ARHITECTURA SISTEMELOR DE CALCUL

Note de curs

Dr. Ing. Inf. Marius Gh. ROGOBETE

București 2021

Cuprins

1. Evoluția arhitecturii calculatoarelor

Un sistem de calcul poate fi definit (conform The American Heritage Dictionary of the English Language, Fifth Edition, 2016) ca:

- dispozitiv care lucrează automat, sub controlul unui program memorat, prelucrând date în vederea producerii unor rezultate ca efect al procesării;
- dispozitiv care efectueaza calcule, în special o mașină electronică programabilă care execută operații aritmetice, logice sau care asamblează, stochează, corelează sau efectuează un alt tip de procesare a informației, cu viteză ridicată.

Funcțiile de bază ale unui sistem de calcul (SC) sunt:

- procesarea de date;
- memorarea de date;
- transferul de informații;
- controlul tuturor componentelor SC.

1.1. Clasificarea arhitecturii sistemelor de calcul

Sistemele de calcul se pot clasifica uzual din punct de vedere a puterii de calcul și din punct de vedere al utilizării.

Clasificarea după puterea de calcul este una dinamică, fiind în continuă schimbare datorită evoluției spectaculoase a tehnologiei de procesare:

- Supercalculatoare - sisteme de calcul considerate la momentul apariției drept cele mai performante din lume în ceea ce privește viteza de procesare a datelor;
- Mainframe-uri - mașini multiprocesor, de asemenea cu putere mare de procesare, neorientate însa spre un task precis ci mai degrabă aplicațiilor critice, prelucrărilor simple asupra unui volum mare de date, salvarea și backup-ul acestor date;
- Minicalculatoare - termen folosit în anii 60 și 70 pâna la apariția microcalculatoarelor. Sisteme de calcul de cost relativ redus - tot ce era inferior unui mainframe și unui supercalculator, atât ca putere de procesare cât și ca dimensiune fizica, destinate universitaților, ramuri ale industriei, etc;
- Microcalculatoarele - sisteme de calcul bazate pe folosirea unui microprocesor (de unde și numele), aparute la sfârsitul anilor 70, începutul anilor 80, cost redus, destinate în principal utilizatorului domestic sau companiilor.

Din punctul de vedere al utilizării:

- Stații de lucru (workstations) - de obicei calculatoare din familia microcalculatoarelor (calculatoarele personale spre exemplu) cu putere de procesare medie, capabilități grafice și multimedia ridicate, de obicei conectate la Internet;
- Server-e - oferă diferite servicii stațiilor (clienților). Din punct de vedere hardware un server poate rula atât pe un microcalculator (calculator personal) cu putere de procesare mai ridicată cât și pe arhitecturi hardware dedicate acestui scop (mainframe-uri sau supercalculatoare);
- Microdispozitive (embedded devices) - dispozitive cu putere de calcul relativ redusă, dotate cu un procesor și cu o funcționalitate dedicată unui anumit scop. Exemple: telefoane mobile, PDA, MP3 player-e, GPS-uri, DVD player-e, etc. Aproximativ 80% din procesoarele produse în acest moment sunt dedicate microdispozitivelor.

1.2. Structura fizică a unui calculator, tipuri de calculatoare, exemple de arhitecturi uniprocesor și multiprocesor

Un sistem de calcul (SC) este structural format din:

- hardware - partea de echipamente:
- unitatea centrala de procesare (Central Processing Unit – CPU);
- memoria;
- dispozitivele periferice;
- software - partea de programe:
- soft sistem (aplicații destinate sistemului de calcul și sistemului de operare);
- soft utilizator (restul aplicațiilor);
- firmware - partea de microprograme.

Arhitectura unui sistem de calcul se refera la acele atribute ale sistemului care sunt vizibile programatorului și care au un impact direct asupra executiei unui program:

- setul de instrucțiuni mașină;
- caracteristicile de reprezentare a datelor;
- modurile de adresare;
- sistemul de intrare / ieșire (I/O).

Mulțimea instrucțiunilor mașină (Instruction Set Arhitecture – ISA) este interfața cheie între nivelele de abstractizare, fiind interfata dintre hard și soft. Aceasta permite unor siteme de calcul diferite să ruleze soft identic, caz în care vorbim despre calculatoare compatibile, de exemplu calculatoare compatibile IBM-PC (în prezent bazate pe procesoare AMD sau Intel).
Aceste instrucțiunui sunt utilizate pentru :

- organizarea SC, modul de stocare a informatiei (registri, memorie);
- tipurile și structurile de date (codificări, reprezentări);
- formatul instrucțiunilor;
- setul de instrucțiuni (codurile operațiilor) pe care microprocesorul le poate efectua;
- modurile de adresare și accesare a datelor și instrucțiunilor;
- definirea condițiilor de excepție.

Componentele minimale ale unui sistem de calcul sunt următoarele:

- modulul de control;
- memoria;
- sistemul de intrare / ieșire (input/output);
- structuri de interconectare a componentelor de mai sus (magistrale);

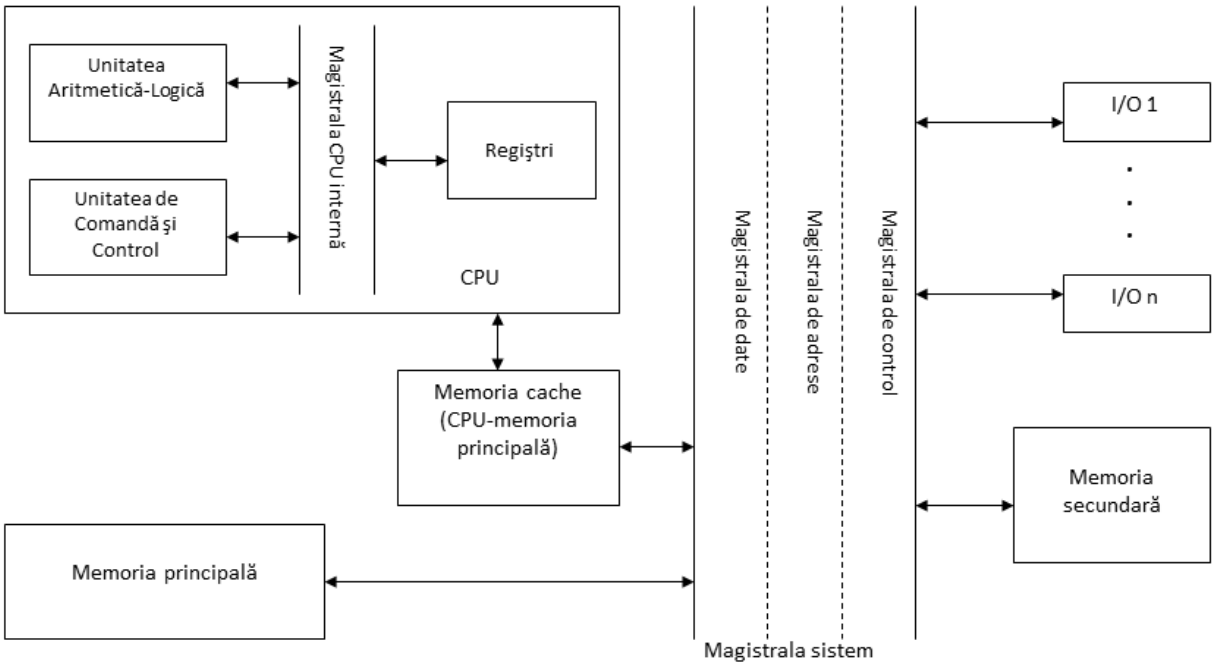


Figura 1.1 Arhitectura unui sistem de calcul.

1.2.1 Tipuri de calculatoare

Sunt două clasificărim date de Flinn și Wang a diverselor tipuri de arhitecturi, generate din arhitectura de baza von Neumann.

1.2.1.1. Clasificarea Flinn

Din punct de vedere conceptual functionarea unui sistem de calcul poate fi vazuta ca actiunea unui *flux de instrucțiuni* (care reprezintă programul) asupra *unui flux de date* (care reprezintă datele de intrare sau rezultate partiale).
Clasificarea Flinn, care are vedere gradul de multiplicitate al celor două fluxuri, identifica patru tipuri de arhitecturi și anume:

- **SISD (*S*ingle *I*nstruction Stream – *S*ingle *D*ata Stream);**
- **SIMD (*S*ingle *I*nstruction Stream – *M*ultiple *D*ata Stream);**
- **MISD (*M*ultiple *I*nstruction Stream – *S*ingle *D*ata Stream);**
- **MIMD (*M*ultiple *I*nstruction Stream – *M*ultiple *D*ata Stream).**

Pentru aceste arhitecturi trei tipuri de componente de sistem respectiv **UC** (*sectiunea de comandă a unității centrale),* **UP** *(sectiunea de prelucrare a unității centrale),* **MM** *(modulul de memorie),* cele două fluxuri fiind **FD** *(fluxul de date)* respectiv **FI** *(fluxul de instrucțiuni).*
Deși diferite între ele cele patru structuri respecta succesiunea evenimentelor specifice arhitecturii von Neumann. Instructiunile și datele sunt extrase din memorie, instrucțiunile sunt decodificate de **UC** care trimite secvența de instrucțiuni catre **UP** pentru execuție.

SISD

Arhitectura SISD realizeaza o execuție secvențiala a instrucțiunilor. De exemplu o înmultire

cu o constanta 3 a 100 de numere implica aducerea pe rând din memorie a celor 100 de numere și înmultirea lor cu respectiva constanta, fiecare rezultat fiind trimis în memorie pe masura efectuarii calculului.
Principalul neajuns al acestei arhitecturi consta în viteza de procesare care la rândul ei este determinată de fracventa ceasului. Este evident ca nici o realizare tehnologica nu va putea face perioada ceasului nula. În consecinta modul strict secvențial de tratare a operațiilor impus de
arhitectura von Neumann plafoneaza la un moment dat viteza de procesare. Aceasta situatie este cunoscuta sub numele *gâtul sticlei lui Neumann(Neumann Bottleneck).*

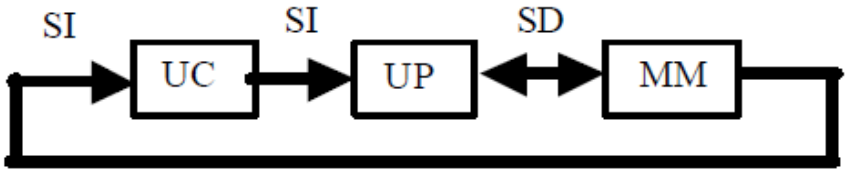


Figura 1.2. Arhitectura SISD

Această limitare este depășită prin introducerea arhitecturilor de tip neserial (*respectiv arhitecturile paralele*).

SIMD

Arhitectura SIMD, prezinta urmatoarele caracteristici:

- există mai multe UP, sub forma *procesoarelor de date*, datorita fluxului de date multiplu, preluat din memoria partajata MP;
- există o singura UC, sub forma *procesorului de instrucțiuni*, care supervizeaza procesoarele UP;
- toate UP primesc setul unic de instrucțiuni și proceseaza fiecare un anume flux de date (UPi proceseaza SDi);
- masinile SIMD pot efectua procesari pe cuvânt (*word-slice)* sau pe bit *(bit – slice).*

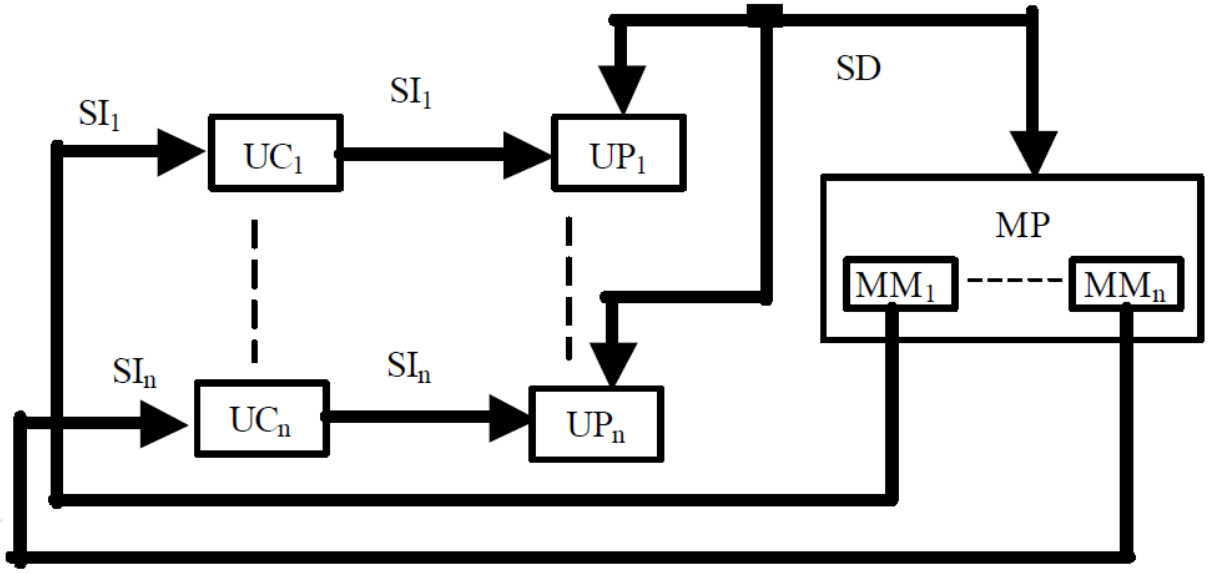


Figura 1.3. Arhitectura SIMD.

Considerând exemplul precedent cele 100 de numere vor fi înmultite simultan cu constanta 3 iar rezultatele vor fi stocate în partitiile MMi ale memoriei. Aceste tipuri de masini lucreaza foarte bine pe seturi de date formate din matrice de dimensiuni foarte mari atunci cand asupra fiecarei date este necesar a se efectua aceiasi operatie. Principala limitare a masinilor SIMD este de natura economica deoarece ariile de procesoare nu sunt componente standard și prin urmare aceste masini sunt unicate și costă foarte mult.

MISD

Arhitectura MISD, numita *macro-pipe-line* prezinta urmatoarele caracteristici:

- fiecare UC lucreaza cu sirul propriu de instuctiuni SI1, SI2, …SIn;
- fiecare UP primeste instrucțiuni diferite, însa opereaza asupra aceluiasi sir de date SD (care suporta în consecinta mai multe prelucrari).

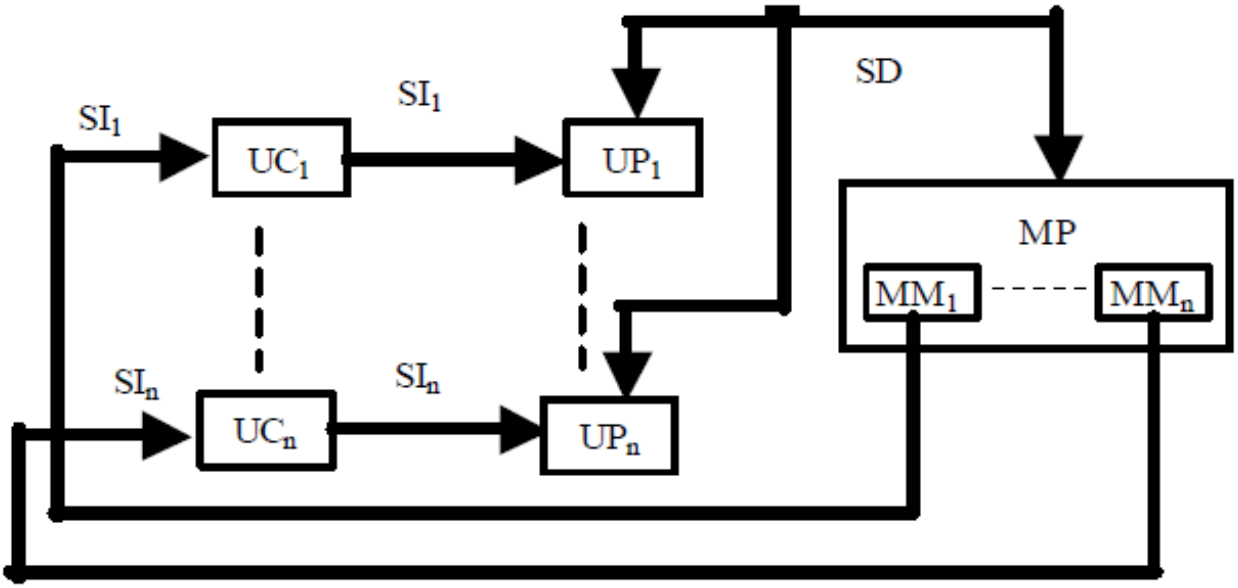


Figura 1.4. Arhitectura MISD.

Desi aceasta mașină este posibil de realizat din punct de vedere teoretic, nu a fost niciodata fabricata în scop comercial, având în prezent doar o valoare teoretica.

MIMD

Arhitectura MIMD realizeaza o prelucrare paralela prin lansarea în execuție a mai multor instrucțiuni în acelasi timp pe diferite seturi de date. În afara elementelor prezentate în figura sunt necesare elemente aditionale de control care să repartizeze instrucțiunea corecta și data respectivă la procesorul ales (simultan la toate procesoarele).

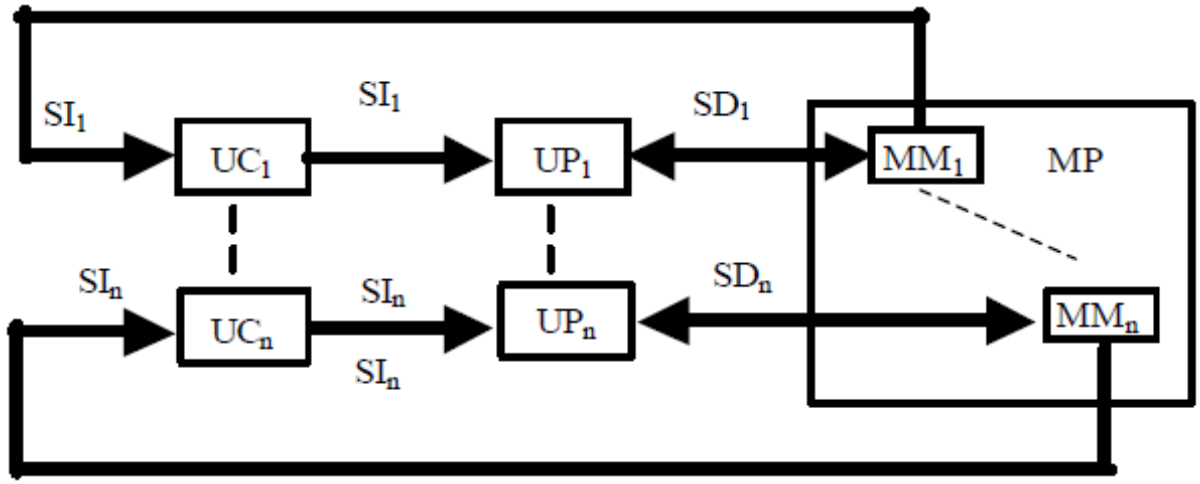


Figura 1.5. Arhitectura MIMD.

Principial există două tipuri de arhitecturi MIMD și anume:

- *shared memory* (intrinseci) dacă spatiul de memorie este accesat în comun de toate cele *n* procesoare;

- *shared nothing* (independente) dacă fiecare procesor are propria memorie.

1.2.1.2. Clasificarea Wang

Aceasta clsificare presupune o organizare matriceala a datelor. O matrice de dimensiune *mxn* presupune existenta a *m* cuvinte, fiecare cuvânt cu lungimea de *n* biți.
Criteriul este reprezentat de gradul de paralelism în procesarea datelor organizate matriceal. Conform acestui criteriu există patru tipuri de arhitecturi și anume:

- **WSBS** (***W**ord **S**erial – **B**it **S**erial)* – se lucreaza pe un singur cuvânt, fiecare cuvânt fiind prelucrat bit cu bit, respectiv *ns1, ms1;*
- **WSBP** (***W**ord **S**erial – **B**it **P**aralel)* – se lucreaza pe un singur cuvânt, bitii fiecarui cuvânt fiind prelucrati simultan, respectiv *n>1, ms1;*
- **WPBS** (***W**ord **P**aralel – **B**it **S**erial)* – se lucreaza pe un singur bit la toate cuvintele simultan, respectiv *ns1, m>1;*
- **WPBP** (***W**ord **P**aralell – **B**it **P**aralel)* – se lucreaza simultan pe toate cuvintele și pe toti bitii fiecarui cuvânt fi, respectiv *n>1,m>1.*

Structura **WPBP** este complet paralela fiind orientata pe prelucrari de matrice *mxn*, în timp ce structurile **WSBP** și **WPBS** sunt partial paralele fiind orientate pe prelucrari vectoriale (WSBP – orizontala *1xn*, WPBS – verticala *mx1).* În ceea ce priveste arhitectura **WSBS** aceasta nu are
elemente de paralelism.

1.2.2. Arhitecturi de sisteme multiprocesor

Există două categorii de sisteme multiprocesor : arhitectura centralizată și cu arhitectură distribuită.

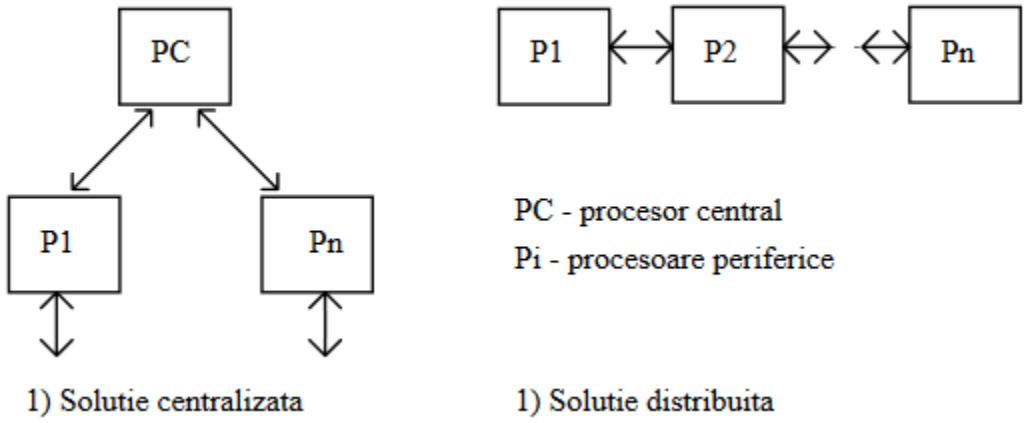


Figura 1.6. Arhitecturi multiprocesor

Soluţia centralizatăeste utilizatăpentru sisteme complexe. Este necesar un mecanism de intercomunicare între procesoare (realizat software sau hardware) care limitează performanţele sistemului.
Pentru soluţia distribuită deciziile se iau local de către procesoarele periferice. Mecanismulde intercomunicare (uzual realizat software) este mai simplu. Este necesară divizarea funcţiilor sistemului în subfuncţii bine determinate care sînt aribuite procesoarelor locale. În practică se utilizează şi soluţii mixte, cu un procesor central și mai multe procesoare locale.

Sistem multiprocesor cu 2 procesoare cu memorie comună

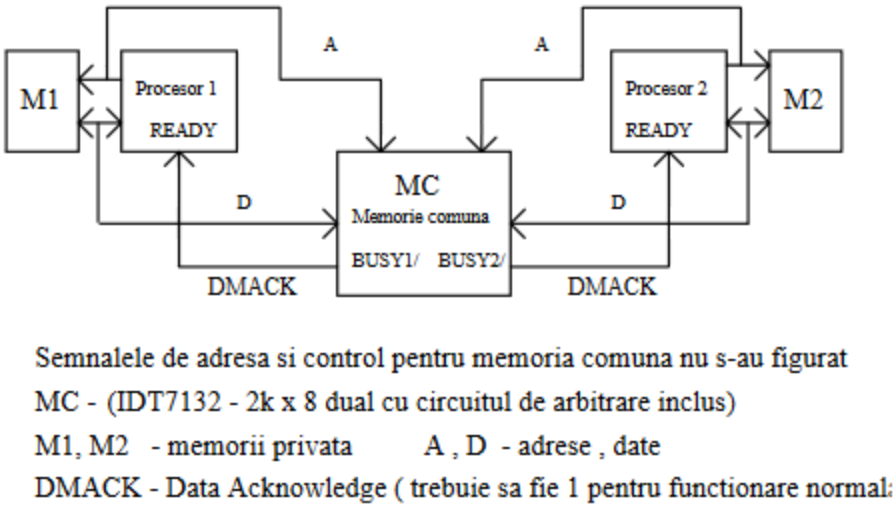


Figura 1.7. Sistem mutiprocesor cu memorie comună

1.3. Arhitectura lui Von Neumann.

Calculatoarele digitale convenționale au o bază conceptuală comună care îi este atribuită lui von Neumann. Modelul von Neumann constă în cinci componente majore:

- unitatea de intrare furnizează instrucțiuni și date sistemului, ce sunt stocate ulterior în
- unitatea de memorie. instrucțiunile și datele sunt procesate de
- unitatea aritmetică și logică (ULA) sub controlul
- unității de control , iar rezultatele sunt trimise la
- unitatea de ieşire.

ULA şi UC poartă denumirea generică de CPU (Unitate Centrală de Procesare).

Programul stocat este cel mai important bloc al modelului von Neumann. Un program este stocat în memoria calculatorului împreună cu datele ce sunt procesate. Înainte de apariția calculatoarelor cu program stocat, programele erau stocate pe medii externe cum ar fi cartele perforate. În calculatorul cu program stocat acesta poate fi manipulat ca și cum ar reprezenta date. Aceasta a dus la apariția compilatoarelor și sistemelor de operare și face posibilă marea versatilitate a calculatoarelor moderne.

Ca urmare, caracterizarea arhitecturii von Neuman se face prin :

- utilizarea memoriei interne pentru a stoca secvențe de control pentru îndeplinirea unei anumite sarcini – secvențe de programe;
- datele, cât și instrucțiunile sunt reprezentate ca siruri de biti și sunt stocate într-o memorie read-write;
- conținutul memoriei se poate accesa în funcție de locatie (adresa), indiferent de tipul informatiei conținute;
- execuția unui set de instrucțiuni se efectueaza secvențial, prin citirea de instrucțiuni consecutive din memorie.

1.3.1. Unitatea centrală (Central Processing Unit – CPU)

Funcțiile unui CPU sunt:

- obtinerea instrucțiunilor care trebuie executate;
- obtinerea datelor necesare instrucțiunilor;
- procesarea datelor (execuția instrucțiunilor);
- furnizarea rezultatelor obtinute.

Componentele de baza ale unui CPU sunt :

- Unitatea Aritmetica-Logica (Arithmetic Logic Unit – ALU);
- Unitatea de Comanda și Control (Control Unit – CU) – decodifica instrucțiunile (FETCH/ DECODE/ READ MEMORY/ EXECUTE/ STORE);
- regiștrii – aceștia sunt dispozitive de stocare temporară a datelor și informatiilor de control (instrucțiunile), de capacitate mica și viteza de acces mare;
- magistrale interne CPU – dispozitive pentru comunicare între componentele CPU și comunicare cu exteriorul, pentru transferul de informatii.

Ceasul sistem

Fiecare procesor (CPU) conține un ceas intern care produce și trimite semnale electrice pe magistrala de control pentru a sincroniza operațiile sistemului. Semnalele alterneaza valori 0 și 1 cu o anumita frecvența. Frecvența cu care se alternează aceste valori se numește ciclu de ceas sau perioada ceasului (clock cycle).

Ciclu de ceas este cea mai mica unitate de timp sesizabilă de catre un procesor.
Frecventa de ceas este numărul de cicluri de ceas pe secunda.

Exemplu: ceasul unui procesor la 300 de Mhz ticaie de exact 300.000.000 ori pe secunda. Un ciclu de ceas al unui astfel de procesor are o durata de 1 / 300.000.000 secunde.

Viteza de calcul

Numărul de cicluri pe instrucțiune (Cycles per Instruction : CPI) determină viteza de procesare. Fiecare instrucțiune dureaza un anumit număr de cicluri de ceas (exemplu: instrucțiunea MOV dureaza intre 2 și 14 cicluri de ceas în funcție de natura operanzilor).
Un procesor ruland la 1400 Mhz executa o aceeasi instrucțiune mai repede decat unul rulând la 800 Mhz – durata ciclului de ceas fiind mai scurtă.

Benchmarking-ul dintre viteza de calcul a procesoarelor din familia x86 versus celor din familia RISC (set redus de instrucțiuni) demonstrează diferența dintre instrucțiunile complexe care dureaza mai multe cicluri vs instrucțiuni simple, primare care se execută rapid.
În prezent s-a atins o anumita limită în ceea ce privește viteza procesoarelor. Mecanisme noi de creștere a vitezei pot fi:

- Pipelining - paralelizarea instrucțiunilor (o instrucțiune trecuta din faza FETCH în faza DECODE, permite unui alte instrucțiuni să treaca în faza FETCH) deci mai multe instrucțiuni se executa în paralel per ciclu de ceas - Instructions per Cycle;
- Cresterea numarului de nuclee (cores) per procesor (dual core, quad core, octal core, etc).

Viteza unui sistem de calcul este influențată de :

- Frecventa procesorului (singura nu e concludenta, vezi Intel vs AMD);
- Capacitatea maxima de memorie care poate fi adresata;

- Capacitatea de paralelizare (pipelining);
- Dimensiunea registrilor interni și a magistralei de date;
- Dimensiunea memoriei CACHE.

Aceasta se masoară în MIPS – milioane de instrucțiuni (intregi) pe secundă sau MFLOPS – milioane de instrucțiuni în virgulă flotantă pe secundă.

Procesor	Frecvență	MIPS
Intel Pentium Pro	200 Mhz	541
AMD Athlon	1.2 Ghz	3.561
Ultra SPARC Niagara 2	1.4 Ghz	22.400
……	……	……
Intel Polaris Prototype (80 nuclee)	5.6 Ghz	1.800.000

Magistrala de date

Reprezintă din punct de vedere hardware mărimea unei locații de memorie ce poate fi accesată de magistrala de date (de exemplu Pentium are o magistrala de date pe 64 biti = 64 linii de date, astfel ca la fiecare "memory cycle" procesorul poate accesa 8 octeți din memorie).
Din punct de vedere software este dimensiunea unui cuvânt de memorie (dimensiunea registrilor CPU).
În plus, cu cât dimensiunea cuvantului de memorie este mai mare, operațiile cu numerele intregi se desfășoară mai rapid (incercati să inmultiti un double cu alt double folosind doar registrii pusi la dispozitie de procesorul 8086).

1.4. Principiile de funcţionare ale maşinii Turing.

O contribuție importanta la dezvoltarea stiintei calculatoarelor a avut-o matematicianul englez ***Alan Turing*** care în 1936 legat de conceptul de *numere calculabile* a sintetizat un automat matematic abstract capabil să opereze cu astfel de numere (numerele calculabile sunt acele numere a caror parte zecimala se poate determină cu un număr finit de iteratii).
Automatul a fost sintetizat pe baza urmatoarelor ipoteze:

a) automatul are un număr *n* finit de stari;
b) automatul se afla în orice moment într-o stare *i* , unde $l \leq i \geq n$, urmând ca la momentul imediat urmator să se afle într-o stare *j*, unde $l \leq j \geq n$,;
c) fiecare dintre cele *n* stari este caracterizată prin urmatoarele informatii:
   - valoarea caracteristicii e_i , care este o valoare curenta a numarului care se calculeaza;
   - funcția f_j care aplicata starii e_i permite obtinerea urmatoarei stari e_j;
   - deplasamentul d_{ij} care va trebui aplicat numarului pentru a se realiza tranzitia din swtarea *i* în starea *j*, respectiv $j = i + d_{ij}$.

Pentru modelul sau abstract *Turing* a propus modelul functional din figura 1.2 unde SR este sistemul reprezentor; P este procesorul iar C/S este capul de citire / scriere.

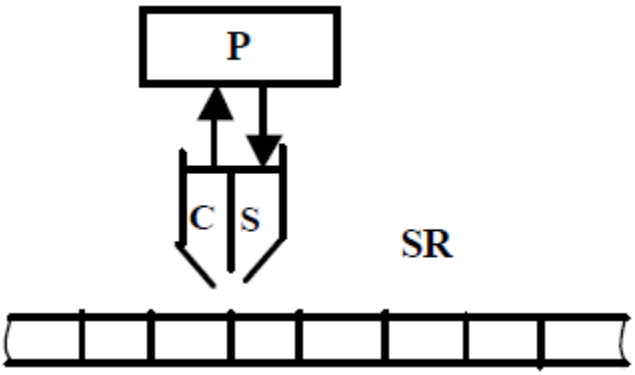


Figura 1.2. Modelul funcțional al mașinii Turing

*Sistemul reprezentor (*sau *memoria* mașinii) este constituit dintr-o banda magnetica de lungime infinita (fara capete) împartita în segmente de lungime egala, fiecare segment putând stoca un număr finit de semne.
Procesorul este un circuit secvențial cu un număr finit de stari, care poate executa urmatoarele instrucțiuni (care practic se constituie în setul de instrucțiuni al masinii):
- schimba simbolul de pe banda de la pozitia curenta;
- pozitioneaza capul de citire cu o pozitie la dreapta;
- pozitioneaza capul de citire cu o pozitie la stânga;
- opreste sistemul.

Capul de citire/scriere poate citi respectiv înscrie informatie de pe /pe banda magnetica.
Pentru a realiza un calcul cu aceasta mașină, se înscriu datele într-un mod convenabil și se descompune algoritmul de calcul în funcție de modul de reprezentare a datelor într-o secvența de instrucțiuni ale masinii. Ultima instrucțiune este cea de oprire a masinii, la care banda trebuie să conțină reultatul calculului.

2. Clasificarea semnalelor

Noţiunea de semnal este o noţiune centrală în electronică și telecomunicaţii.
Un *semnal* este o mărime fizică purtătoare de informaţie. Cel mai adesea, este o funcție scalară de variabila timp, ca în exemplele uzuale următoare:

- Tensiunea sau curentul furnizate de un traductor de temperatură
- Tensiunea de la intrarea unui amplificator de putere
- Tensiunea de la ieşirea modulului *tuner* radio
- Tensiunea de la bornele microfonului
- Cîmpul electromagnetic produs în antena telefonului mobil (la emisie sau la recepție)
- Presiunea aerului în sistemele pneumatice de măsurare şi comandă a proceselor (se foloseşte în mediile cu potenţial de explozie)
- Poziţia deschis-închis a releului electromagnetic cu care se comandă funcţionarea centralei termice
- Succesiunea de valori afişate de ecranul unui voltmetru digital (numeric)
- Poziţia pedalei de acceleraţie, transmisă către carburator.

Semnalul în varianta înregistrată (memorată), se foloseşte în scopul reconstituirii informaţiei iniţiale sau în scopul prelucrării. Exemple:

- Înregistrarea vocii pe bandă de magnetofon
- Înregistrarea vocii de pe un CD
- Înregistrarea numerică a tensiunii afişate pe ecranul osciloscopului
- Înregistrarea numerică a vitezei vîntului într-un punct
- Înregistrarea cursului valutar pe un interval de timp

Există și alte variante de semnale, cu alte variabile sau altă dimensiune a funcției, cum ar fi:

- Denivelarea unui material aproape plan, măsurată în lungul unei axe (funcție scalară de variabilă
- spaţială)
- Semnalul de temperatură, în grosimea unui perete (funcție scalară de timp și de spaţiu)
- Imaginea dată de o cameră de luat vederi (funcție scalară de două variabile spaţiale)
- Secvenţa de imagini date de aceeaşi cameră (funcție scalară, de timp și de două variabile
- spaţiale)
- Semnalul vocal stereo (două funcții scalare de variabila timp, care formează o funcție vectorială
- de variabila timp)
- Semnalele de tensiune de la ieșirea unui traductor de înclinare față de verticală (funcție
- vectorială de variabila timp).

Proprietăţile pe care trebuie să le îndeplinească o mărime fizică pentru a purta informația (implicit: pentru a fi folosită ca semnal) sînt:

- Să poată fi prelucrată (adică să poată fi depusă informaţie, să se poată extrage informaţie şi să se poată aduce modificări informaţiei purtate de acea mărime)
- Să poată fi transmisă la distanţă
- Să fie puţin afectată de perturbaţii

O altă proprietate foarte utilă: posibilitatea semnalului de a fi memorat (întregistrat).

Mărimile folosite ca semnale sînt:

- Tensiunea electrică (vezi semnalul de microfon etc.)

- Curentul electric (vezi ieşirea unor traductoare)
- Deplasarea mecanică (vezi pedala de acceleraţie)
- Presiunea aerului (vezi comandă pneumatică)

Forma sub care se prezintă semnalele depinde de natura mărimii și de scopul în care folosim semnalul. Din punctul de vedere al continuităţii în timp și în valori, folosim două variante:

- *Semnal analogic* (continuu în timp și în valori)
- *Semnal numeric* (discontinuu în timp și în valori, se mai numește semnal în timp discret și cu valori discrete. Semnalul în timp discret se mai numește *semnal eşantion*Semnalul analogic

2.1. Semnalul analogic

Modelul matematic al semnalului analogic este o aplicaţie pe mulţimea numerelor reale, cu valori în mulţimea numerelor reale (sau un interval de numere reale).
În figura 2.1 apare înregistrarea fotografică a unui semnal de pe ecranul osciloscopului care este un semnal analogic. Semnalul acustic care sosește la un microfon, semnalul electric pe care îl produce microfonul, poziţia acului unui instrument de măsură cu ac, semnalul captat de antena unui receptor radio, semnalul electric produs de o cameră video analogică, semnalul afișat de tubul catodic al unui televizor, timpul indicat de un ceasornic mecanic – toate sînt semnale analogice, fiind continue în timp și în valori.

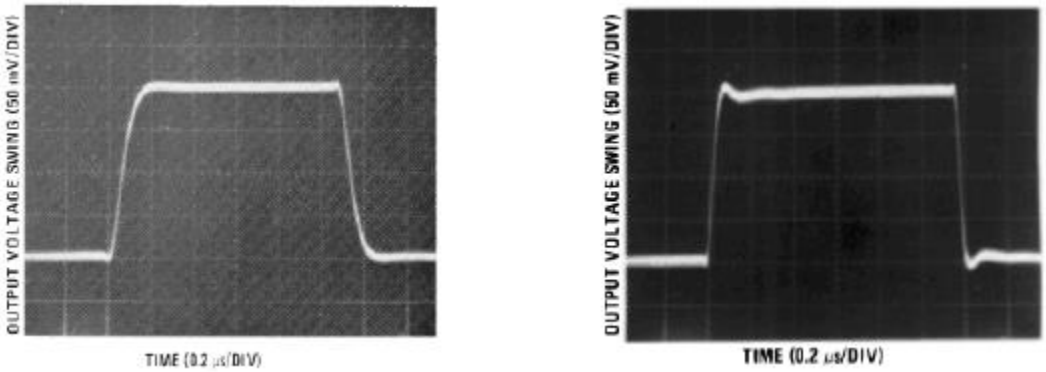


Figura 2.1 Semnal analogic pe ecranul osciloscopului

2.2. Semnalul digital

Modelul matematric al unui semnal numeric este un şir de numere, deci o aplicaţie pe mulţime numărabilă (mulţimea numerelor întregi), cu valori în restricţii ale mulţimii numerelor raţionale sau mulţimii numerelor întregi.
Numerele reprezintă valorile aproximate ale eşantioanelor unui semnal analogic. Exemple: numerele succesive indicate de un voltmetru cu afişaj numeric, indicaţia de temperatură a unui termometru digital, timpul afișat de un ceas digital, semnalul muzical înregistrat pe CD, semnalul produs de o cameră video digitală.
Avantajele semnalelor numerice:

- Posibilitate nelimitată de memorare
- Posibilităţi mari de prelucrare
- Imunitate sporită la perturbaţii
- Versatilitatea circuitelor de prelucrare

Dezavantajele semnalelor numerice

- Circuite mai complicate pentru prelucrare (această particularitate dispare, odată cu dezvoltarea tehnicii numerice)
- Prelucrare încă insuficient de rapidă, pentru frecvenţele mari

2.3. Transformarea semnalelor

Majoritatea semnalelor pe care le folosim provin din „lumea" analogică. Există metode de conversie a semnalelor din analogic în numeric (analog-to-digital conversion) şi din numeric în analogic (digital-to-analog conversion). Scopul conversiei A/N (sau ADC = *Analog-to-Digital Conversion*) este preluarea semnalului în formă numerică, pentru prelucrare sau pentru memorare (exemple: memorarea concertului pe CD, prelucrarea numerică a semnalului din imagine). Scopul conversiei N/A (sau DAC = *Digital-to-Analog Conversion*) este reconstituirea semnalului analogic, pentru transmisiune, afişare sau pentru scopuri audio-video.

Etapele conversiei AD şi DA:
- Eşantionarea și reţinerea eşantionului („sample and hold")
- *Cuantizarea* eşantionului (reprezentarea printr-un nivel discret)
- *Codarea numerică* a nivelului cuantizat, prin care este reprezentat eşantionul

În figura 2.2 este reprezentată o scurtă secvenţă dintr-un semnal analogic, precum şi eşantioanele obţinute de la acest semnal. Semnalul a fost eşantionat la intervale egale (*perioada de eşantionare*). În figura 2.3 sînt reprezentate aproximările eşantioanelor, ca urmare a cuantizării. Se observă că fiecare eşantion ia doar valori discrete, dintr-o mulţime finită. În partea inferioară a figurii 2.2 sînt scrise codurile numerice ale nivelurilor rezultate prin cuantizare (numere în baza 2).
Aceasta este forma în care sînt prelucrare în calculatorul numeric, sau sînt memorate, sau sînt transmise prin sisteme de comunicaţii numerice. Circuitele de conversie ADC şi DAC, precum și introducerea datelor în calculator reprezintă bazele sistemelor de achiziţie a datelor.

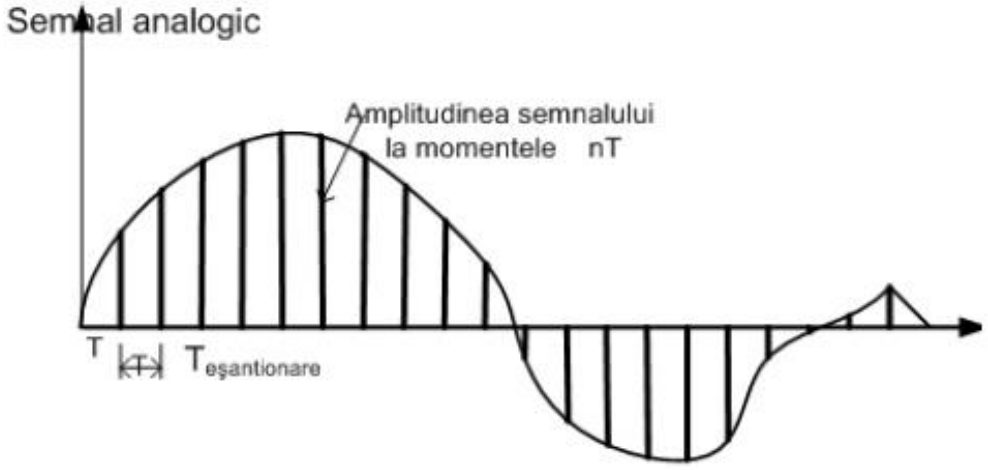


Figura 2.2 Semnalul analogic și semnalul eșantionat

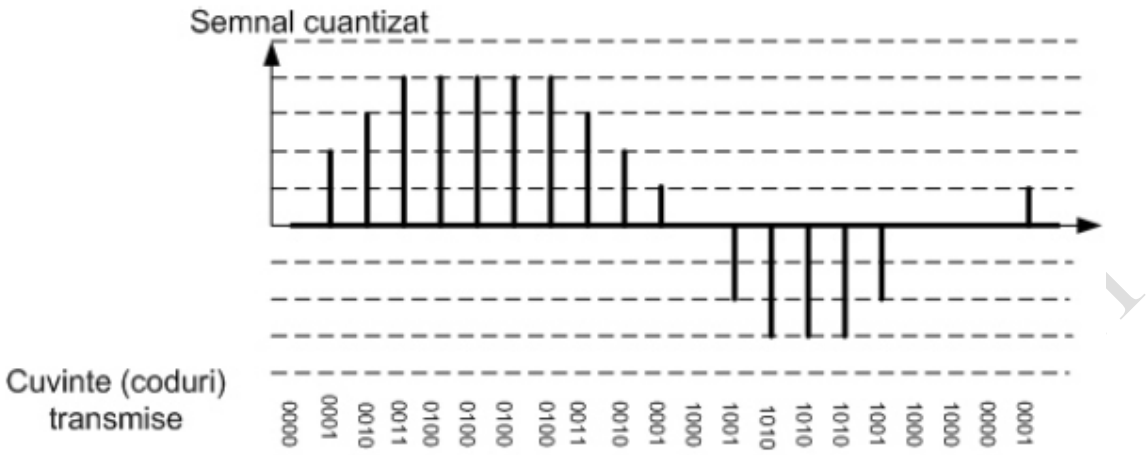


Figura 2.3 Semnalul eșantionat, cuantizat și codat numeric

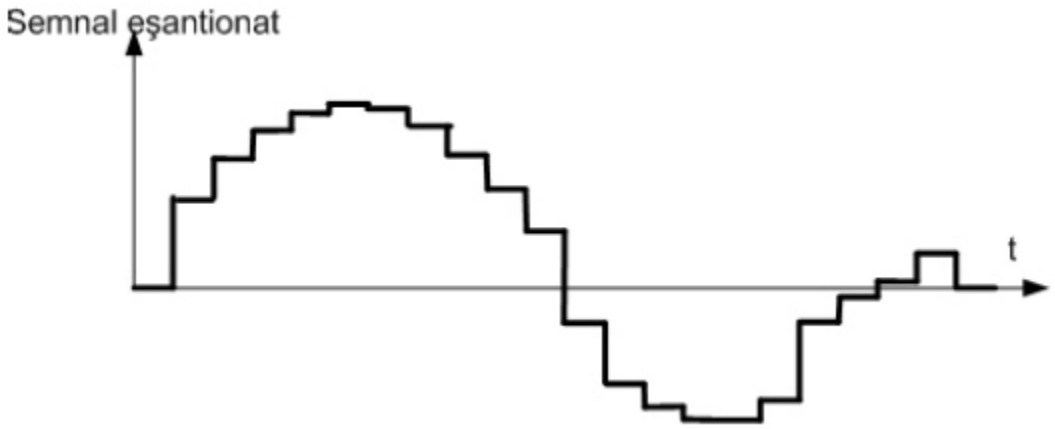


Figura 2.4 Reconstiruirea semnalului analogic (conversie A/N)

În figura 2.4 apare semnalul analogic reconstituit din forma numerică (conversia N/A sau DAC).
Se observă că el este similar cu semnalul original, dar nu este identic. Proprietatea caracteristică este aceea că el este recontituit din aproximări ale eşantioanelor. Aspectul de funcţie în scară provine din aproximarea semnalului doar prin valori discrete. Valorile funcției între momentele de eşantionare sînt aproximate prin menţinerea valorii de la ultimul moment de eşantionare.
Primele două întrebări care apar, la reconstituirea semnalului analogic, sînt:

- cît de fină trebuie să fie cuantizarea (adică cît de dese trebuie să fie nivelurile cu care aproximăm eşantionul)?
- cît de frecventă trebuie să fie eşantionarea (adică cît de mică să fie perioada de eşantionare)?

- *Răspunsul 1*: atît de fină pe cît de mică este eroarea pe care sîntem dispuşi să o acceptăm (să ne gîndim la rezoluţia voltmetrului cu afişare numerică, la care este prezentă exact această problemă).
- *Răspunsul 2:* este mai complicat, va fi tratat la cursul de Semnale şi sisteme. Ca regulă generală: dacă componenta cu frecvenţa cea mai mare din semnalul analogic are frecvenţa *f,* atunci eşantionarea trebuie să se producă cu o frecvenţă mai mare decît 2*f*.

Din punctul de vedere al conţinutului informaţional, semnalele se împart în două categorii: semnale deterministe și semnale întîmplătoare (sau aleatoare).

Semnale deterministe: cele la care evoluţia semnalului este anterior cunoscută. Ele nu aduc nici o informaţie, sînt folosite doar pentru testarea circuitelor şi echipamentelor, în laborator sau în exploatare. Cel mai adesea, semnalele folosite pentru testare sînt periodice.
Forme uzuale: sinus, dreptunghi, triunghi, dinte de fierăstrău, impulsuri etc.
Semnalele întîmplătoare sînt cele a căror evoluţie nu poate fi prezisă, deci poartă cu ele informaţie (cu cît sînt mai puţin predictibile, cu atît aduc mai multă informaţie). Cel mult cunoaştem dinainte proprietăţile statistice ale semnalului întîmplător (domeniul valorilor, frecvenţa cea mai mare a componentelor sale etc.), dar nu evoluţia particulară într-un anumit interval de timp.
Exemple de semnale întîmplătoare: semnalul vocal cules de microfon, curentul absorbit de motorul electric, turaţia motorului, temperatura măsurată într-o încăpere, viteza vîntului, semnalul de date transmis între două calculatoare etc.

3. Bazele logice ale calculatoarelor.

Caracteristica comună de bază a tuturor generaţiilor de calculatoare numerice realizate până în prezent o reprezintă natura discretă a operațiilor efectuate. Teoretic și practic s-a impus utilizarea dispozitivelor care codifică informația în două stări stabile, rezultând efectuarea calculelor în sistem binar. Suportul teoretic al acestuia este algebra logică (booleană). Analiza și sinteza circuitelor de comutație aferente calculatoarelor numerice utilizează algebra booleană ca principal instrument matematic.

Vom prezenta în continuare unele elemente atât ale algebrei booleene cât şi ale unor circuite logice fundamentale.

3.1. Funcţii logice. Porţi logice. Algebra de comutaţie.

3.1.1. Funcţii logice. Tabele de adevăr

Prin definiție, valorile pe care le poate lua o funcție logică *f*, de una sau mai multe variabile logice, pot fi "0" sau "1", ca și valorile variabilelor sale.

$$f(a_1, a_2, \dots, a_i): \{0,1\} \rightarrow \{0,1\} \quad unde \quad a_1, a_2, \dots, a_i \in \{0,1\}, i \in \mathbb{N} \tag{3.1}$$

Funcţia logică conține un număr variabil de termeni. Numărul maxim de valori ce vor fi procesate de funcție este egal cu 2^i (unde *i* este numărul de variabile ale funcției). În aparatura digitală valorile logice "0" și "1" ale variabilelor funcţiei sunt reprezentate prin două nivele de tensiune diferite. Expresiile booleene sau funcţiile logice pot fi reprezentate în mai multe moduri ce vor fi exemplificate pe o funcție oarecare *f*.

a. Reprezentarea cu tabel de adevăr

Tabela de adevăr este cea mai simplă reprezentare a unei funcții booleene. Aceasta cuprinde toate combinațiile posibile ale valorilor variabilelor de intrare și afișează în dreptul fiecăreia, valoarea corespunzătoare, procesată la ieşire pentru funcţia *f*. Cu alte cuvinte, tabelul de adevăr afișează ieşirile pentru toate combinațiile posibile de valori de intrare.

Exemplu: Tabela de adevăr pentru funcție *f* (*A, B, C*) oarecare de trei variabile poate fi:

A	***B***	***C***	***f***
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

b. Forma canonică normal disjunctivă

După cum se poate observa din denumire, este vorba despre o formă care separă operatorii, fiind una dintre formele de reprezentare des întâlnite. Expresia constă din variabile conectate printr-un operator AND rezultând termeni conectaţi cu operatori OR.

Această reprezentara poartă denumirea de sumă de produse sau formă canonică normal disjunctivă (f.c.n.d.).

Fiecare operaţie AND poate fi privită ca un produs booleană, termenul obţinut din variabile conectate de operatori AND fiind un termen-produs. Operatorul OR este asimilat ca o însumare booleană, iar expresia cu termeni produs conectaţi de operatori OR fiind o expresie sumă-de-produse sau forma canonică normal disjunctivă.

În exemplul următor, expresia funcției este o sumă de produse completă pentru o funcție de trei variabile :

$$f(A,B,C)=\overline{A}\overline{B}\overline{C}+\overline{A}\overline{B}C+\overline{A}B\overline{C}+\overline{A}BC+A\overline{B}\overline{C}+A\overline{B}C+AB\overline{C}+ABC. \quad (3.2)$$

Notând $\overline{A}\overline{B}\overline{C}$ cu P_0, $\overline{A}\overline{B}C$ cu P_1, etc., forma canonică normal disjunctivă se poate rescrie astfel:

$$f(A, B, C)= P_0 +P_1+ P_2 +P_3 +P_4 +P_5+P_6+P_7. \quad (3.3)$$

c. Forma canonică normal conjunctivă

Forma canonică normal conjuctivă (f.c.n.c.) este o altă modalitate de exprimare a funcţiilor. Aceasta se obține din operatori AND ce conectează termeni legaţi prin operatori OR. Pentru o funcție logică de trei variabile, forma canonică normal conjunctivă completă se scrie astfel:

$$f(A,B,C)=(A+B+C)(A+B+\overline{C})(A+\overline{B}+C)(A+\overline{B}+\overline{C})\cdot (\overline{A}+B+C)(\overline{A}+B+\overline{C})(\overline{A}+B+C)(\overline{A}+\overline{B}+\overline{C}). \quad (3.4)$$

Notând $(A+B+C)=S_0$, $(A+B+\overline{C})=S_1$ etc, funcția se poate rescrie:

$$f(A, B, C)= S_0 S_1S_2S_3S_4S_5S_6S_7. \quad (3.5)$$

d. Diagrame Veitch-Karnaugh

O reprezentare grafică a formelor canonice este dată de diagramele Veitch-Karnaugh. Aceasta constă dintr-o matrice, unde fiecărui element îi corespunde un termen produs canonic.

Caracteristic pentru diagramele Veitch-Karnaugh este că orice element diferă de elementul său adiacent printr-o singură variabilă. Ca exemplu sunt reprezentate două diagrame Veitch-Karnaugh de trei și patru variabile, rezultând astfel opt, respectiv şaisprezece combinații, fiecăreia dintre aceste combinații fiindui alocată câte un element din diagramă.

	$\overline{A}\,\overline{B}$ 0 0	$\overline{A}B$ 01	AB 11	$A\overline{B}$ 10
$\overline{C}$ 0	P_0	P_2	P_6	P_4
C 1	P_1	P_3	P_7	P_5

Figura 1
Diagrama V-K
pentru 3 variabile de intrare

	$\overline{A}\,\overline{B}$ 0 0	$\overline{A}B$ 01	AB 11	$A\overline{B}$ 10
$\overline{C}\,\overline{D}$ 00	P_0	P_4	P_{12}	P_8
$\overline{C}D$ 01	P_1	P_5	P_{13}	P_9
CD 11	P_3	P_7	P_{15}	P_{11}
$C\overline{D}$ 10	P_2	P_6	P_{14}	P_{10}

Figura 2
Diagrama V-K
pentru 4 variabile de intrare

e. Forma elementară

Termenii formelor elementare nu conţin toate variabilele de intrare, spre deosebire de formele canonice prezentate anterior. Pornind de la forma de reprezentare canonică putem ajunge la una elementară prin operaţia numită minimizare.

Exprimarea unei funcții prin forme elementare oferă avantaje faţă de formele canonice în primul rând la implementarea funcției, deoarece numărul de circuite și componente electronice implicat este minimizat.

Exemplu de scriere a unei funcții sub formă elementară:

$$f(A,B,C)=\overline{A}B+\overline{B}\overline{C} \qquad (3.6)$$

3.1.1.1. Minimizarea funcţiilor logice

Tehnica minimizării permite exprimarea funcției printr-o formă elementară prin transformarea intr-o formă canonică, eliminând variabilele de intrare neutilizate din termenii funcției. Utilizarea expresiei elementare la implementare va costa mai puţin și/sau va opera mai rapid față de implementarea expresiei iniţiale.

Printre cele mai răspândite metode de minimizare este utilizarea diagramele Veitch-Karnaugh. Prin această metodă se face o simplă identificarea vizuală a termenilor care pot fi combinaţi .

Tehnica minimizării cu ajutorul diagramelor Veitch-Karnaugh:

1. Avem dată definiția funcţiei exprimată ca o sumă de produse;
2. Elementele din diagrama Veitch-Karnaugh ce corespund termenilor din expresie sunt marcate cu 1; celelate căsuțe rămase pot fi marcate cu zerouri pentru a indica faptul că funcția va fi *0* în aceste situații, sau vor rămâne necompletate.
3. Se face gruparea suprafeţelor valide de valoare *1*, formate din căsuţe adiacente pe orizontală sau verticală (suprafeţele pot conţine un număr de elemente egal cu puteri ale lui 2).
4. Elementele de-a lungul unei laturi sunt considerate adiacente inclusiv cu cele de pe latura opusă (sus și jos sau stânga şi dreapta), întrucât ele corespund termenilor cu o singură variabilă diferită.
5. Suprafeţele maximale corespund termenilor elementari, iar reprezentarea grafică este ilustrarea teoremei:

$$A\cdot B+A\cdot\overline{B}=A \qquad (3.7)$$

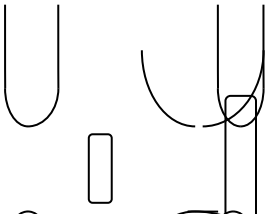
6. Forma elementară se obține ca o sumă de produse, unind prin operatori **AND** termenii elementari rezultaţi în etapa V.

Exemplu: Să se minimizeze funcția

$$f = P_0+P_2+P_5+P_7+P_8+P_9+P_{10}+P_{11}+P_{12}+P_{14}.$$

folosind diagrama V-K .

REZOLVARE:



AB CD	00	01	11	10
00	1		1	1
01		1		1
11		1		1
10	1		1	1

$$f = \overline{A}\,\overline{B}\,\overline{D} + \overline{A}BD + A\overline{D} + A\overline{B}$$

- Pentru construirea diagramei Karnaugh se poate porni și de la f.c.n.c., caz în care suprafeţele maximale vor fi date de căsuţele adiacente conţinând 0 logic.
- Se preferă, totuşi, lucrul cu f.c.n.d., care are avantajul, pe lângă comoditatea oferită de lucrul cu expresii algebrice care conţin sume de produse, și pe acela al implementării cu porţi tip **NAND**, mai răspândite și mai avantajoase tehnologic.

3.1.2. Porţi logice

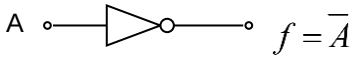
Poarta logică este un circuit electronic cu o singură ieşire și una sau mai multe intrări. Ea acceptă pe fiecare intrare unul din cele două nivele de tensiune, generând la ieşire unul din cele două nivele. De aceea ne referim la tensiunile porţilor logice ca la un **nivel logic** de tensiune **"înaltă"** (**HIGH**) și respectiv un nivel logic de tensiune **"joasă"** (**LOW**). Algebra booleană foloseşte trei operatori fundamentali cu care pot fi definite toate funcţiile logice ce pot fi îndeplinite de porţile logice, și anume:

- **NOT**
- **AND**
- **OR**

Toate funcţiile care se obţin cu ajutorul acestor operatori sunt implementate de circuite numite porţi logice.

3.1.2.1. Poarta NOT

Operarea porţilor logice se face pe semnale de intrare numite **variabile logice** (variabile care pot fi sau adevărate 1, sau false 0). Adesea vrem ca în timpul funcţionări o variabilă să fie modificată, de exemplu din 1 în 0 sau din 0 în 1. Aceasta este operaţia fundamentală **NU,** realizată de **poarta NOT**.
Simbolul de circuit, expresia booleană și tabela de adevăr corespunzătoare unei porţi **NOT** sunt:



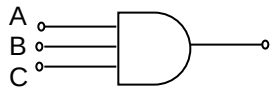
A	**f**
0	1
1	0

- Pentru intrare *1* , ieșirea este *0* și invers.

3.1.2.2. Poarta AND

Este nevoie la proiectarea unui sistem digital, de a stabili momentul în care două semnale logice preiau simultan valoarea logică *1*. Sunt extrem de dese astfel de aplicaţii cu semnale de control în ecare trebuie dată o comandă, dacă mai multe condiţii sau evenimente coexistă. Aceasta funție este îndeplinită de operatorul și **poarta AND.**

Simbolul de circuit, expresia booleană și tabela de adevăr corespunzătoare unei porţi **AND** sunt prezentate mai jos:



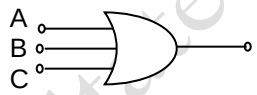
$$f = A \cdot B \cdot C$$

A	B	C	f
0	0	0	0
0	0	1	0
0	1	1	0
0	1	0	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Când toate intrările sunt SUS ieșirea este SUS.
- Când cel puțin o intrae este JOS ieșirea este JOS.

3.1.2.3. Poarta OR

Această poartă semnalează prezența, în mod obișnuit, a cel puţin unui eveniment, lucru indicat prin asocierea variabilei 1. Operaţia **SAU** şi **poarta SAU** corespunzătoare modelează astfel de situații.
Tabelul de adevăr, simbolul de circuit şi expresia booleană corespunzătoare unei porţi **SAU** cu trei intrări vor fi:



$$f = A + B + C$$

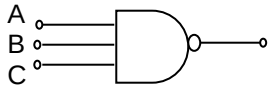
A	B	C	f
0	0	0	0
0	0	1	1
0	1	1	1
0	1	0	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Pentru orice intrare SUS ieșirea va fi SUS.

3.1.2.4. Poarta NAND

Implementarea funcţiilor **AND**, **OR** şi **NOT**, ca dealtfel a oricărei expresii booleene se poate face folosiind **porţi universale**. Una dintre acestea este **poarta NAND (ŞI-NU)**.
Simbolul de circuit, expresia booleană și tabelul de adevăr, pentru o poartă **ŞI-NU (NAND)** cu trei intrări sunt:

A	*B*	*C*	*f*



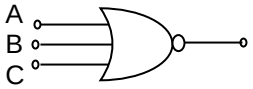
$$f = \overline{A \cdot B \cdot C}$$

0	**0**	**0**	**1**
0	**0**	**1**	**1**
0	**1**	**0**	**1**
1	**0**	**0**	**1**
0	**1**	**1**	**1**
1	**0**	**1**	**1**
1	**1**	**0**	**1**
1	**1**	**1**	**0**

- Orice intrare JOS va produce ieșirea SUS.

3.1.2.5. Poarta SAU-NU (NOR)

Poarta **NOR (SAU-NU)** este o altă **poartă universală**. Expresia booleană, simbolul de circuit şi tabelul de adevăr pentru o poartă **NOR** cu trei intrări**,** sunt:



$$f = \overline{A + B + C}$$

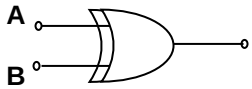
A	*B*	*C*	*f*
0	**0**	**0**	**1**
0	**0**	**1**	**0**
0	**1**	**0**	**0**
1	**0**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**1**	**0**
1	**1**	**0**	**0**
1	**1**	**1**	**0**

- Orice intrare SUS produce ieșirea JOS.
- poartă care realizează operaţia **NOR** în logică pozitivă, realizează operaţia **NAND** în logică negativă și invers.

3.1.2.6. Poarta SAU EXCLUSIV (XOR)

Poarta **EXCLUSIVE OR** (**SAU EXCLUSIV)** are iesșirea în starea "*1*" atunci și numai atunci când o singură intrare este în starea "*1*".

Funcţia booleană, simbolul și tabelul de adevăr pentru o poartă **SAU EXCLUSIV** cu două intrări sunt:



$$f = A \oplus B$$

A	*B*	*f*
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

Această poartă poate fi privită și ca o combinație de porţi **AND** și **OR**.

3.2. Algebră de comutaţie.Logică şi circuite logice combinaţionale. Componente digitale.

3.2.1. Algebra de comutaţie

Când vorbim despre algebra booleană sau algebra de comutație ne referim la o structură algebrică de tip:

$$\boldsymbol{A = (B, O)},$$

Unde:

- $\boldsymbol{B} = \{0, 1\}$ - este mulțimea constantelor de comutație,
- $\boldsymbol{O} = \{AND, OR, NOT\}$ - este mulțimea operatorilor de comutație, definiți astfel:

$$\begin{aligned}
&\forall x, y \in \mathbf{B} \quad x\ AND\ y = \begin{cases} 1\ dacă\ x = 1\ şi\ y = 1 \\ 0\ dacă\ x = 0\ şi\ y = 0 \end{cases} \\
&\forall x, y \in \mathbf{B} \quad x\ OR\ y = \begin{cases} 1\ dacă\ x = 1\ sau\ y = 1 \\ 0\ dacă\ x = 0\ sau\ y = 0 \end{cases} \\
&\forall x \in \mathbf{B} \quad NU\ x = \begin{cases} 1\ dacă\ x = 0 \\ 0\ dacă\ x = 1 \end{cases}
\end{aligned} \tag{3.8}$$

Echivalențe ale operatorilor:

- AND ≡ ŞI,∩,· $(produs\ logic)$;
- OR ≡ SAU,∪, + $(sumă\ logică)$;
- NOT ≡ NU, #, $(produs\ logic)$;

O expresie de comutație este o combinație a unui număr finit de variabile de comutație, constante de comutație și operatori de comutație, definită astfel:

- orice constantă sau variabilă de comutație este o expresie de comutație;
- dacă ***t1*** și ***t2*** sunt expresii de comutație atunci ***t1•t2***, ***t1+t2*** și $\overline{\boldsymbol{t1}}$ sunt expresii de comutație.

Evaluare a unei expresii de comutație se bazează pe următoarele reguli:

- dacă există paranteze, evaluarea se face din interior spre exterior;
- ordinea descrescătoare a priorităților operatorilor este: ***NOT, AND, OR***;
- pentru priorități egale evaluarea se face de la stânga la dreapta.

3.2.1.1. Funcţii de comutaţie

Funcția $f : \mathbf{B}^n \rightarrow \mathbf{B}$ este o funcție de comutație de *n* variabile.

Funcția *complementară* a unei funcții *f*, notată cu $\bar{f}$ este acea funcție care are valoarea *1 / 0* pentru toate combinațiile de valori ale variabilelor pentru care funcția f ia valoarea *0 / 1*.

Specificarea unei funcții de comutație de *n* variabile se poate face în principal prin tabela de adevăr sau prin expresia de comutație.

Tabela de adevăr prezintă valorile funcției pentru toate cele 2^n combinații de valori ale variabilelor.

Exemplu. Se considera funcția $f : \mathbf{B}^3 \rightarrow \mathbf{B}$, data prin tabela de adevăr:

x	*y*	*z*	*f*
0	**0**	**0**	**1**
0	**0**	**1**	**1**

0	**1**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**0**	**0**
1	**0**	**1**	**1**
1	**1**	**0**	**0**
1	**1**	**1**	**0**

Pe baza tabelei de adevăr se obține expresia de comutație a funcției în umătorul mod:

- expresia se scrie sub forma unei sume de produse, câte un termen al sumei pentru fiecare valoare 1 a funcției (în cazul nostru expresia conține trei termeni);
- pentru fiecare termen se scrie un produs al tuturor variabilelor funcției, fiecare variabilă reprezentandu-se în formă directă dacă variabila are valoarea 1 în combinația respectivă de valori din tabelă sau în formă complementată (negată) dacă variabila are valoarea 0. Se obține:

$$f(x,y,z)=\bar{x}\cdot\bar{y}\cdot\bar{z}+\bar{x}\cdot\bar{y}\cdot z+x\cdot\bar{y}\cdot z.$$

Această sumă de produse, în care fiecare produs conține toate variabilele atât în formă directa cât și negată se numește *forma canonică sumă de mintermeni* (un mintermen fiind un produs al tuturor variabilelor în formă directă sau negată). Această formă este unică pentru o funcție dată.

Trebuie reținut că o funcție de comutație poate fi exprimată prin mai multe expresii de comutație echivalente. Ne interesează expresia minimă (suma de produse) obținută aplicând proprietațile de mai sus, pentru exemplul considerat fiind:

$$f(x,y,z)=\bar{x}\cdot\bar{y}\cdot\bar{z}+\bar{x}\cdot\bar{y}\cdot z+x\cdot\bar{y}\cdot z=\bar{x}\cdot\bar{y}\cdot\bar{z}+\bar{x}\cdot\bar{y}\cdot z+\bar{x}\cdot\bar{y}\cdot z+x\cdot\bar{y}\cdot z=$$
$$=\bar{x}\cdot\bar{y}\cdot(\bar{z}+z)+(\bar{x}+x)\cdot\bar{y}\cdot z=\bar{x}\cdot\bar{y}\cdot 1+1\cdot\bar{y}\cdot z=\bar{x}\cdot\bar{y}+\bar{y}\cdot z.$$

Observatii.

1. Orice funcție de comutație de *n* variabile se poate reprezenta în mod unic printr-o expresie de formă canonică sumă de mintermeni;
2. Pentru *n* variabile există 2*n* mintermeni;
3. Pentru o combinație de valori ale celor *n* variabile un singur mintermen are valoarea 1, toti ceilalti mintermeni au valoarea 0;
4. Produsul a doi mintermeni diferiți este 1.

3.2.2. Logică şi circuite logice combinaţionale (CLC)

Independența mărimilor de ieşire ale CLC de timp, altfel zis dependența *numai* de combinațiile aplicate la intrare este caracteristica principală a acestor circuite.

În figura 3.1 este prezentată schema bloc a unui CLC. Intrările sunt *x0, x1, ... xm-1* și generează în exterior ieşirile *y0, y1, ..., yn-1*. O funcție logică (de comutaţie) poate descrie funcţionarea circuitului.

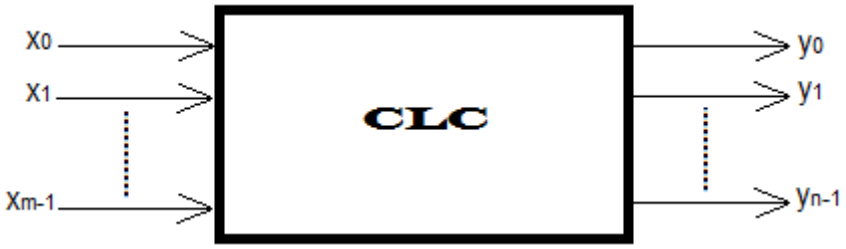


Fig. 3.1. Circuit logic combinaţional

Pornind de la schema circuitului se poate analiza CLC-ul în vederea stabilirii funcţionării, concretizată prin tabela de adevăr sau prin expresiile variabilelor de ieşire funcţie de cele de intrare. Stabilirea structurii circuitului se face prin analiza ce presupune parcurgerea următoarelor etape:

- definirea funcţiilor logice;
- minimizarea acestora;
- obţinerea schemei circuitului.

Tipurile reprezentative de *CLC* din structura unui calculator numeric sunt: *convertoarele de cod, codificatoarele şi decodificatoarele, multiplexoarele și demultiplexoarele, comparatoarele, detectoarele și generatoarele de paritate, ariile logice programabile, memoriile și circuitele aritmetice.*

3.2.3. Convertoare de cod

CLC-ul care permite trecerea dintr-un cod binar în altul sunt convertoarele de cod. Sinteza unui asemenea CLC se va exemplifica pentru un convertor *din cod binar în cod Gray*. În figură se prezintă elementele aferente sintezei acestui tip de convertor, în care $X_3\,X_2\,X_1\,X_0$ reprezintă cuvântul binar aplicat la intrare, iar $Y_3\,Y_2\,Y_1\,Y_0$ cuvântul binar obţinut la ieşire.

X_3	X_2	X_1	X_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Figura 3.2. a) Convertor de cod natural- Gray: tabela de corespondență.

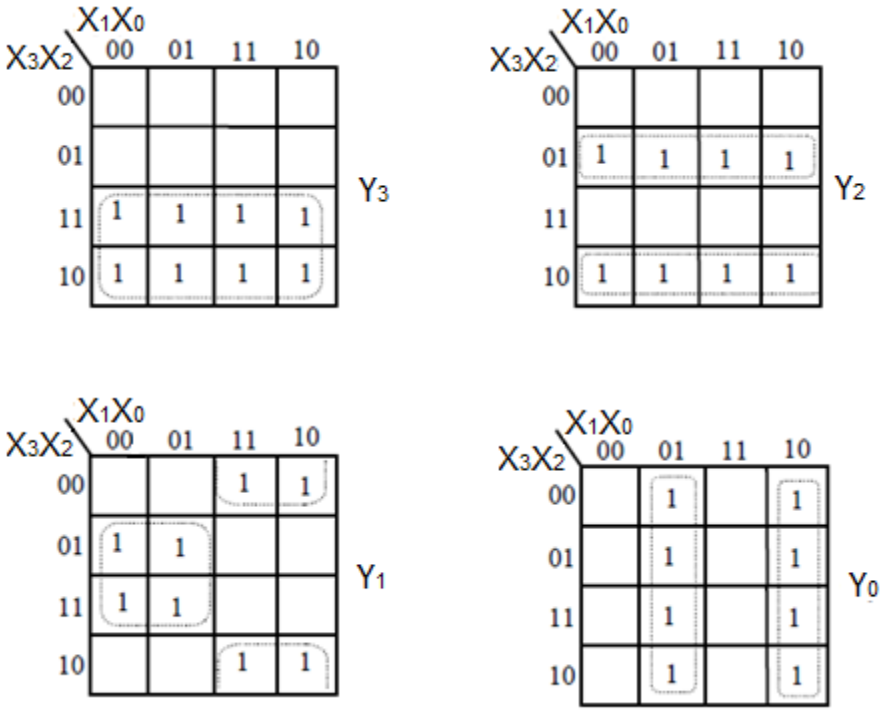


Figura 3.2. b) Convertor de cod natural- Gray: diagramele Karnaugh asociate.

După reducerile în diagramele Karnaugh rezultă:

$Y_3 = X_3$

$Y_2 = \overline{X_2}X_3 + \overline{X_3}X_2 = X_2 \oplus X_3$

$Y_1 = \overline{X_1}X_2 + \overline{X_2}X_1 = X_1 \oplus X_2$

$Y_0 = \overline{X_1}X_0 + \overline{X_0}X_1 = X_1 \oplus X_0$

3.2.4. Codificatoare şi decodificatoare

CLC la care activarea unei intrări, dintr-un grup de *m*, conduce la apariţia unui cuvânt de cod la ieşire format din *n* biţi se numesc *codificatoare*. În figură se prezintă elementele unui codificator cu *m=3* intrări și *n=2* ieșiri.

X_0	X_1	X_2	Y_0	Y_1
0	0	0	0	0
1	0	0	0	1
0	1	0	1	0
0	0	1	1	1

$$Y_0 = \overline{X_0} \cdot (X_1 \oplus X_2)$$

$$Y_1 = \overline{X_1} \cdot (X_0 \oplus X_2)$$

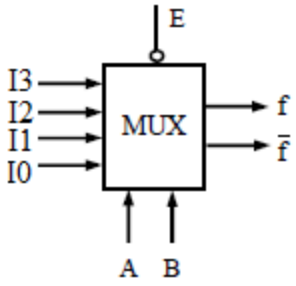
Figura 3.3. Codificator cu m=3 și n=2 (schema bloc, tabela de adevăr, funcții logice)

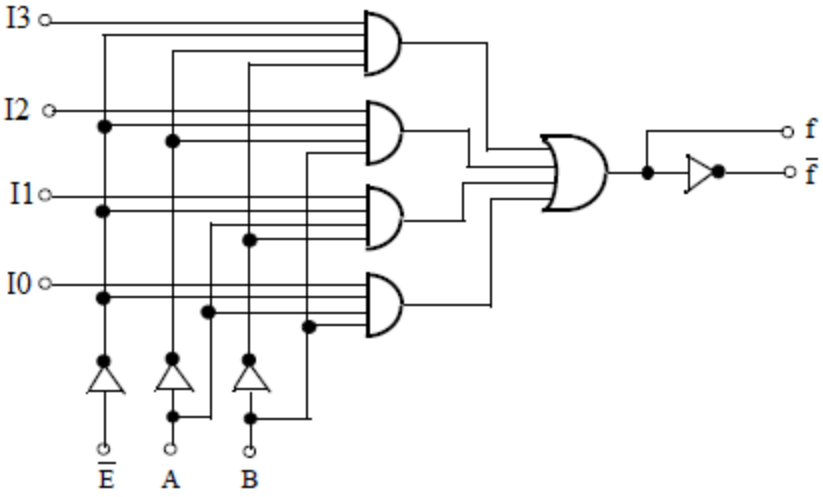
Decodificatoarele sunt acele *CLC* care activează una sau mai multe ieşiri funcţie de cuvântul de cod aplicat la intrare și este necesară în aplicaţii care adresează memoria, în afişarea numerică, multiplexarea datelor etc.

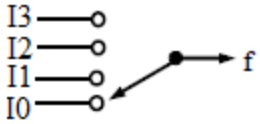
3.2.5. Multiplexoare şi demultiplexoare

Multiplexoarele (MUX) sunt *CLC* care fac transferul datelor de la una din intrările selectate de o adresă (cuvânt de selecţie) către o ieșire unică. Funcţional *MUX* pot fi asimilate cu o reţea de comutatoare comandate. *Multiplexoarele* pot fi analogice sau numerice, ultimele fiind specifice CN.
Exemplu: sinteza unui MUX 4:1 numeric şi se prezintă implementarea cu porţi logice.

E	B	A	f = Canal
0	0	0	I3
0	0	1	I2
0	1	0	I1
0	1	1	I0
1	*	*	-



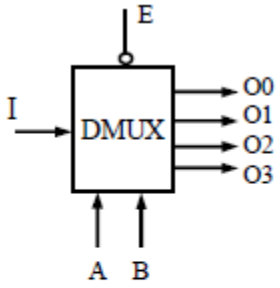


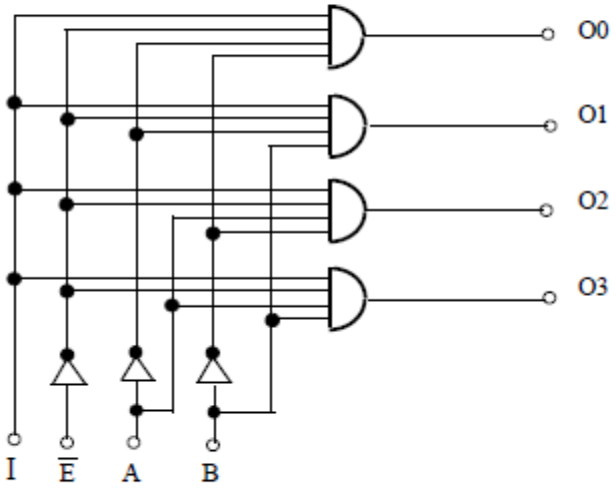


$$f = \overline{E} \cdot (\overline{B} \cdot \overline{A} \cdot I3 + \overline{B} \cdot A \cdot I2 + B \cdot \overline{A} \cdot I1 + B \cdot A \cdot I0)$$

Figura 3.4. Multiplexor numeric 4:1

E	B	A	O0	O1	O2	O3
0	0	0	I			
0	0	1		I		
0	1	0			I	
0	1	1				I
1	*	*	-	-	-	-





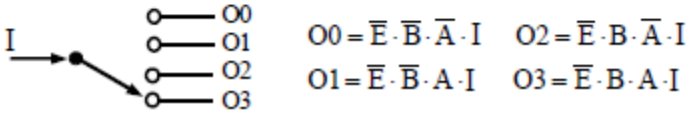
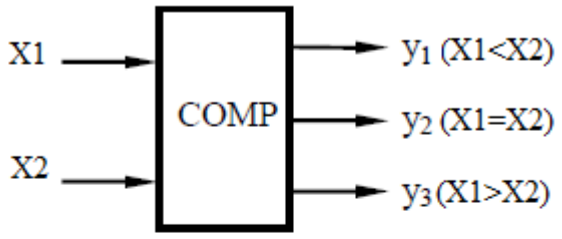


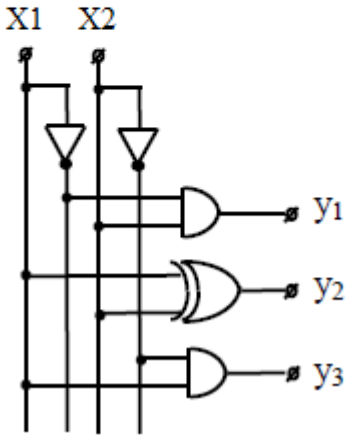
Figura 3.5. Demultiplexor 1:4

3.2.6. Comparatoare

CLC-urile care determină relaţiei dintre două numere sunt *comparatoarele numerice*. Acestea au ieșirile reprezentate de *trei funcții (<, =, >)* ce corespund tipului de relaţie existent între numerele aplicate la intrare.
elemente definitorii ale unui comparator pe un bit sunt prezentate în figura următore.

X1	X2	y_1	y_2	y_3
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0





$y_1 = \overline{X1} \cdot X2$

$y_2 = \overline{X1} \cdot \overline{X2} + X1 \cdot X2 = \overline{X1 \oplus X2}$

$y_3 = X1 \cdot \overline{X2}$

Figura 3.6. Comparator pe un bit

3.2.7. Sumatoare

Un *sumator elementar* este un *CLC* care adună două numere binare *xi, yi* cu un transport de intrare *ci,* generând la ieşire doi biţi: suma *şi* şi transportul *ci+1* către rangul superior, conform tabelului:

x_i	y_i	c_i	Si	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Din tabel rezultă relaţiile :

$$s_i = \overline{x}_i \overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = \overline{x}_i y_i c_i + x_i \overline{y}_i c_i + x_i y_i \overline{c}_i + x_i y_i c_i = c_i (x_i \oplus y_i) + x_i y_i .$$

Relaţiile de mai sugerează obţinerea sumatorului elementar din două semisumatoare:

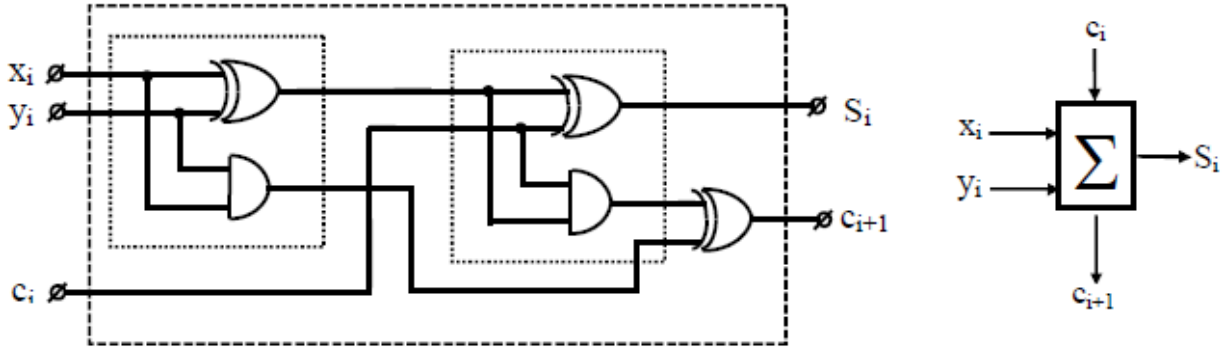


Figura 3.7. Sumatorul elementar

3.3. Maşini cu număr finit de stări

Un **automat finit** (AF) sau o **maşină cu stări finite** este un model de comportament compus din *stări, tranziţii* și *acţiuni*.

- O stare reține comportamentul compus din *stări, tranziţii* și *acţiuni.* Starea stochează informaţii despre trecut, adică reflectă schimbările intrării de la inițializarea sistemului până în momentul de faţă.
- Tranziţia indică o schimbare de stare și este descrisă de o condiţie care este nevoie să fie îndeplinită pentru a declanșa tranziţia.
- Acţiunea este o descriere a unei activităţi ce urmează a fi executată la un anumit moment. Există câteva tipuri de acţiuni:
    - Acţiune *de intrare* executată la intrarea într-o stare.
    - Acţiune *de ieşire* executată la ieşirea dintr-o stare.
    - Acţiune *de intrare de date* executată în funcție de starea prezentă și de datele de intrare.
    - Acţiune *de tranziţie* executată în momentul unei tranziţii.

Logica automatelor finite stabileşte că ieșirea şi starea următoare a unui automat finit este o funcție de valoarea intrării și de starea curentă.

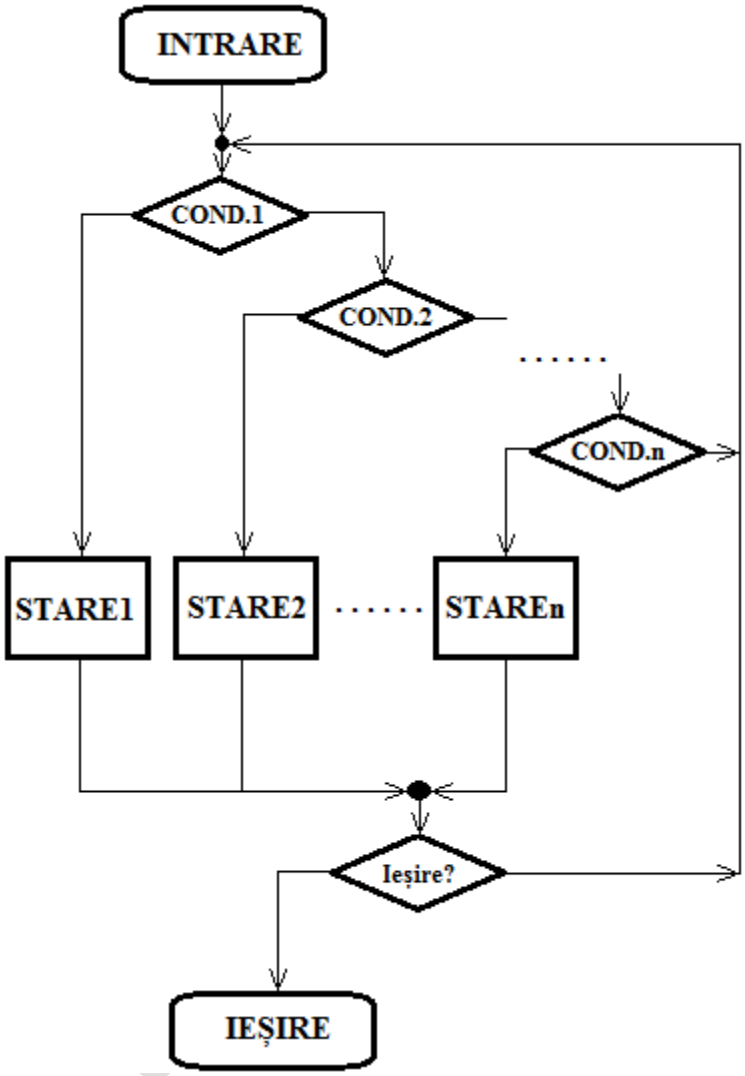


Figura 3.8 Logica automatelor finite

Modelul matematic În funcție de tip, există mai multe definiţii. Un automat finit **acceptor** este un cvintuplu <*Σ, S, s0, δ, F*>, unde:

- *Σ* este alfabetul de intrare (o mulţime finită și nevidă de simboluri).
- *S* este o mulţime finită și nevidă de stări.
- *s0* este starea iniţială, element al lui *S*. Într-un automat finit nedeterminist, *s0* este o mulţime de stări iniţiale.
- *δ* este funcția de tranziţie a stării: $\delta: S \times \Sigma \rightarrow S$.
- *F* este mulţimea stărilor finale, o submulţime (posibil vidă) a lui S.

Un automat finit **transductor** este un sextuplu <*Σ, Γ, S, s0, δ, ω*>, unde:

- *Σ, S* și *s0* își păstrează semnificaţiile din definiţia automatului finit acceptor.
- *Γ* este alfabetul de ieșire (o mulţime finită și nevidă de simboluri).
- *δ* este funcția de tranziţie a stării $\delta: S \times \Sigma \rightarrow S \times \Gamma$.
- *ω* este funcția de ieșire.

Dacă funcția de ieşire este o funcţie de stare şi de alfabetul de intrare (ω: S x $\Sigma \rightarrow \Gamma$), atunci această definiţie corespunde **modelului Mealy**. Dacă funcția de ieșire depinde doar de stare (ω: $S \rightarrow \Gamma$), atunci această definiţie corespunde **modelului Moore**.

Aplicaţie: Circuitul de mai jos implementează un automat cu număr finit de stări (fie acesta *A*). Registrul pe un bit etichetat cu *R* memorează starea curentă, *I* este intrarea şi *O* reprezintă ieșirea.

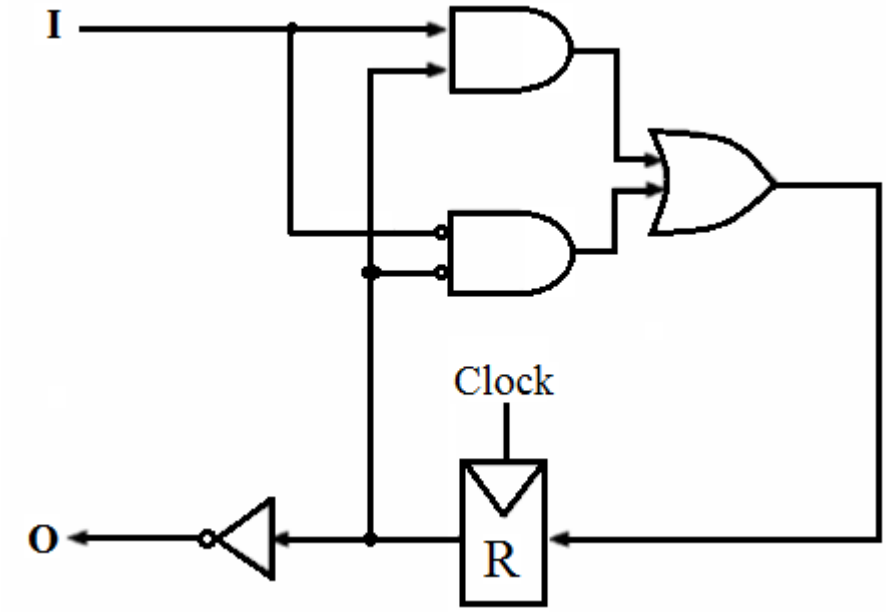


Figura 3.9. Automat cu număr finit de stări.

(a) Câte stări are automatul *A*? Desenaţi diagrama stărilor şi a tranziţiilor pentru automatul A.

(b) Circuitul de mai jos este un automat cu număr finit de stări programabil. Componentele circuitului, sunt etichetate astfel: (M4) - memorie cu patru locații, fiecare conţinând un bit (2^2x1-bit), (M2) - memorie cu două locaţii a câte un bit fiecare (2^1x1-bit), (R) - registru pe un bit.

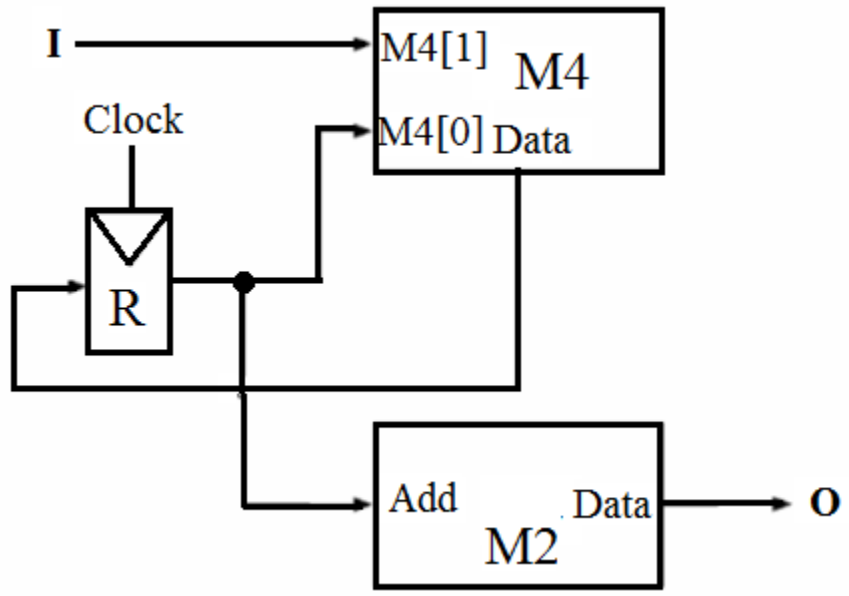


Figura 3.10. Automat programabil cu număr finit de stări.

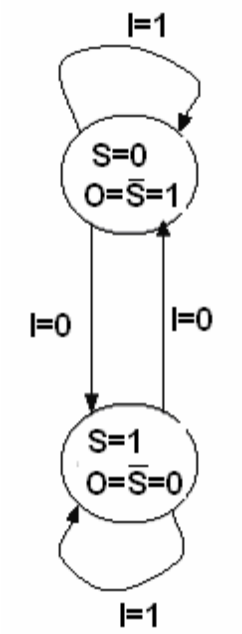
Ce trebuie să conţină locaţiile de memorie M4 și M2 astfel încât acest circuit să reprezinte o implementare hardware a automatului M4?

Adresa(M4[1]M4[0])	Continutul locatiei M4
0 0	
0 1	
1 0	
1 1	

Adresa(M2[0])	Continutul locatiei R
0	1
1	0

Răspuns:

a) Întrucât R este reprezentat pe un bit rezultă că A are două stări. Diagrama stărilor şi a tranziţiilor pentru automatul A arată astfel:



ToDo ----->R

Considerăm o stare a automatului (R=0) și alta (R=1). Din schemă se vă că O= **not** R. De asemenea, pentru orice intrare egală cu 0 (I=0) automatul trece din starea curentă în cealaltă (din R=0 în R=1 sau invers). Pentru orice intrare egală cu 1 (I=1) automatul îşi păstrează starea.

b)

Adresa(M4[1]M4[0])	Conținutul locației
00	1
01	0
10	0
11	1

Se observă că `M4[1]` este **I** (intrarea în automat) iar `M4[0]` este **R** (starea curentă). Ţinem cont de consideraţiile anterioare (intrarea 1 păstrează starea constantă, intrare 0 schimbă starea).

Adresa(M2[0])	Conținutul locației
0	1
1	0

Se observă că **O** (ieșirea) este dat de conținutul locaţiei, şi este negaţia stării R. În finalul acestui curs, anticipând puţin ce va fi prezentat în cursurile următoare, este ilustrată diagrama fluxului de date din cadrul calculatorului LC-3. Aceasta conţine toate structurile logice digitale, combinaţionale şi secvenţiale, care combinate îndeplinesc funcția de procesare a informaţiilor, strează starea const

4. Componentele unui sistem de calcul

Structura unui sistem de calcul

Ansamblul de componente hardware (dispozitive fizice) şi software (programe) ce soluţionează anumite probleme descrise sub forma unui algoritm reprezintă un sistem de calcul.

Structura logică a unui sistem de calcul

Un calculator este un sistem stratificat pe mai multe nivele ierarhice:

1. Nivelul fizic (mașină) este alcătuit din componente electronice și mecanice. La acest nivel se lucrează cu secvenţe de biţi (coduri de instrucțiuni și date).
2. Limbajul de asamblare, permite programarea calculatorului prin instrucțiuni simple exprimate prin mnemonici. Unui mnemonic îi corespunde un cod de instrucțiune.
3. Interfața dintre resursele sistemului de calcul și aplicațiile ce urmează a fi executate este sistemul de operare care oferă un limbaj sau un mediu de operare și un set de rutine predefinite (drivere) pentru lucrul cu aceste resurse.
4. Limbajele de nivel înalt şi mediile de programare. Majoritatea limbajelor de programare dispun de o bibliotecă bogată de funcții prin care se pot accesa resursele calculatorului.
5. Aplicaţiile sunt produse program care facilitează accesul la resursele unui calculator şi pentru personal nespecializat în domeniul calculatoarelor.

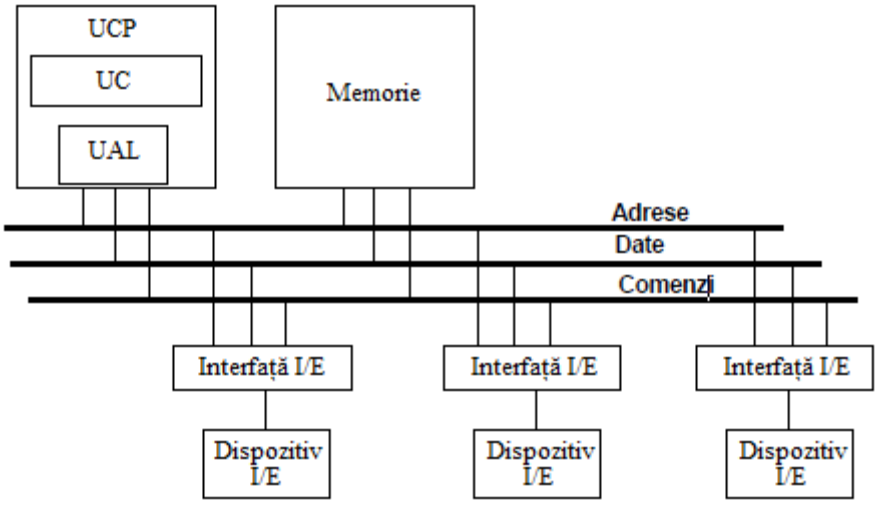


Figura 4.1 Structura fizică a unui calculator

Conform modelului definit de J. Von Neumann, un calculator cuprinde 5 tipuri de componente:

- dispozitive de intrare (ex: tastatură, mouse, interfețe de proces, etc.)
- memorie (interna și externă, volatilă și nevolatilă)
- unitate aritmetico-logică
- unitate de comandă
- dispozitive de ieşire (ex: ecran, imprimantă, etc.).

4.1. Magistrale

Pentru conectarea componentelor sistemului de calcul se folosesc una sau mai multe magistrale. Magistrala se defineşte ca un mediu de comunicație între componentele unui calculator și se compune dintr-un set de semnale prin care se transmit date și comenzi.

Pe magistrală, transferul de date se face pe baza unui set de reguli care stabilesc cine, când și cum comunică pe magistrală; de asemenea stabilesc secvenţa de apariție a semnalelor, intercondiționările existente între semnale și relațiile temporare dintre semnale.

4.1.1. Magistrale și standarde de magistrală

Din punct de vedere conceptual, magistrala este un mediu comun de comunicație între componentele unui sistem de calcul; fizic aceasta este alcatuită dintr-un set de semnale ce facilitează transferul de date și sincronizează componentele sistemului.

Funcție de numărul semnalelor utilizate pentru transferul de date, magistralele pot fi de două tipuri: *magistrale paralele* și *magistrale seriale*.

O magistrală se compune din urmatoarele tipuri de semnale:

- semnale de date - linii bidirectionale utilizate pentru transferul de date
- semnale de adresa - specifică adresa modulului destinatie
- semnale de comandă - specifică directia de transfer și tipul de modul I/E
- semnale de control - reglarea condițiilor de transferare a datelor
- semnale de întrerupere - aparitia unor evenimente interne sau externe
- semnale de ceas - folosite pentru sincronizare
- semnale de alimentare - folosite pentru alimentarea modulelor sistemului
- semnale de control al accesului - în cazul magistralelor multimaster.
-

4.1.2. Clasificarea magistralelor

1. După modul de lucru (în raport cu semnalul de ceas):

- magistrale sincrone la care ciclurile de transfer sunt direct corelate cu semnalul de ceas (e.g. magistrala Pentium)
- magistrale asincrone la care nu există o legatura directa intre evolutia în timp a unui ciclu de transfer și ceasul sistemului (e.g. ISA, IDE, PCI)

2. După numărul de unități master conectate pe magistrală

- magistrale unimaster: un singur modul master pe magistrală ; nu necesită mecanisme de arbitrare a magistralei
- magistrale multimaster: trebuie să conțină semnale de arbitrare și un protocol de transfer al controlului

3. După modul de realizare a transferului de date

- magistrale cu transfer prin cicluri (magistrale secvențiale) ciclurile de transfer se desfășoară secvențial. La un moment dat cel mult un ciclu de transfer este în curs de desfășurare.
- magistrale tranzacționale – transferul de date se efectueaza prin tranzacții; o tranzacție este divizată în mai multe faze și mai multe tranzacții se pot desfășura simultan cu condiția ca tranzacțiile să fie în faze diferite (fiecare fază a unei tranzacții folosește un subset din mulțimea semnalelor magistralei) și deci factorul de creșterea vitezei este egal cu numarul de faze în ale tranzacției (magistrala procesorului Pentium)

Sistemele pot avea o singura magistrală (single bus) sau magistrale multiple (multiple bus).

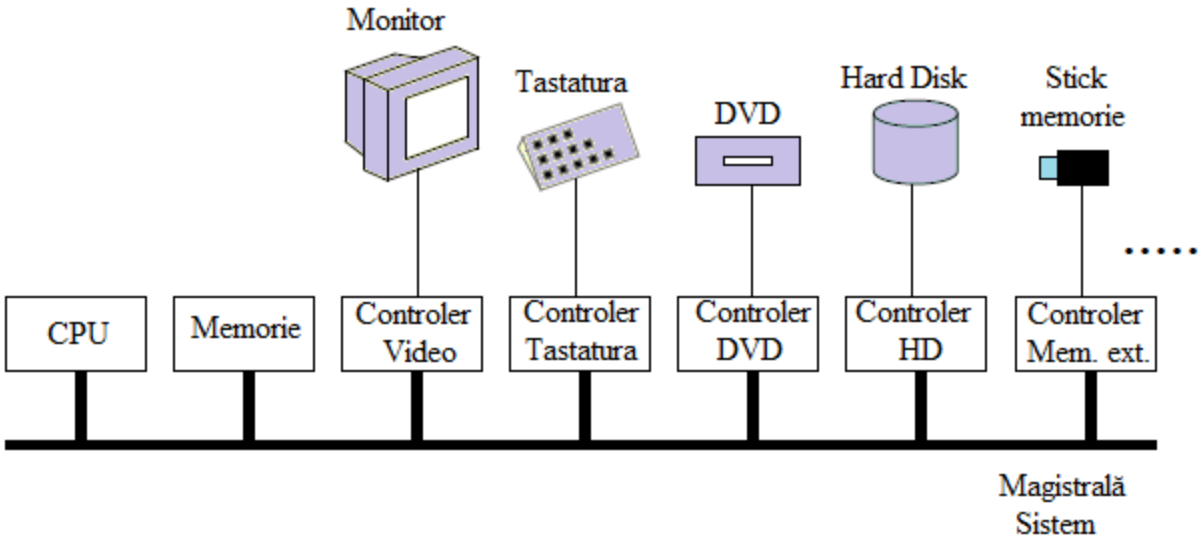


Figura 4.2. Bus sistem tip single

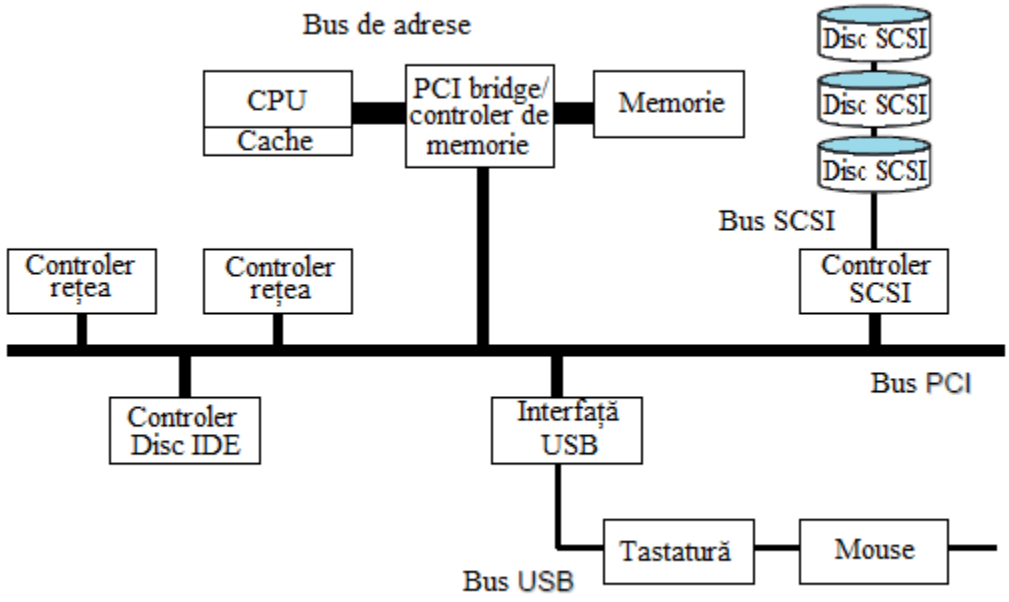


Figura 4.3. Bus sistem tip multiple

4.1.3. Standarde ale magistralelor

Magistralele au caracteristici constructive definite de cerințele funcționale ale procesoarelor pentru care au fost concepute ceea ce duce la dependența de producător a mediului de comunicație. Standardele de magistrală s-au adaptat familiilor de procesoare produse de marile companii din industria de calculatoare ex : Intel - Multibus , IBM – PC-bus, DEC - Unibus, Hewlet-Packard - GPIB, etc.). Printre aceste magistrale unele sunt deschise, adică standardul este accesibil si

utilizabil de orice alt producator, dar altele au restrictii de utilizare (ex : IBM - magistrala Microchannal a firmei din calculatoarele PS 2).

Grupuri de producători au dezvoltat standarde comune (CAMAC, Fastbus, Futurebus) unde se asigură interoperabilitatea dintre modulele diverșilor dezvoltatori.

Pentru a verifica standardele interoperabilitate componentele sunt supuse la teste de conformanță. Standardizarea este definită de foruri internaționale cum ar fi IEEE, ANSI și IEC. Cele mai utilizate magistrale sunt prezentate mai jos, însă nu toate sunt formal standardizate.

IEEE	**Denumire**	**Utilizare**
488	GPIB	Magistrală pentru instrumente de laborator
583	CAMA	Magistrală pentru achizitie de date si instrumentație
696	S100	Microsisteme de dimensiune medie
796	MULTIBIS I, II	Microsisteme de dimensiune medie
P896	Futurebus	Sisteme multiprocesor
P996	PC Bus	Pentru calculatoare personale
P1014	VME Bus	Sisteme microprocesor performante (bazate pe familia Motorola 68000)
P119	Nubus	Sisteme multiprocesor
	Unibus	Minicalculatoare , PDP 11
	QBus	Minicalculatoare, VAX
	SCASI	Magistrala pentru periferice (disc, bandă)

Tabelul 4.1 Standarde de magistrală

Pot conlucra mai multe standarde de magistrală într-un sistem de calcul, fiecare specializat pe transferul de date între anumite componente de sistem. De exemplu, pentru transferul dintre procesor și memorie se folosește magistrala de mare viteză, pentru conectarea unor periferice rapide (disc, interfata video) se utilizează magistrala cu acces multiplu (multimaster) iar o magistrală de mică viteză se folosește pentru periferice lente. Pentru achiziția datelor de proces în aplicațiile de control se utilizează o magistrală adaptată de instrumentație.

După cum se vede, s-au dezvoltat familii de magistrale pentru a acoperi toată gama de cerințe. Acestea sunt variante ale unei magistrale de bază fiind oarecum compatibile între ele (interfețele software dezvoltate necesită modificări minore pentru compatibilizare). Unele dintre aceste semi-standarde de magistrale sunt dezvoltate pentru:

- procesoarele Intel – MULTIBUS I, MULTIBUS II și extensii ale acestora
- procesoare Motorola – magistralele VME
- calculatoarele DEC – Unibus, Qbus si VAXBI
- familia GPIB (a Hewlet Packard), IEEE 488 si Camac.

Dintre standardele impuse pentru calculatoarele personale se pot aminti:

- ISA (Industrial Standard Architecture)– magistrală de sistem a primelor calculatoare personale compatibile IBM PC si care se regăsește în majoritatea calculatoarelor personale
- EISA (Extended ISA) - varianta extinsă a magistralei ISA
- VESA Local Bus (Video Electronics Standard Association)– magistrala proiectată inițial pentru accelerarea transferului intre procesor si interfața video-grafica care s-a extins și pentru alte tipuri de interfețe de mare viteză (ex. disc)

- SCSI (Small Computer System Interface) – magistrală pentru conectarea dispozitivelor de stocare externa a datelor (disc, banda magnetica)
- PCI – magistrală de mare viteză adaptată cerințelor procesoarelor din familia Intel.

4.1.3.1. Magistrala ISA

Magistrala ISA s-a impus odată cu calculatoarele personale compatibile IBM PC XT și AT. Prin acest bus sunt conectatea în sistem interfețele de intrare/ieșire specifice unui calculator personal: interfața de disc, interfața video, interfața de sunet, interfețe utilizator, etc. și este o magistrală asincronă, ce poate să transfere date pe 8 și respectiv 16 biți.

Semnalele magistralei se regăsesc pe sloturile de extensie ale calculatorului personal . Un slot se compune din doi conectori, de 64 si respectiv 36 de pini.

Transferul de date se realizează pe bază de cicluri și se disting 5 tipuri de cicluri funcție de direcția de transfer:

1. ciclu de citire memorie
2. ciclu de scriere memorie
3. ciclu de citire port de intrare
4. ciclu de scriere port de ieșire
5. ciclu de rezolvare a întreruperii (identificare a sursei de întrerupere)

Diagrama de timp prezintă un ciclu de transfer ce presupune activarea și dezactivarea unor semnale (mai jos este prezntată diagramele de timp pentru un ciclu de citire memorie și pentru un ciclu de scriere memorie).

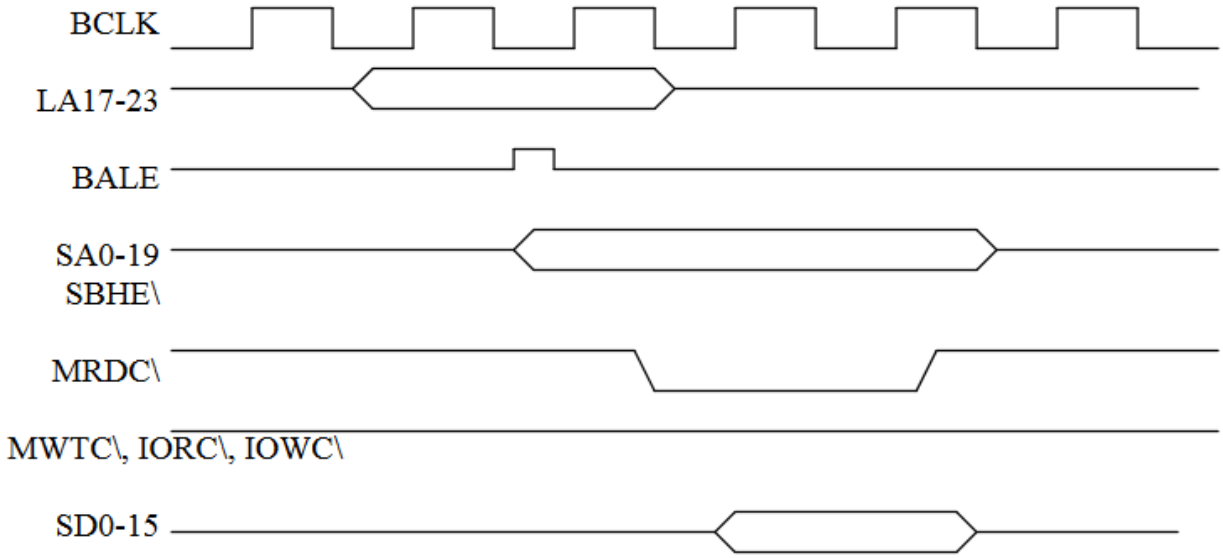


Figura 4.4. Ciclul citire memorie

Tipul de ciclu este indicat prin activarea semnalului de comandă corespunzător :

- *MRDC* (Memory Read Command),
- *MWTC* (Memory Write Command),
- *IORC* (Input Output Read Command),
- *IOWC* (Input Output Write Command)
- *INTA* (Interrupt Acknowledge).

Iar:

- *BCLK* = Bus Clock; semnal de ceas; frecvența semnalului variază tipic între 4.77 MHz si 8MHz
- *LAn* = Lachable Address ; linii de adrese nememorate (sunt valide pe durata semnalului BALE); n=16..23
- *BALE* = Bus Address Latch Enable ; frontul căzător al acestui semnal indică prezența unei adrese valide pe magistrală ; adresa este înscrisă în buffere pe frontul urcător al semnalului
- *SA0-19* = Semnale de adresă, menținute valide pe toată durata ciclului de transfer
- *SBHE* = System Bus High Enable ; semnal ce indică un transfer pe partea superioară a magistralei de date (SD8-15); împreună cu SA0 controlează lățimea cuvântului de date care se transferă
- *SD0-15* = System Data lines ; linii bidirecționale pentru transferul de date.

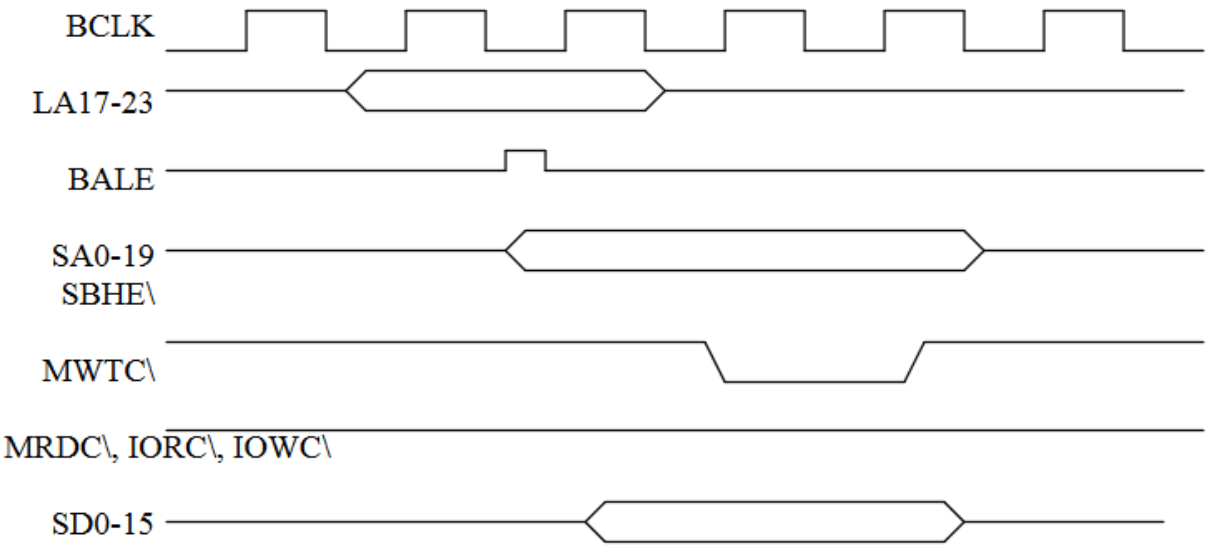


Figura 4.5. Ciclul scriere memorie

Diferența dintre cele două diagrame este dată de:

- la un ciclu de citire, întâi se activează semnalul de comandă (MRDC\) și apoi data citită apare pe liniile de date
- la un ciclu de scriere procesorul pune întâi data pe liniile de date și apoi activează semnalul de scriere.

Pentru ambele cazuri semnalele de adresa trebuie sa fie valide pe toată durata ciclului de transfer.

Ciclurile de citire si scriere a porturilor de intrare/iesire au diagrame de timp similare, cu diferența că în locul semnalelor de comandă pentru memorie (MRDC\ și MWTC\) se activează semnalele corespunzătoare pentru interfețele de intrare/iesire (IORC\ si IOWC\).

Pentru modulele mai lente ciclul de transfer se poate prelungi prin dezactivarea temporară a semnalului CHRDY ; în acest caz ciclul se prelungește cu un număr întreg deperioade de ceas.

4.1.3.2. Magistrala PCI

Magistrala Peripheral Component Interconnect (PCI) este deosebit de fiabilă (bune performanțe și cost redus) fiind adaptată cerințelor de viteză ale procesoarelor de generație mai nouă. Standardul a fost impus de firma Intel și apoi adoptat de alte companii importante (Machintosh, SUN, etc).

Aceasta dispune de 32 de linii de date (cu varianta extinsă 64 linii) si 32 de linii de adresă la o frecvență de lucru uzuală de 33MHz, rezultând o viteză maximă de transfer de 132 Mbps (264 pentru varianta pe 64 de biti). Liniile de adresă și date sunt multiplexate pentru a reduce spațiul ocupat de conector si implicit pentru a reduce costurile.

În mod uzual magistrala lucrează în rafală (burst) speculând astfel posibilitățile rapide de transfer pentru circuitele de memorie dinamică. La inițierea unui transfer se transmite adresa de unde începe blocul de date urmând ca datele sa fie transferate fără validarea fiecărei celule de memorie, acestea fiind consecutive (concatenate).De precizat că nu există limită privind lungimea ciclului de transfer in rafală.
Sunt posibile trei tipuri de adrese (spații de adresare) :

- memorie,
- intare/iesire,
- configurare.

La inițializarea sistemului, pentru configurarea automată (plug-and-play) a plăcilor PCI, un dispozitiv PCI este configurat cu informații referitoare la tipul dispozitivului, producător, caracteristici constructive, registrul pentru adresa de bază, registrul pentru latență, etc.

Fiecărei placi i se asignează o adresă de bază și eventual un nivel de întrerupere. Aceste plăci pot conține o memorie ROM cu programe pentru inițializarea si utilizarea eficientă a cardului.

Standardul permite cuplarea mai multor unități de tip master. Unitatea care face legătura dintre magistrala PCI și magistrala procesorului gazdă va arbitra centralizat accesul la bus-ul master. Latența primită la inițializare de către fiecare dispozitiv PCI master indică timpul maxim de păstrare a controlului magistralei de către dispozitivul respectiv.

Adaptarea magistralei PCI la magistralele noilor variante de procesoare se produce printr-un circuit specializat de tip bridge, ce leagă cele două magistrale.

În concluzie, ținând cont de performanțele ridicate ale magistralei PCI, aceasta a fost adoptată ca standard de multe companii cu mari șanse de rămâne în continuare în topul magistralelor.

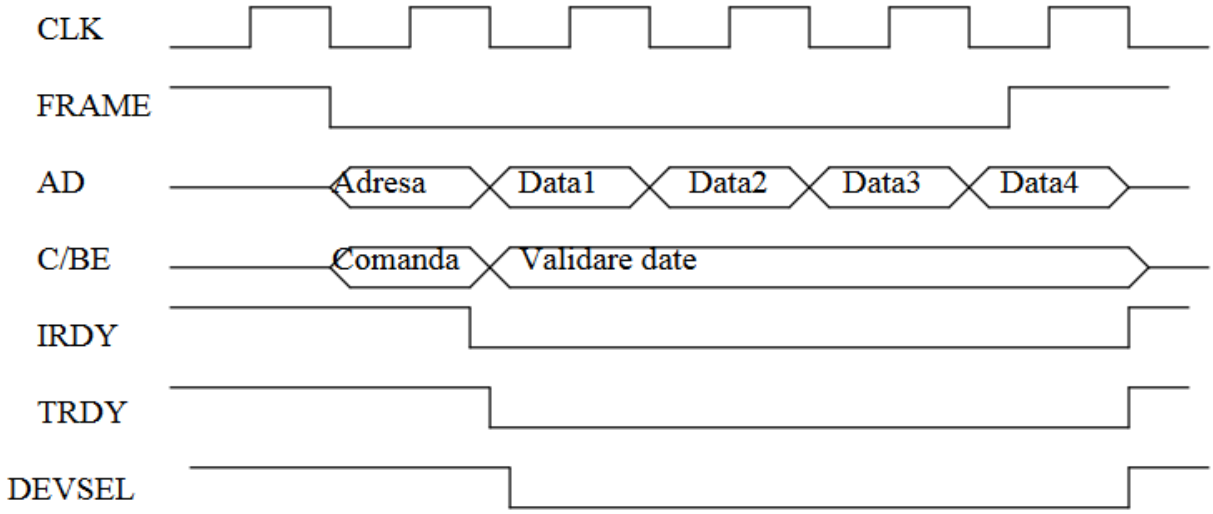


Figura 4.6. Ciclul de transfer în rafală al magistralei PCI

Unde:

- *CLK* = Clock ; semnal de ceas (uzual max. 33MHz)
- *FRAME* = Semnal folosit pentru a indica dacă ciclul este în fază de adresă sau în fază de date

- *AD* = Address/Data lignes; semnale de adrese și date multiplexate; n=0..31
- *C/BE* = Comand, Byte Enable; semnale multiplexate în timp: în primă fază a ciclului de transfer indică tipul ciclului iar în partea a doua controlează lățimea cuvântului transferat; n=0..3
- *IRDY* = Initiator Ready
- *TRDY* = Target Ready ; unitatea adresată pregatită pentru transfer
- *DEVSEL* = Device Select

4.2. Unitatea centrală, procesorul (execuţia unei instrucţiuni, tipuri de unităţi centrale, exemple de microprocesoare)

Microprocesorul este elementul central al structurii unui sistem de calcul, fiind responsabil cu aducerea din memorie, decodificarea şi execuţia instrucțiunilor maşină, codificate binar. În conformitate cu aceste instrucțiuni, microprocesorul generează secvenţial toate semnalele (adrese, date, comenzi) necesare memoriilor şi interfeţelor pe care le gestionează.

Acesta conţine regiştri interni, unităţi de execuție, o unitate de comandă cablată sau microprogramată, bus-uri interne de interconectare etc. În general microprocesorul este integrat pe un singur circuit.

În sfera comercială, primul microprocesor (pe 4 biţi) s-a realizat în anul 1971 la compania *Intel* şi a fost proiectat de către inginerul *Tedd Hoff*. S-a numit Intel 4004. (În domeniul militar existau asemenea sisteme integrate complexe; spre ex. în comandă avioanelor militare americane F14A a existat un microsistem pe 20 de biţi, cu procesare *pipeline* a instrucţiunilor).

Magistrala (bus) de adrese este unidirectională de tip *tree state* (TS). Prin intermediul acestui bus microprocesorul pune adresa de acces a memoriei sau a porturilor I/O (*Input/Output*).

Microprocesorul poate "vedea" în exterior exclusiv din memorii şi interfețe de intrare/ ieșire. Acestea sunt resursele ce pot fi accesate (scris /citite) de către microprocesor. Aşadar, acesta nu "vede" în mod direct perifericele ci interfeţele specifice de I/O.

Magistrala (bus) de date este de tip bidirecţional TS. Prin intermediul acestui bus microprocesorul aduce din memorie instrucțiunea, respectiv citeşte data (operandul) din memorie sau dintr-un port de intrare (arhitectura *Princeton* de memorie).

Pentru scriere microprocesorul plasează pe *bus-ul de date* rezultatul pe care doreşte să-l scrie în memorie sau într-un port de ieşire. Pentru citire, bus-ul preia rezultatul din memorie sau dintr-un port de intrare. Microprocesorul activează, în ambele cazuri, adresa respectivă pe bus-ul de adrese împreună cu semnalele de comandă aferente (*Read, Write*, Memorie, Interfaţă etc.) pe bus-ul de comenzi.

Pe bus-ul de stări, dispozitivele *slave* (memorii, interfețe) comunică informaţii referitoare la modul de desfăşurare al transferului (ex. semnalul emis spre microprocesor "aşteaptă", având semnificaţia că transferul de date comandat nu este încă încheiat).

Unitatea centrala de prelucrare (UCP) reprezintă elemental principal al unui calculator, executând practic programele stocate în memorie. Componentele care asigură disponibilitățile UCP sunt:

a. **U**nitatea de **C**omandă (UC);
b. **U**nitatea **A**ritmetica și **L**ogica (UAL);
c. **R**egistrele **G**enerale (RG).

a. Unitatea de Comandă aduce instrucțiunile din memorie, determină tipul lor după care descompune fiecare instrucțiune într-o secvența de faze. Fiecare fază este caracterizată de un set de microcomenzi a căror finalitate este reprezentată de efectuarea unor operații elementare în unitățile de execuție ale UCP respectiv *registre, UAL, memorie, interfețe de intrare / ieșire*. Unitatea de comandă conține mai multe module care fac posibilă realizarea funcțiilor sale.

- ***R**egistrul de **I**nstructiuni* **(RI)** păstrează codul instrucțiunii în curs de execuție.

- *Numărătorul de **P**rogram* (**NP**) conține adresa instrucțiunii curente.
- ***R**egistrul de **S**tare* (**RS**) este format dintr-un set de bistabile de stare care contin informatii legate de modul de execuție a instrucțiunilor (validarea întreruperilor, execuția pas cu pas, etc.), rezultatele operațiilor aritmetice și logice (depasire de cxapacitate de reprezentare, transport spre rangul superior, etc.), sau informatii legate de conținutul anumitor registre (par sau impar, zero, etc.).
- ***B**locul **C**ircuitelor de **C**omandă* (**BCC**) semnalele de comandă (specifice tipului unei instrucțiuni, fazei curente și informatiei de stare)necesare executiei instrucțiunilor.
- ***G**eneratorul de **T**act* (**GT**) generează semnalul de tact (a carui frecventa determină viteza de lucru) pentru functionarea sincrona a întrgului calculator.
- ***G**eneratorul de **F**aze* (**GF**) creează succesiunea specifică de faze care compun o instrucțiune. Faza următoare este determinată de faza curenta, tipul instrucțiunii și informația de stare din UCP.

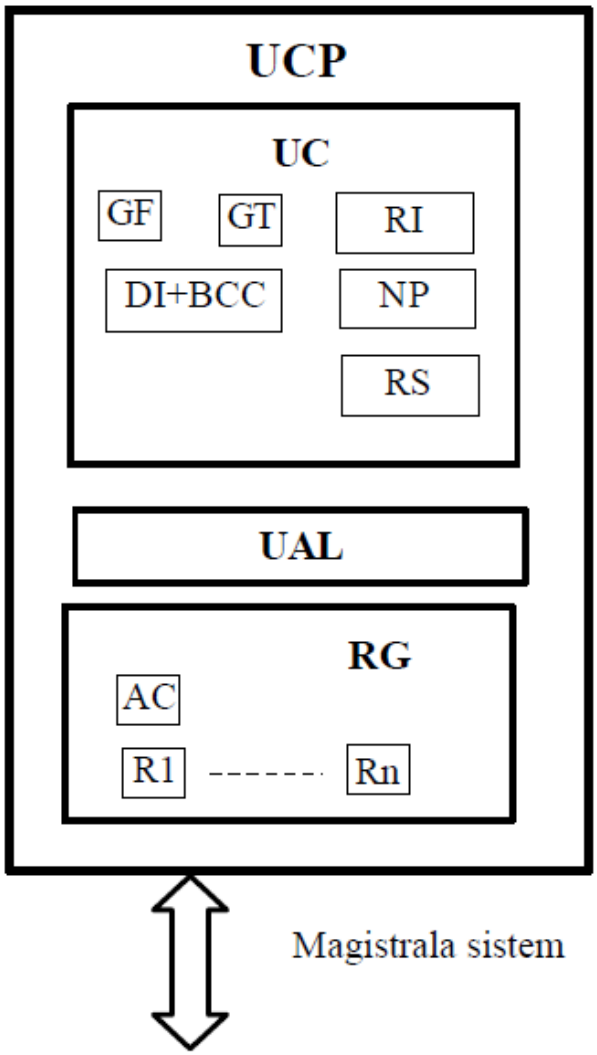


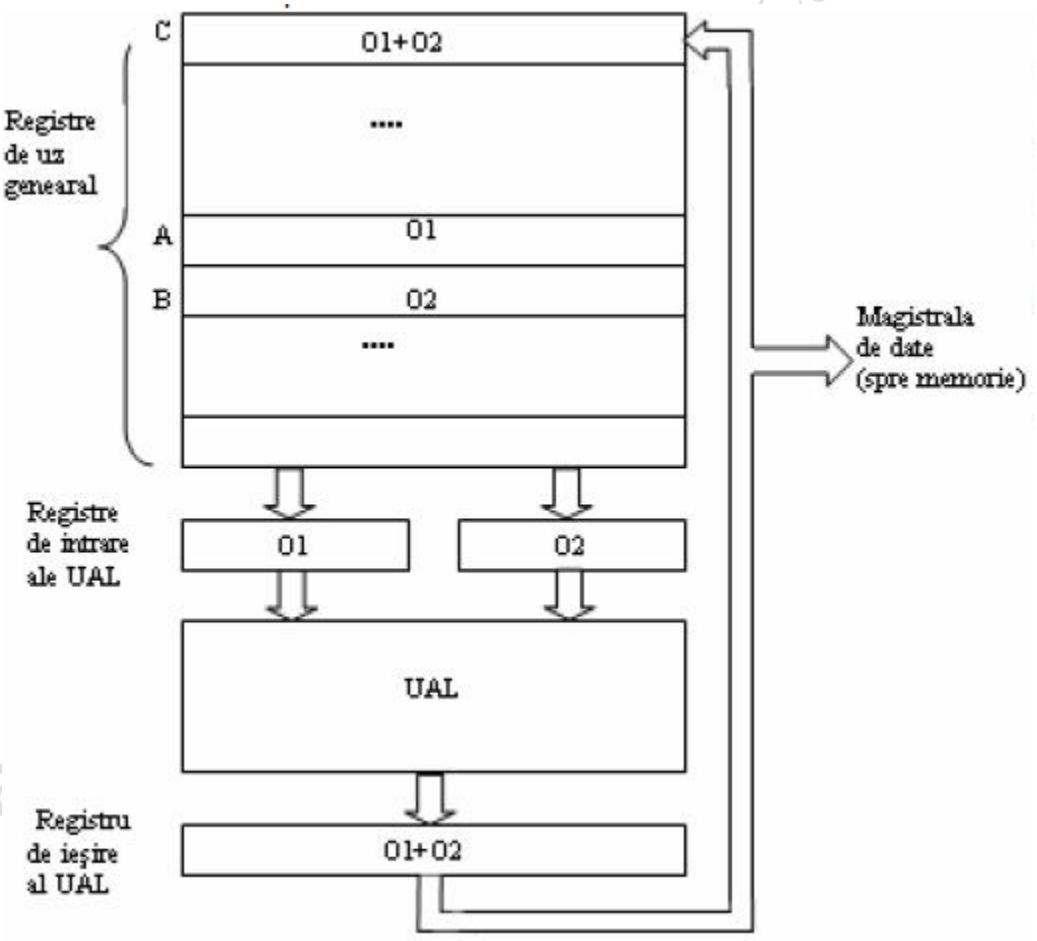
Figura 4.4. Structura UCP

b. Unitatea Aritmetica și Logica (UAL) execută totalitatea operațiilor aritmetice (adunare, scădere, înmulțire, împărțire) și logice (Si, SAU, NU, SAU EXCLUSIV, etc.). O importantă aparte pentru UAL prezintă registrul acumulator AC, care de multe ori este tratat ca făcând parte din aceasta.

c. Registrele Generale (RG) pastrează date cu care lucrează programul în execuție, având rolul unor locații de memorie rapidă. Registrul acumulator este implicat în toatalitatea operațiilor efectuate de UAL pastrând unul din operanzi și rezultatul.

4.2.1. Structura unui procesor

Unitatea aritmetica și logica. Registre mai importante Procesorul este format în principal din unitatea aritmetica și logica (UAL) și mai multe registre, care sunt legate împreuna prin magistrale interne. Unul dintre cele mai importante registre al unui procesor este registrul numărător de program (program counter, PC). Acesta conține totdeauna adresa din memorie a instructinunii care se executa (sau urmeaza să se execute), și îsi modifica conținutul în ritmul executarii programului. De asemenea, un alt registru important este și registrul de instrucțiuni, care păstrează instrucțiunea în curs de execuție. În plus, UCP conține mai multe registre de uz general. UAL executa asupra datelor de intrare operații aritmetice (adunari, scaderi, înmultiri,împartiri), operații logice (SI, SAU) și alte operații simple. În figură se prezintă o parte a UCP, numită „calea de date von Neumann", având ca element central UAL.



În continuare se exemplifica realizarea unei adunari de catre aceasta structura.

- Cei doi operanzi, O1 și O2, se află în două registre de uz general, A și B.
- Operanzii sunt transferati în cele două registre de intrare ale UAL.
- UAL calculeaza rezultatul adunarii, și depune rezultatul în registrul de ieșire.

– Rezultatul este mutat în alt registru de uz general al UAL, C.Apoi, dacă se dorește, conținutul registrului C poate fi transferat în memorie.

Majoritatea instrucțiunilor este de tipul registru-registru sau registru-memorie.
În primul caz, operanzii (operandul) sunt extrasi din registre și adusi în registrele de intrare ale UAL, iar rezultatul este depus într-un alt registru.
În al doilea caz, operanzii (operandul) sunt adusi din memorie în registre, de unde pot fi folositi ca date de intrare pentru UAL.
Rezultatul este depozitat înapoi în memorie. În principiu, pasii pe care îi realizeaza procesorul la execuția unei instrucțiuni sunt urmatorii:

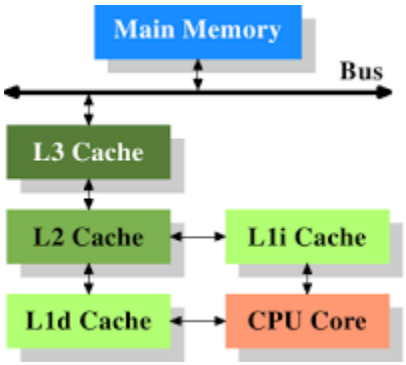
1. Transferă instrucțiunea din memorie în registrul de instrucțiuni (extrage instrucțiunea).
2. Schimbă numărătorul de program astfel încât acesta să indice adresa urmatoarei instrucțiuni (conținutul acestuia creste cu un număr egal cu numărul de octeți al instrucțiunii în curs de execuție).
3. Determină tipul instrucțiunii proaspat extrase (decodifica).
4. Daca instrucțiunea are nevoie de un operand (cuvânt) din memorie, determină unde se gaseste acesta.
5. Extrage (aduce) cuvântul respectiv în unul dintre registrele UCP, dacă este cazul.
6. Executa instrucțiunea.
7. Salt la pasul 1 pentru a începe execuția instrucțiunii următoare.Deseori, aceasta secvență de pași este denumită ciclul extrage-decodifică-execută (fetch-decode-execute).

Trebuie precizat ca există situații când execuția programului conține ramificatii sau salturi, adica instrucțiunile executate nu sunt situate la locații succesive în memorie.În acest caz la punctul 6 de mai sus efectul instrucțiunii este ca în registrul numărător de program este introdusa adresa instrucțiunii de salt, care este diferita de adresa instrucțiunii introduse la punctul 2.

Tehnica pipeline
În scopul cresterii vitezei de execuție a instrucțiunilor, procesoarele folosesc tehnica numita pipeline (banda de asamblare sau conducta).Astfel, execuția unei instrucțiuni este împartita în mai multe parti, de fiecare parte ocupându-se o componenta hardware dedicata (unitate) a procesorului. Toate unitățile hardware pot să functioneze în paralel. De exemplu la o banda de asamblare cu 4 unități (numite și segmente) avem: S1 pentru extragere instrucțiune, S2 pentru decodificare, S3 pentru extragere operanzi iar S4 pentru execuție. Functionarea este prezentata în continuare: -pe perioada ciclului 1, unitatea S1 extrage instrucțiunea 1 (I1) din memorie; -pe perioada ciclului 2, unitatea S1 extrage instrucțiunea 2 (I2) din memorie, iar unitatea S2 decodifica I1; -pe perioada ciclului 3, unitatea S1 extrage instrucțiunea 3 (I3) din memorie, unitatea S2 decodifica I2, iar unitatea S3 extrage operanzii pentru I1; -pe perioada ciclului 4, unitatea S1 extrage instrucțiunea 4 (I4) din memorie, unitatea S2 decodifica I3, unitatea S3 extrage operanzii pentru I2, iar unitatea S4 executa I1. Astfel, după 4 cicluri, instrucțiunea 1 este executata complet, instrucțiunea 2 este executata în proportie de 75%, etc.

Memoria intermediară
În general, procesoarele pot lucra la viteze mult mai mari în raport cu memoriile. Introducerea unei memorii mai mari în interiorul UCP i-ar creste acesteia dimensiunile și costul. Memoria intermediara sau cache (de la frantuzescul cacher, a ascunde) este o memorie de capacitate mai mica dar foarte rapida. Memoria principala a calculatorului (identificata de obicei ca RAM) are o capacitate foarte mare dar este mai lenta. Ideea de baza a memoriei intermediare este urmatoarea: cuvintele de memorie cele mai frecvent utilizate sunt pastrate în memoria intermediara; doar când UCP nu gaseste un cuvânt, îl cauta în memoria principala. Orice procesor are un controler de memorie cache care este responsabil cu aducerea datelor din memoria

principala în cea de tip cache. O caracteristica a memoriei cache este raportul dintre cache hit (numărul de accesari reusite la memoria cache, adica atunci când datele de care procesorul avea nevoie au fost gasite în memoria cache) și numărul total de accesari.

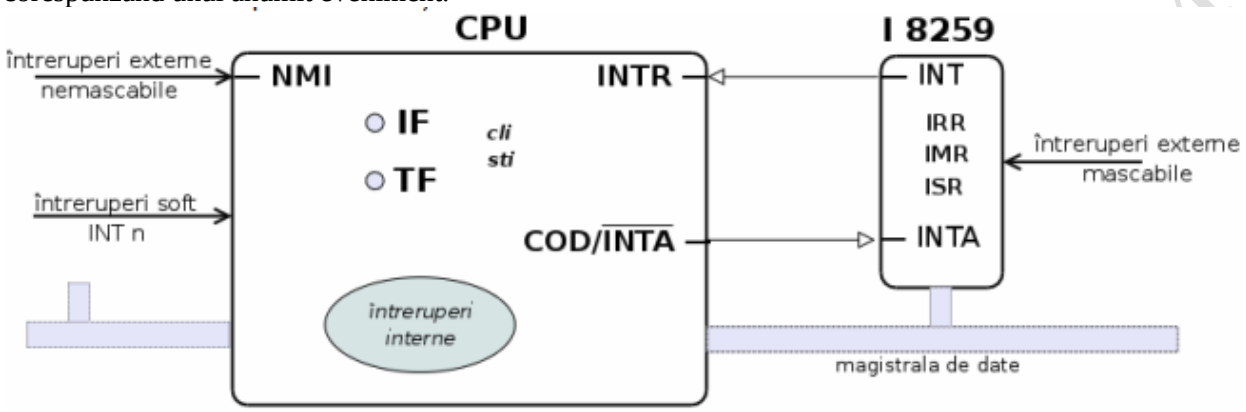


Termenul de cache miss se refera la cazurile în care datele solicitate de procesor nu au fost încarcate în prealabil în memoria cache, urmând să fie preluate din memoria principala, mai lenta. De exemplu, un procesor Intel Core Duo 6600 are o memorie intermediara pe două niveluri. Exista o pereche de memorii intermediare de 32 KB pentru instrucțiuni și 32 KB pentru date, chiar în cipul UCP (level 1- nivelul 1), precum și o memorie intermediara unificata, situata tot în interiorul capsulei circuitului integrat care gazduieste UCP, legata de UCP printr-o cale de mare viteza (nivelul doi), de 4 MB.
Ambele memorii ruleaza la frecventa cu care lucreaza și procesorul, adica 2 GHz. Procesorul Itanium a fost construit chiar cu 3 niveluri de memorie cache. Pentru alte procesoare dual core sau quad core memoria cache L2 este în gama 6MB -12 MB. Acesata memorie este folosita în comun de toate procesoarele.

Modul protejat
Initial, procesoarele (8088 și 80286) rulau în asa numitul mod real. În acest mod, sistemul de operare (asigura gestionarea resurselor hardware și software ale calculatorului și interfata cu utilizatorul) era de tipul single-tasking, adica numai un program rula la un moment dat. Sistemul de operare folosea instrucțiuni de 16 biti și putea adresa 1 MB de memorie. Nu există protectie incorporata pentru ca un program să nu suprascrie peste un alt program sau chiar peste sistemul de operare. Astfel, dacă rulau mai multe programe, întregul sistem se putea bloca. O importanta facilitate a procesoarelor, aparuta odata cu generatia Intel 386, o reprezintă existenta modului protejat. Conform acestui mod nu se pot face scrieri la orice adresa din spatiul de memorie. Acest mod este folosit pentru facilitarea unui mod de lucru multitasking foarte bun. Modul multitasking înseamna ca pe un calculator pot rula în acelasi timp mai multe programe aplicatie, fara ca ele să interfere unul cu celalalt. Sistemele de operare scrise pentru aceste calculatoare opereaza cu instrucțiuni pe 32 de biti și pot adresa întregul spatiu de memorie. Un program eronat nu putea deteriora alte programe sau sistemul de operare. Putea fi terminat, în timp ce restul sistemului continua să functioneze fara a fi afectat.

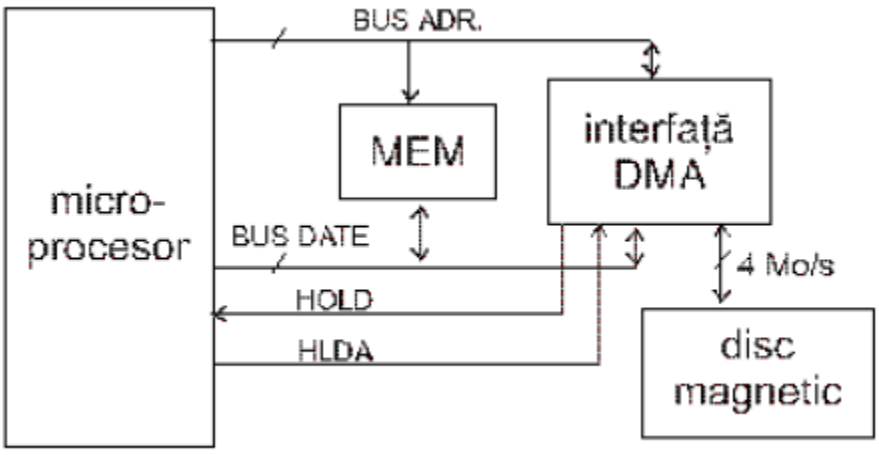
Întreruperi
Un alt aspect important al procesoarelor este conceptul de întrerupere. În principiu prin întrerupere se întelege întreruperea executiei normale a unui program datorita unui eveniment care a avut loc în exteriorul sau în interiorul UCP. Ca efect, programul în rulare va fi întrerupt și se va face un salt la un alt program a carui adresa depinde de evenimentul care a generat întreruperea. Programul respectiv trebuie să realizeze niste actiuni în funcție de evenimentul (sursa) care a generat întreruperea. Dupa efectuarea acelui program, procesorul va continua cu rularea programului initial, din punctul în care a fost întrerupt. O alternativa mult mai putin eficienta este ca UCP să verifice periodic dacă anumite evenimente au avut loc. În acest caz s-ar pierde timp în situatiile în care verificarea se face degeaba. În plus, s-ar putea ca anumite evenimente să apara între două verificari consecutive, iar deservirea lor ulterioara să fie prea târzie. Semnalele (cererile) de întreruperi pot veni de la diverse surse. De exemplu, apasarea unei taste, receptia unui octet la portul

serial sau conectarea unui dispozitiv la un port USB genereaza întreruperi. Acestea sunt întreruperi hardware. Întreruperile initiate în urma executiei unui program se numesc întreruperi software (de exemplu la realizarea unei împartiri cu zero).
IRQ (Interrupt ReQuest line) reprezintă liniile prin care dispozitivele hardware pot cere întreruperi procesorului. Fiecare dispozitiv (porturi, tastatura, hard disc,...) are alocata o astfel de linie, numita IRQi, cu i având valori între 1 și 15. Cele 15 linii sunt conectate la intrarile a două circuite speciale numite controllere de întreruperi, legate în cascada, de unde o singura linie este conectata la o intrare speciala (pin) al procesorului. Procesoarele din familia Intel accepta 256 de tipuri de întreruperi (15 hardware), fiecare corespunzând unui anumit eveniment.



Prin constructie, s-a stabilit modalitatea de determinare a adresei unde se va face saltul pentru deservirea întreruperii, astfel: în memorie există o tabela, cu 1024 de locații, situata chiar la începutul memoriei; pentru fiecare întrerupere există alocati 4 octeți, în funcție de care se calculeaza adresa de salt. Aceasta tabela este creata în timpul procedurii de initializare a sistemului, de catre programul conținut în memoria ROM BIOS.

Tehnica DMA
O alta caracteristica standard a procesoarelor din PC-uri este tehnica DMA. De fiecare data când are loc un transfer între un dispozitiv de intrare-ieșire (prin intermediul unui port) catre memorie sau invers, acesta are loc în două etape: în prima octetul este citit de UCP într-unul dintre registre, iar în a două octetul este scris în memorie.



Daca numărul de octeți care trebuie transferat este mare, aceasta metoda are dezavantajul ca UCP nu poate face altceva în acel timp și, respectiv, transferul se face în doi pași. Pentru a elimina acest neajuns, proiectantii PC-urilor au decis să incorporeze în acesta un circuit (procesor) suplimentar, numit controller de acces direct la memorie (Direct Memory Access, DMA). Astfel, UCP poate să-i spuna controller-ului

de DMA să transfere un bloc de octeți din memorie într-un dispozitiv de ieșire, sau în sens invers, dintr-un dispozitiv de intrare în memorie. Un PC are 8 canale DMA, folosite de placa de sunet, unitățile de hard disc și de discheta, etc.
Când PC-ul este pus sub tensiune, procesorul sare automat la adresa FFFF0h, asteptând să gaseasca aici instrucțiuni de executat. Rezulta ca la adresa respectiva trebuie să existe memorie ROM, care păstrează informația în permanenta. Deoarece aceasta adresa este cu exact 16 octeți mai jos de sfârsitul primului megaoctet (MB) de memorie, care reprezintă tot ceea ce vede procesorul la pornire, la adresa respectiva trebuie înscrisa o adresa de salt la o adresa mai mica.
Un **program** reprezintă o lista secvențiala de instrucțiuni. Un procesor poate să faca în principal urmatoarele trei tipuri de instrucțiuni:

- operații aritmetice și logice pe baza UAL
- poate transfera un operand dintr-o locatie de memorie în alta
- poate testa rezultatul operațiilor efectuate și apoi, funcție de rezultatul testului, poate face salturi în program.

Exista mai multe tipuri de limbaje de programare:

- -Limbaje de nivel înalt (C, Pascal, BASIC)
- -Limbaje de nivel coborât, numite limbaje de asamblare.

Orice program trebuie translatat într-un limbaj mașină pe care îl întelege procesorul. Acest lucru este realizat de programe numite complilatoare, interpretoare sau asambloare.
Exemple de instrucțiuni în limbaj de asamblare:

- LOADA *mem* – încarcă registrul A cu conținutul locatiei de memorie de la adresa *mem*.
- LOADB *con* – încarcă registrul B cu constanta *con*.
- SAVEB *mem* – salveza registrul B în locatia de memorie de la adresa *mem*.
- ADD – aduna registrele A și B și depune rezultatul în registrul C.
- SUB – scade registrele A și B și depune rezultatul în registrul C.
- JUMP *addr* – salt la adresa *addr*
- JNEQ *addr* – salt la adresa *addr*, dacă operandul A nu este egal cu B.

Fiecare instrucțiune în limbaj de asamblare este reprezentată prin 1, 2 sau mai mulți octeți. Aceste numere binare reprezintă limbajul mașină. Astfel, un program este reprezentat printr-un număr de octeți în memorie.

4.3. Tipuri de memorie (memoria principală, memoria de lucru, memoria cache, memoria ROM, memoria video, memoriile secundare)

Memoria poate fi văzută intr-o primă abordare ca o stivă de locații binare (cuvinte), fiecare cuvânt fiind caracterizat de o adresă binară unică.

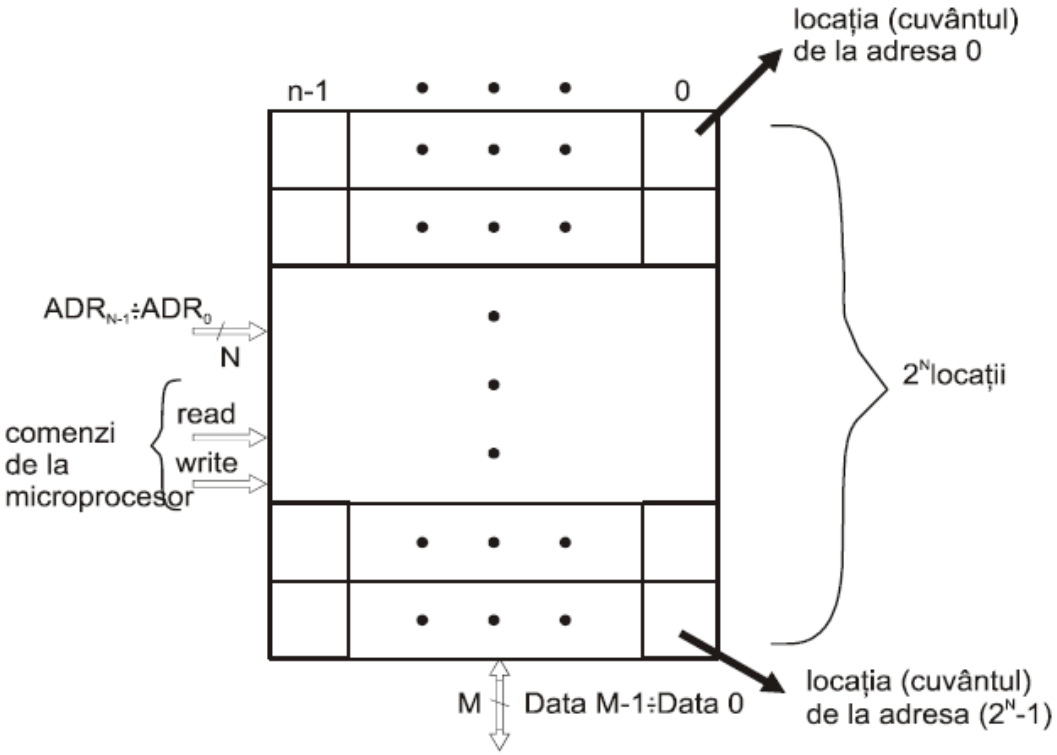


Figura 4.4. Schemă generică de memorie

În general M=8,16,32,64<=> lărgime bus date microprocesor (microsistem pe M biti)
Memoria este caracterizatã prin 2 parametri de bazã:

- capacitatea (nr. de locații pe care le conține)
- latența (timpul de acces) care este o caracteristicã intrinsecã a circuitului de memorie și reprezintã în principiu timpul scurs din momentul furnizării adresei de către microprocesor până în momentul în care memoria a încheiat operația comandatã (citire sau scriere).

Fireste, se doresc capacităţi cât mai mari și latente cât mai mici, cerinţe în general contradictorii.

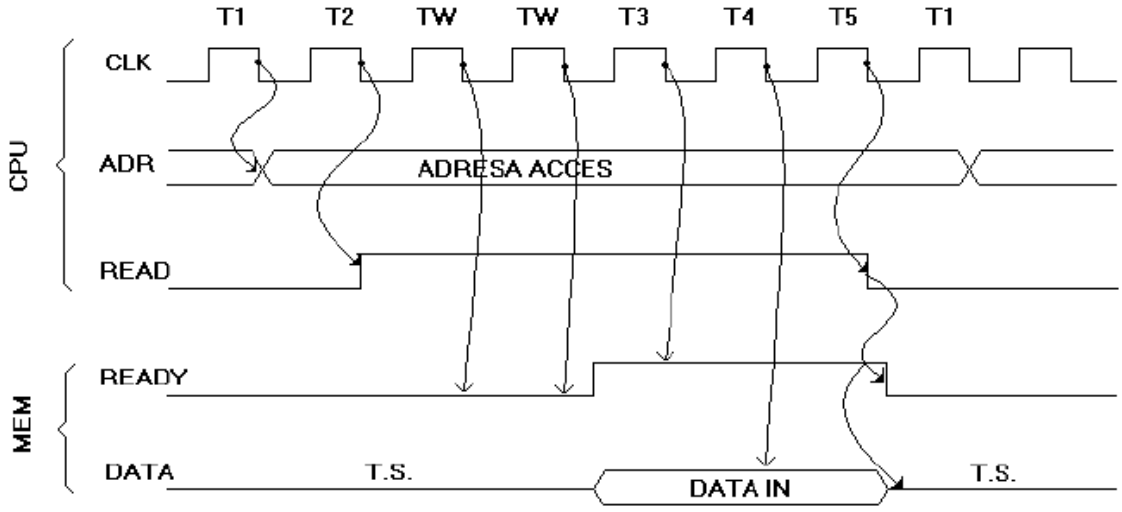


Figura 4.5. Cilclu extern citire memorie

Între busul de adrese și memorie există un decodificator $N:2^N$ ca în figură:

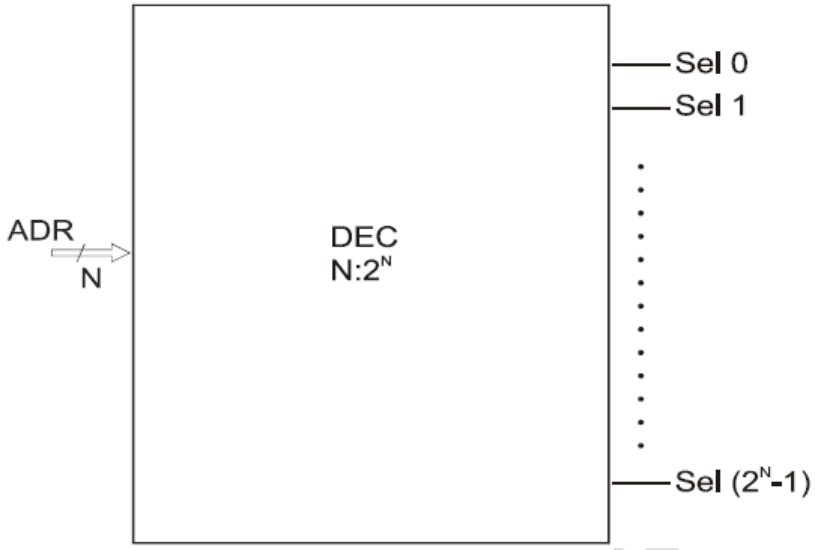


Figura 4.5. Decodificator $N:2^N$

Cu 10 biţi de adrese => 2_{10} cuvinte = 1k cuvânt (kilo)
Cu 20 biţi de adrese => 2_{20} cuvinte = 2_{10} k cuvânt = 1M cuvânt (mega)
Cu 30 biţi de adrese => 2_{30} cuvinte = 2_{10} M cuvânt = 1G cuvânt (giga)
Cu 40 biţi de adrese => 2_{40} cuvinte = 2_{10} G cuvânt = 1T cuvânt (terra)
M = 8 => 1 cuvânt = 1 octet

4.3.1. Clasificare memoriei

Din punct de vedere tehnologic memoriile se împart:

- ROM (*Read Only Memory*) – EPROM, EEPROM
- RAM (*Random Acces Memory*)
- SRAM (static)
- DRAM (dinamic) – uzual este memoria pricipală

Memoriile EPROM sunt memorii rezidente care păstrează conținutul şi după decuplarea tensiunii de alimentare. Ele sunt reprogramabile în sensul în care pot fi șterse prin expunere la raze ultraviolete și reînscrise pe baza unui dispozitiv special numit programator de EPROM –uri.
EPROM-urile păstrează aşa numitul **program monitor** înscris de către fabricant care este primul program procesat de către sistem imediat după alimentarea (resetarea) sa. Acest lucru este absolut necesar întrucât conținutul memoriilor RAM este neprecizabil imediat după alimentare. Prin urmare, imediat după RESET conținutul PC-ului este inițializat şi va pointa spre prima instrucțiune din programul monitor rezident în EPROM.Rolul programului monitor este de a efectua o testare sumară a microprocesorului și a celorlalte componente ale microsistemului, după care va iniția încărcarea sistemului de operare *(Linux, Windows* etc.) de pe disc în memoria RAM. După aceasta programul monitor dă controlul sistemului de operare rezident acum în RAM. De asemenea ROM-ul conține rutinele de intrare – ieșire BIOS.

4.3.1.1. SRAM

Sunt memorii deosebit de rapide, timp de acces de t_0 = 1 ns ÷ 15 ns, având capacitate de integrare redusă (sute de Ko per circuit).

4.3.1.2. DRAM (memoria principală)

Constituie peste 95 % din memoria oricărui sistem de calcul de uz general datorită faptului că oferă densităţi mari de integrare (64 Mbiţi –256 Mbiţi / chip) și timpi de acces "relativ rezonabili" t_0=30 ns ÷ 60 ns.

Totuşi, decalajul între timpul de acces ridicat al acestor memorii și viteza mare de execuţie a microprocesorului, constituie una dintre marile probleme tehnologice și arhitecturale în ingineria calculatoarelor. Fiind realizate în tehnologie MOS puterea absorbită este redusă. Din păcate au 2 mari dezavantaje:

1. Accesare (citire / scriere) complicată. Circuitele DRAM sunt organizate sub o formă matricială, pe linii și coloane. Bitul ce se dorește a fi accesat se află la intersecţia liniei cu coloana selectată.

2. Necesitatea regenerării memoriei DRAM.

Bitul de informaţie DRAM este implementat sub forma unui condensator. Din păcate acest condensator se descarcă în timp și prin urmare cu timpul poate să piardă informația pe care o memorează => deci că periodic el trebuie reîncărcat (*refresh*, regenerare).

Regenerarea se face pe întreg rândul din matricea de memorare. Conform catalogului un anumit rând "ţine minte" informația circa 2 ms. După acest interval, întreg rândul trebuie regenerat. Algoritmul de regenerare va trebui să fie unul de tip secvenţial care să regenereze rând după rând în mod ordonat. Rezultă că rata de regenerare a 2 rânduri succesive *i* respectiv (*i+1*) trebuie să se facă la un interval de maxim 2 ms/N, unde N=nr. de rânduri al circuitului de memorie DRAM.

De exemplu, considerând N=128 rânduri (DRAM 4116, 4164) => rata de regenerare 2 ms/128 ~ 15,6 μs.

Prin urmare vom avea accese la DRAM din 2 părţi:

- din partea μprocesorului care citește / scrie conform programului pe care îl execută
- din partea unui automat de regenerare care regenerează periodic, rând după rând, memoria DRAM.

Posibilele conflicte la memoria DRAM între microprocesor și automatul de regenerare vor trebui gestionate corespunzător, eventual acceptând chiar prin blocaje reciproce, rezultând scăderea performanţei. Ar fi utilă implementarea unei "regenerari transparente" (care să nu blocheze deloc procesorul). Acest deziderat necesită compromisuri între viteza de procesare și gradul de transparenţă al regenerării.

Standardizarea bus-urilor şi a protocoalelor aferente a condus la conceptul fecund de **microsistem de dezvoltare**. Într-un asemenea microsistem, prin simpla conectare la bus-ul comun (date, adrese, comenzi, stări) standard a unor noi module compatibile (memorii, interfețe) se permite o dezvoltare în timp a microsistemului inițial. Spre exemplu, cunoscutele microsisteme standardizate de tip IBM PC, constituie astfel de sisteme de dezvoltare construite în jurul unor magistrale standardizate.

4.3.2. Memoria principală

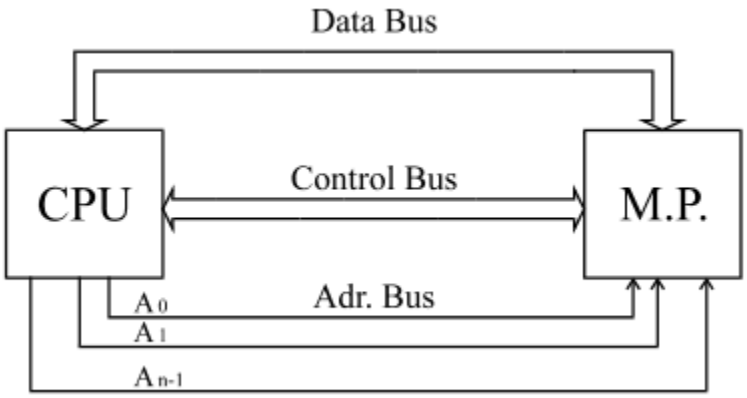
Memoria principală este o memorie operativă adică aici se încarcă programele lansate în execuție și datele aferente.

• Este o memorie construită în tehnologie semiconductoare
• Este o memorie volatilă
• Este o memorie cu acces direct din punctul de vedere al procesorului
• Este o memorie cu acces aleator:

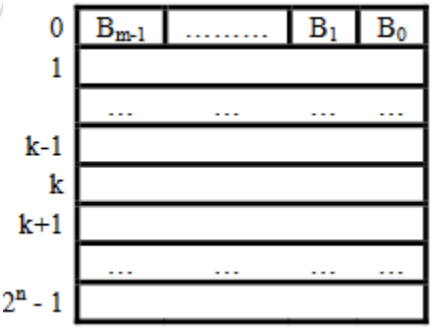
- Memoriile cu acces aleator sunt caracterizate prin faptul că fiecare locaţie poate fi accesată independent, timpul de acces pentru oricare locaţie fiind acelaşi - independent de poziţia locaţiei
- Fiecare celulăconţine linii de date (citire sau scriere - numite linii de bit la majoritatea celulelor bipolare sau unipolare) şi linii de selecţie.

Unitatea de memorie în calculatoarele contemporane se realizează cu ajutorul modulelor integrate. Legătură dintre CPU și memoria principală se realizează prin trei magistrale:

- magistrala de adrese ADRBUS;
- magistrala de date DATA BUS;
- magistrala de control CONTROL BUS.



Magistrala de adrese este unidirecţională, cea de date este bidirecţională, iar cea de control conține linii de comandă, dar și linii informaţie de stare. Memoria este organizata pe locații:

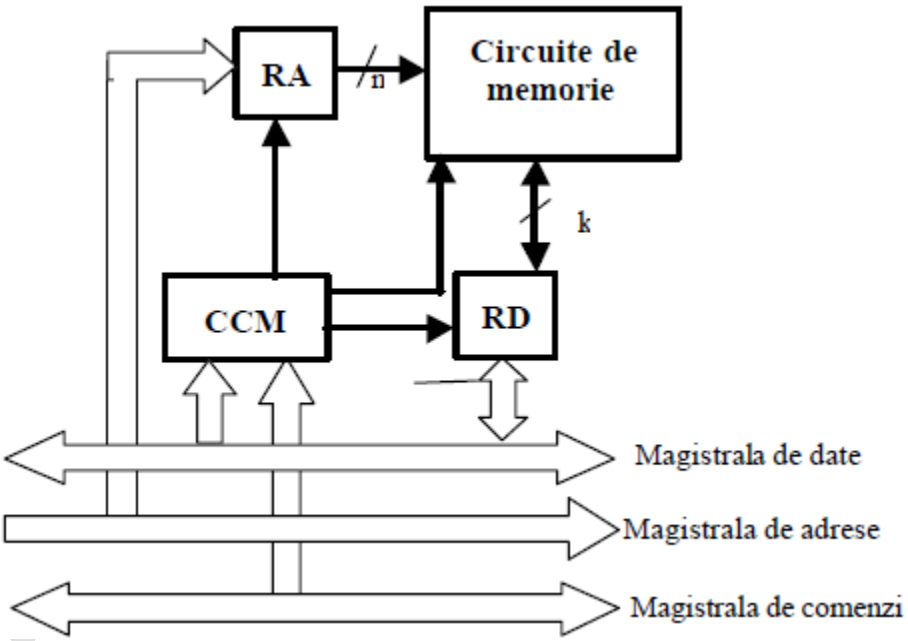


O locaţie are m ranguri notate Bm-1, ... B1, B0.

Citirea şi scrierea memoriei se face paralel, deci, se scrie sau se citeşte un cuvânt pe m biţi. Adresarea memoriei se face prin magistrala de adrese (ADRBUS) și se presupune că această magistrală are n linii. Pentru fiecare combinație de biți ai acestei magistrale se defineşte o anumită adresă. Deci, legătura între conținutul magistralei și o anumită locaţie se face printr-un proces de decodificare. Dacă de adrese are *n* linii atunci mulţimea adreselor este *2n*.

$$A = \{0,1,...2_n - 1\}$$

unde A este spaţiul adreselor.
Pe de altă parte, dându-se o geometrie a memoriei rezultă mulţimea locaţiilor fizice în care sunt memorate cuvinte care conţin m biţi.
Atunci când lăţimea cuvântului memorie nu este suficientă pentru reprezentarea datei, atunci se vor utiliza mai multe locații consecutive. Dacă de exemplu, o dată se reprezintă pe două cuvinte în memorie (UIA), atunci pentru a reprezenta data, se va folosi prima locaţie pentru reprezentarea cuvântului mai puţin semnificativ (de la adresa j), iar locaţia următoare (adresa j+1) pentru cuvântul mai semnificativ.
Procesoarele moderne au facilităţi de gestiune a memoriei. Operaţiile de gestiune se referă la implementarea memoriei virtuale, protecţia memoriei etc.
Unitatea de gestiune a memoriei (Memory Management Unit - MMU) poate fi încorporată în unitatea de control din CPU sau poate fi externă unităţii de control. În particular, în cazul microprocesoarelor pot fi încorporate în microprocesoare sau sunt livrate ca un modul separat - modulul MMU).
Memoria principala *(MP)* conține informația în curs de prelucrare cum ar fi programul în execuție și date cu care opereaza acesta. Pentru a se realiza o viteza ridicata de transfer MP este conectata direct la magistralele sistemului. Adresa locatiei referite este pastrata în ***R****egistrul de* ***A****drese (****RA****),* iar datele citite/scrise din/în memorie se păstrează în ***R****egistrul de* ***D****ate (RD).* RA și RD vor mai fi întâlnite și la microprocesoare ca registre tampon. Prezenta lor este absolut necesara deoarece magistralele nu trebuie retinute pe toata durata transferului în/din memorie.
Circuitele de control ale memoriei (CCM) genereaza semnalele de selectie și comandă a modulelor de memorie.



4.3.3. Memoria cache

Memoria cache este o memorie situată din punct de vedere logic între CPU (Central Processing Unit - unitate centrală) și memoria principală (uzual DRAM - Dynamic Random Access Memory), mai mică, mai rapidă și mai scumpă (per byte) decât aceasta și gestionată – în general prin hardware – astfel încât să existe o cât mai mare probabilitate statistică de găsire a datei accesate de către CPU, în cache. Aşadar, cache-ul este adresat de către CPU în paralel cu memoria principală (MP): dacă data dorită a fi accesată se găsește în cache, accesul la MP se abortează, dacă nu, se accesează MP cu penalizările de timp impuse de latenţa

mai mare a acesteia, relativ ridicată în comparație cu frecvența de tact a CPU. Oricum, data accesată din MP se va introduce și în cache.
Memoriile cache sunt implementate în tehnologii de înaltă performanţă, având deci un timp de acces foarte redus dacă sunt integrate în microprocesor (cca. 1 – 5 ns la ora actuală). Ele reduc timpul mediu de acces al CPU la MP, ceea ce este foarte util.
Se defineşte un acces al CPU cu hit în cache, un acces care găseşte o copie în cache a datei accesate. Un acces cu miss în cache este unul care nu găseşte o copie în cache a datei accesate de către CPU și care, prin urmare, adresează MP cu toate penalizările de timp care derivă din accesarea acesteia.
Se defineşte ca parametru de performanţă al unei memorii cache rata de hit, ca fiind raportul statistic între numărul acceselor cu hit în cache respectiv numărul total al acceselor CPU la memorie. Măsurat pe benchmark-uri (programe de test) reprezentative, la ora actuală sunt frecvente rate de hit de peste 90 %.
În esență, eficiența memoriilor cache se bazează pe 2 principii de natură statistică și care caracterizează intrinsec noțiunea de program: principiile de localitate temporală și spațială. Conform principiului de localitate (vecinătate) temporală, există o mare probabilitate ca o dată (instrucțiune) accesată acum de către CPU să fie accesată din nou, în viitorul imediat. Conform principiului de localitate spaţială, există o mare probabilitate ca o dată situată în imediata vecinătate a unei date accesate curent de către CPU, să fie şi ea accesată în viitorul apropiat (pe baza acestui principiu statistic se aduce din MP în cache un întreg bloc și nu doar strict cuvântul dorit de către CPU). O buclă de program – structură esențială în orice program – exemplifică foarte clar aceste principii și justifică eficienţa conceptului de cache.
Din punct de vedere arhitectural, există 3 tipuri distincte de memorii cache în conformitate cu gradul de asociativitate: cu mapare directă, semiasociative și total asociative.

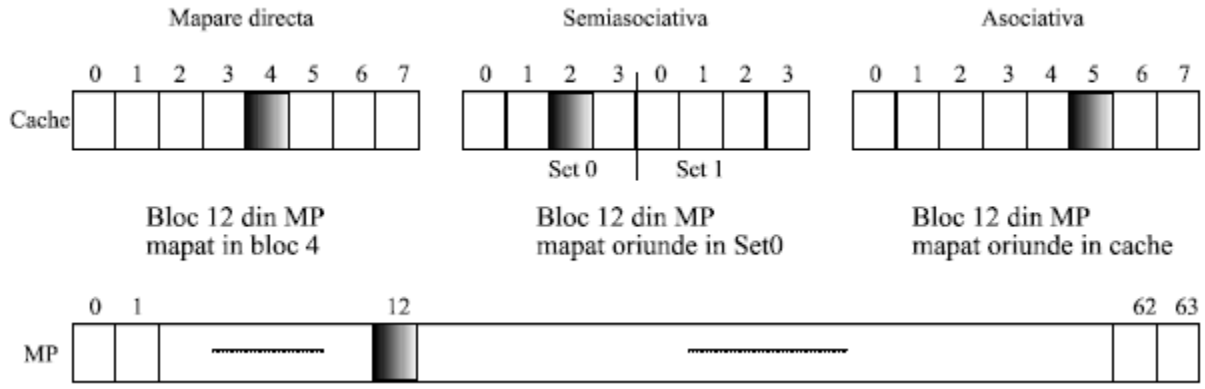


Figura 4.6. Scheme de mapare în cache

La cache-urile cu mapare directă, ideea principală constă în faptul că un bloc din MP poate fi găsit în cache (hit) într-un bloc unic determinat. În acest caz regula de mapare a unui bloc din MP în cache este:

(**Adresa bloc MP**) modulo (**Nr. blocuri din cache**)

Strictețea regulii de mapare conduce la o simplitate constructivă a acestor memorii dar și la fenomenul de interferență al blocurilor din MP în cache. Astfel, de exemplu, blocurile 12, 20, 28, 36, 42 etc. nu pot coexista în cache la un moment dat întrucât toate se mapează pe blocul 4 din cache. Prin urmare, o buclă de program care ar accesa alternativ blocurile 20 şi 28 din MP ar genera o rată de hit egală cu zero.
La cache-urile semiasociative există mai multe seturi, fiecare set având mai multe blocuri componente. Aici, regula de mapare precizează strict doar setul în care se poate afla blocul dorit, astfel:

(**Adresa bloc MP**) modulo (**Nr. seturi din cache**)

În principiu blocul dorit se poate mapa oriunde în setul respectiv. Mai precis, la un miss în cache, înainte de încărcarea noului bloc din MP, trebuie evacuat un anumit bloc din setul respectiv. În principiu, în mod uzual, există implementate două-trei tipuri de algoritmi de evacuare: pseudorandom (cvasialeator, uşor de implementat), FIFO (sau *round-robin*, se evacuează blocul cel mai vechi din cache. Contorul aferent se încarcă doar la încărcarea blocului în cache şi nu la fiecare hit per bloc, ca la algoritmul LRU) şi LRU ("*Least Recently Used*").
Dacă un set din cache-ul semiasociativ conține N blocuri atunci cacheul se mai numește "tip N-way set associative". Mai nou, se implementează algoritmi de evacuare predictivi, care anticipează pe baze euristice utilitatea de viitor a blocurilor memorate în cache, evacuându-l pe cela mai puţin valoros. Deşi aceşti algoritmi depăşesc în mod normal cadrul acestui curs de iniţiere în domeniul microprocesoarelor, în continuare se va prezenta totuşi unul, integrat în arhitectura numită *Selective Victim Cache*.
Este evident că într-un astfel de cache rata de interferenţă se reduce odată cu creşterea gradului de asociativitate (N "mare"). Aici, de exemplu, blocurile 12, 20, 28 şi 36 pot coexista în setul 0. Prin reducerea posibilelor interferenţe ale blocurilor, creşterea gradului de asociativitate determină îmbunătăţirea ratei de hit şi deci a performanţei globale. Pe de altă parte însă, asociativitatea impune căutarea după conținut (se caută deci într-un set dacă există memorat blocul respectiv) ceea ce conduce la complicaţii structurale și deci la creşterea timpului de acces la cache şi implicit la diminuarea performanţei globale. Optimizarea gradului de asociativitate, a capacităţii cache, a lungimii blocului din cache etc., nu se poate face decât prin laborioase simulări software, variind toţi aceşti parametrii în vederea minimizării ratei globale de procesare a instrucțiunilor [instr./cicli].
În fine, memoriile cache total asociative, implementează practic un singur set permiţând maparea blocului practic oriunde în cache. Ele nu se implementează deocamdată în siliciu datorită complexităţii deosebite și a timpului prohibit de căutare. Reduc însă (practic) total interferenţele blocurilor la aceeaşi locaţie cache și constituie o metrică superioară utilă în evaluarea ratei de hit pentru celelalte tipuri de cache-uri (prin comparație).
Pentru a reduce rata de miss a cache-urilor mapate direct (fără să se afecteze însă timpul de hit sau penalitatea în caz de miss), cercetătorul *Norman Jouppi* (DEC) a propus conceptul de "victim cache". Aceasta reprezintă o memorie mică complet asociativă, plasată între primul nivel de cache mapat direct și memoria principală. Blocurile înlocuite din cache-ul principal datorită unui miss sunt temporar memorate în victim cache. Dacă sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mică decât cea a memoriei principale. Deoarece victim cache-ul este complet asociativ, multe blocuri care ar genera conflict în cache-ul principal mapat direct, ar putea rezida în victim cache fără să dea naştere la conflicte. Decizia de a plasa un bloc în cache-ul principal sau în victim cache (în caz de miss) este făcută cu ajutorul unei informaţii de stare asociate blocurilor din cache. Biţii de stare conţin informaţii despre istoria blocului. Această idee a fost propusă de McFarling, care folosește informația de stare pentru a exclude blocurile sigure din cache-ul mapat direct, reducând înlocuirile ciclice implicate de acelaşi bloc. Această schemă, numită excludere dinamică, reduce miss-urile de conflict în multe cazuri. O predicţie greşită implică un acces în nivelul următor al ierarhiei de memorie contrabalansând eventuale câştiguri în performanţă. Schema este mai puţin eficace cu blocuri mari, de capacităţi tipice cache-urilor microprocesoarelor curente.
Pentru a reduce numărul de interschimbări dintre cache-ul principal şi victim cache, Stiliadis și Varma au introdus un nou concept numit *selective victim cache* (SVC).

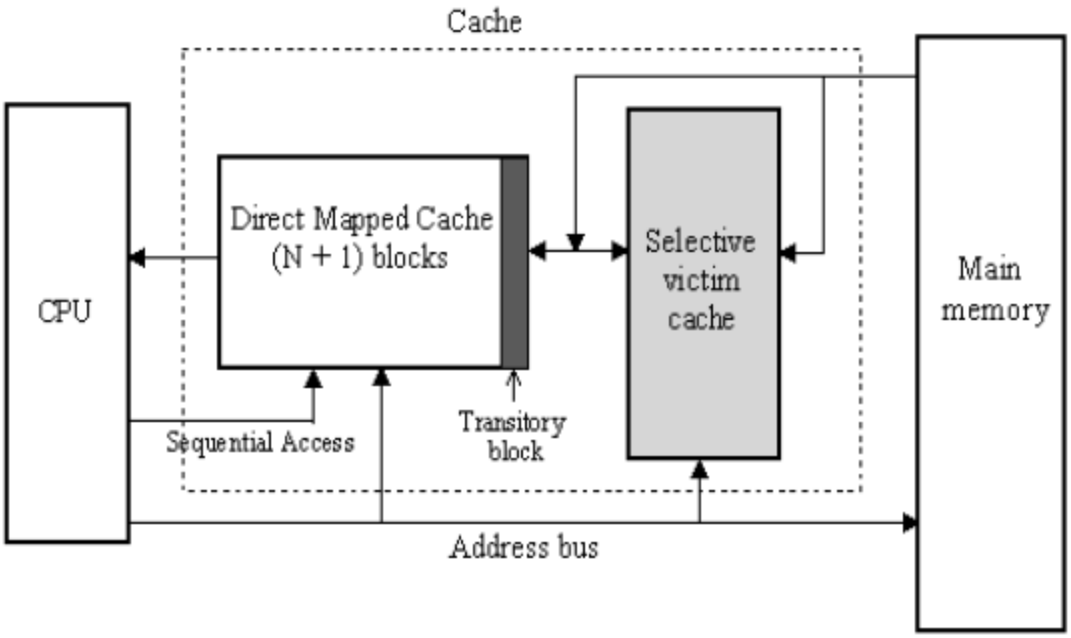


Figura 4.7. Ierarhia de memorie pentru schema SVC

Cu SVC, blocurile aduse din memoria principală sunt plasate selectiv fie în cache-ul principal cu mapare directă fie în selective victim cache, folosind un algoritm de predicţie euristic bazat pe istoria folosirii sale. Blocurile care sunt mai puţin probabil să fie accesate în viitor sunt plasate în SVC și nu în cache-ul principal. Predicţia este de asemenea folosită în cazul unui miss în cache-ul principal pentru a determină dacă este necesară o schimbare a blocurilor conflictuale. Algoritmul obiectiv este de a plasa blocurile, care sunt mai probabil a fi referite din nou, în cache-ul principal și altele în victim cache.

Metrici de performanţă

Metricile de performanţă folosite sunt rata de miss la nivelul L1 de cache și timpul mediu de acces la ierarhia de memorie. În cazurile cu victim cache simplu și cel cu victim cache selectiv, folosim de asemenea și numărul de interschimbări între cache-ul principal și victim cache ca metrică de comparație. Rata de miss la nivelul L1 de cache se bazează pe numărul de referinţe propagate în nivelul următor al ierarhiei de memorie. În caz de miss în cache-ul principal acestea sunt servite de către victim cache, fiind plătită o penalitate pentru accesul în victim cache precum și pentru interschimbările de blocuri rezultate între cache-ul principal şi victim cache. Timpul mediu de acces la ierarhia de memorie ia în calcul şi aceste penalizări și de aceea este un bun indicator al performanţei memoriei sistemului, desigur mai complet decât rata de miss în nivelul L1 de cache.

Timpul mediu de acces pentru un cache mapat direct se calculează astfel:

$$T_d = clk \times \frac{R + p \times M_d}{R} = clk\left(1 + p \cdot \frac{M_d}{R}\right)$$

Unde *R* este numărul total de referinţe generate de programele de tip trace, *Md* reprezintă numărul total de accese cu miss în cache. Pentru fiecare miss, sunt necesari *p* cicli suplimentari. Presupunem că cei *p* cicli includ toate “cheltuielile” CPU-ului. În cazul victim cache-ului simplu, o penalitate de *p* cicli este produsă la fiecare miss în primul nivel al ierarhiei de memorie.

Folosirea victim cache-ului selectiv determină îmbunătăţiri ale ratei de miss precum şi ale timpului mediu de acces la memorie, atât pentru cacheuri mici cât și pentru cele mari (4Kocteți - 128 Kocteți). Simulări făcute pe trace-uri de instrucțiuni a 10 benchmark-uri SPEC’92 arată o îmbunătăţire de aproximativ 21% a

ratei de miss, superioară folosirii unui victim cache simplu de 16 Kocteți cu blocuri de dimensiuni de 32 octeți; numărul blocurilor interschimbate între cache-ul principal şi victim cache s-a redus cu aproximativ 70%.

4.3.4. Memoria virtuală (secundară)

Este descris in capitolul 7.1.

4.3.5. Memoria video

Din punct de vedere constructiv, memoria video este de tip RAM dinamic (DRAM), în mai ulte variante. Memoria RAM dinamica are avantajul densitatii mari de integrare și deci al raportului capacitate/pret mare,dezavantajul fiind, în schimb, necesitatea reamprospatarii periodice a informatiei conținute [DRAM refresh]. Astfel încât, în intervalul de reâmprospătare [duty cycle], nu se pot face accesele normale de citire/scriere la memorie.
Un alt dezavantaj il reprezintă faptul ca la un moment dat se poate face doar un singur acces la memorie (citire sau scriere). La nivelul placii grafice au insa loc două tipuri de transfer de date: transferul informatiei de la unitatea centrala a calculatorului, la memoria video; și transferul datelor din memoria video spre registrii de deplasare, pentru afisarea imaginii pe ecran. Deci capacitatea totala de acces la memorie va trebui impartita la 2.
La adresabilitati mari de pixel și la adancimi mari de culoare, spre monitor vor trebui transferate cantitati enorme de date pentru ca afisarea să nu se altereze (rata de reâmprospatare a imaginii [monitor refresh rate] va fi mare), ceea ce cu unele tipuri de memorie video nu se reuseste.
In incercarea de a elimina dezavantajele enumerate mai sus (și deci de a creste capacitatea totala de acces la memoria video), s-au conceput diverse alte variante de memorie RAM, printre care se remarca prin performante, varianta VRAM [Video RAM].
Memoria VRAM este o memorie DRAM de tip dual-port, adica suporta două accese simultane (citire sau scriere), dubland astfel capacitatea totala de acces la ea. Necesitatea reamprospatarii informatiei din memorie se păstrează, fiind tot RAM dinamica. In schimb conține un registru de deplasare propriu, cu incarcare paralela (echivalentul unui ciclu de citire), de capacitate foarte mare (un rand intreg din matricea memoriei). Acest registru poate fi sincronizat separat de accesele normale la memorie, și e utilizat pentru reamprospatarea video. Astfel, dacă la memoria DRAM clasică, reamprospatarea imaginii afisate ocupa un procent de 50% din accesele la memoria video, în cazul memoriei VRAM, operația ocupă 0.5%.
Capacitatea de memorie a celor mai moderne plăci video variază de la 128 MB la 16 GB. Din moment ce memoria video trebuie să fie accesată de către GPU (**G**raphics **P**rocessing **U**nit) și circuitele de afișaj, se folosește adesea memorie specială mulți-port sau de mare viteză, cum ar fi VRAM, WRAM, SGRAM etc. În jurul anului 2003, memoria video era de obicei bazată pe tehnologia DDR. În timpul și după acest an, producătorii s-au îndreptat spre DDR2, GDDR3, GDDR4, și chiar GDDR5 utilizată în special de către ATI Radeon HD 4870. Rata efectivă a ceasului memorie în plăcile grafice moderne este, în general, cuprinsă între 400 MHz și 3.8 GHz.
Memoria video poate fi utilizată pentru stocarea altor date, precum și a imaginii de pe ecran, cum ar fi Z-bufferul, care gestionează coordonatele de adâncime în grafica 3D, texturi, buffer-e vertex, programe compilate de shader.
Memoria grafică este în general clasificată în două tipuri majore: dedicată și partajată. Memorie grafică dedicată, după cum sugerează și numele, este memoria care este disponibilă pentru utilizarea exclusivă de către subsistemul grafic. Aplicațiile non-grafice și alte subsisteme din sistemul de operare nu pot accesa acest tip de memorie. Un exemplu de memorie grafică dedicată este de memoria care este prezentă fizic pe adaptoarele grafice "discrete". Acest lucru a fost denumit în mod obișnuit ca "la bord" sau "de memorie video locală"-care este, aproape de unitatea de procesare grafică (GPU). Memoria dedicată, cu toate acestea, nu se limitează la memorie on-board. O porțiune de memorie de sistem poate fi, de asemenea, dedicată subsistemelor grafice. Această porțiune de memorie de sistem nu este niciodată disponibilă pentru alte

subsisteme sau aplicații și sunt deținute exclusiv de către subsistemele grafice. Memoria partajată de sistem este o porțiune din memoria sistemului, care pot fi utilizată de către subsistemele grafice atunci când este necesar. Pentru adaptoarele grafice discrete, acest tip de memorie este adesea menționată ca "memoria video non-locală"-care este, departe de cea a GPU-ului. Memorie partajată este disponibilă pentru alte subsisteme sau aplicații non-grafice atunci când nu este utilizată de către subsistemele grafice. Astfel, acesta nu este garantat să fie disponibilă pentru grafică, deoarece ar putea fi deja în uz.
Diferențele dintre un adaptor grafic "discret" și unul "integrat" pot fi evidențiate în contextul de memorie grafică dedicată versus partajată. Adaptoarele grafice discrete sunt de obicei conectate la sistem prin port grafic accelerat (AGP), PCI, sau PCI Express bus. Cele mai multe adaptoare discrete au o anumită cantitate de memorie grafică dedicată cu un bus foarte larg și rapid de memorie locală de a acces, oferind performante mult mai bune decât memoria sistemului. Adaptoarele grafice discrete pot accesa, de asemenea, și utilizeaza memorie de sistem prin intermediul AGP sau PCI-Express bus de memorie video non-local discutat mai devreme. Deoarece memoria de sistem este accesat în magistrala sistemului, accesul este mult mai lent decât accesul la memoria locală. Adaptoarele grafice discrete împart, în general, o porțiune din memoria sistemului cu CPU. De obicei, aceste adaptoare nu cer pentru utilizarea dedicată de memorie de sistem pentru grafica, lăsând astfel mai multe resurse disponibile pentru restul sistemului.

4.3.5.1. Memorii secundare

4.4. Dispozitive de intrare/ieşire (organizare, conectare, interfețe).

Sistemul de intrare/ieşire permite introducerea/extragerea informaţiilor şi este constituit din:

- **Dispozitive de memorare externă (ME)**, de exemplu *SSD, hard-disc, stick memorie, floppy-disc, compact-disc,* etc.
- **Dispozitive de intrare (DI)**, de exemplu tastatură, *mouse,* etc.
- **Dispozitive de ieşire (DE)**, de exemplu monitor, imprimantă, etc.

Comunicarea între aceste componente se realizează prin intermediul unor magistrale. O magistrală reprezintă un grup de linii de conexiune ce permit transmiterea de semnale. Există două tipuri de magistrale într-un calculator:

- **Magistrale de Adrese (MA)** transmit numai adrese de memorie şi conectează **UCP** cu memoria **RAM**.
- **Magistrale de Date (MD)** transmit date și instrucțiuni și conectează **UCP**, memoria **RAM** şi celelalte componente ale sistemului.

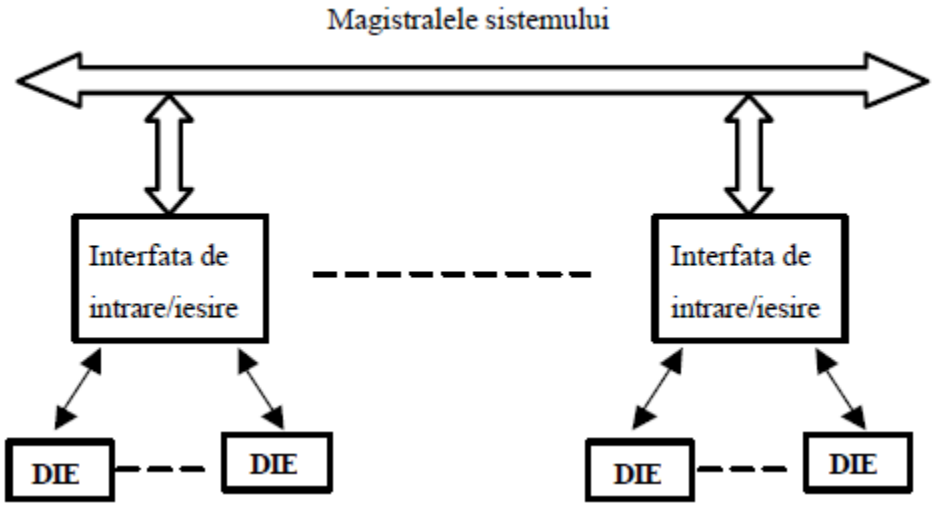
Dispozitivele de intrare/ieșire (I/O devices) sunt tastatura, mouse, monitor, imprimanta, placa de retea, etc. cu următoarele elementele principale:

- controller-e de intrare/ieșire
- comunicatia sistemului de operare cu dispozitivele de intrare/ieșire = programare /O(intreruperi/drivere)
- interfetele puse la dispozitia utilizatorului

Un dispozitiv I/O este orice dispozitiv care permite introducerea sau extragerea de informatie din calculator:

- dispozitive de intrare: tastatura, mouse, joystick;
- dispozitive de ieșire: monitor, imprimanta, boxe;
- dispozitive de intrare & ieșire: placa de retea, modem, harddisk, floppy disk, USB stick (dispozitive de stocare)

Dispozitivele de intrare/ieșire (DIE) asigura introducerea și extragerea informatiei în/din calculator. Numite și *Dispozitive Periferice (DP)* acestea pot îndeplini mai multe categorii de funcții cum ar fi comunicarea dintre calculator și utilizatorul uman, comunicarea între calculatoare, legatura cu procese reale etc.
DIE se conecteaza la unitatea centrala prin intermediul *interfetelor de intrare/ieșire* , a caror complexitate variaza foarte mult de la simple *registre* pâna la *controlere inteligente realizate cu microprocesor.*



Din punctul de vedere al sistemului de operare DIE sunt vazute ca *fisiere* iar interfetele în calitate de *canale de comunicație.*

Dispozitive de ieșire.

Din punctul de vedere al suportului folosit pentru vizualizarea informatiei extrase din calculator există două tipuri de dispozitive de ieșire:

- *dispozitive hardcopy* – exemplu: imprimante cu ace, laser, cu jet de cerneala, termica; plotterele cu penita sau electrostatic - creeaza imaginea fixa, stabila pe un suport fizic cum ar fi hârtia sau folia de plastic;
- *dispozitive de afisare* – exemplu: dispozitive cu tub catodic monocrom sau color (CRT), tuburile cu memorare directa (DVST), dispozitivele de afisare cu cristale lichide (LCD), ecrane cu plasma, dispozitive de afisare electroluminiscente, etc. - sunt caracterizate de o anumita frecventa de reâmprospatare a imaginii pe un ecran, putându-se obține și efecte dinamice.

Dispozitive de intrare.

Servesc la introducerea datelor sub diverse forme în sistem. Exista sase dispozitive logice de introducere a datelor grafice și anume: *locator* (locator), *flux* (stroke), *optiune* (choise), *culegere* (pick), *text* (text), *valoare* (valuator). Pentru acestea au fost realizate dispozitive fizice cum ar fi: *tableta grafica, mouse, trackball, joystick, ecranul senzitiv, creionul optic, butoane, chei, tastatura, valoare.*

Execuţia unui proces de intrare/ieşire

Viteza de lucru a perifericelor diferă în funcție de performanţele lor tehnologice (foarte mică la dispozitivele exclusiv mecanice) dar este mult mai mică decât a unităţii centrale, care funcţionează pe principii exclusiv electronice. Printr-o analogie umană, noi gândim adesea mult mai repede decât putem scrie. După ce unitatea de comandă cere perifericelor execuția unei operații de intrare-ieșire, ea asteaptă un timp îndelungat (față de timpii ei de lucru) pentru terminarea operației. Astfel, apare un gol în activitatea unității de comandă. Problema a fost rezolvată prin introducerea unui bloc auxiliar de comandă care să preia controlul asupra operațiilor de intrare-ieșire efectuate cu ajutorul perifericelor după inițierea lor de către

unitatea de comandă. Prin intermediul procesorului auxiliar care controlează operațiile cu perifericele se permite cuplarea, indirectă, la unitatea centrală, a mai multor dispozitive periferice. Această unitate funcţională a calculatoarelor a apărut odată cu generatia a două de calculatoare sub numele de de *canal de intrare-ieșire*. Rolul său a fost preluat la sistemele de calcul medii mari de unitatea de schimburi multiple (USM), la minicalculatoare – de dispozitivul de control al magistralei de comunicatii iar la microcalculatoare – de o extensie a magistralei. În timp ce procesorul specializat în operații de intrare-ieșire controlează schimbul de date între memorie şi periferice, blocul de comandă poate superviza execuția altor operații, dintr-un alt program. Modul de lucru cu o unitate centrală şi mai multe programe rezidente în memorie spre execuție se numește *multiprogramare* sau *multitasking*. Programele aflate în memoria internă a unui calculator care funcţionează în regim de multitasking se găsesc în diverse stări: pot astepta terminarea unei operații de intrare-ieșire, pot aştepta să fie lansate în execuție iar un singur program este prelucrat la un moment dat de unitatea centrală; politica de servire a acestor programe este stabilită de algoritmii implementaţi în sistemul de operare.
După terminarea unei operații de intrare-ieșire, unitatea de comandă este anunţată, printr-un semnal numit *întrerupere*, că citirea sau scrierea s-a încheiat, astfel încât poate continua execuția programului respectiv. Noua unitate componentă a sistemului de calcul poate fi încorporată în schema generală a unui calculator.

Interfețe

Interfeţele și dispozitivele de intrare-ieşire ale unui calculator personal sunt adaptate pentru modul de lucru cu un singur utilizator. De exemplu: există o singură intrare de tastatură și se foloseşte o singură interfaţă de afişare. Informaţiile afişate se păstrează într-o zonă de memorie direct accesibilă procesorului. Astfel pot fi implementate aplicaţii care necesită o viteză mare de schimbare a informaţiilor afişate. În cazul sistemelor mulți-utilizator interfaţa utilizator este asigurată de mai multe dispozitive inteligente de tip display care încorporează o tastatură și un dispozitiv de afişare; legatura cu calculatorul gazdă se realizează prin canale seriale. Viteza de schimbare a informaţiilor afişate este limitată de viteza relativ mică a canalului serial.
Configuraţia tipică de dispozitive și interfeţe de intrare-ieşire pentru un calculator personal este următoarea: tastatură, monitor de vizualizare, unitate de disc flexibil , unitate disc rigid, unitate de disc optic, dispozitiv de indicare (mouse), imprimantă, interfaţă de reţea, sistem audio (interfaţă soundblaste și boxe). Tastatura este compusă dintr-o matrice de contacte (taste) şi un microsistem bazat pe un microcontrolor. Microsistemul are rolul de a detecta eventualele contacte închise (taste apăsate) și de a transmite această informaţie sistemului de calcul. La calculatorul IBM-PC comunicaţia dintre tastatură și calculator se realizează printr-o legatură serială sincronă. Alimentarea tastaturii se face de la calculator prin cablul de comunicație (prin semnale separate de alimentare).
Unitatea de disc rigid (hard-disk) are rolul de a păstra programe și date care sunt folosite în mod uzual într-un anumit sistem de calcul. Tot pe hard-disk se păstrează de obicei și sistemul de operare al calculatorului, care se încarcă la inițializarea sistemului. O parte a spaţiului de pe disc se poate utiliza pentru extinderea memoriei interne prin tehnica denumită memorie virtuală.
Unitatea de disc optic permite citirea informaţiilor digitale înregistrate pe un suport optic. Ca şi funcţionalitate această unitate se situează între unitatea hard-disk şi cea de disc flexibil: informaţiile înregistrate pe discul optic pot fi transportate între calculatoare. Scrierea se face secvenţial pe tot discul sau pe o porţiune a sa (înscriere multisesiune). Există diferite standarde pentru viteza de transmisie a datelor citite: X4, X8, X32, X40,X48
Monitorul video are rolul de a afişa rezultatele execuţiei unui program precum şi informaţiile de operare. Partea “inteligentă” a ieşirii video este interfaţa video. Există mai multe standarde pentru implementarea interfeţei video: MGA, CGA, EGA, VGA și SVGA. Diferenţele constau în: rezoluţie pe orizontală (320-1024 puncte), rezoluţie pe verticală 200800 puncte), paleta de culori (2-256 culori) şi facilităţi de citire-scriere a informaţiilor grafice. Primele 3 variante de interfaţă generează semnale digitale, iar ultimele semnale analogice. De obicei tipul monitorului trebuie să fie în concordanţă cu tipul interfeţei video. Imprimanta este un dispozitiv de ieșire ce permite tipărirea pe hârtie a rezultatelor unei prelucrări pe

calculator. De obicei legătura cu calculatorul se realizează pe canalul paralel. Imprimanta poate să lucreze în mod alfanumeric (caz în care acceptă coduri ASCII și secvenţe specifice de comandă – ESC), sau în mod grafic (caz în care informaţiile transmise descriu prin puncte o imagine grafică). Există mai multe standarde pentru limbajul de comunicație între calculator și imprimantă (ex: EPSON, IBM, Hewlet-Packard, etc.). Acestea diferă mai ales prin modul grafic. O imprimantă poate să emuleze (să înţeleagă) mai multe tipuri de limbaje.
Interfaţa de reţea permite cuplarea calculatorului într-o reţea locala (LAN). Cel mai răspândit standard de rețea LAN pentru calculatoare personale este Ethernet 10Base5. Conectarea în rețea se poate face cu cablu coaxial, cu cablu bifilar înfăşurat (UTP-Unshielded Twisted Paire) sau prin fibră optică (mai rar). Pentru comunicație se pot folosi mai multe pachete de protocoale: Novel-Netware, Windows-Netware, TCP/IP și altele.
Mouse-ul este un dispozitiv de indicare, util mai ales pentru sistemele de operare şi programele aplicative care au o interfaţă grafică utilizator bazată pe meniuri. Mişcările mouse-ului sunt transformate în mişcări ale unui cursor pe ecran cu ajutorul unui program de interfaţă (driver). Prin intermediul celor două sau trei butoane se pot selecta funcții ale programului care rulează. Există mai multe standarde pentru limbajul de comunicație între mouse și calculator. Legatura se realizează printr-un canal serial.
Interfaţa audio permite redarea înregistrărilor audio, mixarea diferitelor surse de sunet (CD, fisier, microfon), înregistrarea semnalelor audio de intrare, generarea de efecte speciale, filtrarea semnalelor de intrare, etc. Interfaţa este alcatuită dintr-o placă de sunet (soud-blaster) și boxe. O ieșire a unităţii de disc optic este conectată la placa de sunet pentru a permite redarea discurilor de muzică.

5. Nivelul fizic

5.1. Microprocesorul, întreruperile, magistralele de comunicaţie (magistralele sincrone şi asincrone, arbitrajul magistralei)

O magistralã (bus) defineşte setul de reguli după care informația circulă în interiorul unui sistem de calcul, între mai multe sisteme de calcul sau între sisteme de calcul și echipamente specializate. Un standard de magistralã conține mai multe tipuri de specificaţii:
• mecanice - conectorii, cablurile, topologia de cuplare;
• electrice - nivelele de tensiunie, curenţii impedanţe;
• funcţionale - descrierea semnalelor.
Unele calculatoare au impus un standard de magistralã prin popularitatea lor (PC AT au impus magistrala ISA, apoi PCI). Uneori standardele sunt create de grupuri de producători care au interese comune. Un astfel de standard este SCSI (Small Computer Interface System). Unele standarde sunt adoptate de organisme ca IEEE. Magistralele realizează legătura electrică între procesor, memorie, dispozitive I/O
și definesc cel mai jos nivel pentru un protocol de comunicație. Magistrala este o cale de comunica
ţie folosită în comun de mai multe blocuri funcţionale şi este realizată fizic de un set de fire conductoare.
O magistralã trebuie să fie versatilă și să aibă un preţscăzut. Datorită versatilităţii, odată definită o schemă de conexiuni, este foarte uşor să se adauge noi dispozitive în sistem sau acestea să poată fi uşor mutate dintr-un calculator în altul, dacă cele două calculatoare sunt construite cu acelaşi tip de magistralã.
Pe de altă parte, folosirea unei magistrale poate fi dezavantajoasă datoritã limitărilor de vitezã de transfer a datelor (throughput) pe care aceasta le poate crea. Viteza la care poate lucra o magistralã este impusă în primul rând de factori fizici ca: lungimea traseelor şi numărul dispozitivelor ataşate (acestea din urmă aduc constrângeri atât ca număr efectiv cât şi ca diversitate din punct de vedere al ratelor de transfer
și a timpilor de întârziere).
O magistralã este organizată în grupe de semnale care au aceeaşi funcţionalitate:

- un set de semnale de control;
- un set de semnale de date.

Semnalele de control sunt folosite pentru a indica cereri, acceptări sau pentru a indica tipul informa
ţiei de pe liniile de date. Setul semnalelor de date este folosit pentru a transporta informația între blocurile funcţionale ale calculatorului de la sursã la destinaţie (de la iniţiator la ţintã). Informaţia poate fi date, comenzi sau adrese. Adesea magistralele de date sunt despãrţite în douã seturi pentru a separa informația ce reprezintã adrese. Astfel există și o magistralã de adrese pentru ca datele și adresele să poată fi vehiculate într-o singurã tranzacţie de la sursã la destinaţ
ie. Dupăfuncţionalitatea semnalelor magistralele pot fi magistrale de control, magistrale de date şi magistrale de adrese. O clasificare a magistralelor poate fi fãcutã dupã blocurile care se conectează la aceasta. Conform acestui criteriu magistralele pot fi:

- magistrale procesor- memorie;
- magistrale I/O;
- magistrale de fundal.

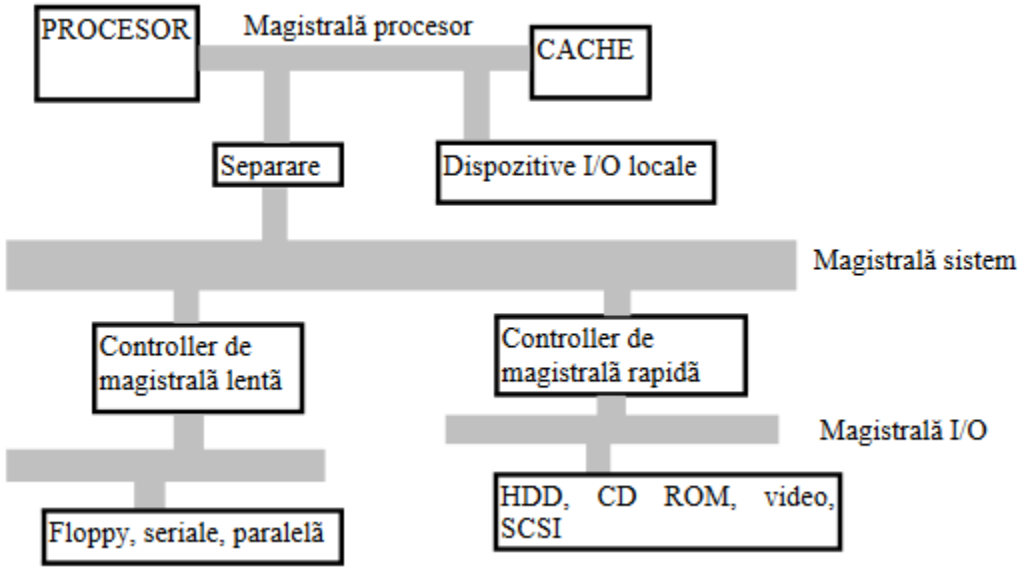
Magistrala procesor memorie este scurtã şi de mare vitezã. Dupã cum îi spune numele, este conceputã pentru a lega procesoarele de memorie.
Magistralele I/O sunt în primul rând mai lungi și suportă o gamă largă de rate de transfer ca o necesitate a conectării unei game largi de dispozitive I/O la sistem. De obicei magistralele I/O nu sunt conectate direct la procesor, ci folosesc fie o magistralã procesor memorie, fie o magistralã de fundal.
Magistralele de fundal sunt proiectate astfel încât atât procesorul, memoria cât şi dispozitivele I/O să folosească aceeaşi cale de comunicație. La acest tip de magistralã se face un compromis pentru a satisface atât necesitãţile comunicaţiei între procesor și memorie cât şi ale comunicaţiei dintre dispozitivele I/O, sacrificând desigur performanţa sistemului.
Magistralele I/O şi magistralele de fundal sunt definite prin specificaţii, deseori publice, iar magistralele procesor memorie sunt specifice (proprietatea producãtorului).
O altă clasificare a magistralelor se poate face după metoda de comunicație folosită:

- magistrale sincrone;
- magistrale asincrone.

Magistrale sincrone dispun de o linie de tact în liniile de control, condusã de un oscilator cu cuarţ, între 5-133MHz. Toate transferurile au loc după un protocol fix care este raportat la semnalul de tact, într-un numãr întreg de cicluri, numite cicluri de bus. Pentru cã protocolul este predeterminat şi logica implicatã este simplă, magistralele sincrone pot fi foarte rapide. Avantajul magistralelor sincrone este simplitatea deoarece nu există un dialog (protocoalele pot fi implementate prin automate finite simple). Dezavantajul este cã dacă un transfer este mai scurt el trebuie totuşi să dureze un numãr întreg de cicluri de bus, ca urmare îmbunătãţirile tehnologice ale plãcilor I/O nu duc la mărirea vitezei. Într-un cuvânt, dacă avem plãci lente și rapide pe o magistralã sincronă, transferurile au loc la viteza celei mai lente. Un alt dezavantaj este cã, din cauza deformării semnalului de ceas, liniile magistralei nu pot fi foarte lungi și nici forte încărcate.
La magistrale asincrone transferurile pot dura oricât, motiv pentru care acestea se poate adapta uşor la o mare varietate de dispozitive conectate. Coordonarea transferurilor de la sursã la destinaţie se face pe baza unui protocol de dialog conversaţional (handshaking). Un astfel de protocol constă într-o serie de paşi parcurşi de sursã şi destinaţie astfel încât trecerea la pasul următor se face numai cu acordul ambelor pãrţi. Pentru implementarea unui astfel de protocol sunt necesare linii de control suplimentare (DATA REQUEST, DATA READYetc.).

Aşa cum este firesc, cel puţin două blocuri sunt conectate la o magistralã. Pentru a evita haosul pe magistralã, controlul acesteia este fãcut de un MASTER de magistralã. Un MASTER trebuie să poată să iniţieze cereri pentru acces la magistralã şi să poată să controleze toate tranzacţiile. Un procesor este capabil să fie MASTER pe când memoria poate fi numai SLAVE. Cel mai simplu sistem este un sistem cu un singur MASTER, acesta fiind procesorul. Implicarea procesorului în toate tranzacţiile este ineficientă, de aceea există alternativa de magistralã cu mai mulți MASTER. Într-o astfel de magistralã

fiecare MASTER este capabil să inițieze un transfer. În aceste condiţii trebuie să existe un mecanism de arbitrare a accesului la magistralã ce decide care MASTER va prelua controlul. Arbitrul primește cererea de administrare a magistralei (BUS REQUEST) pe care o onorează sau nu (GRANT) în urma unei analize. O magistralã cu mai mulți MASTER conține linii speciale pentru arbitrare. Există mai multe metode de arbitrare. Toate metodele respectă principiul priorităţii și al corectitudinii (fairness).
Pentru cuplarea diferitelor subsisteme la magistrale trebuie ţinut cont de diafonie, de reflexii, de impedanţele liniilor etc. Pentru legarea semnalelor la liniile de transmisie se folosesc circuite driver, aşa cum sunt de exemplu 74HCxxx sau 74AHCxxx pentru CMOS şi 74ABTxxx pentru TTL.
Structura tipicã a unei unităţi centrale aratã cã magistrala este ierarhizatã, în funcție de vitezã, pe două nivele; unul de vitezã mare (PCI) şi unul de vitezã micã (ISA).



La magistrala de vitezã mare se conectează echipamente periferice rapide (hard disc, rețea Ethernet, video, SCSI etc.) iar la cea de vitezã micã se conectează echipamente periferice lente (unitatea de disc flexibil, canalele seriale, canalul paralel, etc.).

5.1.2. Moduri de lucru între microprocesor şi interfetele I/O

Legătura între procesor și EP (Echipamente Periferice) se realizează prin canale I/O (de intrare/ie şire) prin intermediul magistralei. Evoluţia în timp a canalelor I/O este în acelaşi timp o evoluţie a creşterii complexităţii și performanţelor. Pot fi enumerate următoarele etape:

1. CPU controlează direct EP;
2. Este adăugat un modul I/O (o interfaţă serială sau paralelă, programabile). CPU comandă EP prin transfer programat (direct sau prin interogare);
3. Aceeaşi configuraţie ca la 2, dar transferul are loc prin întreruperi;
4. Modulul I/O are acces direct la memorie prin DMA. Modulul poate muta informaţia direct în memorie, accesul CPU fiind necesar doar la începutul și sfârşitul transferului;
5. Modulul I/O foloseşte un microcalculator sau un microcontroller, cu instrucțiuni proprii. CPU programează procesorul I/O pentru un transfer, pe care acesta îl execută folosind instrucțiunile proprii.Când transferul se termină, procesorul I/O întrerupe CPU pentru a comunica rezultatele transferului;

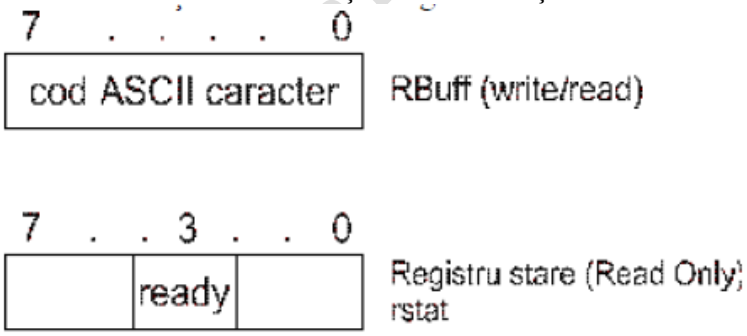
6. Microcontrollerul are memorie locală. El poate controla astfel mai multe EP cu o intervenţie minimă din partea CPU. Memoria locală poate fi folosită și ca buffer de date, realizând astfel o rată de transfer mare.

Evoluţia microcontrollerelor integrate a fost paralelă cu cea a procesoarelor. Dacă la procesoare s-a urmărit o creştere a vitezei de prelucrare prin creşterea tactului, creşterea mărimii memoriei CACHE (uzual 256K integrat), lărgimea busului de date și adrese, la microcontrollere s-a urmărit integrarea de cât mai multe subsisteme utile (memorie EPROM, RAM, convertoare AD și DA). Dacă preţul unui procesor nu a scăzut sub câteva zeci de dolari (PII/450MHz), preţul unui microcontroller poate ajunge la câţiva dolari, ceea ce încurajează eforturile proiectanţilor de a realiza module I/O inteligente cu microcontroller, sau alte aplicaţii cu microcontroller.
Conexiunea între un modul I/O şi unitatea centrală poate fi:

- Punct-la-punct; linii dedicate pentru transferul de date (de exemplu la un PC AT tastatura, RS232 etc.);
- Multipunct; la liniile de interfaţă se pot lega mai multe EP (module I/O; ca la SCSI)

Dacă o legătură punct cu punct se numeşte indiscutabil interfaţă, o legătură multipunct se poate numi atât interfaţă cât și magistrală. Interfaţa serială multipunct USB (Universal Serial Bus) conține în numele ei denumirea de magistrală.

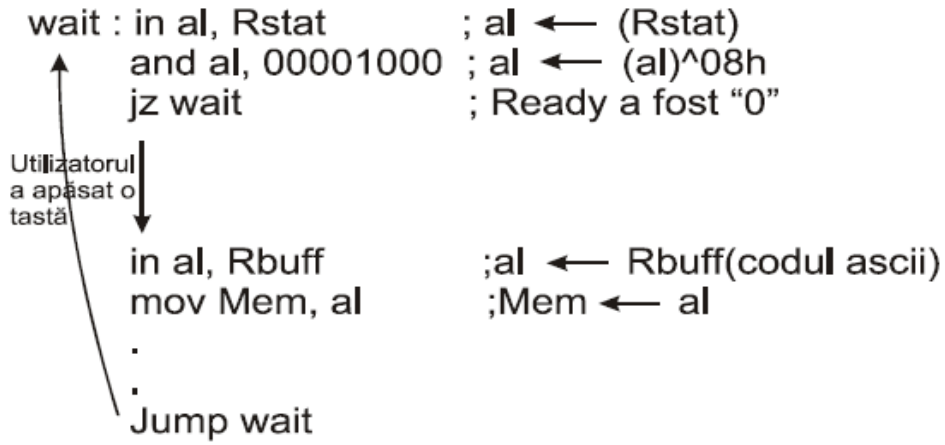
Modul de lucru prin interogare (“polling”)
Se bazează pe testarea de către microprocesor a unui bit de stare asociat dispozitivului periferic. Microprocesorul nu va iniţializa transferul cu perifericul decât în momentul în care bitul de stare semnifică faptul că perifericul este pregătit pentru transfer (nu lucrează la un transfer iniţiat anterior). Să considerăm de exemplu interfaţa cu o tastatură. Această interfaţă trebuie să conţină minimum 2 registre.



RBuff va memora un octet care reprezintă codul ASCII (Unicode) al tastei apăsate de către utilizator.

Exemple:
“A” = 41h ⇔0100.0001
“a” = 61h ⇔0110.0001
“0” = 30h
“ ” = 20h

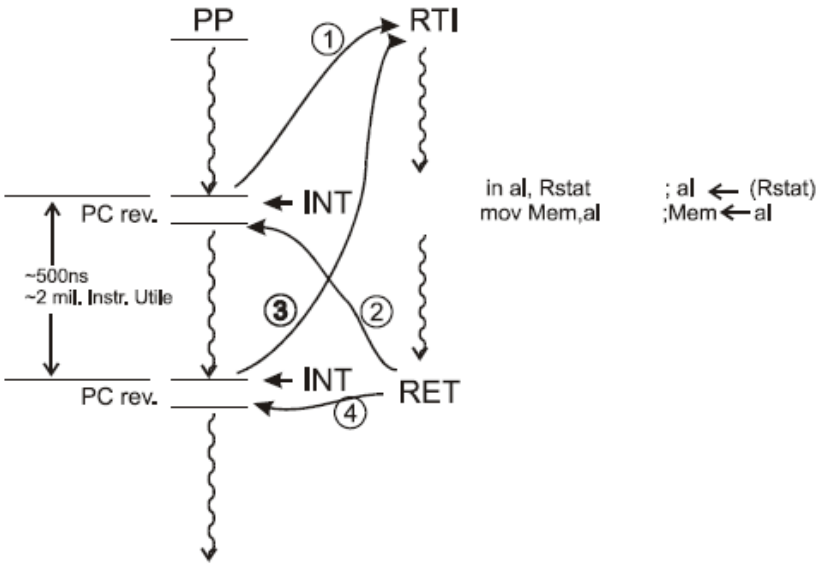
Bitul *Ready* din registrul de stare este un bit de tip *Read Only* cu următoarea semnificaţie: dacă registrul RBuff se încarcă cu un octet (utilizatorul a apăsat o tastă) atunci *Ready* se pune automat pe “1” arătând microprocesorului că poate să preia codul din RBuff. Bitul *Ready* se va reseta automat odată cu preluarea codului din Rbuff de către microprocesor.
Un program - absolut principial - de gestiune a tastaturii s-ar scrie ca mai jos:



Dezavantajul acestei metode constă în faptul că microprocesorul aşteaptă un timp T, neacceptabil de mare la nivelul vitezei sale, pentru a inspecta dacă perifericul este sau nu este pregătit. Considerând că utilizatorul apasă o tastă la interval de 500 ms și că o instrucţiune a microprocesorului durează cca. 250 ns (vezi justificarea anterioară) => că "pierde" 500 ms / 250 ns = 2 milioane instrucțiuni în bucla de aşteptare în loc să execute instrucțiuni utile. Acest dezavantaj este eliminat de metoda următoare de comunicare procesor-interfaţă.

Modul de lucru prin întreruperi hardware

Se bazează pe generarea unui semnal de întrerupere INT de la interfaţă (port) spre microprocesor ori de câte ori acesta doreşte un serviciu de la microprocesor. Ca urmare a recepţionării semnalului INT microprocesorul va abandona programul principal (PP) urmând să intre intr-o aşa numită rutină **tratare a întreruperii** în care va satisface cererea interfeţei. La finele rutinei de tratare a întreruperii printr-o instrucțiune de tip RETURN, microprocesorul va reveni în PP, în general dar nu întotdeauna, pe instrucțiunea imediat următoare ultimei instrucţiuni din PP executate. In cazul exemplului cu tastatura anterior considerat, interfaţa va genera întreruperea INT ori de câte ori utilizatorul a apăsat o tastă, adică registrul RBuff este "plin", deci conţine codul (ASCII, Unicode etc.) al caracterului tastat.



Aşadar RTI după ce execută serviciul necesar perifericului (în cazul acesta preluare şi depozitare caracter în memorie) revine în PP, unde până când perifericul cere un nou serviciu (de ex. se apasă din nou o tastă),

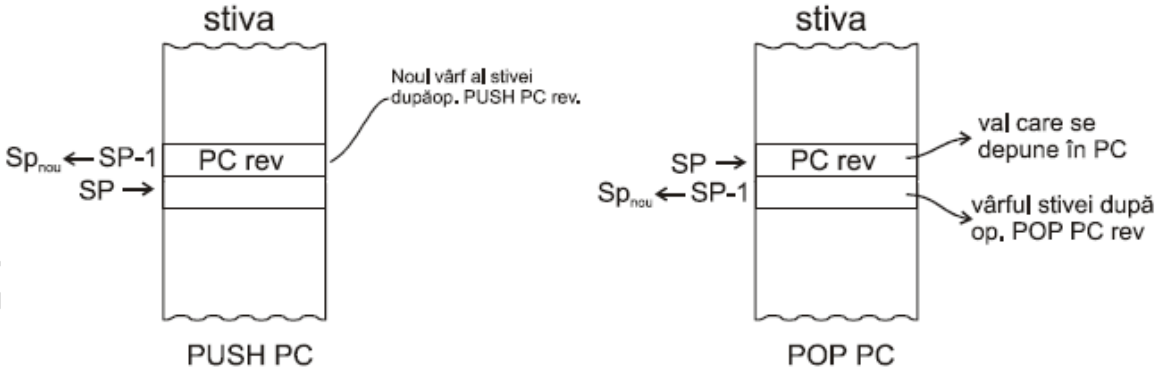
microprocesorul va executa instrucțiuni utile din PP (sistem de operare, program utilizator etc.) și deci nu mai este necesar să mai aştepte inutil ca în cazul 1.
Totalitatea acţiunilor executate de către microprocesor din momentul apariţiei semnalului de întrerupere INT până în momentul procesării primei instrucţiuni din RTI formează aşa numitul **protocol hardware de acceptare a întreruperii** (săgeţile 1 și 3 din figură). În principiu acest protocol se desfăşoară în următoarele etape succesive:
1. Odată sesizată întreruperea INT de către microprocesor acesta îşi va termina instrucţiunea în curs de execuție după care, dacă anumite condiţii sunt îndeplinite (nu există activată o cerere de întrerupere sau de bus mai prioritare etc.), va trece la pasul 2. În general, microprocesoarele examinează activarea întreruperilor la finele ultimului ciclu aferentinstrucţiunii în curs de execuție.
2. Recunoaşterea întreruperii: microprocesorul va iniţializa aşa numitul **ciclu de achitare a întreruperii**. Pe parcursul acestui ciclu extern va genera un semnal de răspuns (achitare) a întreruperii INTACK (*interrupt acknowledge*) spre toate interfețele de intrare - ieșire. Ca urmare a recepţionării INTACK interfaţa care a întrerupt va furniza microprocesorului prin intermediul bus-ului de date un aşa numit octet **vector de întrerupere** (**VI**). Acest VI este diferit pentru fiecare periferic în parte, individualizându-l deci într-un mod unic. Pe baza acestui VI şi conform unui algoritm care diferă de la microprocesor la microprocesor, acesta va determină adresa de început a RTI, adresă ce va urma să o introducă în PC. Firește, la VI diferiţi vor corespunde adrese de început diferite.
3. Microprocesorul va salva într-o zonă specială de program numită **memorie stivă**, PC-ul aferent instrucțiunii imediat următoare instrucțiunii executate de către microprocesor din PP (PCrev), pentru a putea şti la finele RTI unde să revină exact în PP.
Memoria stivă este o zonă de memorie RAM caracterizată la un moment dat de aşa numitul vârf al stivei **adică de ultima locaţie ocupată din stivă**. Acest vârf al stivei este pointat (adresat) permanent de conținutul unui registru special dedicat, existent în orice microprocesor modern, numit SP (*stack pointer*).
În memoria stivă sunt posibile 2 tipuri de operații:

operaţia PUSH Reg care se desfăşoară astfel:

$$\begin{cases} SP \leftarrow (SP) - 1 \,(cuvânt = octet) \\ (Reg) \rightarrow Mem \,|\, adr.SP \end{cases}$$

operaţia POP Reg:

$$\begin{cases} (Reg) \leftarrow Mem \,|\, adr.SP \\ SP \rightarrow (SP) + 1 \end{cases}$$

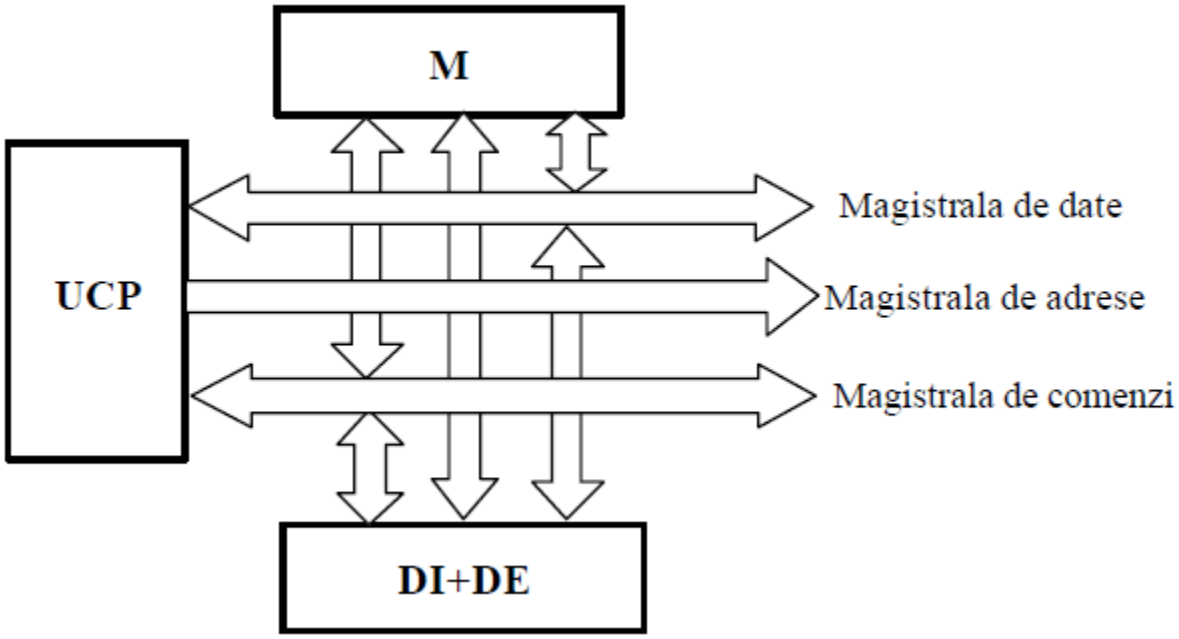


Stiva este o memorie de tip LIFO (*last în first out*) și care spre deosebire de PC în procesarea secvenţială, "creşte" (PUSH) de obicei înspre adrese descrescătoare evitându-se astfel suprapunerea zonelor de program (cod) cu cele de stiva.
4. Intrarea în RTI se face simplu prin introducerea adresei de început a RTI calculată în pasul 2, în registrul PC. Normal în continuare microprocesorul va aduce şi executa prima instrucțiune din RTI protocolul de tratare fiind în acest moment încheiat și controlul fiind preluat de RTI a perifericului care a fost întrerupt.

După cum s-a observat protocolul de tratare salvează în stiva doar PC-ul de revenire (la anumite microprocesoare se mai salvează registrul de stări - *flags*). Acest fapt se poate dovedi insuficient având în vedere că în cadrul RTI pot fi alterați anumiți regiștri interni ai microprocesorului.
Această alterare a regiştrilor poate fi chiar catastrofală la revenirea în PP. Din acest motiv cade în sarcina celui care scrie RTI să salveze (instrucțiuni PUSH) respectiv să returneze corespunzător (instrucţiuni POP) aceşti regiştri.

5.2. Magistrala procesorului, magistrala de date, magistrala de adese, magistrala I/O

Magistrala se definește ca un mediu de comunicație între componentele unui calculator.
O magistrală se compune dintr-un set de semnale prin care se transmit date și comenzi. Transferul de date pe magistrală se face pe baza unui set de reguli. Aceste reguli stabilesc cine, când și cum comunică pe magistrală; stabilesc secvența de apariție a semnalelor, intercondiționările existente între semnale și relațiile de timp între semnale.
Clasificarea magistralelor s-a făcut în capitolul 4.1.
Schema bloc a unui microsistem (Microprocesor, amplificatoare de magistrale, magistrale de adrese, date comenzi și stări, module memorie ROM şi RAM, porturi I/O lente, porturi I/O rapide – interfețe DMA, program incarcator - POST, programe BIOS) este prezentată mai jos:



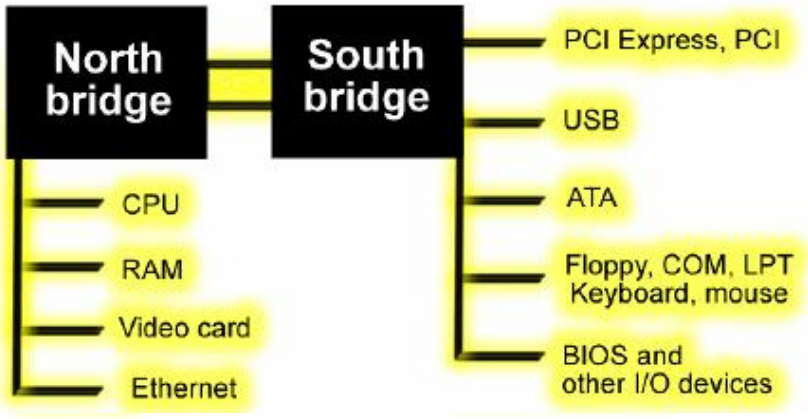
Liniile de comunicație între cele patru module sunt grupate în **magistrale.** Informatia vehiculata *date, adrese și semnale de comandă,* astfel încât în structura calculatorului există trei magistrale, respectiv:

- magistrala de date;
- magistrala de adrese;
- magistrala de comenzi.

5.3. Exemple şi comparatii.

Magistrale în calculatoare personale

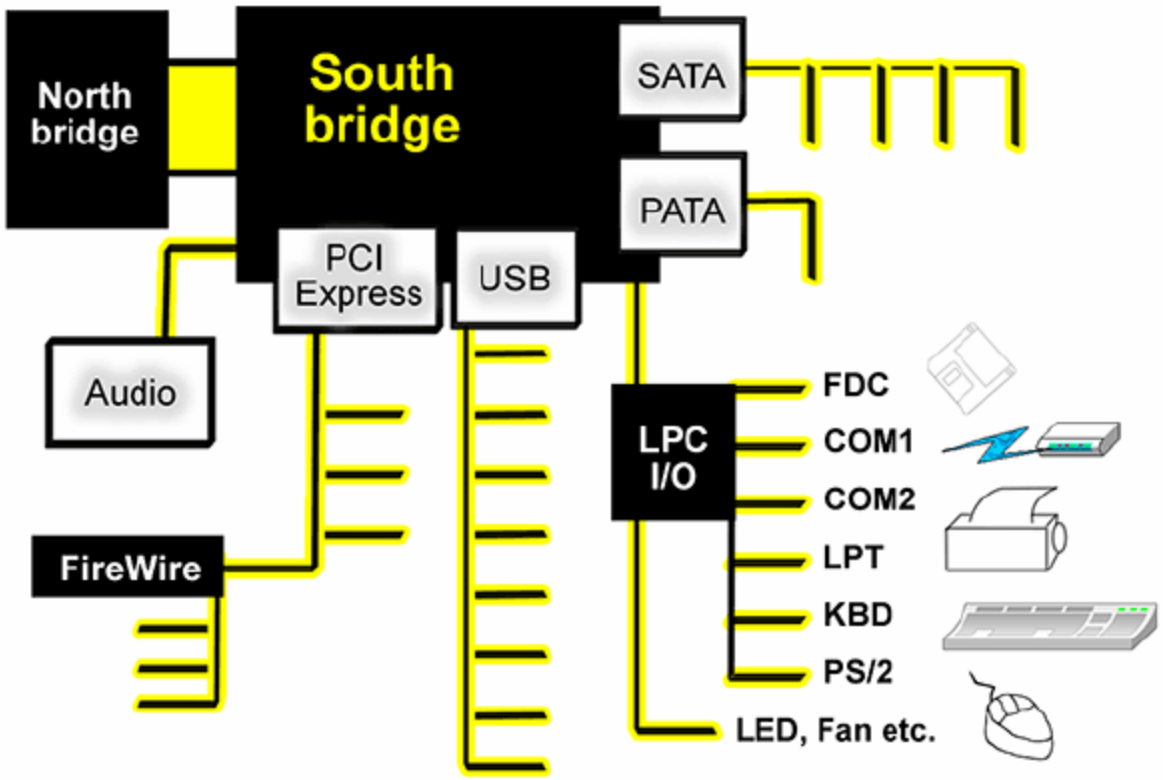
Clasificarea mgistralelor dintr-un calculator personal este prezentată mai jos.



Bus	The north bridge's buses	The I/O buses
Variants	FSB, RAM, AGP, PCI Express X16, CSA	ISA, PCI, PCI Express, USB, ATA, SCSI, FireWire
Connects	CPU, RAM, Video, Ethernet	All other devices.
Clock freq.|	66 - 1066 MHz	Typically 10-33 MHz.
Maximum capacity	> 3 GB/sec.	Typically 20-500 MB/sec. per bus

Name	Devices
KBD, PS2, FDC, Game	Keyboard, mouse, floppy disk drive, joystick, etc.
ROM, CMOS	BIOS, setup, POST.
ATA	Hard disk, CD-ROM/RW, DVD etc.
PCI and PCI Express	Network card, SCSI controller, video card, sound cards and other adapters.
USB	Mouse, scanner, printers, modem, external hard disks and much more.
Firewire	Scanner, DV camera, external hard disk
SCSI	Hard disks, CD-ROM drives, scanners, tape devices etc. (older)
LPT, COM	Parallel and serial devices such as printers, modems, etc

- **Magistrale I/E , "south bridge" și controlerul "Super I/O"**



Functiile "South bridge"

Componenta	Descriere
DMI	Direct Media Interface este conexiunea cu memoria cu o latime de banda de max 2 GB/sec.
PCI Express	Bus hi-speed pt. adaptoare I/O
PCI ports	Standard I/O bus.
Serial ATA	Controler pentru pana la patru hard disc-uri SATA
Matrix Storage	Advanced Host Controller Interface pentru RAID 0 and 1 pe aceleasi drive-uri.
Ultra ATA/100	Controler pentru dispozitive PATA ca hard discuri, unitati DVD- si CD.
USB ports	Porturi Hi-speed USB 2.0.
7.1 channel audio	Dispozitiv sunet cu surround, Dolby Digital and Digital Theater System.
AC97 modem	Modem integrat.
Ethernet	Controler retea 10/100 Mbs.

Functiile "Super I/O"

Controler	Descriere
FDC	Controler floppy disk compatibil 82077
UART	Port serial, controler compatibil 16550, incluzand IrDA (infrarosu)
LPT	Port imprimanta paralel
Game	Pentru un joystick
KBD	Controler tastatura compatibil 8042
PS/2	Port mouse
LED	Control LED'uri panou frontal PC
Fan	Control ventilator

6. Compilatoare și asambloare

Software pentru generarea de programe în limbaj maşină sunt de regulă compilatoarele şi asambloarele.
Un **compilator** este un program care transformă programul scris într-un limbaj de programare de nivel înalt în limbaj maşină. Compilatoarele pentru un limbaj de programare vor avea partea „din față" (cea care recunoaşte construcțiile din limbajul de programare de nivel înalt) identică, iar „partea din spate" (cea care creează codul maşină) diferită pentru fiecare tip de sistem de calcul. Se poate ca acelaşi program compilat cu compilatoare diferite pentru acelaşi sistem de calcul să producă cod diferit.
În procesul de compilare al unui program, programul sursă, scris într-un limbaj de programare de nivel înalt, este transformat în cod în limbaj de asamblare iar mai apoi codul în limbaj de asamblare este transformat în cod maşină de către asamblor. Aceste traduceri au loc în faza de compilare respectiv de asamblare. Programul obiect care rezultă poate fi link-editat cu alte programe obiect în faza de linkeditare. Programul link-editat, stocat de regulă pe disc, este încărcat în memoria principală în faza de încărcare a programului și este executat de către procesor în faza de execuție (run-time).

Pentru activitatea de programare sunt utile generatoarele de programe. Acestea transformă programul sursă într-un nou format. Din această categorie fac parte: macrogeneratorul, asamblorul (relocabil, absolut), compilatoarele, interpretoarele, editorul de legături, bibliotecarul, editoarele de texte etc.
Macrogeneratorul analizează un text sursă conţinând descrieri într-un limbaj special și prezintă la ieşire un fişier cu text scris în limbaj de asamblare, care poate fi prelucrat ulterior de asamblorul relocabil sau cel absolut. De exemplu, folosind TASM (al firmei Borland) se pot asambla programe în format (cod) Intel. Pentru a putea avea portabilitate între sistemele de tip Microsoft şi Linux, pentru procesoare Intel și compatibile se poate utiliza NASM. Pentru testarea programelor scrise în limbajul MMIXAL, pentru procesorul MMIX, se poate utiliza un simulator MMIX.
Programul sursă, scris de utilizator în limbaj de (macro)asamblare, este constituit dintr-un număr de linii. Fiecare linie conține o instrucțiune a limbajului de asamblare sau o directivă de macrogenerare. Rolul macrogeneratorului este de a expanda macroinstrucțiunile existente şi a rezolva, pe cât posibil, blocurile condiţionale.
Macrogeneratorul recunoaşte şi analizează: directivele de control numeric (tipul bazei de numeraţie), directivele de terminare (**.END**), directivele de asamblare condiţională (**.IF**, **.IFF**, **.IFT** etc.), directivele de definire a macroinstrucţiunilor (**.MACRO**, **.ENDM**), directivele de control (mesaje de eroare), directivele de generare și substituire, directivele de apelare a macrodefiniţiilor dintr-o bibliotecă etc.
Asamblorul relocabil transformă modulele sursă scrise în limbaj de asamblare într-un modul obiect relocabil, o tabelă de simboluri și un listing de asamblare. Asamblorul relocabil acceptă la intrare unul sau mai multe fişiere sursă în limbaj de asamblare obţinute folosind macrogeneratorul sau scrise de utilizator. Ieşirile asamblorului constau dintr-un fişier obiect (**.obj**, **.o, .mmo** etc.), un fişier de listare (**.lst**) și o tabelă de simboluri (**.map**).
Asamblorul absolut transformă modulele scrise în limbaj de asamblare (ieşire a macrogeneratorului sau scrise de utilizatori) într-un program executabil (**.tsk, .cmd, .out**, etc.)
Compilatoarele efectuează translatarea programului sursă în program obiect relocabil. Pentru ca un astfel de modul să devină program executabil este necesară editarea legăturilor. Funcţia principală a editorului de legături este de a transforma şi înlănţui modulele obiect relocabile rezultate în procesul de asamblare şi/sau compilare pentru a obţine un program executabil acceptabil de programul încărcător al sistemului de operare. În faza de editare a legăturilor pot fi incorporate și module obiect situate în biblioteci relocabile.
Un compilator este structurat în patru componente principale:

- analizor lexical,

- analizor sintactic şi semantic,
- generator de cod și
- optimizator.

Toate aceste componente gestionează un tabel de simboluri.
Analizorul lexical realizează o primă translatare a textului programului sursă într-un şir de entităţi lexicale ce constituie reprezentări interne ale unor categorii sintactice precum: cuvinte cheie, identificatori, constante, delimitatori, operatori etc. Astfel, în fazele următoare se poate lucra cu simboluri de lungime fixă și cu un număr mai mic de categorii sintactice. Analizorul lexical va completa tabelul de simboluri.
Analizorul sintactic şi semantic verifică corectitudinea sintactică a instrucțiunilor şi colectează atributele semantice ale categoriilor sintactice din program. Ieşirea analizorului sintactic şi semantic o constituie o reprezentare codificată a structurii sintactice și un set de tabele ce conţin atributele semantice ale diferitelor categorii sintactice (simboluri, constante etc.)
Generatorul de cod va traduce ieşirea analizorului sintactic şi semantic în format binar relocabil sau în limbaj de asamblare.
Optimizatorul are rolul de a prelucra ieşirea generatorului de cod cu scopul de a minimiza memoria necesară la execuție şi a elimina redundanţele din corpul programului. Unele compilatoare efectuează anumite optimizări înainte de generarea codului.

O funcție importantă a compilatorului constă în detectarea erorilor din programul sursă şi corectarea sau acoperirea lor. Spre deosebire de compilatoare, interpretoarele efecuează și executarea programului odată cu traducerea programului sursă, furnizând la ieșire rezultatele programului.
Mai precis, diferenţa faţă de un compilator este aceea că interpretorul nu produce un program obiect ce urmează a fi executat după interpreatare, ci chiar execută acest program.
Din punct de vedere structural, un interpretor se aseamănă cu un compilator, dar forma intermediară obţinută nu e folosită la generarea de cod, ci pentru a uşura decodificarea instrucţiunii sursă în vederea executării ei. Este cazul interpretorului sistemului AUTOCAD care analizează linia de comandă și, dacă comandă este corectă, realizează funcția solicitată.
Un editor de texte este un program on-line (răspunsul la comenzi este imediat), ce acceptă comenzi introduse de la terminal pentru a scrie şi/sau şterge caractere, linii sau grupuri de linii din programul sursă sau din orice fişier text.
Editoarele de texte pot lucra în mod linie şi/sau mod ecran. Ele pun la dispoziţie comenzi privind deplasarea cursorului în text (pe linie, în cadrul unui bloc sau în cadrul întregului fişier), manipularea blocurilor de text (marcare, deplasare, copiere, ştergere), comenzi de formatare a ieşirii etc. Exemplificăm prin Edit (Ms-DOS, Windows), Notepad (Windows), vi sau emacs (UNIX) etc.

Cu toate că majoritatea programelor se scriu în limbaje de nivel înalt, programatorii pot scrie programe sau secvenţe din unele programe, care trebuie să ruleze foarte repede, în asamblare. În plus, s-ar putea să nu existe compilatoare pentru anumite procesoare speciale sau compilatoarele să nu poată fi folosite pentru a efectua anumite operații speciale. În aceste cazuri, programatorul este nevoit să recurgă din nou la limbajul de asamblare.
Limbajele de programare de nivel înalt ne permit să ignorăm arhitectura sistemului de calcul țintă. Pe de altă parte, la nivel limbaj maşină, arhitectura este cel mai important aspect de care trebuie ţinut cont.

6.1. Limbaje de nivel înalt, limbaje de nivel scăzut

Limbajele de programare de nivel înalt sunt independente de mașină (microarhitectura hardware) pe care se procesează. Nivelul hardware este abstractizat din punct de vedere al limbajului (operațiile nu depind de arhitectura setului de instrucțiuni – ISA) [Patt03, Zah04]. *C* este primul limbaj de nivel mediu (considerat

de nivel înalt de către alţi cercetători) destinat creării de sisteme de operare (anterior acestuia sistemele de operare erau scrise în limbaje de asamblare). De asemenea, în C este permisă manipularea structurilor hardware ale calculatoarelor (regiştrii, memorie, porturi). C este un limbaj de programare standardizat, compilat, implementat pe marea majoritate a platformelor de calcul existente azi. Este apreciat pentru eficienţa codului obiect pe care îl poate genera, și pentru portabilitatea sa. A fost dezvoltat la începutul anilor 1970 de Brian Kernighan și Dennis Ritchie, care aveau nevoie de un limbaj simplu și portabil pentru scrierea nucleului sistemului de operare UNIX. Sintaxa limbajului C a stat la baza multor limbaje create ulterior și încă populare azi: C++, Java, JavaScript, C#. *C* este un limbaj *case sensitive*.
Instrucţiunile din C care rezolvă o problemă sunt mult mai puţine decât cele ale limbajului asamblare aferent calculatorului LC-3 care rezolvă aceeaşi problemă [Patt03]. De asemenea, LC-3 nu asigură instrucțiuni de înmulţire și împărţire. Limbajele de nivel înalt sunt mai expresive decât cele de nivel mediu sau jos (asamblare). Sunt folosite simboluri pentru variabile și expresii simple pentru structuri de control (if-else, switch-case) sau repetitive (for, while, do-while). Permite compilarea condiţionată. Deşi limbajul C oferă o libertate foarte mare în scrierea codului, acceptând multe forme de scriere care în alte limbaje nu sunt permise, acest lucru poate fi și un dezavantaj, în special pentru programatorii fără experienţă.

6.1.1. Compilare vs. Interpretare

Programele de nivel înalt (*High Level Languages* – HLL) pot fi translatate în instrucțiuni maşină (ISA corespunzătoare arhitecturii hardware pe care se va procesa) prin două metode:

- **Interpretare:** Codul sursă HLL este „*interpretat*” de o maşină virtuală (cele mai cunoscute sunt maşinile virtuale Java – JVM) care translatează „*secţiuni*” din codul sursă în limbaj maşină și îl execută direct pe arhitectura gazdă, trecând apoi la următoarea *secţiune* de cod sursă. Dintre avantajele interpretării faţă de metodele tradiţionale de compilare s-ar menţiona simplitatea implementării hardware dar şi faptul că nu necesită o zonă mare de memorie pentru stocarea programului compilat. Principalul dezavantaj îl reprezintă viteza scăzută de execuție a aplicaţiei, dar şi necesitatea existenţei în momentul interpretării atât a codului sursă HLL cât şi a maşinii virtuale care realizează interpretarea.
- **Compilare:** Compilarea directă translatează codul sursă (poate fi Java, C, C++, Fortran, Pascal) în instrucțiuni maşină, direct executabile pe un procesor ţintă, dar nu le execută ca în cazul interpretoarelor. Orice modificare a codului sursă necesită recompilare. Procesul este realizat static şi poate îngloba tehnici de optimizare de tip analiza dependenţelor de date dintre instrucțiuni, analiză interproceduralã. Se caracterizează printr-o lipsă de portabilitate.

6.2. Descrierea componentelor unui compilator

Compilatorul translatează un program sursă scris într-un limbaj de nivel înalt (C, Pascal) în acelaşi program - obiect, de regulă scris în limbaj de asamblare. În literatura de specialitate, este denumită *fază* a unui compilator o succesiune de operaţiuni prin care un program de la intrare suferă anumite modificări. Prin *trecere* aparţinând unui compilator se înţelege o citire a programului dintr-un fişier, transformarea lui conform unor faze și scrierea rezultatului în alt fişier(de ieșire).

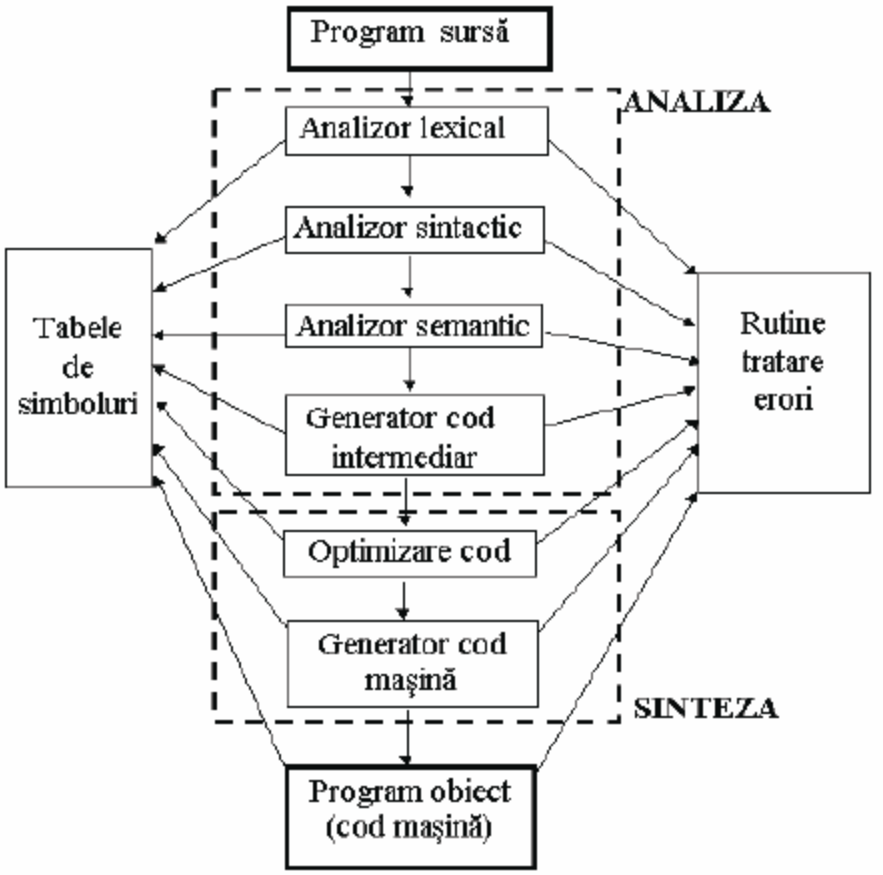


Figura anterioară ilustrează cele două faze majore ale procesului de compilare propriu-zisă:

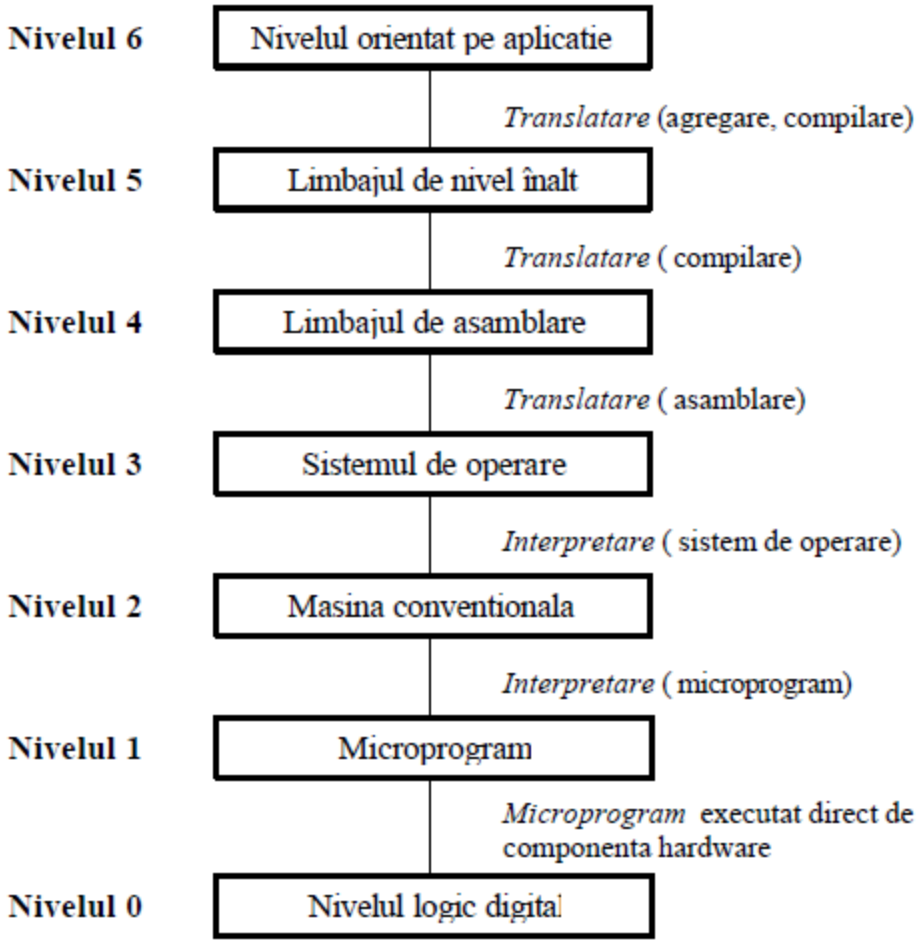
- **analiza** (de tip "*front end*") - în care se identifică părţile constituente fundamentale ale programului (variabile, expresii, declaraţii, definiţii, apeluri de funcţii) şi se construieşte o reprezentare internă a programului original, numită „*cod intermediar*" (*analiza lexicală* produce un -> şir de atomi lexicali -> *analiza sintactică* generează -> arborele sintactic -> *analiză semantică* construieşte o reprezentare a programului sursă în-> *cod intermediar*). Este dependentă de limbajul de nivel înalt şi nu de maşina pe care se va procesa.
- **sinteza** (de tip "*back end*") - generează cod maşină eventual optimizat. Se disting două etape
  - optimizare cod intermediar (scheduling) pentru o anumită maşină;
  - generare de cod maşină (generare cod într-o gamă variată de formate: *limbaj maşină absolut, limbaj maşină relocabil* sau *limbaj de asamblare* urmat de alocare de resurse). Această fază este puternic dependentă de mașină pe care se va executa codul obiect generat.

Înaintea primei faze din cadrul procesului de compilare trebuie realizată de cele mai multe ori o **preprocesare**. Preprocesorul translatează un program al cărui limbaj sursă este de nivel înalt (C, Pascal) într-un program destinaţie (obiect) scris tot într-un limbaj de nivel înalt C. Practic are loc interpretarea *directivelor de preprocesare* (încep cu # - *include, define, if !defined*(**…**)**…***endif*). De exemplu, preprocesorul trebuie să introducă conţinutul fişierului <*stdio.h*> în codul sursă al aplicaţiei create acolo unde apare respectiva directivă de preprocesare, sau, să înlocuiască anumite valori constante declarate ca şiruri de caractere cu valorile numerice aferente, dacă acest lucru se specifică printr-o directivă de tip *#define* în cadrul programului.

Analiza lexicală reprezintă prima fază a procesului de compilare, şi care este responsabilă de transformarea programului sursă văzut ca o succesiune de caractere (text) într-o succesiune de atomi lexicali și de atribute ale lor. Conform definiţiei din , un atom lexical este o entitate indivizibilă a unui program - identificatori de variabile/funcții, constante de diverse tipuri, operatori, cuvinte cheie (if, else, switch/case…), delimitatori. Atributele atomilor lexicali sunt: clasă, tip, lungime, loc în tabela de simboluri, dacă a fost definit sau nu, etc. Clasa atributului se referă la faptul că este cuvânt-cheie, identificator de variabilă, constantă de tip întreg sau real etc.
Toate fazele unui compilator dispun de **rutine de tratare a erorilor** şi lucrează în comun cu una sau mai multe tabele de simboluri în care se păstrează informaţii despre cuvintele cheie ale limbajului, identificatorii de variabile (tip, lungime, adresă stocare în memorie etc.), etichete de instrucțiuni, identificatori de proceduri și funcții și alte informaţii utile despre programul în curs de compilare. **Tabela de simboluri** este creată în faza de analiză a programului sursă şi folosită în scopul validării numelor simbolice facilitând generarea de cod . Tabela de simboluri realizează o asociere simbolică între numele (identificatorul) unei variabile și caracteristicile acesteia (tip, domeniu de vizibilitate, deplasament faţă de începutul zonei de date statice sau dinamice, stivă). Compilatorul foloseşte tabela de simboluri pentru a urmări domeniul de utilizare și valabilitate a unui nume şi de a adăuga informaţii despre un nume. Tabela de simboluri este cercetată de fiecare dată când un nume este întâlnit în textul sursă. Mecanismul asociat tabelei de simboluri trebuie să permită adăugarea a noi intrări și găsirea informaţiilor existente în mod eficient. Tabela de simboluri este similară celei generate de asamblor însă cuprinde mai multe informaţii decât aceasta. În asamblare, toţi identificatorii (variabilele) erau etichete (*labels*) şi informaţiile reprezentate de acestea erau adrese. Tabela de simboluri conține informaţii legate de toate variabilele din program.
Generatorul de cod constituie faza finală a unui compilator. Primeşte la intrare reprezentarea intermediară a programului sursă împreună cu informaţia din tabela de simboluri - folosită la determinarea run-time a adresei obiectelor de date, desemnate de nume în reprezentarea intermediară, și produce la ieşire un program obiect echivalent. Codul de ieşire trebuie să fie corect și de înaltă calitate, aceasta însemnând că el trebuie să folosească eficient resursele maşinii pentru a putea fi procesat cât mai rapid.
Ieşirile generatorului de cod se găsesc într-o gamă variată de forme: *limbaj de asamblare*, *limbaj maşină relocabil* sau *limbaj maşină absolut*. Producând un program într-un limbaj de asamblare, caracterizat de o mai mare lizibilitate, generarea codului se realizează mult mai uşor. Pot fi generate instrucțiuni simbolice precum şi utilizate facilităţile de macro-asamblare ajutătoare la generarea de cod. Producând un program în limbaj maşină absolut, există avantajul că programul poate fi plasat într-o locaţie fixă a memoriei și poate fi imediat executat. Producând un program în limbaj maşină relocabil, acesta permite subprogramelor să fie compilate separat (similar cu fişierul *Makefile* din cadrul setului de instrumente SimpleScalar 3.0). O mulţime de module-obiect relocate pot fi *legate* (*linked*) împreună și apoi încărcate pentru execuție de către un *loader*. Câştigul de flexibilitate care dă posibilitatea compilării separate a subrutinelor şi apelarea altor programe compilate anterior dintr-un modul obiect este tributar costurilor suplimentare pentru legare şi încărcare.

6.3. Programarea în limbaj de asamblare

Limbajul de asamblare permite accesul la unele dintre resursele fizice ale masinii în condițiile utilizarii unor mnemonice mai usor de manevrat decât codurile numerice asociate instrucțiunilor limbajului maşină și are nivelul 4 – nivelul limbajului de asamblare:



Există câte un astfel de limbaj specific fiecărei familii de procesoare.

6.3.1. Familia de procesoare Intel x86

Familia de procesoare Intel x86 cuprinde seria de procesoare 8086, 8088, 80286, '386, '486, Pentium, Pentium Pro, şi variantele ultime Pentium II,III şi IV. Toate aceste procesoare împărtăşesc o arhitectură comună și utilizează același set de instrucțiuni de asamblare. Îmbunătăţirile arhitecturale aduse în decursul timpului la noile versiuni de procesoare s-au făcut în aşa fel încât să se menţină compatibilitatea cu imaginea iniţială a unui procesor I8086. Chiar dacă în interior un procesor din ultima generaţie diferă semnificativ de structura primului procesor din familie, acest lucru este vizibil în mică măsură pentru un programator. De aceea nu se mai vorbeşte de o anumită arhitectură particulară de procesor ci de o arhitectură corespunzătoare unui set de instrucțiuni de asamblare. Astfel pentru seria de procesoare Intel pe 16 și 32 de biţi se operează cu imaginea arhitecturală denumită ISA x86 (Instruction Set Architecture).
În acelaşi mod pentru procesoarele pe 64 de biţi s-a introdus arhitectura ISA 64. Această arhitectura diferă semnificativ de cea a procesoarelor pe 16 şi 32 de biţi, iar setul instrucțiunilor de asamblare este diferit. Astfel un program scris pentru arhitectura ISA 64. nu mai este compatibil la nivel de coduri de instrucțiune cu arhitectura ISA x86. Este interesant de observat că Intel a propus această nouă arhitectură chiar înainte de lansarea pe piaţă a unui procesor pe 64 de biţi.
În ceea ce priveşte arhitectura ISA x86, aceasta a suferit anumite modificări impuse de introducerea, în versiunile mai noi de procesoare, a unor elemente arhitecturale şi funcţionale noi. Însă aceste elemente noi

nu modifică componentele existente deja, iar programele scrise pentru primele versiuni de procesoare rulează și pe versiunile noi fără nici o modificare.
Cunoașterea arhitecturii ISA x86 este strict necesară pentru a putea scrie programe în limbaj de asamblare.

Componentele

În continuare se prezintă acele componente ale unui procesor care sunt vizibile și accesibile pentru un programator în limbaj de asamblare. Astfel, un procesor dispune de un set de registre interne folosite pentru păstrarea temporară a datelor, a adreselor sau a instrucţiunilor. Există registre generale folosite în majoritatea operațiilor aritmetico-logice și registre speciale, care au o destinaţie specială.
La arhitectura ISA x86 pe 16 biți există 8 registre generale pe 16 biţi, registre denumite în felul următor: AX, BX, CX, DX, SI, DI, BP, SP. Dintre acestea primele patru se folosesc cu precădere în operații aritmetico-logice pentru păstrarea operanzilor, iar următoarele patru mai ales pentru păstrarea și calculul adreselor de operanzi. Primele patru registre se pot adresa şi pe jumătate, adică la nivel de octet. Astfel fiecare registru are o parte superioară "HIGH" (biţii D15-D8) și o parte inferioară "LOW" (biţii D7-D0). Denumirea lor este AH şi AL pentru registrul AX, BH şi BL pentru registrul BX şi așa mai departe.
La procesoarele pe 32 de biți (începând de la '386) aceste registre s-au extins la 32 de biţi. Astfel registrele pot fi adresate pe 32 de biţi (cu denumirile EAX,EBX, ECX, ... ESP) sau în mod uzual pe 16 sau 8 biţi. Partea superioara a unui registru pe 32 de biți (D31-D16) nu se poate adresa individual.
În principiu registrele generale respectă principiul de ortogonalitate, adică pentru majoritatea operațiilor oricare registru este utilizabil. La unele procesoare (ex: procesoarele Motorola) acest principiu este respectat strict. În cazul procesoarelor Intel anumite operații speciale impun utilizarea anumitor registre, ceea ce înseamnă că există o oarecare specializare între registre. Această specializare decurge de multe ori și din denumirea registrelor. Astfel:

- registrul AX (EAX în varianta pe 32 de biţi) – se foloseşte ca registru "acumulator" în majoritatea operațiilor aritmetice și logice, adică păstrează unul dintre operanzi și apoi rezultatul operaţiei; la operațiile de înmulţire și împărţire în mod obligatoriu primul operand şi rezultatul se păstrează în acest registru
- registrul BX (EBX) – se foloseşte pentru în operații aritmetico-logice sau pentru calculul adresei operandului la "adresarea bazată"
- registrul CX (ECX) – se foloseşte pentru operații aritmetico-logice sau în mod implicit, la anumite instrucțiuni (ex: instrucţiuni de buclare), pe post de contor
- registrul DX (EDX) – se foloseşte pentru operații aritmetico-logice sau pentru păstrarea adresei porturilor la instrucțiunile de intrare/ieşire; de asemenea la operaţiile de înmulţire și împărţire se foloseşte ca extensie a registrului acumulator
- registrele SI și DI (ESI, EDI) – denumite și registre index, se folosesc cu precădere pentru calculul adresei operanzilor la "adresarea indexată"; SI adresează elementele şirului sursă (source index), iar DI elementele şirului destinaţie (destination index)
- registrul BP (EBP) – se foloseşte cu precădere pentru calculul adresei operanzilor la "adresarea bazată", alături de registrul BX
- registrul SP (ESP) – se foloseşte aproape în exclusivitate pentru adresarea stivei (stack pointer); conţinutul registrului se incrementează și se decrementează automat la orice operaţie cu stiva

În afara registrelor generale un procesor mai are și registre speciale sau de control. La procesoarele Intel numai o parte din aceste registre sunt vizibile și accesibile pentru programator. Aceste registre controlează regimul de lucru al procesorului, sau permit efectuarea unor operații speciale de manipulare a spaţiului de memorie.
Arhitectura ISA x86 operează cu următoarele registre speciale:

- registrul PC – "program counter" – păstrează adresa instrucţiunii care urmează; nu este adresabil direct (prin nume) dar conținutul său se poate modifica prin execuția instrucțiunilor de salt
- registrul PSW – "program status word" – păstrează indicatorii de stare ai procesorului;
    - o parte din indicatori caracterizează rezultatul obţinut în urma unei anumite instrucţiuni:
        - ZF – rezultat 0,

        - SF – semnul rezultatului
        - OF – "overflow" indică o depăşire de capacitate la ultima operaţie aritmetică
        - PF – indicator de paritate (arată paritatea rezultatului)
        - CF – indicator de transport ("carry"), arată dacă s-a generat un transport
        - AC – indicator de transport auxiliar, arată dacă după primii 4 biţi s-a generat un transport
    - o parte din indicatori controlează modul de lucru al procesorului:
        - IF – indicator de întrerupere (interrupt"), dacă este setat (1 logic) atunci se blochează toate întreruperile mascabile, în caz contrar (0 logic) se validează
        - DF – indicatorul de direcţie, arată direcţia de parcurgere a şirurilor de caractere la instrucțiunile pe șiruri (în ordine ascendentă sau descendentă a adreselor)
- registrele de segment – se utilizează pentru calculul adresei operanzilor şi a instrucţiunilor; cu ajutorul lor memoria este divizată în segmente; există 4 registre segment în versiunea iniţială ISA x86 şi 6 registre segment în versiunea pe 32 de biţi:
    - registrul CS – registrul segment de cod, folosit pentru adresarea instrucțiunilor (a codurilor); acest registru păstrează adresa de început (descriptorul în varianta extinsă) pentru segmentul de cod (segmentul unde se află programul)
    - registrul DS – registrul segment de date, folosit pentru adresarea operanzilor din memorie; acest registru păstrează adresa de început (sau descriptorul) pentru segmentul de date
    - registrul SS – registrul segment de stivă, folosit pentru adresarea memoriei stivă; păstrează adresa de început (descriptorul) segmentului unde se află organizata stiva
    - registrul ES – registrul extra-segmentului de date, folosit pentru păstrarea adresei celui de al doilea segment de date
    - registrele FS și GS – registre segment introduse începând de la versiunea '386 şi care se folosesc pentru adresarea operanzilor

Versiunile mai noi de procesoare conţin şi alte registre speciale, dar acestea au o importanţă mai mică pentru un programator obişnuit. Ele vor fi introduse pe parcurs în momentul în care acest lucru va fi necesar.
Un limbaj de asamblare conține instrucțiuni corespunzătoare unor operații simple care sunt direct interpretate şi executate de procesor. Fiecărei instrucţiuni din limbajul de asamblare îi corespunde în mod strict un singur cod executabil. În contrast, unei instrucțiuni dintr-un limbaj de nivel înalt (ex: C, Pascal, etc.) îi corespunde o secvenţă de coduri (instrucțiuni în cod maşină). Un anumit limbaj de asamblare este specific pentru un anumit procesor sau eventual pentru o familie de procesoare. Instrucţiunile sunt în directă corelaţie cu structura internă a procesorului. Un programator în limbaj de asamblare trebuie să cunoască această structură precum şi tipurile de operații permise de structura respectivă.
Un program în asamblare scris pentru o anumită arhitectură de procesor nu este compatibil cu un alt tip de procesor. Pentru implementarea unei aplicaţii pe un alt procesor programul trebuie rescris. În schimb programele scrise în limbaj de asamblare sunt în general mai eficiente atât în ceea ce priveşte timpul de execuţie cât şi spaţiul de memorie ocupat de program. De asemenea, programarea în limbaj de asamblare dă o mai mare flexibilitate şi libertate în utilizarea resurselor unui calculator. Cu toate acestea astăzi utilizarea limbajului de asamblare este mai puţin frecventă deoarece eficienţa procesului de programare este mai scăzută, există puţine structuri de program şi de date care să uşureze munca programatorului, iar programatorul trebuie să cunoască structura procesorului pentru care scrie aplicaţia. În plus programele nu sunt portabile, adică nu rulează şi pe alte procesoare.

6.4. Directive şi instrucţiuni ASM.

Un program scris în limbaj de asamblare conține instrucțiuni și directive. Instrucţiunile sunt traduse în coduri executate de procesor; ele se regăsesc în programul executabil generat în urma compilării și a editării de legături. Directivele sunt construcţii de limbaj ajutătoare care se utilizează în diferite scopuri (ex:

declararea variabilelor, demarcarea segmentelor și a procedurilor, etc.) și au rol în special în fazele de compilare și editare de legături. O directivă nu se traduce printr-un cod executabil şi în consecinţă NU se execută de către procesor.

6.4.1. Sintaxa unei instrucţiuni în limbaj de asamblare

O instrucţiune ocupă o linie de program și se compune din mai multe câmpuri, după cum urmează (parantezele drepte indică faptul că un anumit câmp poate să lipsească):

[<eticheta>:] [<mnemonica> [<parametru_1> [,<parametru_2>]] [;<comentariu>]

- <eticheta> - este un nume simbolic (identificator) dat unei locații de memorie care conține instrucţiunea care urmează; scopul unei etichete este de a indica locul în care trebuie să se facă un salt în urma executării unei instrucțiuni de salt; eticheta poate fi o combinație de litere, cifre şi anumite semne speciale (ex: _), cu restricţia ca prima cifră să fie o literă
- <mnemonica> - este o combinație de litere care simbolizează o anumită instrucțiune (ex: add pentru adunare, mov pentru transfer, etc.); denumirile de instrucțiuni sunt cuvinte rezervate şi nu pot fi utilizate în alte scopuri
- <parametru_1> - este primul operand al unei instrucțiuni și în acelaşi timp și destinaţia rezultatului; primul parametru poate fi un registru, o adresă, sau o expresie care generează o adresă de operand; adresa operandului se poate exprima și printr-un nume simbolic (numele dat unei variabile)
- <parametru_2> - este al doilea operand al unei instrucțiuni; acesta poate fi oricare din variantele prezentate la primul operand și în plus poate fi și o constantă
- <comentariu> - este un text explicativ care arată intenţiile programatorului și efectul scontat în urma execuţiei instrucțiunii; având în vedere că programele scrise în limbaj de asamblare sunt mai greu de interpretat se impune aproape în mod obligatoriu utilizarea de comentarii; textul comentariului este ignorat de compilator; comentariul se considera până la sfârşitul liniei curente

Într-o linie de program nu toate câmpurile sunt obligatorii: poate să lipsească eticheta, parametrii, comentariul sau chiar instrucțiunea. Unele instrucțiuni nu necesită nici un parametru, altele au nevoie de unul sau doi parametri. În principiu primul parametru este destinaţia, iar al doilea este sursa.

Constantele numerice care apar în program se pot exprima în zecimal (modul implicit), în hexazecimal (constante terminate cu litera 'h') sau în binar (constante terminate cu litera 'b'). Constantele alfanumerice (coduri ASCII) se exprimă prin litere între apostrof sau text între ghilimele.

Clase de instrucțiuni

În această prezentare nu se va face o prezentare exhaustiva a tuturor instrucțiunilor cu toate detaliile lor de execuție. Se vor prezenta acele instrucțiuni care se utilizează mai des și au importanţă din punct de vedere al structurii și al posibilităţilor procesorului. Pentru alte detalii se pot consulta documentaţii complecte referitoare la setul de instrucțiuni ISA x86 (ex: cursul AoA – The Art of Assembly Language, accesibil pe Internet).

6.4.2. Instrucţiuni de transfer

Instrucţiunile de transfer realizează transferul de date între registre, între un registru şi o locaţie de memorie sau o constantă se încarcă într-un registru sau locaţie de memorie. Transferurile de tip memorie-memorie nu sunt permise (cu excepţia instrucțiunilor pe şiruri). La fel nu sunt permise transferurile directe între două registre segment. Ambii parametrii ai unui transfer trebuie să aibă aceeaşi lungime (număr de biţi).

Instrucţiunea MOV

Este cea mai utilizată instrucțiune de transfer. Sintaxa ei este:

[<eticheta>:] MOV <parametru_1>, <parametru_2> [;<comentariu>]

unde:
<parametru_1> = <registru>| <reg_segment>|<adresa_offset>|<nume_variabilă>|<expresie>
<parametru_2> = <parametru_1>|<constantă>
<registru> = EAX|EBX|.....ESP|AX|BX|....SP|AH|AL|....DL
<expresie> = **[**[<registru_index>][+<registru_bază>][+<deplasament>]**]**
; aici parantezele drepte marcate cu bold sunt necesare
<registru_index> = SI| DI |ESI | EDI
<registru_bază> = BX|BP |EBX| EBP
<deplasament> = <constantă>

Exemple:

```
mov ax,bx                   et1:  mov ah, [și+100h]
mov cl, 12h                       mov al, 'A'
mov dx, var16                     mov și, 1234h
mov var32,eax               sf:   mov [și+bx+30h], dx
mov ds, ax                        mov bx, cs
```

Exemple de erori de sintaxă:

```
mov ax, cl          ; operanzi inegali ca lungime
mov var1, var2  ; ambii operanzi sunt locații de memorie
mov al, 1234h       ; dimensiunea constantei este mai mare decât cea a registrului
mov ds, es          ; două registre segment
```

Instrucţiunea LEA, LDS şi LES

Aceste instrucțiuni permit încărcarea într-un registru a unei adrese de variabile. Prima instrucțiune LEA ("load effective address") încarcă în registrul exprimat ca prim parametru adresa de offset a variabilei din parametrul 2. Următoarele două instrucțiuni încarcă atât adresa de offset cât și adresa de segment; LDS încarcă segmentul în registrul DS, iar LES în ES.

LEA parametru_1>,<parametru_2>
LDS <parametru_1>, <parametru_2

Exemple:

```
lea  și, var1       ; SI<= offset(var1)
lds bx, text        ; DS<= segment(text), BX<=offset(text)
lea di, [bx+100]; DI<= BX+100
```

Instrucţiunea XCHG

Această instrucțiune schimbă între ele conținutul celor doi operanzi.

XCHG <parametru_1>, <parametru_2>

Atenţie: parametrul 2 nu poate fi o constantă.

Exemple:

```
xchg al, bh
xchg ax,bx
```

Instrucţiunile PUSH şi POP

Cele două instrucțiuni operează în mod implicit cu vârful stivei. Instrucţiunea PUSH pune un operand pe stivă, iar POP extrage o valoare de pe stivă și o depune într-un operand. În ambele cazuri registrul indicator de stivă (SP) se modifică corespunzător (prin decrementare și respectiv incrementare) astfel încât registrul SP să indice poziţia curentă a vârfului de stivă. Sintaxa instrucțiunilor este:

PUSH <parametru_1>
POP <parametru_1>

Transferul se face numai pe 16 biţi. Aceste instrucțiuni sunt utile pentru salvarea temporară şi refacerea conținutului unor registre. Aceste operații sunt necesare mai ales la apelul de rutine și la revenirea din rutine.

Exemple:

```
push ax          push var16
pop bx                 pop var16
```

6.4.3. Instrucţiuni aritmetice

Aceste instrucţiuni efectuează cele patru operații aritmetice de bază: adunare, scădere, înmulţire și împărţire. Rezultatul acestor instrucțiuni afectează starea indicatorilor de condiţie.

Instrucţiunile ADD și ADC

Aceste instrucţiuni efectuează operaţia de adunare a doi operanzi, rezultatul plasându-se în primul operand. A două instrucțiune ADC (ADD with carry) în plus adună și conținutul indicatorului de transport CF. Această instrucţiune este utilă pentru implementarea unor adunări în care operanzii sunt mai lungi de 32 de biţi.

ADD <parametru_1>,<parametru_2>
ADC <parametru_1>,<parametru_2>

Exemple:

```
add ax, 1234h
add bx, ax
adc dx, var16
```

Instrucţiunile SUB și SBB

Aceste instrucțiuni implementează operaţia de scădere. A două instrucţiune, SBB (Subtract with borrow) scade și conținutul indicatorului CF, folosit în acest caz pe post de bit de împrumut. Ca şi ADC, SBB se foloseşte pentru operanzi de lungime mai mare.

SUB <parametru_1>,<parametru_2>
SBB <parametru_1>,<parametru_2>

Instrucţiunile MUL şi IMUL

Aceste instrucțiuni efectuează operaţia de înmulţire, MUL pentru întregi fără semn şi IMUL pentru întregi cu semn. De remarcat că la operațiile de înmulţire și împărţire trebuie să se țină cont de forma de reprezentare a numerelor (cu semn sau fără semn), pe când la adunare și scădere acest lucru nu este necesar. Pentru a evita dese depăşiri de capacitate s-a decis ca rezultatul operaţiei de înmulţire să se păstreze pe o

lungime dublă faţă de lungimea operanzilor. Astfel dacă operanzii sunt pe octet rezultatul este pe cuvânt, iar dacă operanzi sunt pe cuvânt rezultatul este pe dublu-cuvânt. De asemenea se impune ca primul operand şi implicit şi rezultatul să se păstreze în registrul acumulator. De aceea primul operand nu se mai specifică.

MUL <parametru_2>
IMUL <parametru_2>

Exemple:

```
mul dh          ; AX<=AL*DH
mul bx          ; DX:AX<= AX*BX    DX este extensia registrului acumulator AX
imul var8       ; AX<=AL*var8
```

Instrucţiunile DIV şi IDIV

Aceste instrucțiuni efectuează operaţia de împărţire pe întregi fără sem și respectiv cu semn. Pentru a crește plaja de operare se consideră că primul operand, care în mod obligatoriu trebuie să fie în acumulator, are o lungime dublă faţă de al doilea operand. Primul operand nu se specifică.

DIV <parametru_2>
IDIV <parametru_2>

Exemple:

```
div   cl        ; AL<=AX/CL și AH<=AX modulo CL (adică restul împărţirii)
div    bx       ; AX<= (DX:AX)/BX și DX<=(DX:AX) modulo BX
```

Instrucţiunile INC şi DEC

Aceste instrucțiuni realizează incrementarea și respectiv decrementarea cu o unitate a operandului. Aceste instrucţiuni sunt eficiente ca lungime și ca viteză. Ele se folosesc pentru contorizare și pentru parcurgerea unor şiruri prin incrementarea sau decrementarea adreselor.

INC <parametru>
DEC <parametru>

Exemple:

```
inc   și        ; SI<=SI+1
dec  cx         ; CX<=CX-1
```

Instrucţiunea CMP

Această instrucțiune compară cei doi operanzi prin scăderea lor. Dar rezultatul nu se memorează; instrucţiunea are efect numai asupra indicatorilor de condiţie. Această instrucțiune precede de obicei o instrucțiune de salt condiţionat. printr-o combinație de instrucțiune de comparare și o instrucțiune de salt se pot verifica relaţii de egalitate, mai mare, mai mic, mai mare sau egal, etc.

CMP <parametru_1>, <parametru_2>

Exemplu:

```
cmp ax, 50h
```

6.4.4. Instrucţiuni logice

Aceste instrucțiuni implementează operațiile de bază ale logicii booleene. Operaţiile logice se efectuează la nivel de bit, adică se combină printr-o operaţie logică fiecare bit al operandului 1 cu bitul corespunzător din operandul al doilea. Rezultatul se generează în primul operand.

Instrucţiunile AND, OR, NOT şi XOR

Aceste instrucțiuni implementează cele patru operații de bază: ŞI, SAU, Negaţie şi SAU-Exclusiv.

AND <parametru_1>,<parametru_2>
OR <parametru_1>, <parametru_2>
NOT <parametru_1>
XOR <parametru_1>,<parametru_2>

Exemple:

```
and   al, 0fh
or     bx, 0000111100001111b
and   al,ch
xor   ax,ax           ; şterge conţinutul lui ax
```

Instrucţiunea TEST

Această instrucțiune efectuează operaţia ŞI logic fără a memora rezultatul. Scopul operaţiei este de a modifica indicatorii de condiţie. Instrucţiunea evită distrugerea conținutului primului operand.

TEST <parametru_1>,<parametru_2>

Exemple:

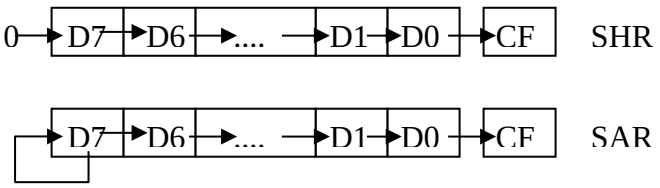
```
test al, 00010000b          ; se verifică dacă bitul D4 din al este zero sau nu
test bl, 0fh                ;se verifica dacă prima cifră hexazecimală din bl este 0
```

6.4.5. Instrucţiuni de deplasare şi de rotire

Instrucţiunile SHL, (SAL), SHR şi SAR

Aceste instrucţiuni realizează deplasarea (eng. shift) la stânga şi respectiv la dreapta a conţinutului unui operand. La deplasarea "logică" (SHL, SHR) biţii se copiază în locaţiile învecinate (la stânga sau la dreapta), iar pe locurile rămase libere se înscrie 0 logic. La deplasarea "aritmetică" (SAL, SAR) se consideră că operandul conține un număr cu semn, iar prin deplasare la stânga şi la dreapta se obține o multiplicare și respectiv o divizare cu puteri ale lui doi (ex: o deplasare la stânga cu 2 poziţii binare este echivalent cu o înmulţire cu 4). La deplasarea la dreapta se doreşte menţinerea semnului operandului, de aceea bitul mai semnificativ (semnul) se menține și după deplasare. La deplasarea la stânga acest lucru nu este necesar, de aceea instrucțiunile SHL și SAL reprezintă aceeaşi instrucțiune.

În figura de mai jos s-a reprezentat o deplasare logică și o deplasare aritmetică la dreapta. Se observă că bitul care iese din operand este înscris în indicatorul de transport CF



Formatul instrucțiunilor:

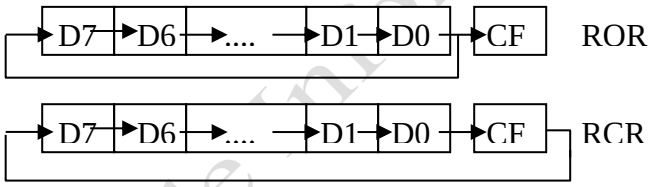
SHL <parametru_1>, <parametru_2>
SAL <parametru_1>, <parametru_2>
SHR <parametru_1>, <parametru_2>
SAL <parametru_1>, <parametru_2>

Primul parametru este în concordanţă cu definiţiile anterioare; al doilea parametru specifică numărul de poziţii binare cu care se face deplasare; acest parametru poate fi 1 sau dacă numărul de pași este mai mare atunci se indica prin registrul CL. La variantele mai noi de procesoare se acceptă şi forma în care constanta este diferită de 1.

Exemple:
```
shl    ax,1
shr    var, cl
```

Instrucţiunile de rotire ROR, ROL, RCR, RCL

Aceste instrucțiuni realizează rotirea la dreapta sau la stânga a operandului, cu un număr de poziţii binare. Diferenţa faţă de instrucţiunile anterioare de deplasare constă în faptul că în poziţia eliberată prin deplasare se introduce bitul care iese din operand. Rotirea se poate face cu implicarea indicatorului de transport (CF) în procesul de rotaţie (RCR, RCL) sau fără (ROR, ROL). În ambele cazuri bitul care iese din operand se regăseşte în indicatorul de transport CF. Figura de mai jos indică cele două moduri de rotaţie pentru o rotaţie la dreapta.



Formatul instrucțiunilor:

ROR <parametru_1>, <parametru_2>
ROL <parametru_1>, <parametru_2>
RCR <parametru_1>, <parametru_2>
RCL <parametru_1>, <parametru_2>

Referitor la parametri, se aplică aceleaşi observaţii ca şi la instrucțiunile de deplasare.

6.4.6. Instrucţiuni de salt

Instrucţiunile de salt se utilizează pentru controlul secvenţei de execuție a instrucțiunilor. În principiu există două tipuri de salt:

- instrucțiuni de ramificare și ciclare (buclare)
- instrucțiuni de apel și revenire din rutine

6.4.7. Instrucţiunile de apel şi revenire din rutine

Utilizarea rutinelor (a procedurilor) permite structurarea programelor scrise în limbaj de asamblare şi implicit scade complexitatea procesului de proiectare și programare. Se recomandă ca anumite operații care se repetă să fie scrise sub formă de rutine, urmând a fi apelate de mai multe ori. Uneori se justifică utilizarea de rutine chiar și numai cu scop de structurare a programului.

Instrucţiunile CALL şi RET

Instrucţiunea CALL realizează un salt la adresa exprimată în instrucțiune. Înainte de salt procesorul salvează pe stivă adresa de revenire în programul apelant (adresa instrucțiunii de după instrucțiunea CALL). Instrucţiunea RET se plasează la sfârşitul rutinei și realizează revenirea în programul apelant. În acest scop se extrage de pe stivă adresa de revenire și se face salt la această adresă. O rutină conține mai multe instrucțiuni RET dacă există mai multe ramificări în secvenţa rutinei.

Formatul instrucțiunilor:

CALL <adresă>
RET [<constantă>]

unde:

<adresa> - este o etichetă (numele rutinei) sau o expresie evaluabilă ca şi o adresă
<constanta> indică numărul de poziţii cu care trebuie să se descarce stiva înainte de revenirea din rutină; în mod uzual acest parametru lipseşte

Exemple:

```
call    rut_adunare
call    1000:100
ret
ret     2       ; descărcarea stivei cu 2 unităţi
```

Funcţie de poziţia rutinei faţă de instrucţiunea de apel compilatorul va genera o adresă "apropiată" (eng. near) care este de fapt adresa de offset a rutinei, sau o adresă "îndepărtată" (eng. far), care conține și adresa de segment. Al doilea caz se aplică atunci când rutina se află în alt segment decât instrucţiunea de apel. programatorul poate cere o adresă "near" sau "far" în mod explicit printr-o directivă de declarare a procedurii.

6.4.8. Instrucţiunile de ramificare şi ciclare

Aceste instrucțiuni modifică secvența de execuție a instrucțiunilor. Există instrucțiuni de salt necondiţionat (care se execută în orice condiţie) și instrucțiuni de salt condiţionat. cele din urmă determină execuţia unui salt în măsura în care o anumită condiţie este îndeplinită. O condiţie poate fi starea unui indicator de condiţie sau o combinație a acestora. Formatul instrucțiunilor este:

JMP <adresă>
J<cc> <adresă>

unde:

<adresă> este o etichetă sau o expresie evaluabilă ca şi o adresă
<cc> este o combinație de litere care sugerează condiţia care se aplică

În primul tabel s-au indicat variantele de salt condiţionat care implică testarea unui singur indicator de condiţie. Următoarele două tabele prezintă instrucțiunile de salt condiţionat utilizate după o operaţie de comparare a două numere.

Instrucţiunea	Condiţia	Instrucţiuni echivalente	Explicaţii
JC	CF=1	JB,JNAE	salt dacă a fost un transport
JNC	CF=0	JNB,JAE	salt dacă nu a fost un transport
JZ	ZF=1	JE	salt dacă rezultatul este zero
JNZ	ZF=0		salt dacă rezultatul nu este zero
JS	SF=1		salt dacă rezultatul este negativ
JNS	SF=0		salt dacă rezultatul este pozitiv
JO	OF=1		salt la depăşire de capacitate

JNO	OF=0		salt dacă nu este depăşire
JP	PF=1		salt dacă rezultatul este par
JNP	PF=0		salt dacă rezultatul nu este par

Instrucţiuni de salt condiţionat utilizate după compararea a două numere fără semn:

Instrucţiune	Condiţie	Indicatori testaţi	Instrucţiuni echivalente	Explicaţii
JA	>	CF=0,ZF=0	JNBE	salt la mai mare
JAE	>=	CF=0	JNB,JNC	salt la mai mare sau egal
JB	<	CF=1	JNAE,JC	salt la mai mic
JBE	<=	CF=1 sau ZF=1	JNA	salt la mai mic sau egal
JE	=	ZF=1	JZ	salt la egal
JNE	!=	ZF=0	JNZ	salt la diferit

Instrucţiuni de salt utilizate după compararea a două numere cu semn (reprezentate în complement faţă de doi).

Instrucţiune	Condiţie	Indicatori testaţi	Instrucţiuni echivalente	Explicaţii
JG	>	SF=OF sau ZF=0	JNLE	salt la mai mare
JGE	>=	SF=OF	JNL	salt la mai mare sau egal
JL	<	SF!=OF	JNGE	salt la mai mic
JLE	<=	SF!=OF sau ZF=1	JNG	salt la mai mic sau egal
JE	=	ZF=1	JZ	salt la egal
JNE	!=	ZF=0	JNZ	salt la diferit

Observaţie: relaţiile de ordine (mai mic, mai mare, etc.) se stabilesc între primul şi al doilea parametru al operaţiei de comparare.

Exemple:

```
cmp    ax, 100h
je     et1             ; salt dacă ax=100h
cmp    var1,al
jb     mai_mic         ; salt dacă var1<al
add    dx,cx
jo     eroare          ; salt dacă apare depăşire de capacitate
```

Instrucţiunile de ciclare LOOP, LOOPZ, LOOPNZ

Aceste instrucţiuni permit implementarea unor structuri de control echivalente cu instrucțiunile "for", "while" "do-until" din limbajele de nivel înalt. Aceste instrucțiuni efectuează o secvență de operații: decrementarea registrului CX folosit pe post de contor, testarea condiţiei de terminare a ciclului și salt la etichetă (la începutul ciclului) în cazul în care condiţia nu este îndeplinită.

Sintaxa instrucțiunilor:

```
LOOP        <adresă>
LOOPZ       <adresă>
LOOPNZ      <adresă>
```

unde: <adresa> este o etichetă sau o expresie evaluabilă la o adresă

Pentru instrucţiunea LOOP condiţia de terminare a ciclului este CX=0, adică contorul ajunge la 0. Pentru instrucţiunea LOOPZ în plus ciclul se încheie şi în cazul în care ZF=0. La instrucţiunea LOOPNZ ieșirea din buclă se face pentru ZF=1.

Exemplu:

```
        mov     cx, 100
et1:    .....
        loop    et1     ; ciclul se execută de 100 de ori
```

6.4.9. Instrucţiuni de intrare/ieşire

Aceste instrucțiuni se utilizează pentru efectuarea transferurilor cu registrele (porturile) interfeţelor de intrare/ieşire. Trebuie remarcat faptul că la procesoarele Intel acestea sunt singurele instrucțiuni care operează cu porturi.

Instrucţiunile IN şi OUT

Instrucţiunea IN se foloseşte pentru citirea unui port de intrare, iar instrucțiunea OUT pentru scrierea unui port de ieșire. Sintaxa instrucțiunilor este:

```
IN      <acumulator>, <adresă_port>
OUT     <adresă_port>, <acumulator>
```

unde:

<acumulator> - registrul AX pentru transfer pe 16 biţi sau AL pentru transfer pe 8 biţi
<adresa_port> - o adresă exprimabilă pe 8 biţi sau registrul DX

Se observă că dacă adresa portului este mai mare decât 255 atunci adresa portului se transmite prin registrul DX.

Exemple:

```
în      al, 20h
mov dx, adresa_port
out     dx, ax
```

6.4.10. Instrucţiuni pe şiruri

Aceste instrucțiuni s-au introdus cu scopul de a accelera accesul la elementele unei structuri de tip şir sau vector. Instrucţiunile folosesc în mod implicit registrele index SI şi DI pentru adresarea elementelor şirului sursă şi respectiv destinaţie. În mod implicit registrul DS păstrează adresa de segment a sursei iar registrul ES adresa de segment a destinaţiei. După efectuarea operaţiei propriu-zise (specificată prin mnemonica instrucțiunii), registrele index sunt incrementate sau decrementate automat pentru a trece la elementele următoare din şir. Indicatorul DF determină direcţia de parcurgere a şirurilor: DF=1 prin incrementare, DF=0 prin decrementare. Registrul CX este folosit pentru contorizarea numărului de operații efectuate. După fiecare execuție registrul CX se decrementează.

Instrucţiunile MOVSB, MOVSW

Aceste instrucțiuni transferă un element din şirul sursă în şirul destinaţie. Instrucţiunea MOVSB operează pe octet (eng. move string on byte), iar MOVSW operează pe cuvânt. La operațiile pe cuvânt

registrele index se incrementează sau se decrementează cu 2 unităţi, deoarece un cuvânt ocupă 2 locaţii în memorie. Instrucţiunile nu au parametrii; programatorul trebuie să încarce în prealabil adresele şirurilor în registrele SI și DI, şi lungimea şirului în CX.

Exemplu:

```
        mov     și, offset sir_sursa            ; "offset" este un operator care determină adresa
        mov     di, offset sir_destinatie       ; de offset a variabilei
        mov     cx, lung_sir
et:     MOVSB                                   ; DS:[SI]=>ES:[DI], SI++, DI++, CX--
        jnz     et
```

Instrucţiunile LODSB, LODSW, STOSB și STOSW

Primele două instrucțiuni realizează încărcarea succesivă a elementelor unui şir în registrul acumulator. Următoarele două instrucțiuni realizează operaţia inversă de salvare a registrului acumulator într-un şir. Şi la aceste instrucțiuni registrele index (SI pentru încărcare şi DI pentru salvare) se incrementează sau se decrementează automat, iar registrul CX se decrementează. Terminaţiile "B" respectiv "W" indică lungimea pe care se face transferul: octet sau cuvânt.

Instrucţiunile CMPSB, CMPSW, SCASB și SCASW

Aceste instrucțiuni realizează operații de comparare cu elemente ale unui şir. Primele două instrucţiuni compară între ele elementele a două şiruri, iar ultimele două compară conținutul registrului acumulator cu câte un element al şirului (operaţie de scanare).

Instrucţiunile REP, REPZ, REPE, REPNZ, REPNZ

Aceste instrucţiuni permit execuţia multiplă a unei instrucțiuni pe şiruri. Prin amplasarea unei astfel de instrucțiuni în faţa unei instrucțiuni pe şiruri obligă procesorul execuția repetată a operaţiei până ce condiţia de terminare este satisfăcută. La prima variantă, REP, condiţia de terminare este CX=0. La instrucțiunile REPZ şi REPE operaţia se repetă atâta timp cât rezultatul este zero sau operanzii sunt egali. La REPNZ și REPNE operaţia se repetă atâta timp cât rezultatul este diferit de zero sau operanzii sunt diferiţi.

Exemple:

```
        mov     și, offset sir_sursa
        mov     di, offset sir_destinatie
        mov     cx, lungime_sir
        rep     movsb                   ; transferă şirul sursă în şirul destinaţie
```

6.4.11. Instrucţiuni speciale

În această categorie s-au inclus acele instrucțiuni care au efect asupra modului de funcţionare al procesorului.

Instrucţiunile CLC, STC, CMC

Aceste instrucțiuni modifică starea indicatorului CF de transport. Aceste instrucţiuni au următoarele efecte:

CLC şterge (eng. clear) indicatorul, CF=0
STC setează indicatorul, CF=1
CMC inversează starea indicatorului, CF=NOT CF

Instrucţiunile CLI și STI

Aceste instrucţiuni şterg şi respectiv setează indicatorul de întrerupere IF. În starea setată (IF=1) procesorul detectează întreruperile mascabile, iar în starea inversă blochează toate întreruperile mascabile.

Instrucţiunile CLD şi STD

Aceste instrucţiuni modifica starea indicatorului de direcţie DF. Prin acest indicator se controlează modul de parcurgere a şirurilor la operaţiile pe şiruri: prin incrementare (DF=0) sau prin decrementare (DF=1).

Instrucţiunile NOP, HLT şi HIT

Instrucţiunea NOP nu face nimic (eng. no operation). Această instrucţiune se foloseşte în scopuri de temporizare, pentru întârzierea unor operaţii sau pentru implementarea unor bucle de aşteptare. Instrucţiunea HLT (eng. halt) determină oprirea procesorului. Instrucţiunea HIT determină oprirea temporară a procesorului până la apariţia unui semnal de întrerupere sau de iniţializare (reset).

Instrucţiunea CPUID

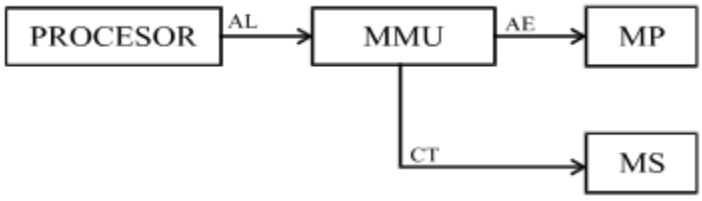
Această instrucţiune permite identificarea tipului de procesor pe care rupeaza programul.

7. Nivelul sistemului de exploatare

7.1. Memoria virtuală, conceptual de paginare , conceptul de segmentar

Memoria virtuală reprezintă o tehnică de organizare a memoriei prin intermediul căreia programatorul "vede" un spaţiu virtual de adresare foarte mare şi care, fără ca programatorul să "simtă", este mapat în memoria fizic disponibilă. Uzual, spaţiul virtual de adrese corespunde suportului disc magnetic, programatorul având iluzia prin mecanismele de memorie virtuală (MV), că deţine o memorie (virtuală) de capacitatea hard-discului și nu de capacitatea memoriei fizice preponderentă DRAM (limitată la 64 ÷1024 Mo la ora actuală). În cazul MV, memoria principală este analoagă memoriei cache între CPU (*Central Processing Unit*) și memoria principală, numai că de această dată ea se situează între CPU și discul hard. Deci memoria principală (MP) se comportă oarecum ca un cache între CPU și discul hard. Prin mecanismele de MV se măreşte probabilitatea ca informaţia ce se doreşte a fi accesată de către CPU din spaţiul virtual (disc).

Pentru implementarea conceptului de memorie virtuală se foloseşte următoarea schemă de principiu:



AL reprezintă adresa logică care este generată de CPU prin citirea instrucţiunii curente. Prin prelucrarea adresei logice în MMU rezultă adresa fizică . MMU realizează o alta funcție de translatare

$$f_{T^*}: AL \rightarrow AE.$$

f_T realizează translatarea între adresele logice (virtuale) şi cele fizice (reale). Dacă adresa logică nu are corespondent fizic în memoria principală , atunci se generează comenzile de transfer (CT) pentru transferul între memoria externă şi memoria principală.

Realizarea practică a conceptului de memorie virtuală se bazează pe transferul în blocuri între memoria externă şi memoria principală . Cea mai răspândită metodă de implementare este metoda segmentării memoriei. S-a arătat că fiecare instrucţiune la nivel cod mașină este formată din 2 câmpuri mari:

OPCODE	ADRESĂ
n	l
	L

În câmpul de adresă apar elementele constiutive ale adresei logice din care, prin efectuarea unor calcule, se determină adresa efectivă sau adresa fizică.
În cazul adresei efective:

$$AE=f(AL)$$

Unde

- *AL:* adresa logică;
- *AE*: adresa efectivă.

Pe parcursul timpului vitezele de transfer au evoluat astfel:

- Interfața IDE (pe cablul paralel de legătură între placa de bază și HDD) asigura 133MB/sec.
- Prima interfață SATA serială a crescut viteza de transfer la 150MB/sec.
- Evoluția la SATA2 a realizat dublarea acestei performanțe la 300MB/sec.
- Interfața SATA3 a dus la o la viteză de 540 - 560MB/sec
- SSD -urile sunt compatibile atât cu SATA2 cât și cu SATA3 iar viteza de transfer a crescut de peste 4 ori.

De obicei, spațiul virtual de adresare este împărţit în entităţi de capacitate fixă (4 ÷ 64 Ko actualmente), numite pagini. O pagină poate fi mapată în MP sau pe disc.

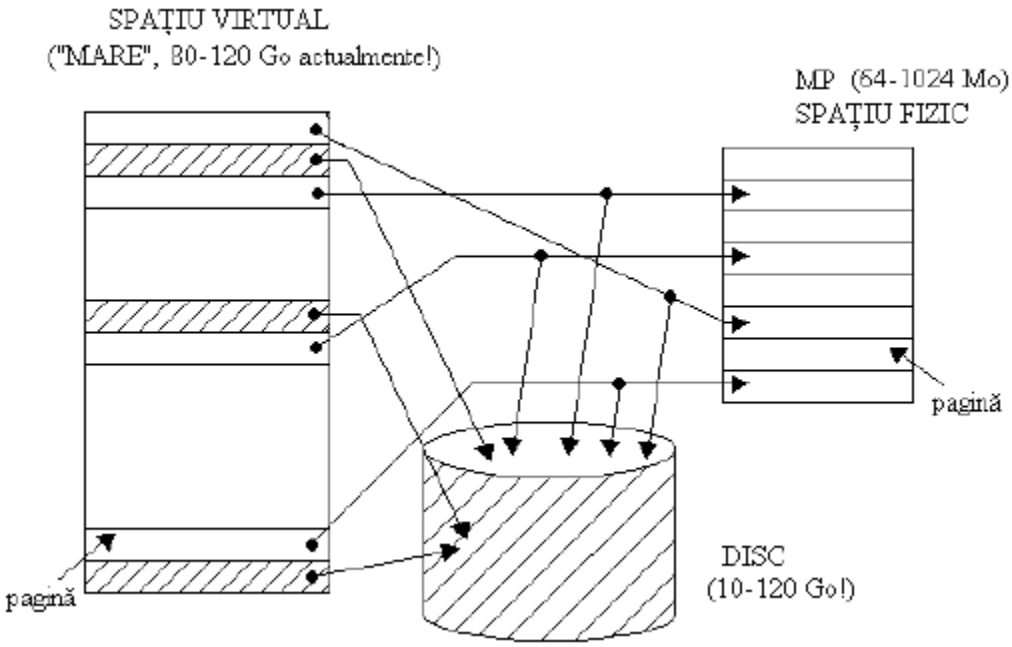


Figura 4.8. Maparea adreselor virtuale în adrese fizice

7.1.1. Paginarea

În general, prin mecanismele de MV, MP conține paginile cel mai recent accesate de către un program, ea fiind după cum am mai arătat, pe post de "cache" între CPU și discul hard. Transformarea adresei virtuale emisă de către CPU întro adresă fizică (existentă în spațiul MP) se numește mapare sau translatare. Așadar mecanismul de MV oferă o funcție de relocare a programelor (adreselor de program), pentru că adresele virtuale utilizate de un program sunt relocate spre adrese fizice diferite, înainte ca ele să fie folosite pentru accesarea memoriei. Această mapare permite aceluiași program să fie încărcat și să ruleze oriunde ar fi încărcat în MP, modificările de adrese realizându-se automat prin mapare (fără MV un program depinde de obicei în execuția sa de adresa de memorie unde este încărcat).

MV este un concept deosebit de util în special în cadrul sistemelor de calcul multiprogramate care - de exemplu prin "*time-sharing*" – permit execuția cvasi-simultană a mai multor programe (vezi sistemul de operare WINDOWS XX, NT, Unix, Ultrix etc.). Fiecare dintre aceste programe are propriul său spațiu virtual de cod și date (având alocate un număr de pagini virtuale), dar în realitate toate aceste programe vor partaja aceeași MP, care va conține dinamic, pagini aferente diverselor procese.

În implementarea MV trebuie avute în vedere următoarele aspecte importante:

- paginile să fie suficient de mari (4 ko ÷16 ko … 64 ko) astfel încât să compenseze timpul mare de acces la disc (9 ÷12 ms).
- organizări care să reducă rata de evenimente PF, rezultând un plasament flexibil al paginilor în memorie (MP)
- PF-urile trebuie tratate prin software şi nu prin hardware (spre deosebire de miss-urile în cache-uri), timpul de acces al discurilor permitând lejer acest lucru.
- scrierile în MV se fac după algoritmi tip "Write Back" și nu "Write Through" (ar consuma timp enorm).

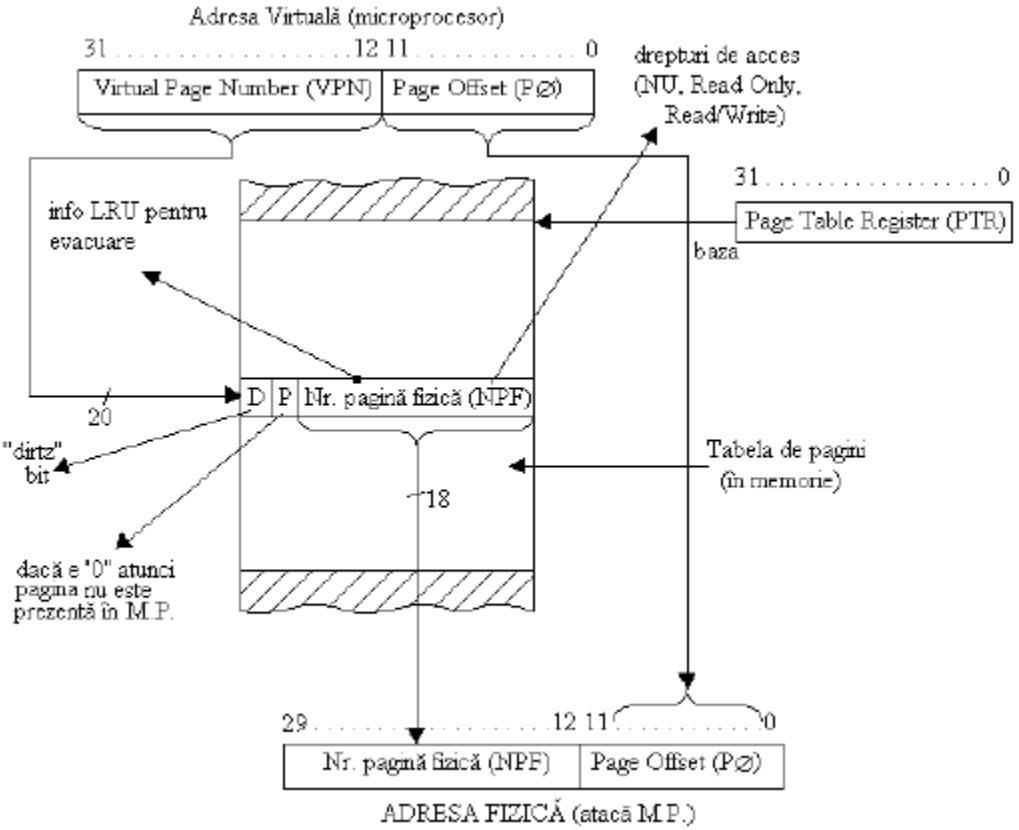


Figura 4.9. Translatare adresă virtuală în adresă fizică

Fiecare program are propria sa tabelă de pagini care mapează spaţiul virtual de adresare al programului într-un spaţiu fizic, situat în M.P.
Tabela de pagini + PC + registrele microprocesorului formează starea unui anumit program. Programul + starea asociată caracterizează un anumit proces (task). Un proces executat curent este activ, altfel el este inactiv. Comutarea de taskuri implică inactivarea procesului curent şi activarea altui proces, inactiv până acum rezultând deci ca fiind necesară salvarea/restaurarea stării proceselor. Desigur, sistemul de operare (S.∅.) trebuie doar să reâncarce registrul pointer al adresei de bază a paginii (PTR) pentru a pointa la tabela de pagini aferentă noului proces activ.

7.1.2. Segmentarea

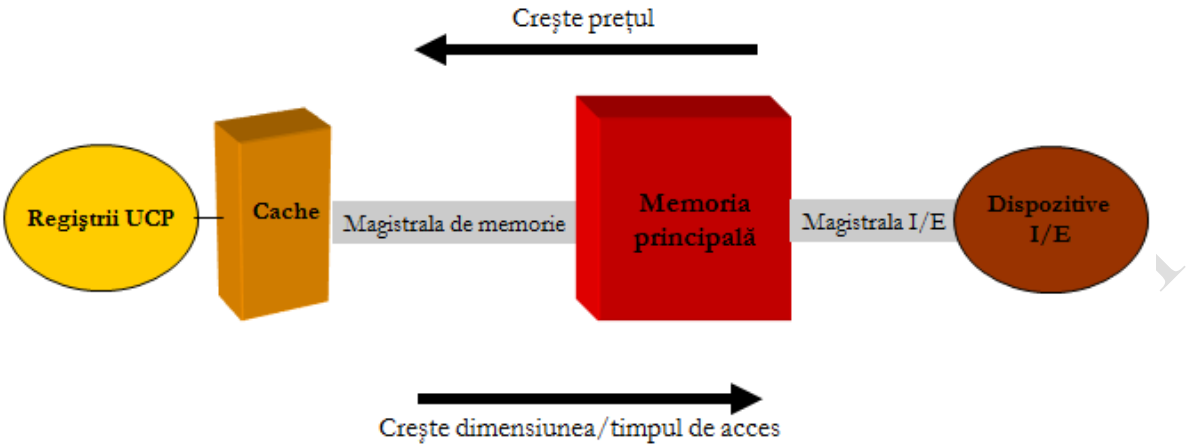
Constituie o altă variantă de implementare a MV, care utilizează în locul paginilor de lungime fixă, entităţi de lungime variabilă zise segmente. În segmentare, adresa virtuală este constituită din 2 cuvinte: o bază a segmentului și respectiv un deplasament (offset) în cadrul segmentului.
Datorită lungimii variabile a segmentului (de ex. 1 octet ¸ 2^{32} octeți la arhitecturile Intel Pentium), trebuie făcută și o verificare a faptului că adresa virtuală rezultată (baza + offset) se încadrează în lungimea adoptată a segmentului. Desigur, segmentarea oferă posibilităţi de protecţie puternice și sofisticate a segmentelor. Pe de altă parte, segmentarea induce şi
numeroase dezavantaje precum:
2 cuvinte pentru o adresă virtuală, necesare având în vedere lungimea variabilă a segmentului. Asta complică sarcina compilatoarelor și a programelor

- încărcarea segmentelor variabile în memorie mai dificilă decât la paginare
- fragmentare a memoriei principale (porţiuni nefolosite)
- frecvent, trafic ineficient MP-disc (de exemplu pentru segmente “mici”, transferul cu discul e complet ineficient – accese la nivel de sector = 512 octeți)

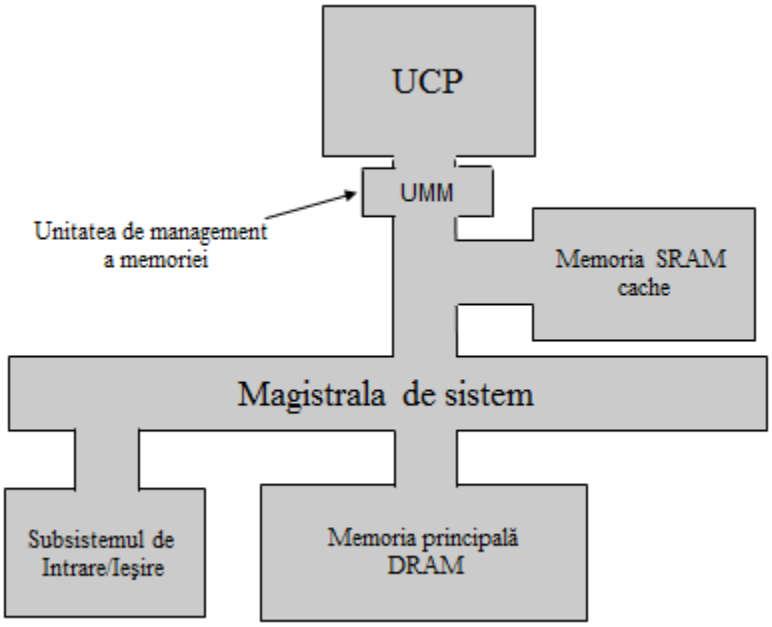
Există în practică și implementări hibride segmentare – paginare

7.2. Exemple de gestionare a memoriei virtuale.

O schema a ierharhiei memoriei este prezentată alăturat.

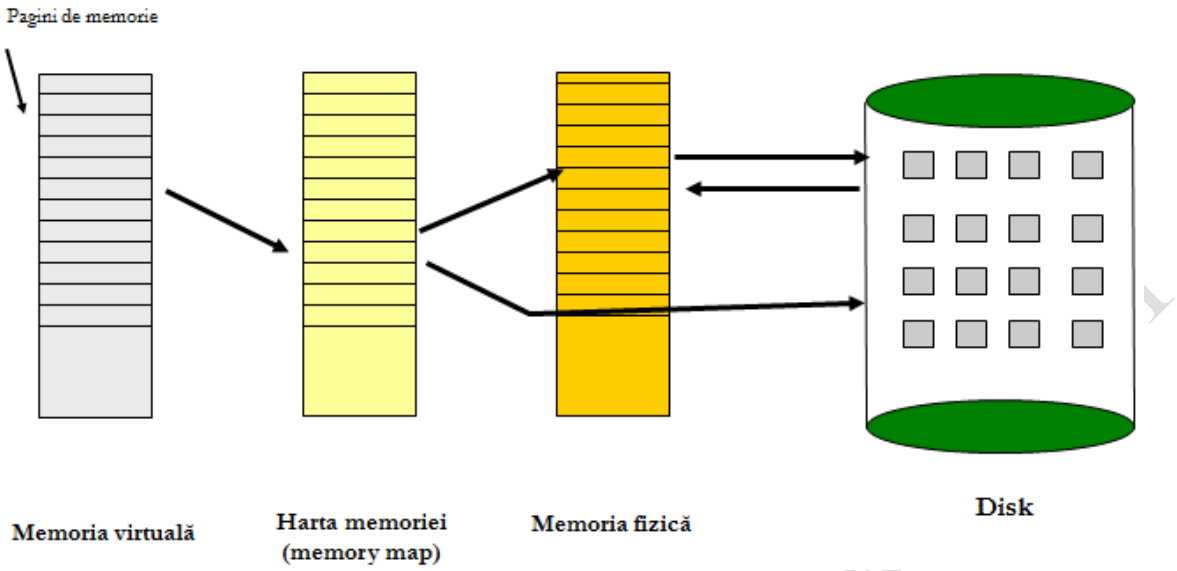


Unitatea de management a memoriei virtuale



7.2.1. Memoria virtuală

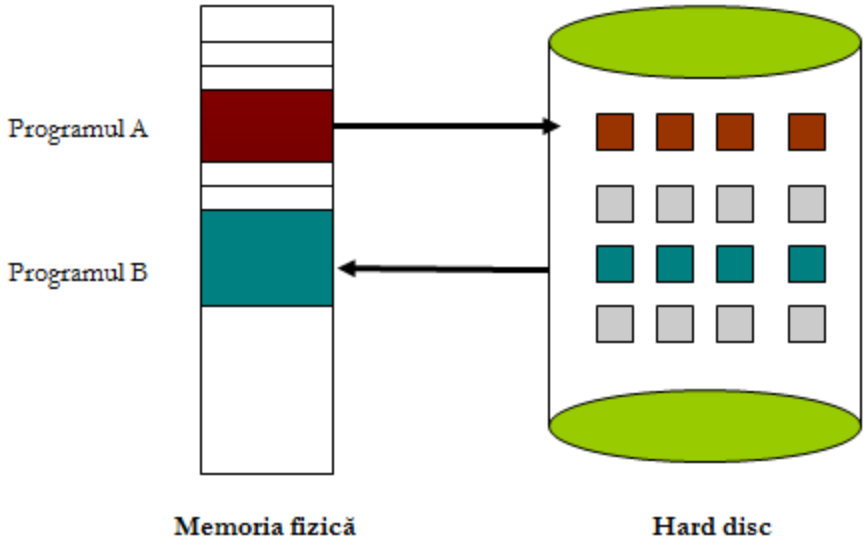
Reprezintă separarea conceptuală a memoriei logice disponibile pentru aplicatii fata de memoria fizica. În acest mod putem avea o memorie virtuala de dimensiuni mari chiar cu o memorie fizica de dimensiuni reduse :



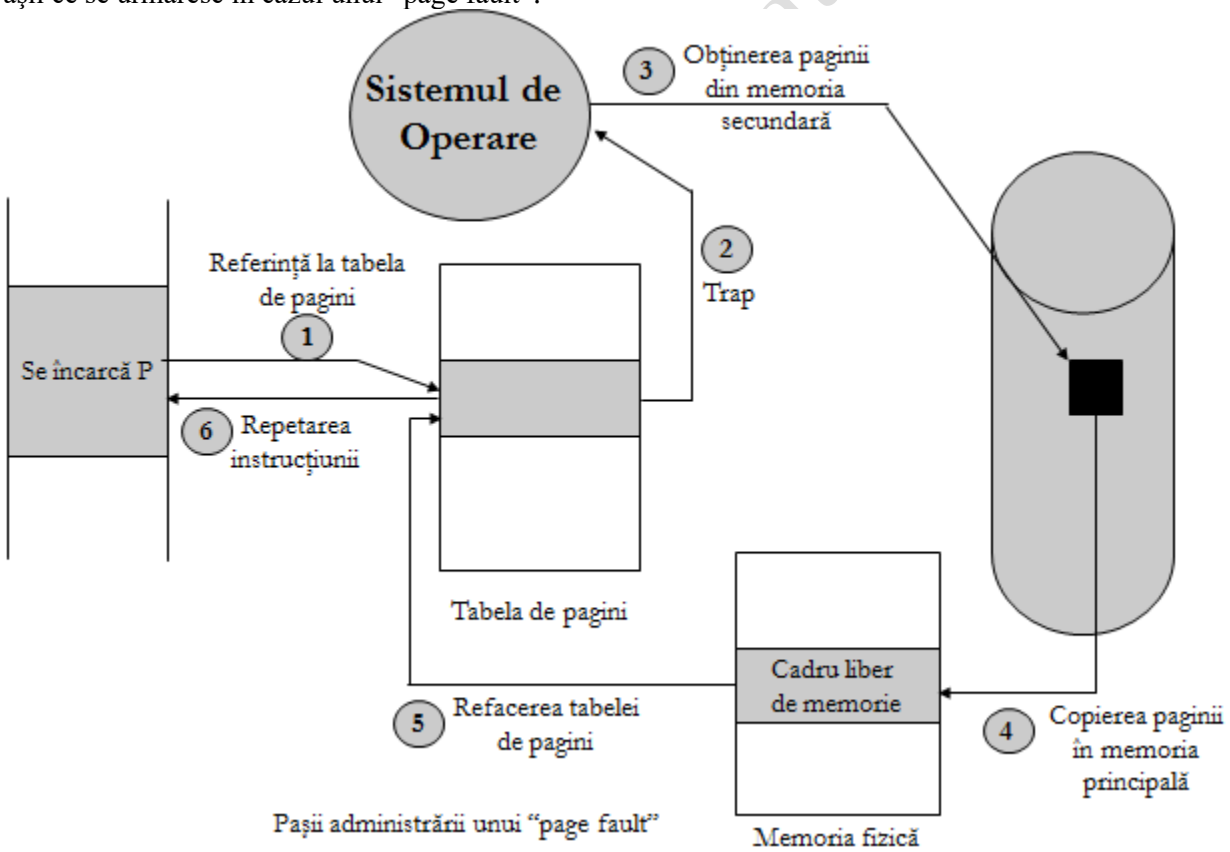
- În acelaşi sens, nu toate obiectele (date sau instrucțiuni) pot fi la un moment dat în memoria principală. Dacă avem memorie virtuală, atunci unele dintre obiecte se pot afla pe disc. Spaţiul de adresare este de regulă împărţit în blocuri de lungime fixă – pagini.
- La un moment dat, paginile se află fie în memoria principală, fie pe disc
- Atunci când se cere un obiect care nu este în cache sau în memoria principală, apare un "page-fault" – moment în care întreaga pagină este mutată de pe disc în memoria principală. Aceste "page-fault" durează mai mult şi atunci sunt controlate de software şi UCP nu face pauză.
- De regulă, UCP comută către alt task atunci când apare un acces la disc. Memoria cache şi memoria principală au aceeaşi relaţie ca și cea existentă între memoria principală și disc.
- În orice moment, un calculator rulează mai multe procese, fiecare având propriul spaţiu de adrese de memorie. Ar fi foarte costisitor să se dedice un întreg spaţiu de adresare pentru fiecare proces, având în vedere că multe dintre procese folosesc doar o mică parte a spaţiului propriu de adrese. A apărut astfel necesitatea partajării unei părţi a memoriei între mai multe procese.
- Acest procedeu poartă numele de "memorie virtuală" – memoria fizică se divide în blocuri care sunt alocate diferitelor procese.
- Inerentă unei astfel de abordări este o schemă de protecţie ce restricţionează accesul proceselor la blocuri ce aparţin altor procese. Majoritatea formelor de memorie virtuală reduc, de asemenea, timpul de pornire a unui program, deoarece nu tot codul sau datele trebuie să fie deja în memoria fizică înainte ca programul să înceapă.
- Însă partajarea între procese a memoriei nu este adevăratul motiv pentru care s-a inventat memoria virtuală. Dacă un program devine prea mare pentru memoria fizică, este sarcina programatorului să îl facă să încapă în ea. Au rezultate acele suprapuneri (overlay).
- Blocurile de memorie în cazul memoriei virtuale se numesc *pagini* sau *segmente*. UCP foloseşte adrese virtuale ce sunt translatate (hardware cât și software) în adrese fizice ce accesează memoria principală. Acest procedeu se numește procedeul de mapare a memoriei sau de translatare a adreselor. Astăzi memoria virtuală intervine la nivel de memorie principală și disc magnetic

Cererea de pagini (*demand paging*)

Atunci când o pagină de memorie este referită (fie că este vorba despre cod sau date) şi ea nu se află în memorie atunci ea este adusă de pe disc şi se re-execută instrucţiunea



Paşii ce se urmăresc în cazul unui “page fault”:



Paşii administrării unui “page fault”

În cazul supra-alocării memoriei trebuie să renunţăm la ceva deja existent în memorie.
Supra-alocarea apare atunci când programele au nevoie de mai multe pagini de memorie decât cele existente fizic.
Metoda de abordare: dacă nici o pagină fizică nu este liberă, se caută una care nu este utilizată la momentul respectiv şi se eliberează.

8. Arhitecturi evoluate

Arhitectura Setului de Instrucţiuni
Arhitectura setului de instrucțiuni (ISA - Instruction Set Architecture) este folosită pentru a abstractiza funcţionarea internă a unui procesor. ISA defineşte "personalitatea" unui procesor: cum funcţionează procesorul d.p.d.v. al programatorului, ce fel de instrucțiuni execută, care este semantica acestora, în timp ce microarhitectura se referă la cum este implementată ISA, pipeline-ul de instrucțiuni, fluxul de date dintre unităţile de execuţie (dacă sunt mai multe), registre şi cache. Astfel, o ISA poate fi implementată de diferite microarhitecturi: la ARM, ARMv6 este un exemplu de ISA, şi este implementată de 4 procesoare, la Pentium numele ISA este IA-32 și este implementată de diferite procesoare produse de Intel, AMD, Via, Transmeta.
Astfel, ISA este cea mai importantă parte a design-ului unui procesor; alte aspecte cum sunt interacţiunea cu cache-ul, pipeline-ul, fluxul de date în procesor putând fi schimbate de la o versiune la alta a procesorului.
La ora actuală există două filozofii de design pentru un procesor:

- Complex Instruction Set Computer (**CISC**)
- Reduced Instruction Set Computer (**RISC**).

În afară de acestea există și ISA-uri pentru procesoare specializate, cum sunt GPU-urile pentru plăci grafice și DSP-urile pentru procesare de semnal.

8.1. Masinile RISC (evolutia arhitecturii sistemelor de calcul, principii de proiectare a masinilor RISC),

8.1.1. RISC (Reduced Instruction Set Computer) au următorele caracteristicile:

- set redus de instrucțiuni: ~100-200; complexitatea care este eliminată din ISA este de fapt transferată programatorului în limbaj de asamblare şi compilatoarelor; codul conține mai multe instrucţiuni, şi în multe cazuri (mai ales în trecut (anii '70, '80)) acest fapt constituie o problemă
- instrucţiunile sunt de obicei codificate pe lungime fixă; permite o implementare mai simplă a UC
- execuţia instrucțiunilor într-un singur ciclu (folosind pipeline)
- **arhitectură load-store** - numai instrucţiunile de tip LOAD și STORE lucrează cu memoria
- un număr mai mare de registre (din cauza load-store): 32-64; familii de arhitecturi: ARM, AVR, AVR32, MIPS, PowerPC (Freescale, IBM), SPARC (SUN)

Exemplu: încărcarea unui cuvânt de date din memorie într-un registru la arhitectura AVR32:

```
ld.w Rd, Rp++
```

`Rd` - registrul destinație
`Rp` - registrul ce conține adresa cuvântului; aceasta este incrementată după copierea conținutului.

8.1.2. CISC (Complex Instruction Set Computer)

Arhitecturile de tip CISC pun accentul pe hardware având instrucțiuni complexe ce reduc dimensiunea codului și simplificând partea de software. Cele mai importante caracteristici sunt:

- instrucțiunile pot accesa memoria, având deja încorporate load-ul şi store-ul
- varietate mare de moduri de adresare a memoriei

- instrucțiunile au dimensiuni variabile și necesită mai mulți cicli de ceas pentru a fi executate
- necesită un nivel suplimentar între hardware şi ISA, controlul microprogramului - instrucțiunile sunt implementate în microcod;
- familii de arhitecturi: IA-32, x86-64

Exemplu:
La IA-32:

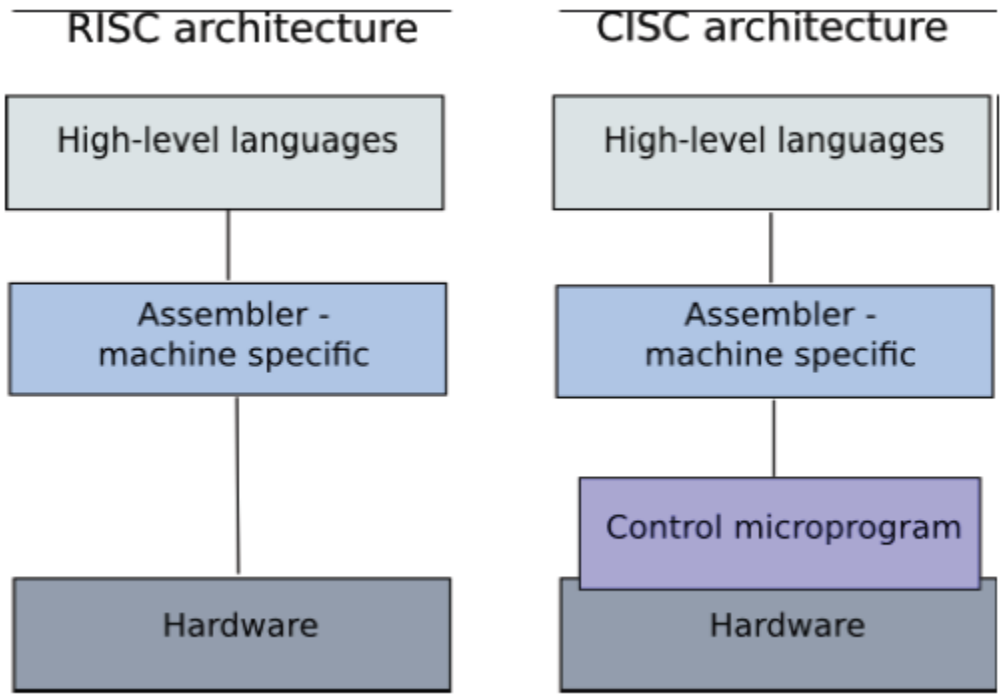
```
add [ebx],eax
```

(se adună operandul de la adresa data de registrul ebx cu cel din registrul eax și rezultatul se stocheaza la adr primului operand)

La o arhitectura RISC:

```
load r1, $10
load r2, $11
add r1,r2
store $10, r1
```

`$10, $11` sunt locații de memorie (adrese), numele instrucțiunilor sunt generice, nu aparțin unei arhitecturi anume
Pentru o arhitectura RISC am avea intai de incarcat operanzii din memorie folosind instrucțiuni de tip LOAD și apoi să apelam instrucțiunea pentru inmultire, în final stocand rezultatul în memorie folosind o instrucțiune de tip STORE.



8.1.3. RISC vs CISC

Regula 80-20: 80% din timp se folosesc 20% din instrucțiunile unui procesor. Multe instrucțiuni CISC sunt nefolosite de către programatori, iar majoritatea instrucțiunilor complexe pot fi sparte în mai multe instrucțiuni simple. Ambele abordări fac compromisuri pentru a minimiza una dintre aceste fracții: în cazul

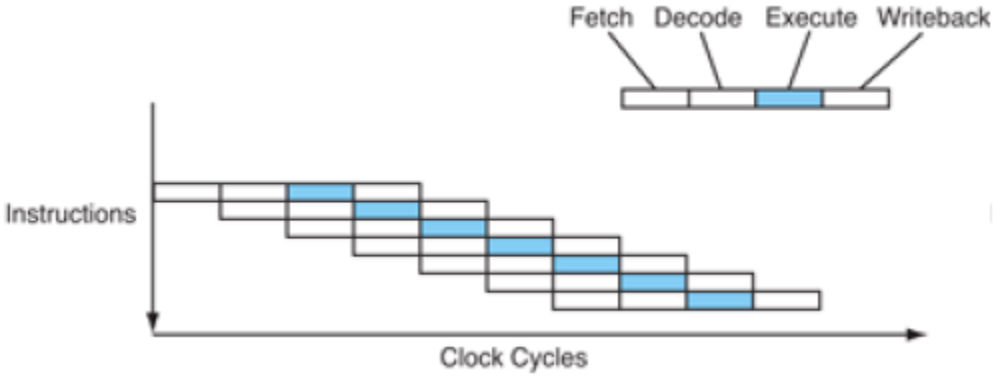
RISC, se preferă un număr mai mare de instrucțiuni pe program pentru a micşora numărul de cicli pe instrucțiune, în timp ce la CISC este invers.
Totuşi, lucrurile nu stau atât de simplu. Deşi arhitecturile RISC au fost devansate în anii 80 de către CISC datorită lipsei software-ului şi al unui segment de piaţă, precum şi datorită preţului ridicat al memoriei, o dată cu progresele tehnologice, procesoarele cu ISA RISC au monopolizat domeniul embedded. În plus, cele două arhitecturi au început să se apropie, majoritatea procesoarelor actuale având o arhitectura hibridă, de exemplu Intel Pentium având un interpretor CISC peste un nucleu RISC.

RISC	CISC
costuri mici de fabricatie, mai puține tranzistoare	hardware mai complex, costuri mai mari
consum redus de energie	consum mai mare
mai multe registre	mai mult hardware, controlul microprogramului
mai multe linii de cod per program	puţin cod

8.2. Arhitecturi paralele, exemple.

8.2.1. Paralelism - Pipelining

Chiar dacă un program este în mod tradițional interpretat secvenţial, instrucțiune cu instrucţiune, o parte dintre acestea nu sunt dependente între ele și pot fi executate în paralel. Această metodă de a extrage paralelism la nivelul unui singur flux de instrucțiuni de numește paralelism la nivel de instrucțiunie sau Instruction-level parallelism (ILP).
O metodă de a implementa ILP este banda de asamblare (**pipeline**). Banda de asamblare presupune că atunci când o instrucțiune *i* este executată, instrucțiunea *i+1* este decodificată și instrucțiunea *i+2* este citită, astfel fiecare subsistem din procesor este ocupat la un moment dat.

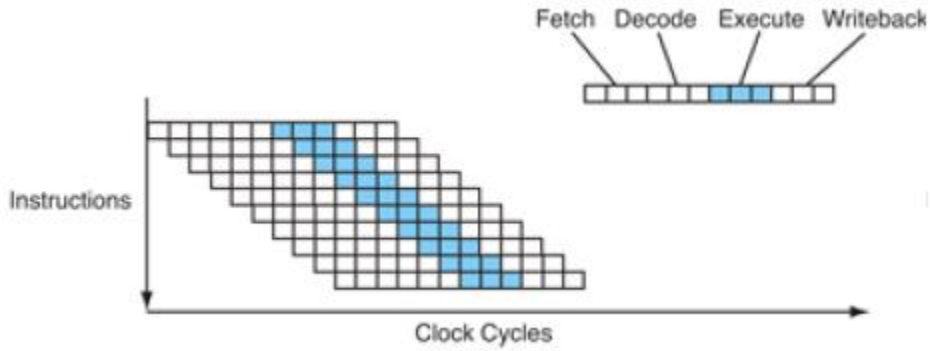


Modelul de bază pentru un pipeline la o arhitectura RISC este cel din 4 etape:

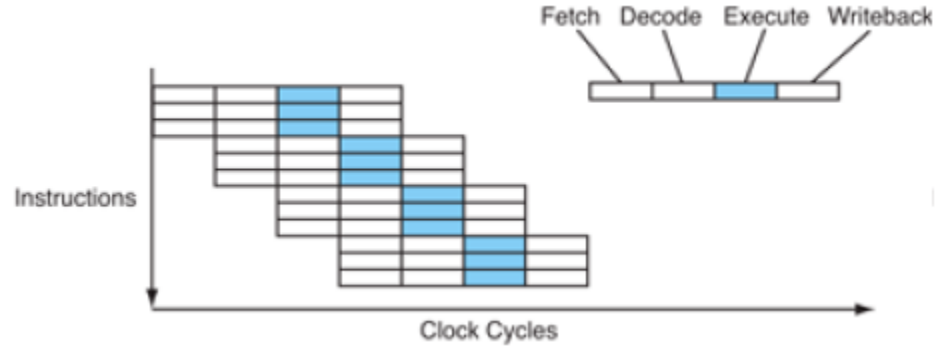
1. Instruction Fetch,
2. Instruction Decode,
3. Execute,
4. Register Write Back.

- În stagiul **Fetch** se aduce instrucţiunea din memorie, de la adresa dată de contorul program (PC - *program counter*).
- În stagiul **Decode** se decodifică instrucțiunea
- în **Execute** se execută efectiv operaţia de către unitatea aritmetico-logica,
- în **Writeback** se stochează rezultatele în regiştrii sau memorie.

Pentru CISC, avem de obicei un număr mai mare de etape (>12).
Banda de asamblare ***superpipeline*** este mai lungă decât banda obişnuită (cele 4 stagii F, D, E, W sunt împărţite la rândul lor în mai multe stagii).



Teoretic cu cât banda de asamblare este mai segmentată, mai multe instrucțiuni pot fi executate în paralel şi, fiecare stagiu făcând mai puține operații, se poate mări frecvența ceasului. Însă, alți factori caracteristici ai design-ului pot scădea performanţele şi nu întotdeauna un procesor cu superpipelining este mai bun decât unul cu bandă de asamblare normală. Pentru o arhitectură și un set de instrucțiuni date există un număr optim de etape ale benzii de asamblare. Peste acest număr se reduce performanţa.
Pentru procesoarele RISC, unde instrucţiunile se execută într-un singur ceas, această durată de execuție este dată de instrucţiunea cea mai lungă.
Banda de asamblare mărește productivitatea (throughput-ul), adică numărul de instrucțiuni executate în unitatea de timp, însa timpul de execuţie (latența) a unei instrucțiuni nu se micşorează, ci se poate chiar mări.
Pentru a paraleliza și mai mult execuția instrucțiunilor, **procesoarele *superscalare*** introduc benzi de asamblare independente ce funcţionează în paralel. Se măreşte astfel numărul de resurse şi complexitatea procesorului, însă mai multe instrucțiuni pot fi aduse, decodificate ţi executate în acelaţi timp, după cum este ilustrat.



8.2.2. Hazard

În cazul unui procesor cu bandă de asamblare, pentru a obține performanţă maximă, este necesar ca o instrucțiune să fie prezentă în fiecare stagiu al pipeline-ului. Acest lucru nu este posibil însă datorită faptului că instrucțiunile pot avea diferite dependențe între ele ceea ce face ca execuția lor în pipeline să ducă la rezultate nedorite, situaţie care poartă numele de hazard.

8.2.2.1. Tipuri

Există trei tipuri de dependenţe care pot conduce la hazarde:

1. **Structurale**: când două instrucțiuni aflate în execuție în pipeline doresc acces la aceeaşi structură hardware în acelaşi timp.

Exemplu:

```
add r1, r2
load r3, [r4 + offset]
```

- ambele instrucţiuni necesită acces la unitatea de adunare: prima pentru a executa o adunare, iar a două pentru a calcula adresa de la care se face citirea din memorie.

2. **De date**: când două instrucțiuni folosesc acelaşi registru sau aceeaşi locaţie de memorie și cel puţin una dintre ele este o scriere

• **RAW (Read after Write):** instrucțiunea *Ii+1* citeşte o locaţie de memorie scrisă de instrucțiunea I_i

Exemplu:

```
add r1, r2
add r3, r1
```

- valoarea lui r1 este disponibilă în a două instrucțiune abia după ce prima instrucțiune a scris rezultatul

• **WAR (Write after Read)**: instrucțiunea *Ii+1* scrie o locaţie de memorie care este citită de instrucţiunea I_i

Exemplu:

```
add r2, r1
add r1, r3
```

- valoarea lui r1 nu poate fi modificată în a două instrucțiune până când prima instrucțiune nu a terminat citirea lui

• **WAW (Write after Write):** instrucțiunea *Ii+1* scrie o locaţie de memorie care este scrisă şi de instrucţiunea I_i

Exemplu:

```
mov r1, r2
mov r1, r3
```

- este necesar ca cele două scrieri în r1 să se execute în ordinea din program pentru a nu modifica valoarea văzută de instrucțiunile următoare

3. **De control**: când o instrucțiune de salt (branch, jump, call, ret) este executată; instrucțiunile de salt pot modifica PC (contorul program), sărind la o altă instrucțiune decât cea imediat următoare; în acest caz adresa următoarei instrucţiuni care trebuie citită nu se cunoaşte până când instrucțiunea de salt nu este executată

Exemplu:

```
if (a == b) (i1)
      d = a+b (i2)
      c = 1 (i3)
else
      d = a * b (i4)
```

În banda de asamblare se aduc din memorie la rand instrucțiunile i1,i2,i3, insa dacă în urma executiei instrucțiunii *if* (tradusa în assembler printr-o instrucțiune de comparare și apoi una de salt condiționat (jmp - jump) se sare la ramura de else apare un hazard de control. Instrucțiunile i2,i3 nu mai trebuiesc executate, și în schimb trebuie adusă instrucțiunea i4, deci se face 'flush'la pipeline și se aduce instrucțiune i4, pierzandu-se astfel din eficiența prelucrării în paralel a instrucțiunilor.

8.2.2.2. Soluții

O soluție simplă ar fi adăugarea de către programator sau compilator sau automat, de către UC a procesorului, a unor instrucțiuni nop pentru întârzierea execuţiei instrucțiunilor cu dependinţe; aşa numitele *pipeline bubbles* sau *pipeline stalls.*

Forwarding - dacă unul sau mai mulți operanzi ai unei instrucțiuni depind de o instrucțiune aflata înainte în pipeline, rezultatul acesteia poate fi forwardat catre stagiile anterioare. Aceasta soluție necesită hardware suplimentar pentru logica de control.

Exemplu:

```
instr k: add r1,r2
instr k+1: add r1,5
```

Rezultatul obtinut pentru instrucțiunea k în unitatea aritmetico-logică, în stagiul de Execute este forwardat stagiului Decode al instrucțiunii k+1.

Out-of-order execution(OOE) este o tehnica de planificare dinamica(dynamic scheduling) prin care instrucțiunile sunt executate în altă ordine decât cea a programului, însă rezultatele sunt ordonate ca și cum ar fi fost executate normal. Prin această rearanjare, se evită adăugarea de întârzieri, intrând în pipeline instrucțiuni fără dependențe de date între ele. Aceasta soluție necesită hardware suplimentar(registre suplimentare, buffers pentru cozi de așteptare etc), iar algoritmul cel mai cunoscut este cel al lui Tomasulo .

Deoarece registrele generale sunt limitate, iar codul unui program îi refolosește des, generând hazarde de date, putem folosi tehnica de ***register renaming.*** Aceasta minimizează apariția hazardelor, prin adaugarea unor registre suplimentare, ce nu sunt vizibile programatorului. Redenumirea registrelor rezolvă hazardele de date WAR, WAW însa nu și RAW, fiind folosită și în algoritmul Tomasulo de planificare OOE.

Exemplu:

```
instr k add r1,r2,r3
instr k+1 mul r3,r2,r1
instr k+2 sub r1,r2,r4 //hazard WAW intre instr k și k+2
```

După redenumirea registrelor:

```
add l1, l2, l3
mul l5, l2, l4
add l6, l2, l7
```

În cazul hazardelor de control, putem amâna aducerea instrucțiunilor din branch prin adăugarea de întarzieri în pipeline sau intercalarea altor instrucțiuni independente de acel branch, însă soluția cea mai bună este predicția ramificatiilor (**branch prediction**) care minimieaza numărul de cicli pierduți. Această predicție poate fi statică, facută de către compilator prin optimizari de tipul *loop unrolling* (de exemplu în bucla *for,* a cărei condiție este neîndeplinită doar o dată pentru a se ieși din buclă) sau profile-driven(se creează profile ale programelor în urma mai multor execuții).
Predicția dinamică se face de catre procesor prin intermediul unor circuite (*branch predictors*) și în general se bazează pe menținerea unor tabele cu informații despre fiecare dintre ramificații.

9. Bibliografie

1. Patterson, David A.; Hennessey, John L , *Computer Architecture: A Quantitative Approach*, Publisher: Morgan Kaufmann; 5 edition (September 30, 2011), ISBN-13: 978-0123838728
2. Null J., Lobur J., The essentials of computer organization and architecture, Jones & Bartlett Learning, 3rd edition (December 17, 2010), ISBN-13: 978-1449600068.
3. Stallings W., Computer Organization and Architecture, 9th ed., Prentice Hall, March 11, 2012, ISBN-13: 978-0132936330.
4. Chalk B.S., Computer Organization and Architecture, Palgrave Macmillan, 2003.
5. Hyde R., The Art of Assembly Language Programming, No Starch Press, 2nd edition, 2010,ISBN-13: 978-1593272074 .
6. Gorgan D., Sebestyen G. - Structura calculatoarelor, Editura Albastra, 2000.
7. Corneliu Burileanu, Mihaela Ionita, Mircea Ionita, Mircea Filotti – Microprocesoarele x86 – o abordare software, 1999.