiona o serie de tabele de traducere a adreselor logice în adrese fizice.

După terminarea transferului programul întrerupt se poate relua. La transfer, dacă memoria principală este plină, trebuie ștearsă o pagină veche pentru a face loc celei noi. Algoritmul de înlocuire poate fi de tip, de exemplu:

- FIFO - selectează pentru înlocuire paginile care au stat în memorie cel mai lung timp. De fiecare dată când o pagină e încărcată în memorie, numărul sau de identificare este introdus într-o stivă FIFO (primul intrat, primul ieșit). Dacă tamponul FIFO este plin și trebuie adusă altă pagină, se elimină pagina cea mai "veche" în tampon și care e în vârful stivei FIFO. Este un algoritm ușor de implementat dar nu foarte eficient.
- LRU - least recently used (cel mai puțin recent utilizat) este un algoritm mai dificil de implementat, dar mai eficient. Se înlocuiește cea mai puțin recent folosită pagină. Folosește

câte un contor pentru fiecare pagină ce se află în memoria principală. Când se face referire la o pagină, contorul asociat ei este setat la valoarea maximă, iar celelalte contoare asociate paginilor se decrementează. Ele sunt numite adesea registre de vârstă, pentru ca ele indică vârsta cât n-au mai fost referite.

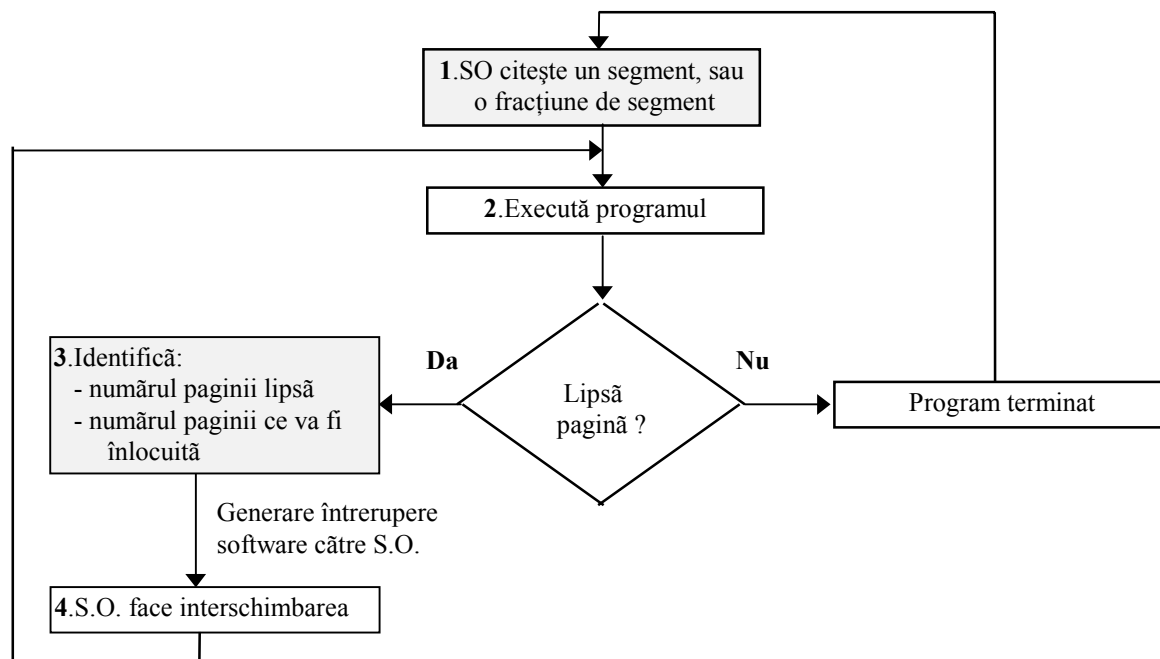


Figura 6.32. Explicativă la mecanismul de încărcare a paginilor virtuale.

6.7.5. Noțiuni privind protecția memoriei

Inventarea multiprogramării, la care un procesor este partajat între mai multe programe aflate în execuție, a condus la noi cerințe privind protecția între programe. Mecanismele de protecție sunt strâns legate de implementarea memoriei virtuale.

Multiprogramarea conduce la conceptul de proces. Un proces este constituit dintr-un program aflat în execuție plus orice stare necesară pentru continuarea rulării sale. Partajarea timpului ("time-sharing") este o variantă a multi-programării care partajează UCP și memoria între diferiți utilizatori simultani, iar mecanismul de partajare dă iluzia că toți utilizatorii au propriile mașini. Astfel că, în oricare dintre variante, trebuie să fie posibil să se comute de la un proces la altul. Această comutare este numită *comutare de proces* sau *comutare de context*.

Un proces trebuie să se execute corect indiferent dacă se execută continuu de la start până la terminare, sau dacă este întrerupt în mod repetat pentru a se face comutarea către alte procese. Responsabilitatea menținerii comportării corecte a proceselor cade atât în sarcina resurselor hardware, care asigură salvarea și restaurarea corectă a proceselor ce rulează pe UCP cât și în sarcina sistemului de operare care trebuie să garanteze că un proces nu interferează cu celelalte și că un proces nu poate modifica datele altor procese. Aceasta înseamnă asigurarea protecției datelor de către sistemul de operare. În afară de protecție, sistemul de operare permite, în anumite condiții, și partajarea codului și a datelor între diferite procese pentru salvare de spațiu de memorie prin reducerea numărului de copii de informații identice.

Cel mai simplu mecanism de protecție este constituit de o pereche de registre care verifică fiecare adresă, astfel încât să nu se permită accesul în afara unor limite alocate în spațiul de adrese.

Aceste registre sunt numite registru - bază și registru limită. O adresă este validă dacă se încadrează între cele două adrese conținute în registre:

$$\text{Bază} \leq \text{Adresă} \leq \text{Limită}$$

La unele sisteme adresa este considerată ca un număr fără semn, care se adună întotdeauna la o adresă de bază, astfel că testul de adresă limită se reduce la:

$$(\text{Bază} + \text{Adresă}) \leq \text{Limită}$$

Dacă proceselor utilizator li s-ar permite să modifice registrele de adrese de bază și limită, atunci acest mecanism de protecție nu ar putea funcționa. Modificarea respectivelor registre este permisă doar sistemului de operare, pentru asigurarea protecției între procese.

Pentru asigurarea protecției sistemul de operare (SO) are trei responsabilități principale:

1. Să furnizeze cel puțin două moduri de execuție indicând că procesul aflat în rulare este un proces utilizator, sau un proces al sistemului de operare. La diferite SO ultimul tip de proces este numit în diferite feluri: *proces kernel* (nucleu), *proces supervisor*, sau *proces executiv*.
2. Să prevadă o porțiune a stării CPU pe care procesul utilizator o poate doar citi, dar nu o poate scrie. Aceasta include registrele bază / limită, indicatori (biți) pentru moduri utilizator / supervisor și indicatori pentru validare / invalidare evenimente de tip excepție.
3. Să prevadă un mecanism prin care UCP poate trece din mod utilizator (user) în mod supervisor și viceversa. Primul sens de trecere este specific apelurilor sistem ("system calls" – apeluri de servicii oferite de SO), implementate ca instrucțiuni speciale care transferă temporar controlul la o locație precisă din spațiul de cod al supervisorului. Registrul contor de program corespunzător locului unde se face un apel sistem este salvat, iar UCP trece în mod supervisor. Revenirea la modul utilizator este similară cu cea produsă la o revenire din procedură.

Adresele de bază și limită constituie minimul unui sistem de protecție. Mecanismul memoriei virtuale oferă alternative mai complexe și mai sigure decât modelul simplu prin bază și limită. Așa cum s-a văzut, adresele virtuale sunt translatate în adrese fizice pe baza unor tabele de traducere. Acest mod de mapare, prin tabele, oferă posibilitatea introducerii de informații pentru controlul erorilor de program (intenționate sau nu) care încearcă să treacă peste mecanismele de protecție. Cea mai simplă cale este introducerea unor indicatori de permisie pentru fiecare pagină sau segment. De exemplu pot exista indicatori care să permită doar citire, doar execuție, sau care să interzică accesul unui proces utilizator la anumite pagini / segmente. Fiecare proces poate face adresare doar către paginile proprii de memorie, procesul utilizator neavând dreptul să modifice tabelele sale de pagină / segment.

Protecția poate fi extinsă chiar pe mai mult decât două niveluri (nu doar utilizator și supervisor), privite ca și inele concentrice de protecție, în centru găsindu-se nivelul de protecție cel mai înalt. În această ierarhie de niveluri de protecție un program poate accesa doar date de pe nivelul său de protecție și de pe nivelurile inferioare în ierarhie. Poate face însă operații de apelare (call) a serviciilor sistemului de operare, servicii oferite de rutine ce se află pe niveluri superioare de protecție. Adesea se face comparația, în oarecare măsură nefericită, cu clasificări de tip militar ale sistemelor: top – secret, secret, confidențial și neclasificat. Programele utilizator ("civili" în exemplul militar) au doar dreptul de acces la nivelul de protecție cel mai de jos: "ne-clasificat. La sistemele de protecție de tip inele concentrice, deosebirea față de exemplul anterior, este că se pot face apelări la rutine situate pe niveluri superioare de protecție, dacă există "chei" de acces către acele niveluri. Poate exista dreptul de apelare a unor servicii ale sistemului de operare, prin mecanismul de comutare a proceselor.

Așa cum am pomenit și mai sus, informația de protecție este setată în registre speciale atașate fiecărei intrări în tabelele de traducere, registre setate doar de rutine de control al sistemului de operare. Drepturile de acces pot fi de tipul:

- a. Atribuirea de privilegii complete de citire și scriere. Aceste drepturi se atribuie programului atunci când execută propriile instrucțiuni.
- b. Read-only (protecție la scriere). Protecția la scriere este utilă la operațiile de partajare a unor rutine de sistem (utilitare, biblioteci, etc.).
- c. Execute only (program protection). Protejează programul la copiere. Restricționează referirea la segment doar în timpul fazei de fetch a instrucțiunii și nu și în timpul fazei de execuție. Asta permite utilizatorului să execute instrucțiunile segmentului de program, dar nu permite citirea instrucțiunilor ca date cu scopul de a copia conținutul lor.

Sistemul de protecție a memoriei, în sensul celor spuse mai sus, se construiește pentru:

- (a) memorie (detectează orice eroare de adresare înainte ca aceasta să creeze erori accidentale sau voite);
- (b) programe (previne ca programele utilizator (de aplicații) să facă modificări ilegale în rutinele SO);
- (c) utilizatori (programele utilizatorilor între ele);
- (d) securitate informație (acces limitat la informațiile unde utilizatorul nu are drept de acces).

CAPITOLUL 7

SISTEMUL DE INTRARE - IEȘIRE

Conținut:

- 7.1. Circuite de interfață
- 7.2. Organizarea ierarhică a magistrelor
- 7.3. Transferuri asincrone de date
 - 7.3.1. Transmisie asincronă cu un singur semnal de control
 - 7.3.2. Transmisie asincronă cu două semnale de control
- 7.4. Modalități de transfer de intrare-ieșire
 - 7.4.1. Transferul de I/O prin program
 - 7.4.2. Transferul I/O prin întreruperi
 - 7.4.3. Transferul prin acces direct la memorie

Obiective educaționale:

- a. Descrierea rolului circuitelor de interfață
- b. Analiza organizării magistrelor sistemului pe mai multe niveluri
- c. Clasificarea și descrierea diferitelor moduri de transfer al datelor cu dispozitivele de intrare - ieșire

7.1. CIRCUITE DE INTERFAȚĂ

Subsistemul de intrare-ieșire (I/O) al unui calculator asigură calea de comunicație a informațiilor între calculator și mediul extern (logica externă care nu lucrează direct cu UCP). Un sistem de calcul de uz general, nu are utilitate practică dacă nu poate recepționa și transmite informații cu exteriorul, într-o formă accesibilă utilizatorului uman. Prin intermediul acestui subsistem, utilizatorul introduce programele și datele dorite în memoria calculatorului, pentru prelucrare, și tot cu ajutorul său rezultatele se înregistrează, sau se afișează în exterior.

Legătura între UCP și dispozitivele de I/O (dispozitivele periferice) se face prin intermediul unor *circuite de interfață de I/O* care asigură, din punct de vedere hardware, schimbul corect de date. Aceste circuite de interfață se cuplează la magistralele calculatorului și ele sunt adresabile la nivel de *registru-port* de intrare-ieșire. Prin *port* înțelegem aici un loc (în general un registru), cu adresă specifică, adresă care constituie o "poartă" prin care calculatorul realizează schimb de informație cu exteriorul; fie culege informația, iar portul este port de intrare (PI), fie transmite informația la un port de ieșire (PO). Registrele port pot fi adresate în spațiul de adrese al memoriei (vorbim atunci de *mapare-organizare a porturilor în spațiul de memorie*) sau pot fi adresate în spațiu separat de cel de memorie (*mapare în spațiul de I/O*). Fiecare port de intrare sau ieșire are o adresă specifică.

Ca urmare adresele porturilor de intrare-ieșire pot fi tratate în două moduri:

1. *ca porturi cu adrese distincte față de adresele de memorie* (semnalele de control și selecție fiind specifice pentru memorie, respectiv pentru spațiul de intrare-ieșire) - *mapare în spațiu separat de I/O*;
2. *adresele porturilor sunt înglobate în spațiul de adrese al memoriei principale*. Această organizare a adreselor (numită și "*mapare a adreselor de I/O în spațiul de memorie*") face ca porturile să fie selectate prin aceleași semnale de adresă și control ca și memoria. În lucrul cu porturile de I/O se vor utiliza instrucțiunile de transfer specifice memoriei.

Este important de observat că, exceptând procesorul și memoria principală, toate circuitele conectate la magistrala de date a sistemului, deci care partajează această resursă a sistemului, sunt privite de procesorul central (UCP) ca porturi de I/O, chiar dacă funcțiile acestora nu se limitează strict la operații de intrare / ieșire. Este o manieră unitară de lucru a UCP cu orice circuit de interfață, circuit care poate include mai multe porturi de intrare / ieșire, indiferent de funcțiile specifice ale acestuia (figura 7.1). Aceasta face ca din punctul de vedere al UCP să se lucreze cu porturile adresabile într-un mod asemănător cu cel folosit la adresarea locațiilor de memorie.

Unele circuite de interfață sunt incluse în circuitele specializate de control ("controllere") ale perifericelor. De exemplu, controllerul unității de disc, controlează operațiile fizice efectuate de discul magnetic. Pentru ca să se poată face cuplarea la calculator controllerul trebuie să respecte niște specificații standardizate de interfațare. Standardul IDE ("Integrated Drive Electronics") a realizat transferarea circuitelor de control către mecanismul discului hard, circuitele cuplate direct la magistralele calculatorului având o structură simplă. Pentru cazul specific al discului hard, standardul de interfață stabilește nu numai modul de lucru cu partea electronică de control, ci și modul în care se face codificarea datelor pe suportul magnetic. Funcția principală a circuitelor interfață de I/O este de a rezolva diferențele *funcționale, electrice și informaționale* dintre calculator și periferic. Circuitele controler au în plus și sarcina specifică de control a unui anumit periferic. Principalele diferențe între UCP și periferie, care impun folosirea circuitelor de interfață, constau în următoarele:

- a. perifericele sunt dispozitive a căror funcționare se bazează pe diferite tehnologii (electromecanice, electromagnetice, electronice). De aceea trebuie să existe dispozitive de

- conversie a valorilor semnalului, pentru o adaptare din punct de vedere electric cu calculatorul;
- ritmul de transfer al datelor este mult mai scăzut la periferice față de UCP. Pentru transferul de date între periferice și UCP sau memorie trebuie deci să existe mecanisme de sincronizare.
 - codurile și formatele datelor în echipamentele periferice pot fi diferite față de codurile și formatele folosite în UCP și memorie.
 - există o varietate de periferice, cu moduri de funcționare diferite și de aceea acestea trebuie controlate adecvat, pentru a nu perturba celelalte periferice conectate la UCP.

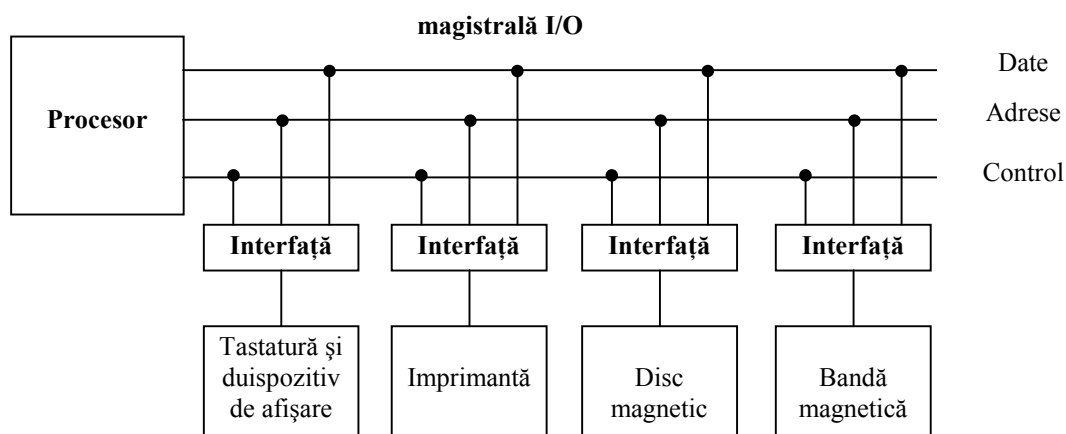


Figura 7.1. Conectarea la magistrala de I/O a dispozitivelor de intrare - ieșire.

Circuitul de interfață rezolvă toate aceste diferențe, fiind inclus între UCP și periferice, pentru a superviza și sincroniza toate transferurile de intrare / ieșire. În plus așa cum am amintit, fiecare periferic poate avea propriul controller care supervizează funcționarea corectă, specifică a respectivului periferic. Interfațarea a două dispozitive fizice (*hardware interfacing*) constă în a proiecta circuitele de interconectare fizică dintre aceste două dispozitive. Iată câteva exemple obisnuite de sarcini de "hardware interfacing":

- interfațarea unei imprimante (printer) la un calculator (de exemplu, pentru PC, posibilă doar prin respectarea standardului de interfațare USB¹ sau Centronix);
- interfațarea unei legături de comunicație serială la un calculator
- interfațarea unui port de I/O digital la un calculator
- interfațarea unor convertoare D/A sau A/D la un calculator
- interfațarea unei unități de dischetă sau a unui controler de disc fix (hard disc) la un calculator.

Notiunea de interfațare poate fi folosită însă și dacă ne referim la programele rulate de calculator (*software interfacing*). Când cele două obiecte ce definesc interfața sunt programe de calculator, sarcinile de interfațare constau în proiectarea unui alt program ce asigură comunicarea dintre primele două programe. De exemplu, un program poate fi un program de control al imprimantei (printer-driver), iar celălalt poate fi un program general de aplicație al utilizatorului. Programul de control al imprimantei este o procedură care tipărește un caracter la imprimantă ori de câte ori este apelat. Interfața software se referă la faptul ca programul de aplicație trebuie să

¹ USB = Universal Serial Bus

cunoască locul în care trebuie să plaseze caracterul de tipărit, înainte de a invoca programul printer driver. După cum se vede din acest exemplu, interfațarea software face ca parametrii să fie transmiși corect de la un program la celălalt. Protocolul de transmitere a parametrilor este stabilit doar de unul din cele doua programe ce comunică prin intermediul interfeței. De exemplu la un PC², ce rulează sub sistemul de operare DOS, dintre cele doua programe ce se interfațează, un program este, în general, un program de aplicație; celălalt program poate fi un program de control pentru un dispozitiv ("device driver"), de exemplu un printer-driver, sau poate realiza doar o anumită funcție simplă (ca de exemplu alocarea memoriei). Al doilea program este cel care definește protocolul pentru transmiterea parametrilor. Distincția dintre un hardware driver (program de control al unui dispozitiv fizic) și o funcție nu prezintă importanță aici, elementul important este modul cum se transmit parametrii.

Circuitul de interfață fiind legat între magistrala de I/O a sistemului și echipamentul periferic, are semnale specifice de cuplare - interfațare cu magistrala, respectiv cu echipamentul periferic. Dacă spre periferic tipurile de semnale și modul de lucru cu acestea depind în mare măsură de caracteristicile dispozitivului periferic, partea dinspre magistrală, face legătura cu memoria și UCP și cuprinde semnale pentru selecția circuitelor de I/O, pentru controlul transferului datelor și semnale de cerere de servicii către UCP.

Transferul datelor între un port de I/O și UCP are loc, în principiu, asemănător cu transferul datelor între procesor și memorie.

Există patru tipuri de semnale de comandă pe care le poate recepționa/transmite o interfață:

1. semnale de control
2. semnale de stare
3. date de ieșire
4. date de intrare

Semnalele de control sunt transmise de UCP pentru a activa perifericul și pentru a-l informa ce operație trebuie să efectueze. De exemplu, o unitate de bandă magnetică poate fi comandată să deplaseze banda înainte cu o înregistrare de date, să deruleze rapid banda la început, sau să pornească citirea unui bloc de înregistrări. Semnalele de control emise sunt specifice pentru fiecare tip de periferic.

Semnalele de stare sunt utilizate pentru a testa diferite condiții ale interfeței și ale perifericului. De exemplu, calculatorul poate testa dacă un periferic este gata pentru un viitor transfer de date. În timpul transferului se pot produce erori care sunt detectate de interfață. Aceste erori sunt marcate prin setarea unor biți dintr-un registru de stare al interfeței, registru ce poate fi citit de procesor.

Semnalele de comandă pentru date de ieșire fac ca interfața să răspundă prin transferarea datelor de la magistrala de date către registrele sale interne. Considerați exemplul cu unitatea de bandă. Calculatorul pornește derularea benzii prin semnale de control. Apoi procesorul monitorizează efectuarea comenzii trimise prin citirea informațiilor de stare. Când banda a ajuns la poziția corectă, procesorul transmite comenzile (adresă și control) și datele de ieșire, iar interfața transferă informația către registrele interne, iar apoi către controllerul unității de bandă, pentru stocare.

Comenzile pentru date de intrare sunt similare cu cele pentru ieșirea datelor, diferind doar sensul de circulație al informației. Procesorul testează starea interfeței pentru a verifica dacă datele cerute pot fi transferate către magistrala de date.

² PC = Personal Computer

7.2. ORGANIZAREA IERARHICĂ A MAGISTRALELOR

În calculatoarele moderne transferul dintre UCP și memorie, respectiv dispozitivele de I/O se face prin magistrale organizate ierarhic, în funcție de viteza dispozitivelor cuplate la fiecare magistrală.

Există mai multe variante de organizare, în funcție de tipul calculatorului (de uz general sau pentru aplicații specifice) și diferențele de viteză între dispozitive. Dacă la o magistrală se conectează mai multe dispozitive performanța sistemului poate scădea. Cu cât sunt mai multe dispozitive conectate la magistrală cu atât crește lungimea acestuia și deci întârzierea la propagarea semnalelor. În plus magistrala unică poate deveni o resursă hardware extrem de solicitată dacă este multiplexată în timp de multe dispozitive, cu viteze de funcționare mult diferite.

De aceea marea majoritate a calculatoarelor utilizează mai multe magistrale, organizate ca o ierarhie de magistrale cu viteze de operare diferite. O structură clasică de magistrale multiple organizate pe niveluri este prezentată în figura 7.2 [Stallings00]. Există o magistrală locală care conectează procesorul cu memoria cache și la care se pot conecta și câteva dispozitive de I/O (locale) de mare viteză. Controllerul de memorie cache interfațează această memorie cu procesorul și de asemenea cu magistrala sistem la care este conectată memoria principală. Avantajul acestui mod de organizare este că transferul direct între dispozitivele de I/O și memorie nu interferează cu activitatea procesorului care lucrează direct cu memoria cache. Dispozitivele de I/O sunt conectate la al treilea nivel ierarhic numit magistrală de extensie. Acest ultim mod de aranjare permite conectarea la magistrala de extensie a unui mare număr și o mare diversitate de dispozitive de I/O și în același timp izolează traficul procesor - memorie față de traficul de I/O.

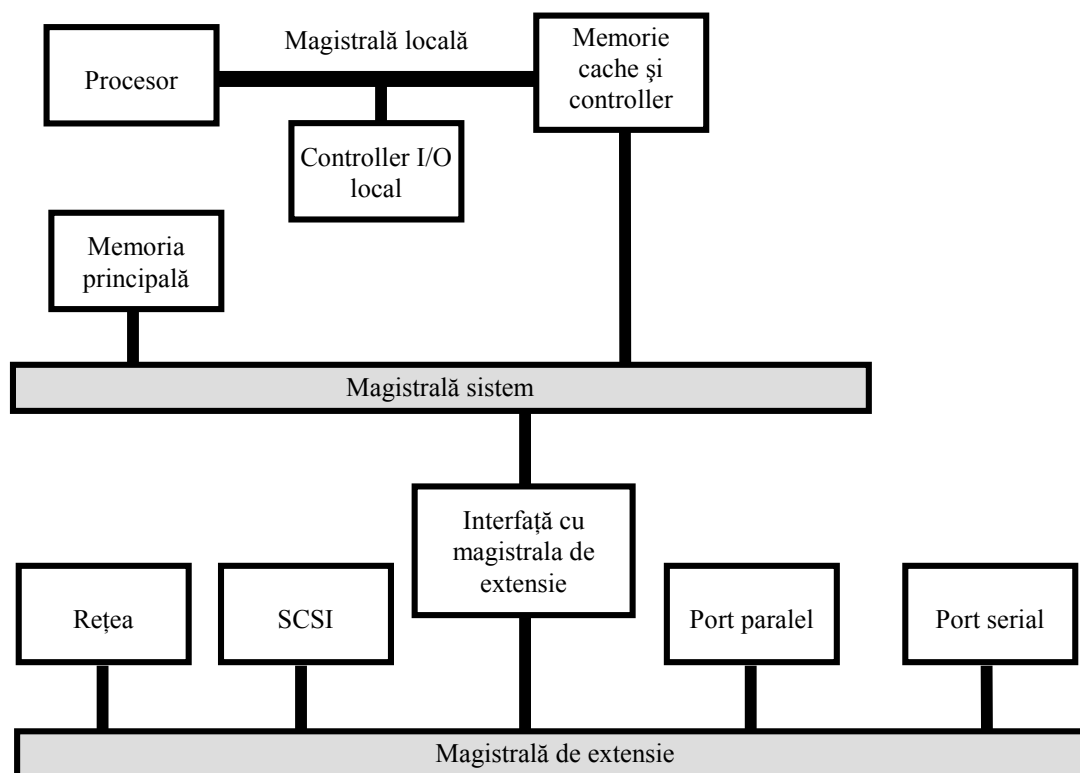


Figure 7.2. Exemplu de organizare ierarhică a magistrelor calculatorului

În figura 7.2 se exemplifică câteva dispozitive de I/O ce pot fi cuplate la magistrala de extensie. Conexiunile de tip rețea includ rețele locale (LAN - local area network), sau conexiuni la

rețele pe arii extinse. Controllerul SCSI (small computer system interface) conectează la magistrala de extensie o magistrală SCSI la care se pot conecta controllere de hard disc locale și alte periferice. Portul serial poate fi folosit pentru conectarea unei imprimante sau a unui scanner. Magistralele de extensie (numite și magistrale de I/O) suportă o gamă largă de rate de transfer, pentru a permite conectarea unei game largi de dispozitive I/O.

Arhitectura clasică a magistrelor este eficientă, dar nu face față la noile dispozitive de I/O ce funcționează la viteze din ce în ce mai mari. Ca urmare au apărut noi organizări, care introduc un nivel intermediu de magistrală de mare viteză (numit uneori magistrală la mezanin) care se interfațează cu o punte (bridge) cu magistrala locală a procesorului. Figura 7.3 prezintă această abordare, în care controllerul de cache este integrat într-o punte, sau dispozitiv tampon, care se conectează la magistrala de mare viteză. La această magistrală de mare viteză se pot conecta circuite controller de mare viteză pentru LAN, (cum ar fi Fast Ethernet la 100 Mbps, controller video-grafic). Dispozitivele de I/O cu viteză mică se cuplează în continuare la magistrala de extensie conectată printr-o interfață cu magistrala de mare viteză. Avantajul acestei aranjări este că dispozitivele de I/O de mare viteză sunt integrate mai aproape de procesor și în același timp pot funcționa în paralel cu procesorul.

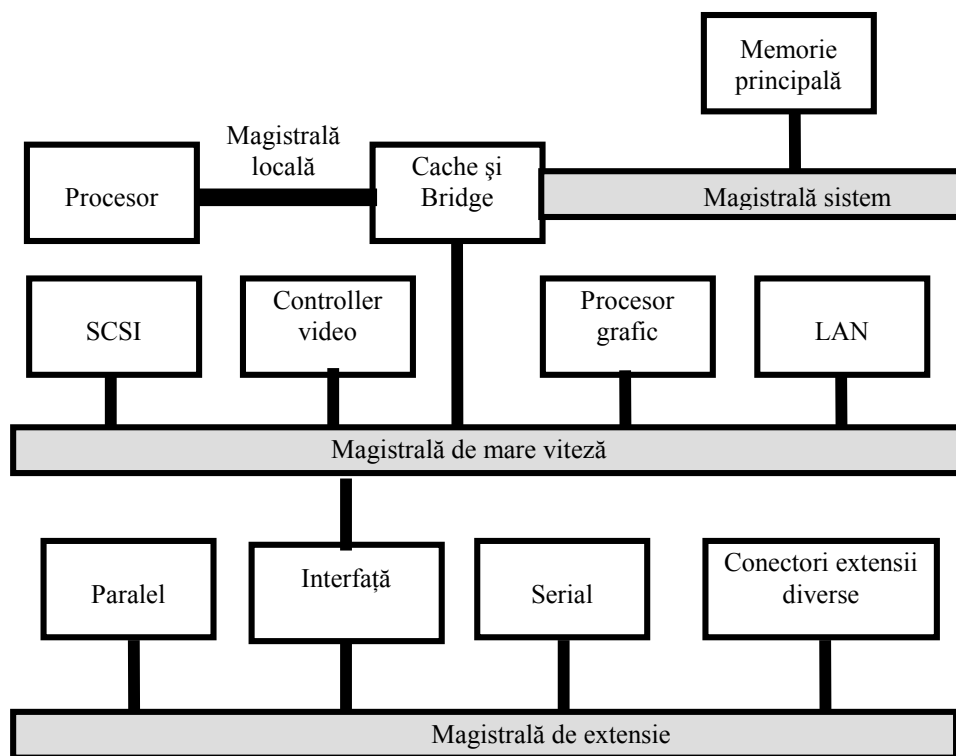


Figura 7.3. Exemplu de organizare ierarhică a magistrelor la care se introduce magistrala de mare viteză “de la mezanin”

Din punctul de vedere al modului de transfer a informațiilor pe magistrale, acestea pot fi sincrone, sau asincrone. Magistralele sincrone sunt comandate de un semnal de ceas local. Toate transferurile pe aceste magistrale, numite cicluri de magistrală, respectă un protocol fix ce impune un anumit număr de impulsuri de ceas. Datorită sincronizării controlul transferurilor este extrem de simplu. Blocurile de interfață (bridge) între magistralele sincrone ce funcționează la diferite frecvențe de ceas trebuie să realizeze adaptarea de viteză în așa fel încât transferurile între magistrale să se facă corect și cât mai rapid. La magistralele asincrone transferurile nu mai trebuie

să se încadreze într-un interval fix de timp, iar controlul transferurilor se face cu ajutorul unor semnale de control între cei doi corespondenți (*handshaking*).

Transferurile cu porturile de I/O sunt în general de tip asincron, cu posibilitatea de a introduce stări suplimentare de așteptare ale UCP pentru a mări durata unui ciclu mașină.

Pentru majoritatea procesoarelor ce pot organiza spațiu separat de adrese pentru porturile de I/O, în ciclurile de transfer cu porturile se introduc automat (de către UCP) stări de așteptare (*wait*), care adaptează viteza UCP la viteza scăzută a dispozitivelor de I/O. Dacă porturile sunt organizate în spațiul de memorie, acest mecanism de sincronizare, cu ajutorul stărilor de *wait*, trebuie construit în exteriorul UCP și el trebuie să acționeze ori de câte ori se face acces la o adresă ce corespunde spațiului de I/O.

7.3. TRANSFERURI ASINCRONE DE DATE

Transferurile asincrone de date între două unități independente cer să se transmită semnale de control între unitățile ce comunică, pentru a se indica momentul la care datele sunt disponibile. Transferul asincron poate fi controlat în două moduri:

1. *cu un semnal de control*, printr-un impuls de *strobe* (*marcare*) furnizat de unul din corespondenți pentru a indica celuilalt momentul la care datele vor fi transferate.
2. *cu două semnale de control*, la care datele ce trebuie transferate sunt însoțite de un semnal de control care indică prezența datelor pe magistrală. Unitatea ce recepționează datele răspunde cu un alt semnal de control pentru a confirma recepția datelor. Se stabilește astfel o legătură de date cu confirmare ("*handshaking*").

Cele două metode de control a transferului asincron pot fi utilizate nu numai la transferurile de I/O, ele fiind folosite în unele sisteme de calcul și la transferurile asincrone procesor-memorie, procesor - port, port - echipament periferic. În continuare vom nota cele două unități care comunică ca "*sursă*" și "*destinație*" a datelor fără a specifica care sunt cei doi interlocutori.

7.3.1. Transmisie asincronă cu un singur semnal de control

Această metodă de transfer asincron a datelor folosește o singură linie de control pentru a indica fiecare transfer. Semnalul de marcă (notat "*Strobe*" în figura 7.4) poate fi activat fie de unitatea sursă (figura 7.4.a) fie de cea destinație (figura 7.4.b). La transferul inițiat de sursă, *Strobe* indică prezența datelor pe magistrala comună și de asemenea poate servi ca semnal ce controlează memorarea datelor de către destinație. Vitezele de lucru ale sursei *S* și destinației *D* trebuie cunoscute reciproc, la proiectarea și construcția sistemului. În funcție de cel mai lent dintre interlocutori, se dimensionează intervalele de timp t_1 , t_2 și t_3 , astfel încât să se efectueze un transfer corect. Dacă destinația este un port de ieșire, acesta trebuie să aibă capacitatea de memorare a datelor. Pentru transferul inițiat de sursă sursa plasează întâi datele pe magistrala comună și după stabilizarea valorilor pentru date, sursa activează linia de *strobe*. Datele și semnalul de *strobe* trebuie să rămână active cel puțin un timp notat t_2 , suficient destinației pentru a citi corect datele. Adesea destinația folosește ultimul front al semnalului de *strobe* pentru a copia conținutul datelor într-un registru intern. În cazul transferului inițiat de destinație, dacă sursa datelor este un port de intrare, în principiu, nu este obligatorie funcția de memorare locală. Ieșirile portului către magistrala locală de date pot fi realizate prin operatori TSL (trei stări logice), activați de semnalul de *strobe*. Pentru transfer inițiat de destinație, semnalul de marcă are funcția unei cereri de transfer de date. În multe din aceste tipuri de transferuri în calculator, impulsurile de *strobe* sunt controlate tot de impulsurile de ceas ale UCP (tranzitiile *strobe* sunt sincronizate cu tranziții ale semnalului de ceas). Astfel că

strobe ar putea fi semnalul de validare al adresei pe magistrală, sau de validare date, semnal de citire-scriere etc.

Inconvenientele acestui tip de transfer constau în lipsa totală a oricărei confirmări privind acceptarea și / sau corectitudinea datelor recepționate, respectiv transmise. Acest mod de transfer se poate utiliza doar între unitățile la care vitezele de lucru sunt cunoscute reciproc. Transferul de acest tip este specific pentru inițializarea unor dispozitive programabile de I/O, la care datele transmise se înscriu în registre de control ale dispozitivelor respective.

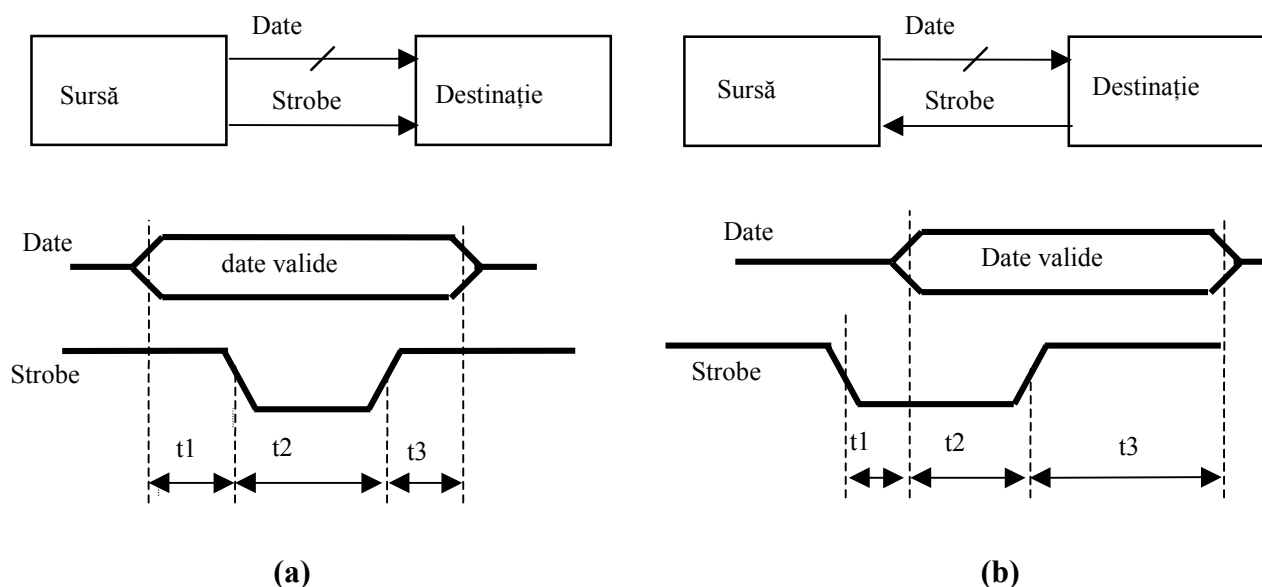


Figura 7.4. Transfer asincron de date cu un singur semnal de control. (a) diagrama bloc și diagrama semnalelor în timp pentru transfer inițiat de sursă. (b) diagrama bloc și diagrama semnalelor în timp pentru transfer inițiat de destinație.

7.3.2. Transmisie asincronă cu două semnale de control

Unul dintre semnalele de control realizează inițierea transferului de date, iar cel de-al doilea confirmă recepția acestora. Procedul este numit *transfer asincron al datelor cu confirmare* ("handshaking"). Se realizează astfel un protocol de transmisie la nivel electric, între sursă și destinație. Și aici, transferul poate fi inițiat de sursă (cu strobe - marcarea datelor de transmis), sau de către destinație (prin strobe - cererea de date). În ambele cazuri interlocutorul va răspunde cu un semnal care indică că datele au fost acceptate, respectiv că datele au fost furnizate (Ready).

Secvența evenimentelor pentru transferul inițiat de sursa datelor este următoarea (figura 7.5):

- sursa plasează datele pe magistrala de date comună
- sursa validează datele prin semnalul de marcă (Strobe)
- destinația confirmă că este gata (Ready) pentru transfer
- sursa dezactivează semnalul Strobe, care de obicei produce și memorarea datelor la destinație iar după un timp dezactivează și datele de pe magistrală
- destinația dezactivează semnalul Ready

Se observă că sursa menține semnalul de strobe activ până când destinația este în măsură să răspundă cu semnalul de ready. Durata semnalelor active se pot prelungi până când cel mai lent

dintre corespondenți poate efectua corect transferul de date.

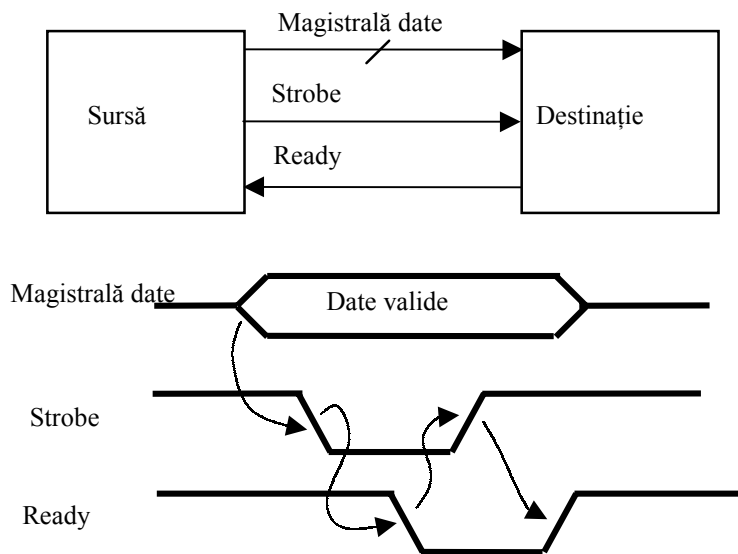


Figura 7.5. Diagrama bloc și evoluția semnalelor în timp pentru transfer de tip handshaking inițiat de sursa datelor.

În cazul când inițierea transferului se face de către destinație formele de undă corespunzătoare controlului transferului sunt prezentate în figura 7.6.

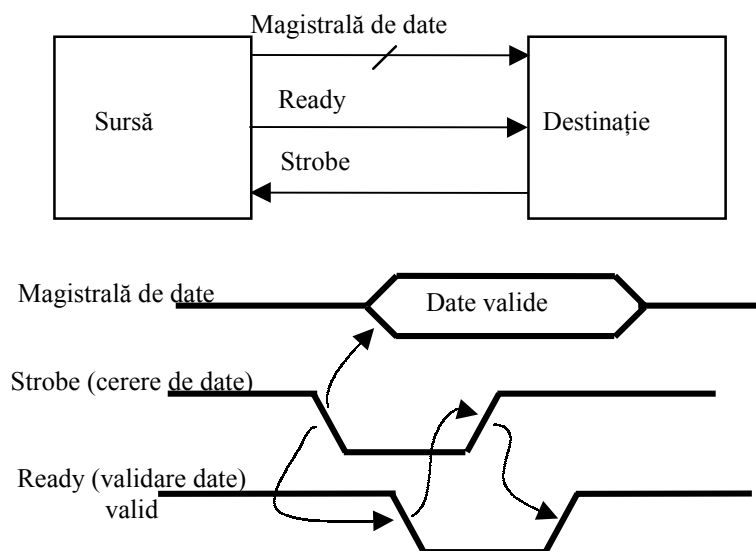


Figura 7.6. Diagrama bloc și evoluția semnalelor în timp pentru transfer de tip handshaking inițiat de destinația datelor.

Secvența evenimentelor pentru transferul inițiat de destinație este următoarea:

- destinația lansează semnalul Strobe, cu semnificația "cerere de date"
- sursa răspunde prin punerea datelor pe magistrala comună și apoi prin activarea semnalului de confirmare (Ready)
- destinația citește datele și confirmă acest lucru prin dezactivarea semnalului de strobe
- sursa consideră transferul încheiat și dezactivează semnalul de ready.

7.4. MODALITĂȚI DE TRANSFER DE INTRARE-IEȘIRE

Din punctul de vedere al circuitelor care controlează transferul și de asemenea al dispozitivului care inițiază un transfer de I/O, modalitățile de transfer se pot clasifica în:

1. *transfer prin program* - transfer inițiat și controlat în totalitate de programul rulat de UCP;
2. *transfer prin întreruperi* - transferul este controlat de UCP ca răspuns la o cerere de întrerupere externă, care inițiază transferul;
3. *transfer prin acces direct la memorie* (DMA - "Direct Memory Acces") - transferul este controlat de un circuit controler DMA (care preia controlul magistralelor sistemului), iar inițierea transferului este făcută fie de o cerere de transfer de la un periferic, fie la inițiativa programului rulat de UCP.

7.4.1. Transferul de I/O prin program

La acest tip de transfer, toate transferurile se inițiază și se controlează prin program. Transferul poate fi *direct*, caz în care procesorul citește un port de intrare (PI) sau scrie un port de ieșire (PO), fără nici o verificare prealabilă a stării perifericului corespunzător portului. Acest mod de transfer se folosește pentru sistemele simple de control numeric, dedicate unor aplicații fixe, în faza de proiectare și testare a transferului (program plus hardware), sau în cazul rulării unor rutine de inițializare a circuitelor programabile de interfață.

Al doilea mod de transfer prin program este transferul prin *interogare*. Intrarea în conversație cu un periferic se face sub controlul programului, de obicei într-o *bucă de interogare* (*polling* - scrutare a stării circuitelor periferice implicate în transfer). UCP interoghează dispozitivele periferice, după o anumită strategie stabilită prin program, dacă doresc sau nu schimb de date sau mesaje, sau dacă sunt active (gata pentru a primi informații). Pentru a determina dacă o operație de I/O este cerută, sau dacă poate avea loc, se pot folosi bistabile de condiție (fanioane) locale, setate conform condițiilor portului. Aceste fanioane se implementează fizic fie ca bistabile singulare, fie sunt incluse în registre de stare ale porturilor. De exemplu un program poate controla transferul de date de la un circuit de conversie analog-numerică. Procesorul declanșează conversia, iar apoi testează periodic o ieșire (un bit) de stare al convertorului. Când bitul indică terminarea conversiei, se încheie rularea buclei de așteptare, preluându-se datele convertite.

Datorită simplității ei, tehnica de comunicare prin interogare programată este recomandată în sistemele în care timpul pierdut în rutina de interogare nu este critic pentru aplicația respectivă. Atunci când programul are de controlat mai multe periferice, pe baza unei liste a porturilor asociate perifericelor, procesorul testează pe rând starea perifericelor și le servește din punctul de vedere al transferului, dacă starea citită permite acest lucru. În acest fel perifericele sunt deservite în mod secvențial și implicit apar întârzieri la servirea acestora, mai ales dacă numărul de periferice controlate este mare. Dacă unele dintre periferice se consideră mai importante decât altele, adresa acestora se poate introduce de mai multe ori în lista de testare din bucla de interogare. Dezavantajul principal al modului de transfer programat este constituit de timpul pierdut de UCP pentru testarea stării perifericelor, chiar dacă acestea nu cer servicii la momentele de timp respective. Acest mod de transfer prezintă însă avantaje din punctul de vedere al costurilor (minim de echipament fizic suplimentar pentru transfer). Alt avantaj este că se cunoaște exact momentul de timp când se testează sau când se face transfer de date cu un periferic; de aceea se spune că acest tip de transfer este *sincron cu programul*.

Ca exemplu al transferului prin program vom presupune mai întâi un transfer între memorie și un port de ieșire, indicând, mai întâi printr-o listă secvența de comenzi folosite:

START

```

                                inițializare POINTER adresă de memorie;
                                inițializare CONTOR al lungimii transferului;
CITIRE_MEM:                   UCP citește date din memorie
TEST_STARE:                   citire stare port
                                dacă e gata de transfer sari la TRANSF
                                dacă nu, incrementează CONTOR_RATĂRI
                                dacă CONTOR_RATĂRI = W salt la EROARE
                                altfel, salt la TEST_STARE
TRANSF:                       UCP scrie data la port, și actualizează CONTOR și POINTER
                                dacă mai sunt date de transmis, salt la CITIRE_MEM
                                altfel salt la FINAL
EROARE:                       .....
FINAL:                        .....

                                END

```

Pentru I8086, dacă la adresa simbolică "**port**" se găsește informația de stare (bitul 0 indicând prin 1 logic, stare gata de transfer), iar la adresa **port+1** se vor transfera datele, fragmentul de program arată astfel:

```

start:    mov dx, port
          lea bx, buffer
          mov cx, count
          mov si, wait
test_st:  in al,dx
          test al,1
          jnz transf
          dec si
          jnz test_st
transf:   inc dx
          mov al,[bx]
          out dx,al
          dec dx
          inc bx
          dec cx
          jnz test_st

```

Dacă sunt mai multe porturi, (portul I având adresa registrului de stare "port1", iar bitul 0 al cuvântului de stare arată dacă portul e gata (1) sau nu (0) de transfer), bucla de interogare ar putea arata astfel:

```

test1:    in al,port1
          test al,1
          jz test2
          call transf1
test2:    in al,port2
          test al,1
          jz test3
          call transf2

          .....

```

```

testN:      in al,portN
            test al,1
            jz test1
            call transfN

```

7.4.2. Transferul I/O prin întreruperi

Așa cum s-a arătat în capitolul 5 evenimentele externe procesorului, deci independente de programul rulat, pot produce *cereri de întrerupere*. Dacă acestea sunt acceptate, se produce *întreruperea* (suspendarea temporară a) programului rulat și *saltul la o rutină specifică de tratare* a cererii de întrerupere. După execuția rutinei de tratare se revine la execuția programului întrerupt. Subsistemul de întreruperi al calculatorului nu este destinat special doar pentru operații de transfer de I/O, dar aici ne vom referi doar la acest aspect. Din punctul de vedere al transferurilor de I/O, avantajele sistemului de întreruperi sunt următoarele [Sztojanov87]:

- permite sincronizarea procesorului cu evenimente externe;
- eliberează procesorul de sarcina testării periodice a perifericelor, sarcină consumatoare de timp. Ca urmare transferul prin întreruperi prezintă o viteză de răspuns mai mare decât transferul prin program;
- posibilitatea de tratare ierarhizată a cererilor de întrerupere simultane sau succesive;

Cererile de întrerupere pot fi recunoscute doar la sfârșitul ciclului instrucțiune curent. În general, pentru efectuarea transferurilor de I/O se folosesc cererile de întrerupere mascabile.

Răspunsul UCP la acceptarea unei cereri de întrerupere mascabilă, consta în următoarele operații succesive:

1. Salvarea stării programului întrerupt. În această fază se salvează automat în stivă cel puțin adresa de revenire la programul întrerupt. Alte informații privind starea programului se pot salva prin instrucțiuni introduse la începutul rutinei de tratare, iar înainte de revenirea la programul întrerupt trebuie introduse instrucțiuni pentru restaurarea acestor informații.
2. Confirmarea acceptării și identificarea întreruptorului. Confirmarea acceptării întreruperii se face prin transmiterea de UCP a unui semnal de acceptare, în urma căruia perifericul trimite, de obicei, un vector de întrerupere pentru identificare. Există și sisteme de identificare care nu folosesc vector de întrerupere.
3. Calcularea adresei de început a rutinei de tratare a întreruperii. Pe baza informației de identificare se calculează unde se va face saltul pentru ca cererea de întrerupere, pentru transferul de date, să fie servită.
4. La terminarea rutinei de tratare, se execută o instrucțiune specială, care comanda refacerea conținutului contorului de program și deci revenirea la programul întrerupt.

Generarea vectorului de întrerupere se face fie direct de către dispozitivul întreruptor (de fapt de către circuitul de interfață cu un anumit periferic), fie centralizat, pentru toate întreruperile mascabile, de către un dispozitiv special numit controller de întreruperi.

Arbitrarea întreruperilor multiple.

În cazul întreruperilor multiple, simultane sau succesive, subsistemul de întreruperi trebuie să realizeze un arbitraj al acestora pentru a stabili care dintre cererile de întrerupere multiple va fi servită la un moment dat. Sistemul de arbitraj alocă priorități cererilor de întrerupere, servind întotdeauna cererile cu prioritatea cea mai mare. Se permite de asemenea ca o rutină de tratare să

poată fi la rândul ei întreruptă de o cerere cu prioritate mai mare. De obicei, pentru un procesor de uz general, prioritățile intrinseci sunt atribuite în următoarea ordine: întreruperile interne (excepțiile) cu prioritatea cea mai mare, apoi urmează întreruperile externe (hardware) nemascabile și în final cele mascabile. Arbitrarea întreruperilor multiple se poate realiza prin mai multe metode:

Arbitrare	• controlată de UCP	• prin software
		• prin hardware
	• controlată de circuit controler de întreruperi	
	• controlată prin hardware, în lanț de priorități	

În cazul arbitrării *controlate de UCP prin hardware*, UCP conține intern circuitele necesare realizării arbitrării. Circuitul de arbitrare primește mai multe intrări de cereri de întrerupere, acestea, sau combinații ale acestora, corespunzând la un anumit nivel prefixat de prioritate. Adresele rutinelor de tratare pot fi fixe (cum este de exemplu cazul microprocesorului I8085) sau variabile (de exemplu la microprocesorul MC68000).

De exemplu, la microprocesorul Intel 8085 există 5 intrări de cereri de întrerupere. Una dintre intrări este numită INT și ea funcționează pentru întreruperi vectorizate, deci pe bază de vector de întrerupere. Celelalte intrări nu au nevoie de vector de întrerupere, pentru că intern ele au alocată câte o prioritate fixă, iar saltul către rutina de tratare se face tot la o adresă fixă. În tabelul 7.1. se indică intrările de întrerupere la microprocesorul I8085, împreună cu ordinea de prioritate alocată (prioritatea 1 fiind maximă) și adresele de salt.

Tabel 7.1. Întreruperi la I8085.

Prioritate fixă	Nume intrare întrerupere	Adresă de salt
1	TRAP	24h
2	RST 5.5	2Ch
3	RST 6.5	34h
4	RST 7.5	3Ch
5	RST 7.5	prin Vector

Arbitrarea *controlată de UCP prin software*, constă într-un suport hardware minimal, extern UCP, care să sprijine operațiile de recepție a cererilor de întrerupere și respectiv de identificare și validare sau nu a întreruptorului, în funcție de prioritatea alocată. Circuitele au ca element principal un registru-port de intrare, în care se alocă câte un bit pentru fiecare dispozitiv întreruptor. Dacă acest bit este setat, dispozitivul a cerut cerere de întrerupere. Simultan cu setarea bitului din registru, semnalul de cerere de întrerupere se transmite și către UCP. Prin citirea registrului port de intrare și compararea cu situația anterioară (memorată într-un registru intern), UCP detectează dacă a apărut o cerere de întrerupere cu prioritate mai mare decât cea a programului curent. În cazul unei întreruperi cu prioritate mai mică se continuă programul curent (după eventuala înregistrare a cererilor în vederea servirii ulterioare). În cazul priorității mai mari, UCP începe execuția subrutinei de servire asociată noii cereri, întrerupând astfel programul curent și actualizând registrul de serviciu. După terminarea execuției subrutinei de servire se reia ultimul program întrerupt. De asemenea, după comutarea contextului, UCP trebuie să revalideze întreruperile pentru a permite unor eventuale cereri de întrerupere prioritare să fie luate în considerare. Cererile de întrerupere ale dispozitivelor individuale de I/O, pot fi activate sau dezactivate prin program, prin intermediul unor bistabili de mască, asamblați de obicei într-un registru de mascare. Astfel că la momente diferite de timp, prin aplicarea unor măști diferite, programul poate modifica ordinea de (prioritate de) servire a cererilor

de întrerupere.

Arbitrarea controlată de circuit controler de întreruperi, este cea mai des întâlnită la microcalculatoarele moderne. Controlerul de întreruperi programabil (PIC³) este un circuit specializat care preia, în principiu, toate sarcinile pe care le avea UCP la arbitrarea prin software: acceptă cereri de întrerupere de la mai multe surse, determină dacă există cel puțin o cerere de întrerupere cu prioritate superioară celei a programului curent, iar în caz afirmativ, generează către UCP o cerere de întrerupere. La primirea confirmării acceptării cererii de întrerupere, circuitul PIC generează pe magistrala de date vectorul de întrerupere asociat celei mai prioritare cereri existente. UCP întrerupe programul în curs și servește cererea prin mecanismul descris anterior. Circuitul controler, este numit și "programabil", pentru că poate funcționa în mai multe moduri de lucru programabile prin înscrierea de către UCP a codurilor corespunzătoare modurilor de lucru în registrele de control ale PIC. De obicei programarea modului de lucru se face printr-o secvență de inițializare, executată de UCP, la pornirea calculatorului. Modul de lucru poate însă să fie modificat ulterior, tot prin program. Circuitul PIC este văzut de UCP ca un circuit de interfață ce cuprinde mai multe porturi de I/O. Modul de alocare a priorităților poate fi:

1. fix
2. rotitor. Acest mod implementează "politețea" în servirea cererilor de întrerupere. Inițial prioritățile sunt ordonate în ordine descrescătoare, dar după servire, fiecare nivel de prioritate devine automat cel mai puțin prioritar.
3. cu mascare. În acest caz prioritățile pot fi modificate dinamic, masca invalidând temporar anumite niveluri de prioritate.

Arbitrarea realizată prin PIC asigură un timp mic de răspuns pentru întreruperile hardware și o flexibilitate mare în alocarea sau modificarea priorităților. În plus, multe dintre circuitele de tip PIC pot fi cascadeate, ceea ce permite extinderea ușoară a sistemului într-un calculator.

Metoda de *arbitrare descentralizată prin lanț de priorități* a întreruperilor presupune existența în fiecare dispozitiv întreruptor a unei logici capabile să asigure protocolul electric de tratare a întreruperilor. Dispozitivele întreruptoare se conectează într-un "lanț de priorități" (daisy chain), ca în figura 7.7.

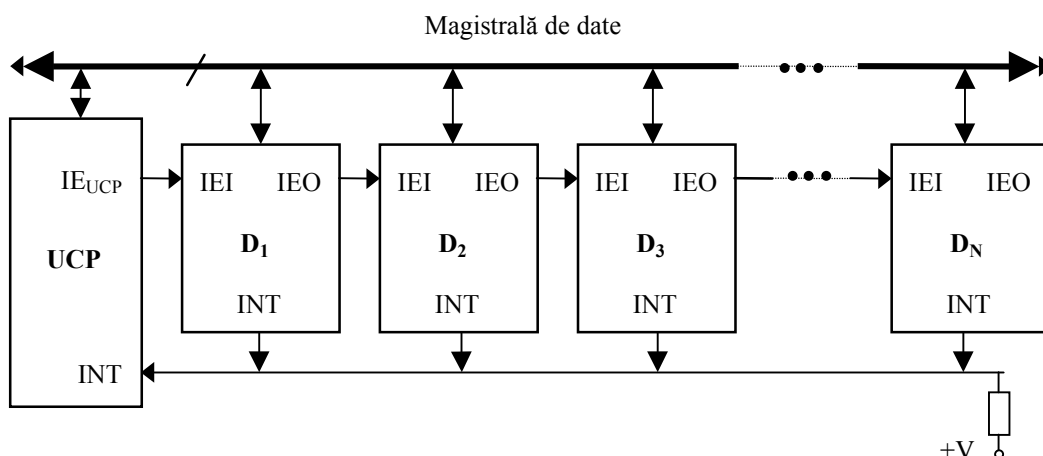


Figura 7.7. Exemplu de arbitrare a cererilor multiple de întrerupere prin lanț de priorități

În figură fiecare dispozitiv D_i , $i = 1, N$ dispune de câte o ieșire \overline{INT} (cu colector / drenă în

³ Programmable Interrupt Controller

gol), ceea ce permite legarea împreună a acestor ieșiri la linia de intrare \overline{INT} a UCP. Se realizează astfel un SAU cablat pe această intrare a UCP. (de fapt legând ieșirile împreună avem un SI cablat între ieșiri active în stare JOS, adică:

$$\overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \dots = \overline{A + B + C + \dots} \quad (7.1)$$

Lanțul de dispozitive conduce la priorități fixe, ce țin de poziția dispozitivului în lanț. Prioritatea cea mai mare este alocată celui mai apropiat dispozitiv față de UCP (D_1), iar prioritate minimă o are ultimul dispozitiv din lanț.

UCP generează semnalul de validare a întreruperilor externe (IE_{UCP}) care se propagă prin lanț, trecând prin celulele succesive ale lanțului. Pentru fiecare dispozitiv, starea intrării de validare IEI (Interrupt Enable Input) este copiată la ieșirea IEO (Interrupt Enable Output) a fiecărui dispozitiv, dacă dispozitivul respectiv nu cere întrerupere către UCP. Dacă un dispozitiv a cerut întrerupere, care a fost recunoscută și se face tratarea acesteia, pe tot timpul servirii între intrarea IEI și ieșirea IEO ale dispozitivului servit nu mai există egalitate. Pe toată această perioadă dispozitivul aduce IEO la zero pentru a nu permite dispozitivelor cu prioritate mai mică să-i întrerupă rutina de tratare. În momentul în care un dispozitiv D_i cere întrerupere, pot exista două situații:

- ⇒ dacă $IEI_i=0$, atunci un dispozitiv mai prioritar decât D_i a cerut întrerupere și servirea sa de către rutina corespunzătoare nu s-a încheiat. Ca urmare dispozitivul D_i va aștepta până când IEI_i trece în stare activă "1".
- ⇒ dacă $IEI_i=1$, atunci D_i poate genera cererea de întrerupere către INT (aducând această linie la zero logic); aceasta pentru că D_i este dispozitivul cel mai prioritar care cere întrerupere la momentul respectiv de timp. Ca urmare, după lansarea cererii de întrerupere, IEO_i devine zero, invalidând cererile de întrerupere de la dispozitivele cu prioritate mai mică din lanț.

Din cele de mai sus rezultă că dacă a apărut o cerere de întrerupere către UCP, în lanț, un singur dispozitiv are $IEI = 1$ și $IEO = 0$, și anume cea servită. Pentru toate celelalte dispozitive ieșirea IEO are valoare logică identică cu intrarea IEI. Dacă UCP acceptă cererea de întrerupere, dispozitivul întreruptor depune pe magistrala de date a sistemului vectorul de întrerupere, pentru a fi identificat și servit corespunzător.

7.4.3. Transferul prin acces direct la memorie

Transferul, prin intermediul UCP, a blocurilor mari de date între memoria principală și periferice, se face relativ lent, pentru că de fiecare dată informația trece și prin registre ale UCP. În plus pentru o cantitate mare de date transferate între memorie și periferic UCP va consuma foarte mult timp.

Transferul prin acces *direct la memorie* (DMA - Direct Memory Acces) este executat direct între memoria principală și dispozitivul periferic, fără ca datele să mai treacă prin UCP, sub comanda unui circuit controler DMA, care controlează temporar magistralele sistemului.

În figura 7.8. se presupune că inițierea transferului se face de către periferic (prin intermediul interfeței) care efectuează o cerere de transfer prin acces DMA spre circuitul controler de DMA (DMAC). Acesta solicită de la UCP controlul magistralelor prin semnalul de *cerere de control a magistralelor*, BR , (Bus Request). Cu o mică întârziere (la sfârșitul ciclului mașină curent) UCP cedează controlul magistralelor, își trece ieșirile către acestea în HiZ (stare de înaltă impedanță) și informează despre aceasta prin semnalul *acordare a controlului magistralelor*, BG , (Bus Grant). Circuitul controler DMAC furnizează adresele pe magistrala de adrese, preia controlul semnalelor de scriere (WR) și citire (RD) și trimite către periferic semnalul de acceptare a transferului prin

DMA. Când dispozitivul I/O primește acest semnal de acceptare, el pune un cuvânt pe magistrala de date (pentru scriere în memoria principală) sau citește un cuvânt de pe magistrala de date (pentru citire din memoria principală).

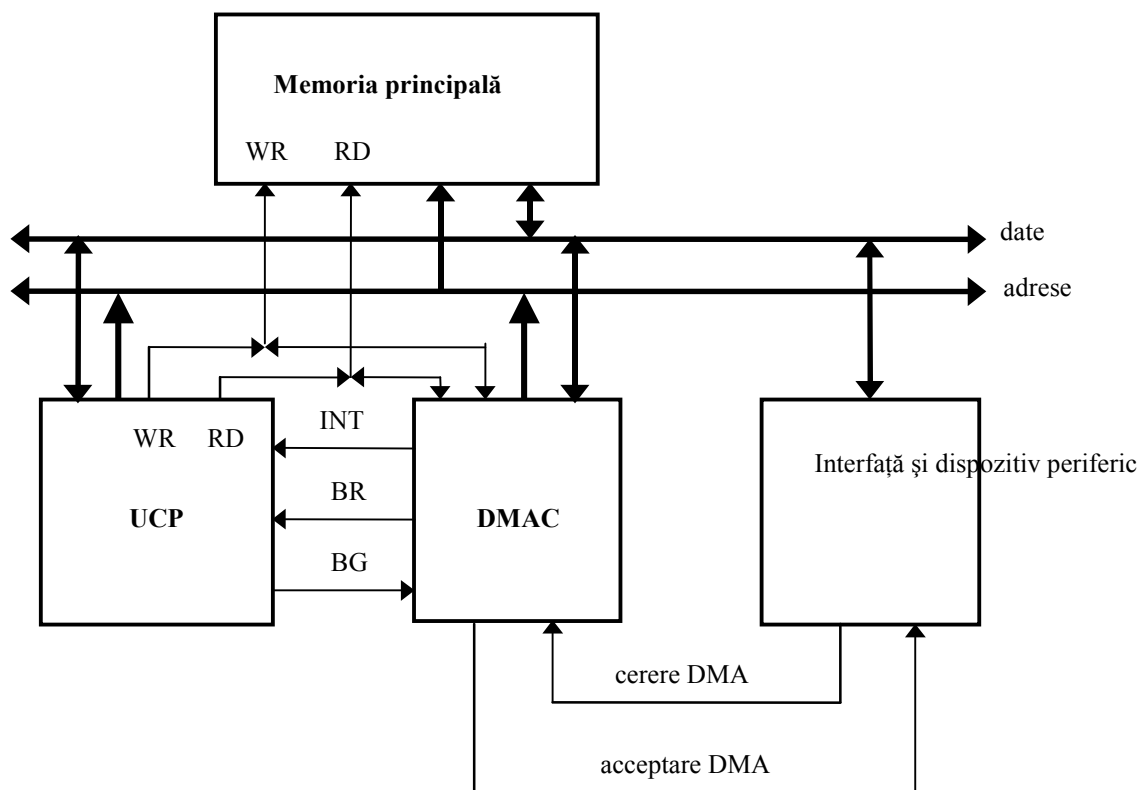


Figura 7.8. Exemplu de transfer DMA (DMAC = Controller DMA)

Transferul de date se va face *direct* între periferic și memoria principală, fără ca UCP să mai fie intermediar al acestui transfer (așa cum se întâmpla la transferul prin program și prin întreruperi). Un *ciclu DMA* reprezintă transferul unui cuvânt din sau în memorie. După terminarea unui ciclu, controllerul DMA poate continua cu alte cicluri DMA (dacă a fost programat să facă acest lucru și dacă cererea DMA de la periferic se păstrează activă), sau poate returna controlul magistrelor către UCP prin inactivarea semnalului BR. Pentru transfer *pe cuvânt*, secvența de semnale de cerere DMA (cerere → BR → BG → acceptare) conduce la desfășurarea unui singur ciclu DMA. Pentru un alt ciclu DMA, secvența de semnale trebuie repetată și abia apoi se transferă un nou cuvânt. Cealaltă variantă de transfer, în *mod rafală-bloc*, presupune transferul mai multor cuvinte pentru fiecare cerere de transfer DMA. Este un mod de mare viteză, care însă dacă controllerul și UCP sunt legate la aceleași magistrale poate menține procesorul în inactivitate pentru o perioadă de timp. În cazul transferului pe cuvânt, întreruperea poate fi insesizabilă, ca timp, pentru procesor, și de aceea uneori acest mod de transfer este numit "*cu furt de ciclu*".

De obicei circuitele controler DMA pot gestiona transferuri cu mai multe periferice simultan, pentru fiecare periferic existând (intern controllerului) un *canal DMA*.

Conform funcțiilor pomenite pe scurt mai sus, structura controllerului DMA trebuie să cuprindă:

- **Logică de comandă și sincronizare.** Modul de lucru al controllerului DMA este programabil. De aceea el este inițializat printr-o rutină de inițializare rulată de UCP, fixându-se astfel: modul de transfer (cuvânt sau bloc-rafală), sensul transferului (la memoria principală sau de

la memoria principală), nivelul activ al semnalelor de interfață cu echipamentul periferic, prioritățile acordate canalelor, modul de tratare al sfârșitului de transfer etc. Circuitul controller DMA are de asemenea o interfață cu UCP, care permite inițializarea registrelor controllerului, comanda ulterioară de către UCP a controllerului, precum și realizarea protocolului BR/BG;

- *Circuite tampon* pentru conectarea la magistralele calculatorului;
- *Logică de arbitrare a priorităților canalelor*;
- *Logică specifică fiecărui canal DMA*, care cuprinde cel puțin următoarele componente:
 - ⇒ un registru *numărător de adrese* care generează adresa curentă (din memoria principală) de transfer;
 - ⇒ un *numărător de cuvinte de transferat*, care se decrementează la fiecare ciclu DMA efectuat (valoarea inițială a acestui contor este egală cu numărul de cuvinte ce se dorește a fi transferat. Atunci când conținutul a ajuns la zero, ciclul de transfer s-a sfârșit);
 - ⇒ un *registru de stare a canalului*, care poate fi citit de UCP și care indică: sensul transferului prin canal, canal activat / dezactivat, prioritatea alocată etc.

Transferul prin DMA prezintă avantaje din punctul de vedere al vitezelor mari de transfer, pentru blocuri mari de date. Este un transfer folosit în aplicații de genul: transfer cu discurile magnetice, transfer cu plăci periferice ce conțin convertoare AD sau DA rapide etc.

BIBLIOGRAFIE

1. [Borcoci95] Borcoci E., Zoican S., Popovici E., Arhitectura microprocesoarelor, partea I, Ed. Media Publishing, București, 1995.
2. [Burileanu94] Burileanu, C., Arhitectura microprocesoarelor, Editura DENIX, București, 1994;
3. [Crutu87] Crutu, Gh., Romanca, M., Fratu, A., Calculatoare, micro sisteme de calcul, Universitatea din Brasov, 1987;
4. [Dodescu80] Dodescu, Gh., Ionescu, D., Misdolea, T., Nisipeanu, L., Pilat, F., Sisteme electronice de calcul și teleprelucrare, Ed. Didactică și Pedagogică, București, 1980;
5. [Furht87] Furht, B., Milutinovic, V., A survey of microprocessor architecture for memory management, IEEE Computer, March 1987, vol.20, no3, pp. 48-66.
6. [Hayes88] Hayes, J., Computer Architecture and Organisation, McGraw Hill Comp., 1988.
7. [Lupu86] Lupu, C., s.a., Microprocesoare, Ed. Militară, București 1986;
8. [Mano93] Mano, M., Computer System Architecture, Prentice-Hall Inc. 1993.
9. [MDE72] Mic dicționar enciclopedic, Editura enciclopedică română, București, 1972
10. [Nicula97] Nicula, D., Arhitecturi de microprocesoare de performanțe ridicate, Teză de doctorat, Universitatea TRANSILVANIA Brașov, 1997
11. [Patterson90] Patterson, D., Hennessy, J., Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers, Inc. 1990;
12. [Patterson96] Patterson, D., Hennessy, J., Computer Architecture - A Quantitative Approach, second edition, Morgan Kaufmann Publishers, Inc. 1996;
13. [Patterson94] Patterson, D., Hennessy, J., Computer Organization & Design, the Hardware Software Interface, Morgan Kaufmann Publishers, Inc. 1994;
14. [Pfaffenberger96] Pfaffenberger, B., Dicționar explicativ de calculatoare, Ed. Teora, București, 1996
15. [Pop2000] <http://www.ida.liu.se/~paupo>, Slides for Advanced Computer Architecture, Paul Pop, Institutionen för Datavetenskap, Linköpings Universitet
16. [SPEC] The Standard Performance Evaluation Corporation, <http://www.spec.org>
17. [Stallings00] Stallings, W., Computer Organization and Architecture, 5th edition, Prentice Hall International, Inc., 2000.
18. [Stefan91] Ștefan, Gh., Funcție și structură în sistemele digitale Ed. Academiei Române, 1991;
19. [Stefan83] Ștefan, Gh., Drăghici, I., Mureșan, T., Barbu, E., Circuite integrate digitale, Ed. Didactică și Pedagogică, București 1983;
20. [Strugaru92] Strugaru, C., Popa, M., Microprocesoare pe 16 biți, Editura TM, Timișoara 1992;
21. [Sztojanov87] Sztojanov, I., Borcoci, E., Tomescu, N., Bulik, D., Petrec, M., Petrec, C., De la poarta TTL la microprocesor, Ed. Tehnică, București, 1987;
22. [Tanenbaum99] Tanenbaum, A., Organizarea structurată a calculatoarelor, ediția a IV-a, Computer Press AGORA, Tg. Mureș, 1999.
23. [Toacse85] Toacse, Gh., Introducere în microprocesoare, Ed. Științifică și Enciclopedică, 1985;
24. [Toacse96] Toacșe, Gh. Nicula, D. Electronică digitală, Ed. Teora, 1996
25. [Weems96] Weems, Ch. Jr., Computer Science Course 635, Notes from Lecture 3, at www.cs.umass.edu/~weems/index.html