

ARHITECTURA SISTEMELOR DE CALCUL

Note de curs

Dr. Ing. Inf. Marius Gh. ROGOBETE

Bucureşti 2022

Cuprins

1.	Evoluția arhitecturii calculatoarelor	6
1.1.	Clasificarea arhitecturii sistemelor de calcul	6
1.2.	Structura fizică a unui calculator, tipuri de calculatoare, exemple de arhitecturi uniprocesor și multiprocesor	7
1.2.1	Tipuri de calculatoare	8
1.2.2.	Arhitecturi de sisteme multiprocesor	11
1.3.	Arhitectura lui Von Neumann	12
1.3.1.	Unitatea centrală (Central Processing Unit – CPU).....	12
1.4.	Principiile de funcționare ale mașinii Turing.	14
2.	Clasificarea semnalelor	16
2.1.	Semnalul analogic	17
2.2.	Semnalul digital.....	17
2.3.	Transformarea semnalelor	18
3.	Bazele logice ale calculatoarelor.....	21
3.1.	Functii logice. Porți logice. Algebra de comutatie.....	21
3.1.1.	Functii logice. Tabele de adevar	21
3.1.2.	Porți logice	24
3.2.	Algebra de comutatie.Logică și circuite logice combinaționale. Componente digitale.	27
3.2.2.	Logică și circuite logice combinaționale (CLC)	29
3.2.3.	Convertoare de cod	29
3.2.4.	Codificatoare și decodificatoare.....	31
3.2.5.	Multiplexoare și demultiplexoare	31
3.2.6.	Comparatoare	33
3.2.7.	Sumatoare.....	34
3.3.	Mașini cu număr finit de stări	35
4.	Componentele unui sistem de calcul.....	40
4.1.	Magistrale	40
4.2.	Unitatea centrală, procesorul (execuția unei instrucțiuni, tipuri de unități centrale, exemple de microprocesoare)	47
4.3.	Tipuri de memorie (memoria principală, memoria de lucru, memoria cache, memoria ROM, memoria video, memoriile secundare)	54
4.3.1.	Clasificare memoriei	56
4.3.2.	Memoria principală	57
4.3.3.	Memoria cache	59

4.3.4. Memoria virtuală (secundară)	63
4.3.5. Memoria video	63
4.4. Dispozitive de intrare/ieșire (organizare, conectare, interfețe).	64
5. Nivelul fizic	67
5.1. Microprocesorul, intreruperile, magistralele de comunicație (magistralele sincrone și asincrone, arbitrajul magistralei).....	67
5.1.2. Moduri de lucru între microprocesor și interfetele I/O	69
5.2. Magistrala procesorului, magistrala de date, magistrala de adese, magistrala I/O	73
5.3. Exemple și comparații.	73
6. Compilatoare și asambloare	76
6.1. Limbaje de nivel înalt, limbaje de nivel scăzut	77
6.1.1. Compilare vs. Interpretare.....	78
6.2. Descrierea componentelor unui compilator	78
6.3. Programarea în limbaj de asamblare	80
6.3.1. Familia de procesoare Intel x86	81
6.4. Directive și instrucțiuni ASM.	84
6.4.1. Sintaxa unei instrucțiuni în limbaj de asamblare	84
Clase de instrucțiuni.....	84
6.4.2. Instrucțiuni de transfer	84
6.4.3. Instrucțiuni aritmetice	86
6.4.4. Instrucțiuni logice	88
6.4.5. Instrucțiuni de deplasare și de rotire	88
6.4.6. Instrucțiuni de salt.....	89
6.4.7. Instrucțiunile de apel și revenire din rutine	90
6.4.8. Instrucțiunile de ramificare și ciclare	90
6.4.9. Instrucțiuni de intrare/ieșire	92
6.4.10. Instrucțiuni pe șiruri	92
6.4.11. Instrucțiuni speciale.....	93
7. Nivelul sistemului de exploatare	94
7.1. Memoria virtuală, conceptual de paginare , conceptul de segmentar	94
7.1.1. Paginarea	96
7.1.2. Segmentarea	97
7.2. Exemple de gestionare a memoriei virtuale.	98
7.2.1. Memoria virtuală.....	99
8. Arhitecturi evoluate.....	101

8.1. Masinile RISC (evolutia arhitecturii sistemelor de calcul, principii de proiectare a masinilor RISC),	101
8.1.3. RISC vs CISC	102
8.2. Arhitecturi paralele, exemple	103
8.2.1. Paralelism - Pipelining	103
8.2.2. Hazard	104
9. Bibliografie	106
Anexa I: Procesorul multi-core	107
Avantaje	111
Dezavantaje	112
Hardware	112
Arhitectură	113
Efecte software	113
Compartimentare	114
Comunicare	115
Aglomerare	115
Cartografiere	115
Licențiere	115
Aplicații încorporate	115
Procesoare de rețea	116
Procesare digitală a semnalului	116
Sisteme eterogene	116
Anexa II: Infineon TriCore	116
Siguranță	117
Anexa III - Estimarea performanței procesorului folosind legea lui Amdahl	117
Ce este legea lui Amdahl?	118
Limitările legii lui Amdahl	119
Pasul 1: Testează-ți programul cu un număr diferit de nuclee CPU	119
Pasul 2: Determinarea fracției de paralelizare	121
Pasul 3: Estimați performanța procesorului folosind fractia de paralelizare	123
Modul simplu: Utilizarea unei foi de calcul Google Doc	123
Concluzie	124

Facultatea de Informatica - UTM

1. Evoluția arhitecturii calculatoarelor

Un sistem de calcul poate fi definit (conform The American Heritage Dictionary of the English Language, Fifth Edition, 2019) ca:

- dispozitiv care lucrează automat, sub controlul unui program memorat, prelucrând date în vederea producării unor rezultate ca efect al procesării;
- dispozitiv care efectuează calcule, în special o mașină electronică programabilă care execută operații aritmetice, logice sau care asamblează, stochează, coreleză sau efectuează un alt tip de procesare a informației, cu viteza 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 ușual 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 însă 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 fizică, destinate universitaților, ramuri ale industriei, etc;
- Microcalculatoare - sisteme de calcul bazate pe folosirea unui microprocesor (de unde și numele), aparute la sfârșitul 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 (calculatoare personale spre exemplu) cu putere de procesare medie, capabilități grafice și multimedia ridicate, de obicei conectate la Internet;
- Server-e - oferă diferite servicii stațiilor (clientilor). 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 centrală 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 referă la acele atribute ale sistemului care sunt vizibile programatorului și care au un impact direct asupra executiei unui program:

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

Mulțimea instrucțiunilor mașină (Instruction Set Architecture – ISA) este interfața cheie între nivelele de abstractizare, fiind interfața dintre hard și soft. Aceasta permite unor sisteme 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 instrucții sunt utilizate pentru :

- organizarea SC, modul de stocare a informației (registri, memorie);
- tipurile și structurile de date (codificări, reprezentări);
- formatul instrucțiunilor;
- setul de instrucțuni (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);

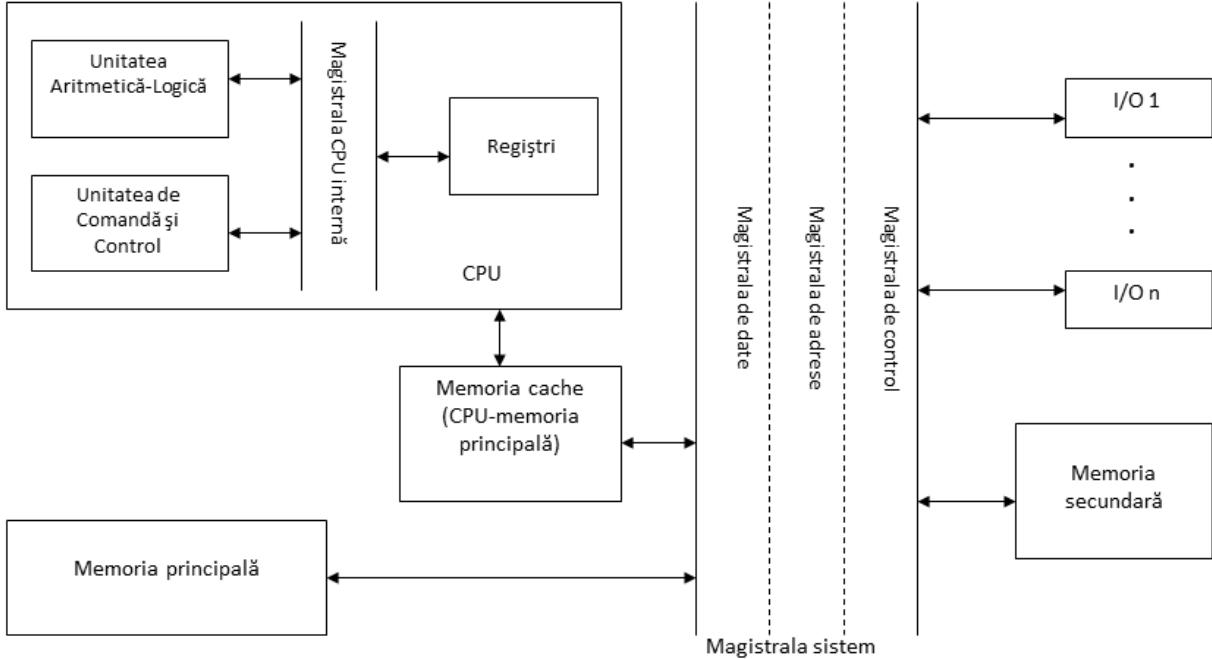


Figura 1.1 Arhitectura unui sistem de calcul.

1.2.1 Tipuri de calculatoare

Sunt două clasificări 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 funcționarea unui sistem de calcul poate fi văzută ca acțiunea unui *flux de instrucțiuni* (care reprezintă programul) asupra *unei fluxuri de date* (care reprezintă datele de intrare sau rezultate parțiale).

Clasificarea Flinn, care are vedere gradul de multiplicitate al celor două fluxuri, identifică patru tipuri de arhitecturi și anume:

- **SISD** (*Single Instruction Stream – Single Data Stream*);
- **SIMD** (*Single Instruction Stream – Multiple Data Stream*);
- **MISD** (*Multiple Instruction Stream – Single Data Stream*);
- **MIMD** (*Multiple Instruction Stream – Multiple Data Stream*).

Pentru aceste arhitecturi trei tipuri de componente de sistem respectiv **UC** (secțiunea de comandă a unității centrale), **UP** (secțiunea 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 respectă succesiunea evenimentelor specifice arhitecturii von Neumann. Instrucțiunile și datele sunt extrase din memorie, instrucțiunile sunt decodificate de **UC** care trimit secvența de instrucțiuni catre **UP** pentru execuție.

SISD

Arhitectura SISD realizează o execuție secvențială a instrucțiunilor. De exemplu o înmulțire

cu o constantă 3 a 100 de numere implica aducerea pe rând din memorie a celor 100 de numere și înmulțirea lor cu respectiva constantă, fiecare rezultat fiind trimis în memorie pe masura efectuării calculului.

Principalul neajuns al acestei arhitecturi constă în viteza de procesare care la rândul ei este determinată de fracția ceasului. Este evident că nici o realizare tehnologică nu va putea face perioada ceasului nula. În consecință modul strict secvențial de tratare a operațiilor impus de arhitectura von Neumann plafonează la un moment dat viteza de procesare. Această situație este cunoscută sub numele *gâtul sticlei lui Neumann* (*Neumann Bottleneck*).

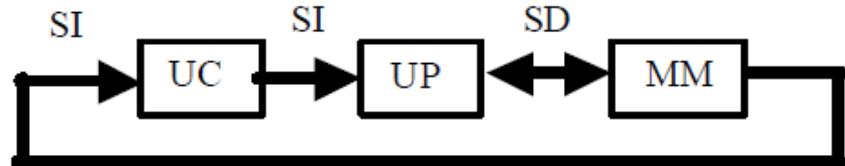


Figura 1.2. Arhitectura SISD

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

SIMD

Arhitectura SIMD, prezintă următoarele caracteristici:

- există mai multe UP, sub formă *procesoarelor de date*, datorită fluxului de date multiplu, preluat din memoria partajată MP;
- există o singură UC, sub formă *procesorului de instrucțiuni*, care supervizează procesoarele UP;
- toate UP primesc setul unic de instrucțiuni și procesează fiecare un anume flux de date (UPI procesează SD_i);
- masinile SIMD pot efectua procesari pe cuvânt (*word-slice*) sau pe bit (*bit – slice*).

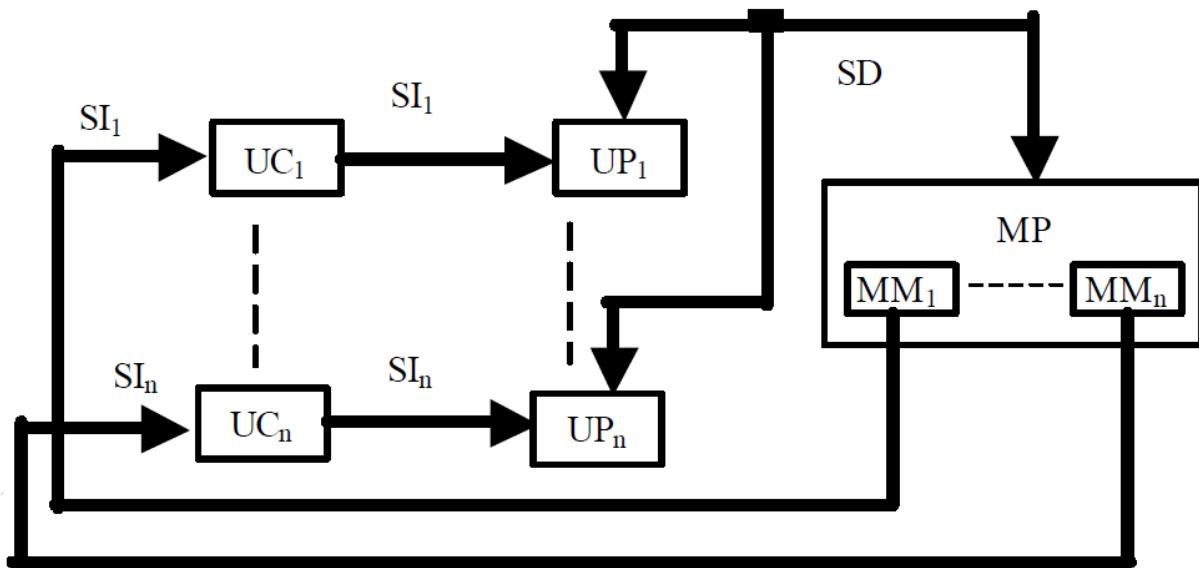


Figura 1.3. Arhitectura SIMD.

Considerând exemplul precedent cele 100 de numere vor fi înmulțite simultan cu constanta 3 iar rezultatele vor fi stocate în partitiile MM_i 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 economică deoarece

ariile de procesoare nu sunt componente standard și prin urmare aceste mașini sunt unice și costă foarte mult.

MISD

Arhitectura MISD, numita *macro-pipe-line* prezintă următoarele caracteristici:

- fiecare UC lucrează cu sirul propriu de instrucțiuni SI_1, SI_2, \dots, SI_n ;
- fiecare UP primește instrucțiuni diferite, însă operează asupra aceluiași sir de date SD (care suportă în consecință mai multe prelucrări).

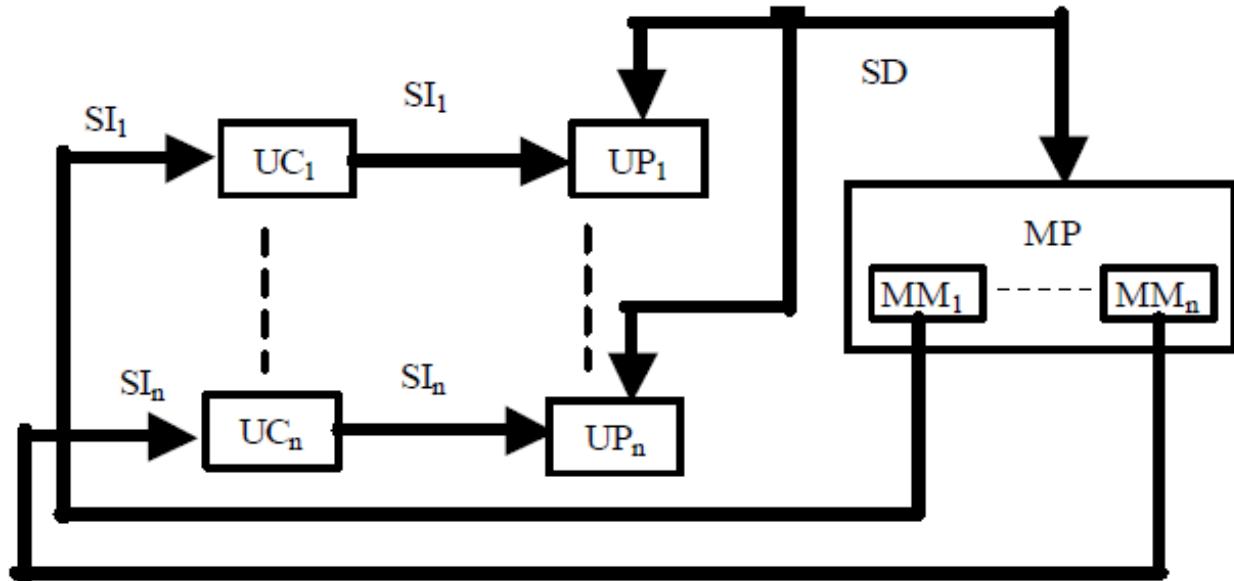


Figura 1.4. Arhitectura MISD.

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

MIMD

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

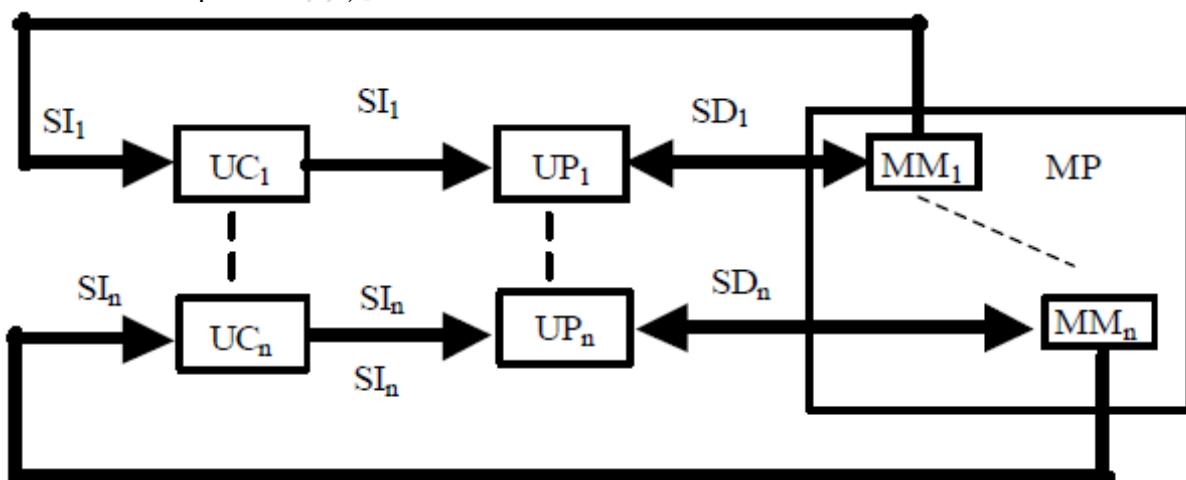


Figura 1.5. Arhitectura MIMD.

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 clasificare presupune o organizare matriceală a datelor. O matrice de dimensiune $m \times n$ presupune existența 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 (Word Serial – Bit Serial)** – se lucrează pe un singur cuvânt, fiecare cuvânt fiind prelucrat bit cu bit, respectiv $ns1, ms1$;
- **WSBP (Word Serial – Bit Paralel)** – se lucrează pe un singur cuvânt, bitii fiecarui cuvânt fiind prelucrați simultan, respectiv $n > 1, ms1$;
- **WPBS (Word Paralel – Bit Serial)** – se lucrează pe un singur bit la toate cuvintele simultan, respectiv $ns1, m > 1$;
- **WPBP (Word Paralell – Bit Paralel)** – se lucrează simultan pe toate cuvintele și pe toți bitii fiecarui cuvânt fi, respectiv $n > 1, m > 1$.

Structura **WPBP** este complet paralela fiind orientată pe prelucrările matriciale $m \times n$, în timp ce structurile **WSBP** și **WPBS** sunt parțial paralele fiind orientate pe prelucrările vectoriale (**WSBP** – orizontală $1 \times n$, **WPBS** – verticală $m \times 1$). În ceea ce privește 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.

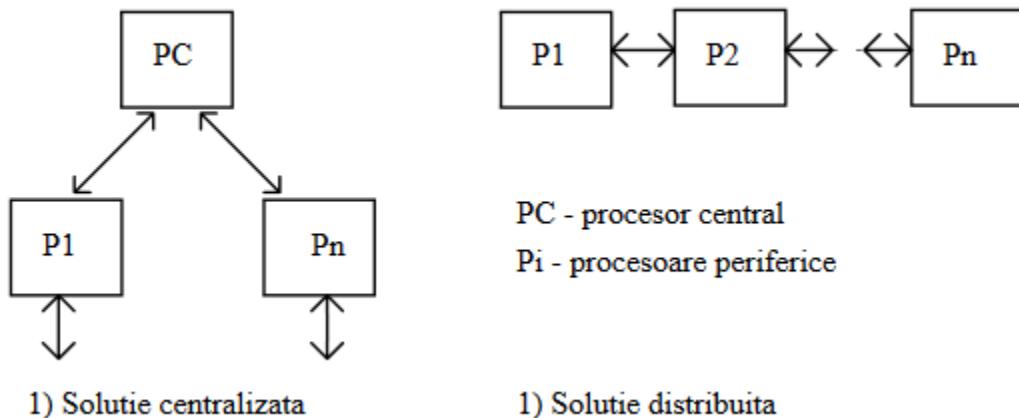
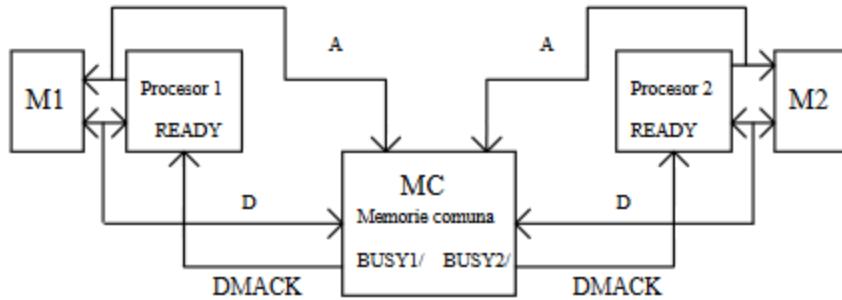


Figura 1.6. Arhitecturi multiprocesor

Soluția centralizată este utilizată pentru sisteme complexe. Este necesar un mecanism de intercomunicare între procesoare (realizat software sau hardware) care limitează performanțele sistemului. Pentru soluția distribuită deciziile se iau local de către procesoarele periferice. Mecanismul de intercomunicare (uzual realizat software) este mai simplu. Este necesară diviziarea funcțiilor sistemului în subfuncții bine determinate care să răspundă procesoarelor locale. În practică se utilizează și soluții mixte, cu un procesor central și mai multe procesoare locale.

Sistem multiprocesor cu 2 procesoare cu memorie comună



Semnalele de adresa și control pentru memoria comună nu s-au figurat

MC - (IDT7132 - 2k x 8 dual cu circuitul de arbitrage inclus)

M1, M2 - memorii私有 A, D - adrese, date

DMACK - Data Acknowledge (trebuie să fie 1 pentru funcționare normală)

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 ieșire.

ULA și 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 locație (adresa), indiferent de tipul informației conținute;
- execuția unui set de instrucțiuni se efectuează secvențial, prin citirea de instrucțiuni consecutive din memorie.

1.3.1. Unitatea centrală (Central Processing Unit - CPU)

Funcțiile unui CPU sunt:

- obținerea instrucțiunilor care trebuie executate;

- obtinerea datelor necesare instrucțiunilor;
- procesarea datelor (execuția instrucțiunilor);
- furnizarea rezultatelor obținute.

Componentele de baza ale unui CPU sunt :

- Unitatea Aritmetica-Logica (Arithmetic Logic Unit – ALU);
- Unitatea de Comandă și Control (Control Unit – CU) – decodifică instrucțiunile (FETCH/ DECODE/ READ MEMORY/ EXECUTE/ STORE);
- registrii – aceștia sunt dispozitive de stocare temporară a datelor și informațiilor de control (instrucțiunile), de capacitate mică și viteză de acces mare;
- magistrale interne CPU – dispozitive pentru comunicare între componentele CPU și comunicare cu exteriorul, pentru transferul de informații.

Ceasul sistem

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

Ciclu de ceas este cea mai mică unitate de timp sesizabilă de către un procesor.

Frecvența de ceas este numărul de cicluri de ceas pe secundă.

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

Viteza de calcul

Numărul de cicluri pe instrucțiune (Cycles per Instruction : CPI) determină viteza de procesare. Fiecare instrucțiune durează un anumit număr de cicluri de ceas (exemplu: instrucțiunea MOV durează între 2 și 14 cicluri de ceas în funcție de natura operanților).

Un procesor rulând la 1400 Mhz execuția unei instrucțiuni mai repede decât unul rulând la 800 Mhz – durata ciclului de ceas fiind mai scurtă.

Benchmarking-ul dintre viteza de calcul a procesoarelor din familia x86 versus cele din familia RISC (set redus de instrucțiuni) demonstrează diferența dintre instrucțiunile complexe care durează mai multe cicluri vs instrucțiuni simple, primare care se execută rapid.

În prezent s-a atins o anumită limită în ceea ce privește viteza procesoarelor. Mecanisme noi de creștere a vitezei pot fi:

- Pipelining - paraleлизarea instrucțiunilor (o instrucțiune trecută din faza FETCH în faza DECODE, permite unui altă instrucțiuni să treacă în faza FETCH) deci mai multe instrucțiuni se execută în paralel per ciclu de ceas - Instructions per Cycle;
- Cresterea numărului de nuclee (cores) per procesor (dual core, quad core, octal core, etc).

Viteza unui sistem de calcul este influențată de :

- Frecvența procesorului (singura nu e concluziva, vezi Intel vs AMD);
- Capacitatea maximă de memorie care poate fi adresată;
- 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 magistrală 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 întregi 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 funcționare ale mașinii Turing.

O contribuție importantă la dezvoltarea științei 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 căror parte zecimală se poate determina cu un număr finit de iteratii).

Automatul a fost sintetizat pe baza urmatoarelor ipoteze:

- automatul are un număr n finit de stări;
- automatul se află în orice moment într-o stare i , unde $l \leq i \leq n$, urmând ca la momentul imediat urmator să se afle într-o stare j , unde $l \leq j \leq n$;
- fiecare dintre cele n stări este caracterizată prin urmatoarele informații:
 - valoarea caracteristicii e_i , care este o valoare curentă a numarului care se calculează;
 - funcția f_i care aplicată stării e_i permite obținerea următoarei stării e_j ;
 - deplasamentul d_{ij} care va trebui aplicat numarului pentru a se realiza tranziția din starea 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.

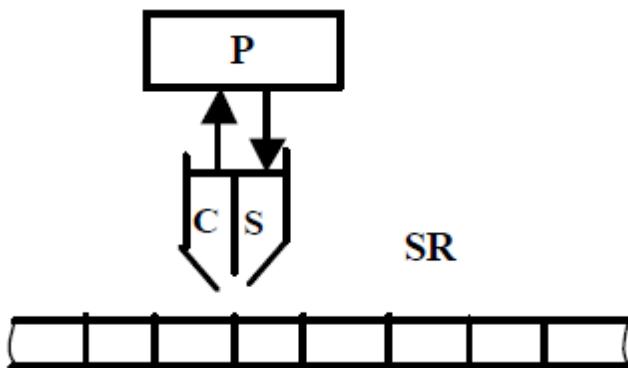


Figura 1.2. Modelul funcțional al mașinii Turing

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;
- pozoneaza capul de citire cu o pozitie la dreapta;
- pozoneaza capul de citire cu o pozitie la stânga;
- opreste sistemul.

Capul de citire/scriere poate citi respectiv înscrise 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 secvență de instrucțiuni ale masinii. Ultima instrucțiune este cea de oprire a masinii, la care banda trebuie să conțină rezultatul calculului.

2. Clasificarea semnalelor

Noțiunea de semnal este o noțiune centrală în electronică și telecomunicații.

Un *semnal* este o mărime fizică purtătoare de informație. Cel mai adesea, este o funcție scalară de variabilă 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 ieșirea 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 și comandă a proceselor (se folosește în mediile cu potențial de explozie)
- Poziția deschis-închis a releeului electromagnetic cu care se comandă funcționarea centralei termice
- Succesiunea de valori afișate de ecranul unui voltmetru digital (numeric)
- Poziția pedalei de accelerare, transmisă către carburator.

Semnalul în varianta înregistrată (memorată), se folosește în scopul reconstituirii informației inițiale sau în scopul prelucrării. Exemple:

- Înregistrarea vocii pe bandă de magnetofon
- Înregistrarea vocii de pe un CD
- Înregistrarea numerică a tensiunii afișate 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 spațială)
- Semnalul de temperatură, în grosimea unui perete (funcție scalară de timp și de spațiu)
- Imaginea dată de o cameră de luat vederi (funcție scalară de două variabile spațiale)
- Secvența de imagini date de aceeași cameră (funcție scalară, de timp și de două variabile spațiale)
- Semnalul vocal stereo (două funcții scalare de variabilă timp, care formează o funcție vectorială de variabilă timp)
- Semnalele de tensiune de la ieșirea unui traductor de înclinare față de verticală (funcție vectorială de variabilă timp).

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

- Să poată fi prelucrată (adică să poată fi depusă informație, să se poată extrage informație și să se poată aduce modificări informației purtate de acea mărime)
- Să poată fi transmisă la distanță
- Să fie puțin afectată de perturbații

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

Mărimele folosite ca semnale sunt:

- Tensiunea electrică (vezi semnalul de microfon etc.)
- Curentul electric (vezi ieșirea unor tructoare)
- Deplasarea mecanică (vezi pedala de accelerare)
- 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 continuității 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 eșantionat*)

2.1. Semnalul analogic

Modelul matematic al semnalului analogic este o aplicație pe mulțimea numerelor reale, cu valori în mulțimea 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, poziția 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 sunt semnale analogice, fiind continue în timp și în valori.

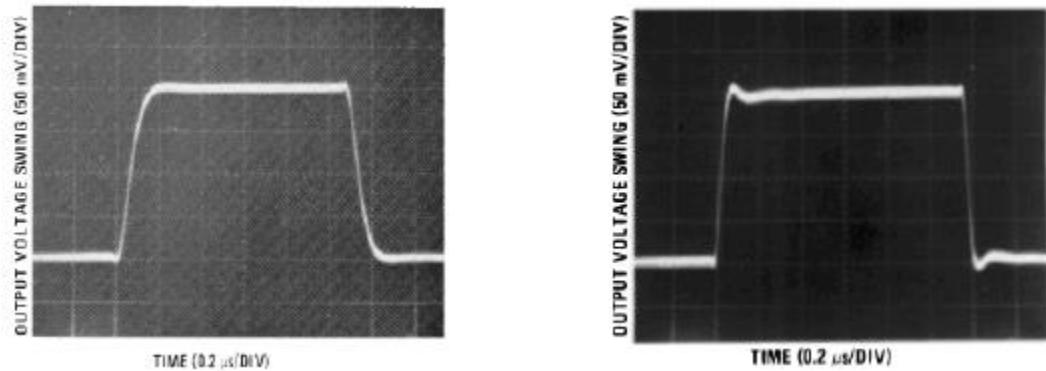


Figura 2.1 Semnal analogic pe ecranul osciloscopului

2.2. Semnalul digital

Modelul matematic al unui semnal numeric este un sir de numere, deci o aplicație pe mulțimea numărabilă (mulțimea numerelor întregi), cu valori în restricții ale mulțimii numerelor raționale sau mulțimii numerelor întregi.

Numerele reprezintă valorile approximate ale eșantioanelor unui semnal analogic. Exemple: numerele succesive indicate de un voltmètre cu afişaj numeric, indicația 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
- Posibilități mari de prelucrare
- Imunitate sporită la perturbații
- 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 frecvențele 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) și 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 (exemplu: 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, afișare sau pentru scopuri audio-video.

Etapele conversiei AD și DA:

- Eșantionarea și reținerea eșantionului („sample and hold”)
- *Cuantizarea* eșantionului (reprezentarea printr-un nivel discret)
- *Codarea numerică* a nivelului cuantizat, prin care este reprezentat eșantionul

În figura 2.2 este reprezentată o scurtă secvență dintr-un semnal analogic, precum și eșantioanele obținute de la acest semnal. Semnalul a fost eșantionat la intervale egale (*perioada de eșantionare*). În figura 2.3 sunt reprezentate aproximările eșantioanelor, ca urmare a cuantizării. Se observă că fiecare eșantion ia doar valori discrete, dintr-o mulțime finită. În partea inferioară a figurii 2.2 sunt scrise codurile numerice ale nivelurilor rezultate prin cuantizare (numere în baza 2).

Aceasta este forma în care sunt prelucrate în calculatorul numeric, sau sunt memorate, sau sunt transmise prin sisteme de comunicații numerice. Circuitele de conversie ADC și DAC, precum și introducerea datelor în calculator reprezintă bazele sistemelor de achiziție a datelor.

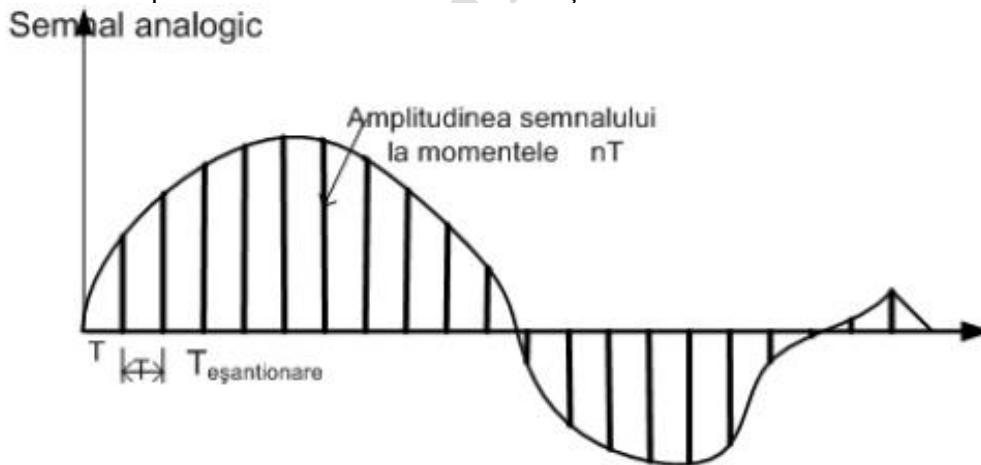


Figura 2.2 Semnalul analogic și semnalul eșantionat

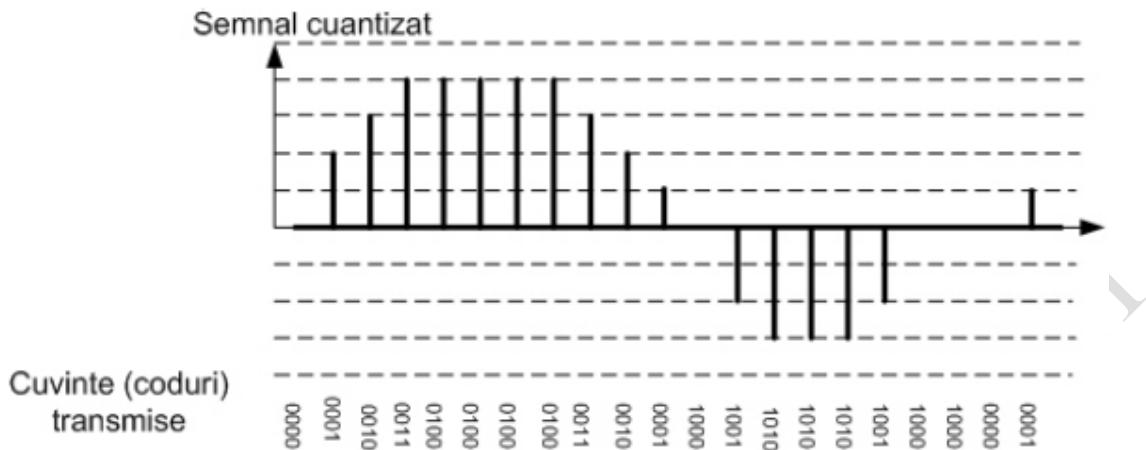


Figura 2.3 Semnalul eșantionat, cuantizat și codat numeric

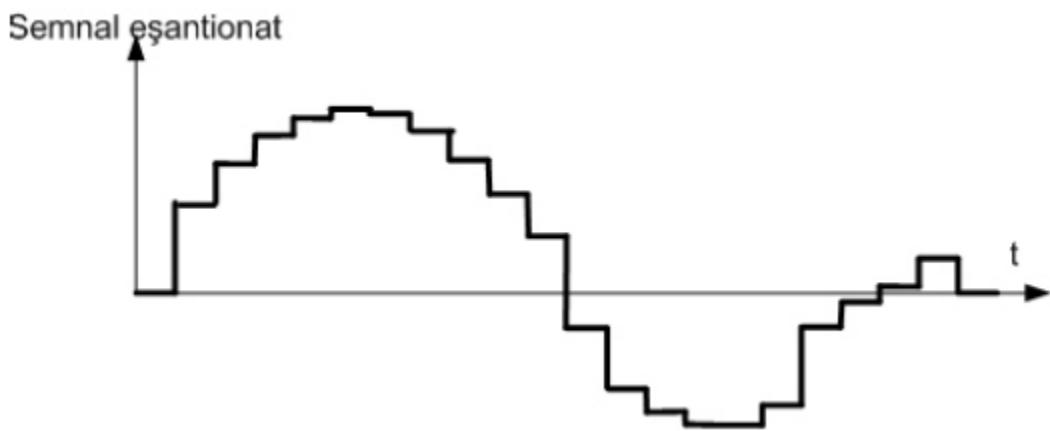


Figura 2.4 Reconstituirea semnalului analogic (conversie A/N)

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 reconstituit din aproximări ale eșantioanelor. Aspectul de funcție în scară provine din aproximarea semnalului doar prin valori discrete. Valorile funcției între momentele de eșantionare sunt approximate prin menținerea valorii de la ultimul moment de eșantionare.

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 eșantionul)?
- cît de frecvență trebuie să fie eșantionarea (adică cît de mică să fie perioada de eșantionare)?
- *Răspunsul 1:* atît de fină pe cît de mică este eroarea pe care sîntem dispuși să o acceptăm (să ne gîndim la rezoluția voltmetrului cu afișare numerică, la care este prezentă exact această problemă).
- *Răspunsul 2:* este mai complicat, va fi tratat la cursul de Semnale și sisteme. Ca regulă generală: dacă componenta cu frecvență cea mai mare din semnalul analogic are frecvența f , atunci eșantionarea trebuie să se producă cu o frecvență mai mare decît $2f$.

Din punctul de vedere al conținutului informațional, semnalele se împart în două categorii: semnale deterministe și semnale întâmplătoare (sau aleatoare).

Semnale deterministe: cele la care evoluția semnalului este anterior cunoscută. Ele nu aduc nici o informație, sănt folosite doar pentru testarea circuitelor și echipamentelor, în laborator sau în exploatare. Cel mai adesea, semnalele folosite pentru testare sănt periodice.

Forme uzuale: sinus, dreptunghi, triunghi, dinte de fierastrău, impulsuri etc.

Semnalele întâmplătoare sănt cele a căror evoluție nu poate fi previsă, deci poartă cu ele informație (cu cât sănt mai puțin predictibile, cu atât aduc mai multă informație). Cel mult cunoaștem dinainte proprietățile statistice ale semnalului întâmplător (domeniul valorilor, frecvența cea mai mare a componentelor sale etc.), dar nu evoluția particulară într-un anumit interval de timp.

Exemple de semnale întâmplătoare: semnalul vocal cules de microfon, curentul absorbit de motorul electric, turăția 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 generațiilor 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 și ale unor circuite logice fundamentale.

3.1. Funcții logice. Porți logice. Algebra de comutație.

3.1.1. Funcții 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.

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

Funcția 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 funcției sunt reprezentate prin două nivele de tensiune diferite. Expresiile booleene sau funcțiile 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 ieșire pentru funcția f . Cu alte cuvinte, tabelul de adevăr afișează ieșirile 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 canonica normal disjunctivă

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

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

Fiecare operație AND poate fi privită ca un produs booleană, termenul obținut din variabile conectate de operatori AND fiind un termen-produs. Operatorul OR este assimilat ca o însumare booleană, iar expresia cu termeni produs conectați de operatori OR fiind o expresie sumă-de-produse sau forma canonica normal disjunctivă.

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

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

Notând $\bar{A}\bar{B}\bar{C}$ cu P_0 , $\bar{A}\bar{B}C$ cu P_1 , etc., forma canonica normal disjunctivă se poate scrie 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 canonica normal conjunctivă

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

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

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

$$f(A, B, C) = S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7. \quad (3.5)$$

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 canonico.

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

	$\bar{A} \bar{B}$ 0 0	$\bar{A}B$ 0 1	AB 1 1	$B\bar{B}$ 1 0
$C \bar{0}$	P_0	P_2	P_6	P_4
$C 1$	P_1	P_3	P_7	P_5

22

$\bar{A} \bar{B}$ 0 0	$\bar{A}B$ 0 1	AB 1 1	$B\bar{B}$ 1 0	
$\bar{C} \bar{D}$ 0 0	P_0	P_4	P_{12}	P_8
$\bar{C} \bar{D}$ 0 1	P_1	P_5	P_{13}	P_9
$\bar{C} D$ 1 1	P_3	P_7	P_{15}	P_{11}
$\bar{C} \bar{D}$ 1 0	P_2	P_6	P_{14}	P_{10}

Figura 1
Diagrama V.K.

Figura 2
Diagrama V-K
 pentru 4 variabile de intrare

e. Forma elementară

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

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

Exemplu de scriere a unei funcții sub formă elementară:

$$f(A, B, C) = \bar{A}B + \bar{B}\bar{C}$$

(3.6)

3.1.1.1. Minimizarea funcțiilor logice

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

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 combinații.

Tehnica minimizării cu ajutorul diagramelor Veitch-Karnaugh:

1. Avem dată definiția funcției exprimată ca o sumă de produse;
2. Elementele din diagrama Veitch-Karnaugh ce corespund termenilor din expresie sunt marcate cu 1; celelalte 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 suprafețelor valide de valoare 1, formate din căsuțe adiacente pe orizontală sau verticală (suprafețele pot conține 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 și dreapta), întrucât ele corespund termenilor cu o singură variabilă diferită.
5. Suprafețele maximale corespund termenilor elementari, iar reprezentarea grafică este ilustrarea teoremei:

$$A \cdot B + A \cdot \bar{B} = A \quad (3.7)$$

6. Forma elementară se obține ca o sumă de produse, unind prin operatori **AND** termenii elementari rezultați 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	00	01	11	10
CD	00			
	00	1	1	1
	01		1	
	11			1
	10	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 suprafețele maximale vor fi date de căsuțele adiacente conținând 0 logic.
- Se preferă, totuși, lucrul cu f.c.n.d., care are avantajul, pe lângă comoditatea oferită de lucrul cu expresii algebrice care conțin sume de produse, și pe acela al implementării cu porți tip **NAND**, mai răspândite și mai avantajoase tehnologic.

3.1.2. Porți logice

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

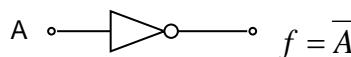
- **NOT**
- **AND**
- **OR**

Toate funcțiile care se obțin cu ajutorul acestor operatori sunt implementate de circuite numite porți logice.

3.1.2.1. Poarta NOT

Operarea porților 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 funcționării o variabilă să fie modificată, de exemplu din 1 în 0 sau din 0 în 1. Aceasta este operația fundamentală **NU**, realizată de **poarta NOT**.

Simbolul de circuit, expresia booleană și tabela de adevăr corespunzătoare unei porți **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 aplicații cu semnale de control în care trebuie dată o comandă, dacă mai multe condiții sau evenimente coexistă. Aceasta funcție este îndeplinită de operatorul și **poarta AND**.

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

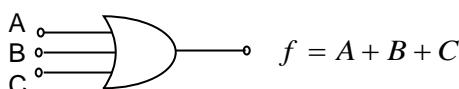


A	B	C	f
0	0	0	0
0	0	1	0
0	1	1	0
0	1	0	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

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

3.1.2.3. Poarta OR

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



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

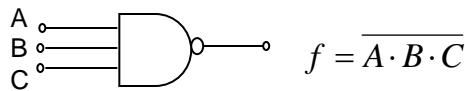
0	1	0	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Pentru orice intrare SUS ieșirea va fi SUS.

3.1.2.4. Poarta NAND

Implementarea funcțiilor **AND**, **OR** și **NOT**, ca de altfel a oricărei expresii booleene se poate face folosind **porți universale**. Una dintre acestea este **poarta NAND (ȘI-NU)**.

Simbolul de circuit, expresia booleană și tabelul de adevăr, pentru o poartă **ȘI-NU (NAND)** cu trei intrări sunt:

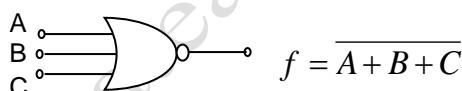


A	B	C	f
0	0	0	1
0	0	1	1
0	1	0	1
1	0	0	1
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

- 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 și tabelul de adevăr pentru o poartă **NOR** cu trei intrări, sunt:



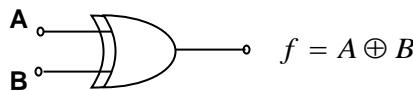
A	B	C	f
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	1	1	0
1	0	1	0
1	1	0	0
1	1	1	0

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

3.1.2.6. Poarta SAU EXCLUSIV (XOR)

Poarta **EXCLUSIVE OR (SAU EXCLUSIV)** are ieșirea în starea “I” atunci și numai atunci când o singură intrare este în starea “I”.

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



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 porți **AND** și **OR**.

3.2. Algebră de comutație. Logică și circuite logice combinaționale. Componente digitale.

3.2.1. Algebra de comutație

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

$$A = (B, O),$$

Unde:

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

$$\forall x, y \in B \quad x \text{ AND } y = \begin{cases} 1 & \text{dacă } x = 1 \text{ și } y = 1 \\ 0 & \text{dacă } x = 0 \text{ și } y = 0 \end{cases}$$

$$\forall x, y \in B \quad x \text{ OR } y = \begin{cases} 1 & \text{dacă } x = 1 \text{ sau } y = 1 \\ 0 & \text{dacă } x = 0 \text{ sau } y = 0 \end{cases}$$

$$\forall x \in B \quad \text{NU } x = \begin{cases} 1 & \text{dacă } x = 0 \\ 0 & \text{dacă } x = 1 \end{cases}$$

(3.8)

Echivalențe ale operatorilor:

- $\text{AND} \equiv \text{SI}, \cap, \cdot$ (*produs logic*);
- $\text{OR} \equiv \text{SAU}, \cup, +$ (*sumă logică*);
- $\text{NOT} \equiv \text{NU}, \#, (\text{produs logic})$;

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 t_1 și t_2 sunt expresii de comutație atunci $t_1 \cdot t_2$, $t_1 + t_2$ și \bar{t}_1 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. Funcții de comutație

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 consideră 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 talelei de adevăr se obține expresia de comutație a funcției în următorul 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 reprezentându-se în formă directă dacă variabila are valoarea 1 în combinația respectivă de valori din tabelă sau în formă complementară (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 directă cât și negată se numește *forma canonica 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 proprietățile de mai sus, pentru exemplul considerat fiind:

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

Observații.

1. Orice funcție de comutație de n variabile se poate reprezenta în mod unic printr-o expresie de formă canonica 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, toți ceilalți mintermeni au valoarea 0;
4. Produsul a doi mintermeni diferiți este 1.

3.2.2. Logică și circuite logice combinaționale (CLC)

Independența mărimilor de ieșire 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 x_0, x_1, \dots, x_{m-1} și generează în exterior ieșirile y_0, y_1, \dots, y_{n-1} . O funcție logică (de comutație) poate descrie funcționarea circuitului.

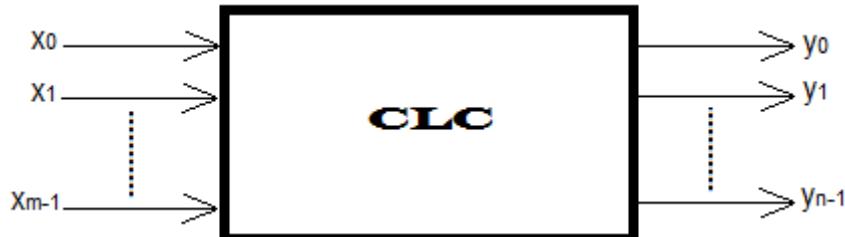


Fig. 3.1. Circuit logic combinațional

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

- definirea funcțiilor logice;
- minimizarea acestora;
- obținerea schemei circuitului.

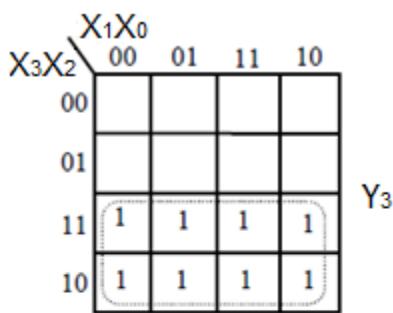
Tipurile reprezentative de CLC din structura unui calculator numeric sunt: *convertoarele de cod, codificatoarele și decodificatoarele, multiplexoarele și demultiplexoarele, comparatoarele, detectoarele și generatoarele de paritate, arile logice programabile, memoriile și circuitele aritmetice*.

3.2.3. Convertor de cod

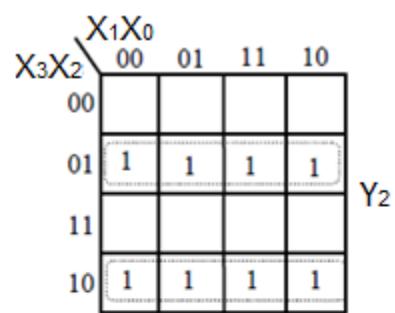
CLC-ul care permite trecerea dintr-un cod binar în altul sunt convertoarele de cod. Sinteză unui asemenea CLC se va exemplifica pentru un convertor *din cod binar în cod Gray*. În figură se prezintă elementele aferente sintezei acestui tip de convertor, în care $X_3 X_2 X_1 X_0$ reprezintă cuvântul binar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

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

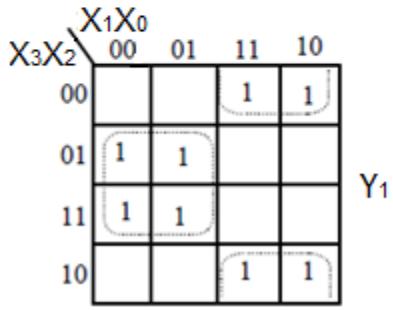
Figura 3.2. a) Convertor de cod natural- Gray: tabela de corespondență.



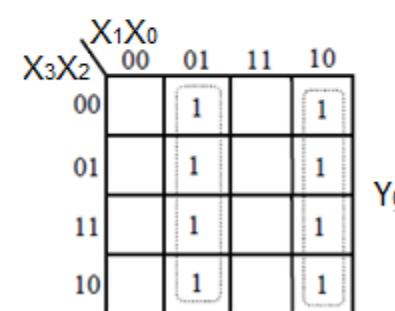
Y_3



Y_2



Y_1



Y_0

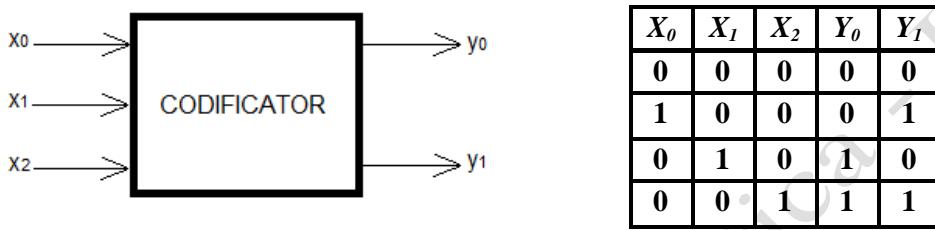
Figura 3.2. b) Convertor de cod natural- Gray: diagramele Karnaugh asociate.

După reducerile în diagramele Karnaugh rezultă:

$$\begin{aligned}
 Y_3 &= X_3 \\
 Y_2 &= \overline{X}_2 X_3 + \overline{X}_3 X_2 = X_2 \oplus X_3 \\
 Y_1 &= \overline{X}_1 X_2 + \overline{X}_2 X_1 = X_1 \oplus X_2 \\
 Y_0 &= \overline{X}_1 X_0 + \overline{X}_0 X_1 = X_1 \oplus X_0
 \end{aligned}$$

3.2.4. Codificatoare și decodificatoare

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



$$\begin{aligned}
 Y_0 &= \overline{X}_0 \cdot (X_1 \oplus X_2) \\
 Y_1 &= \overline{X}_1 \cdot (X_0 \oplus X_2)
 \end{aligned}$$

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

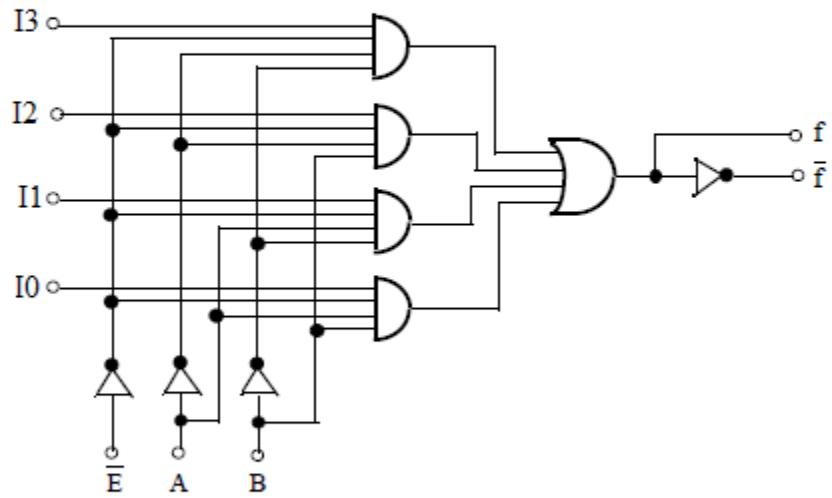
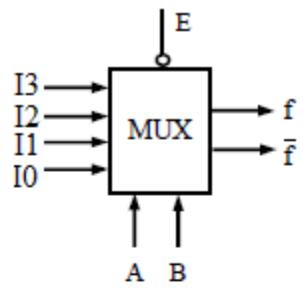
Decodificatoarele sunt acele CLC care activează una sau mai multe ieșiri funcție de cuvântul de cod aplicat la intrare și este necesară în aplicații care adresează memoria, în afișarea numerică, multiplexarea datelor etc.

3.2.5. Multiplexoare și demultiplexoare

Multiplexoarele (MUX) sunt CLC care fac transferul datelor de la una din intrările selectate de o adresă (cuvânt de selecție) către o ieșire unică. Funcțional MUX pot fi asimilate cu o rețea de comutatoare comandate. *Multiplexoarele* pot fi analogice sau numerice, ultimele fiind specifice CN.

Exemplu: sinteza unui MUX 4:1 numeric și se prezintă implementarea cu porti 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

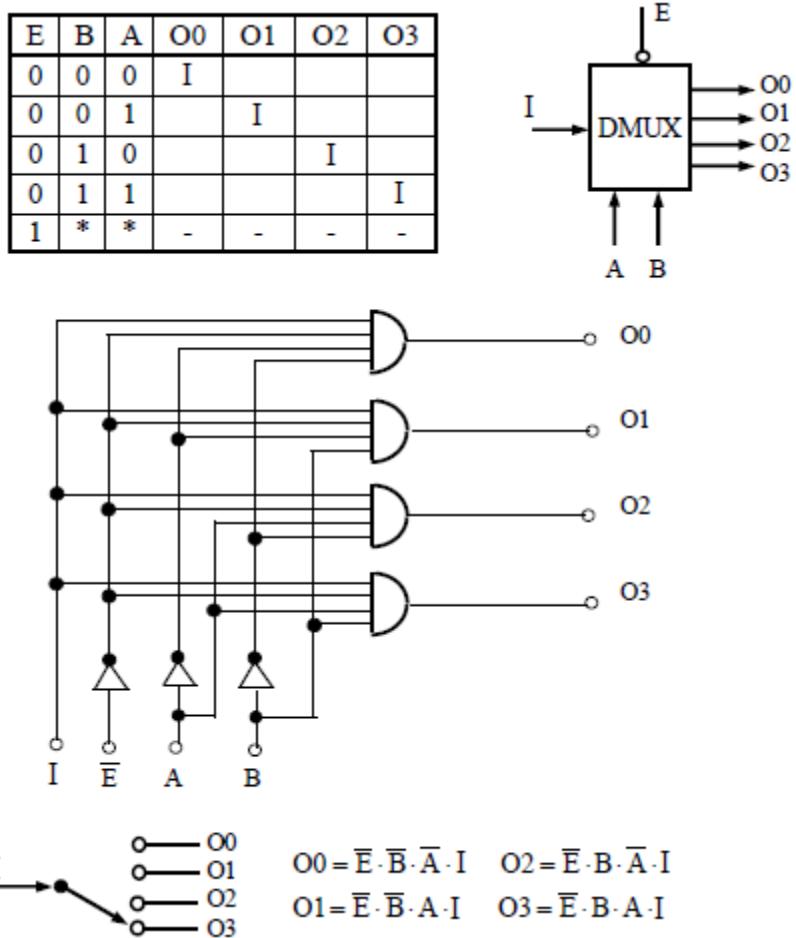


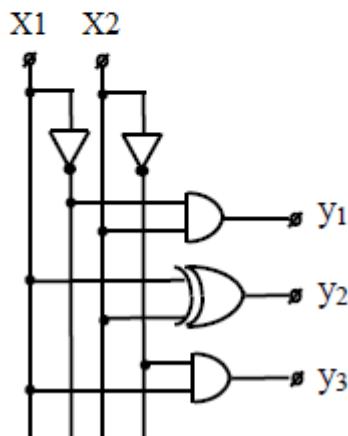
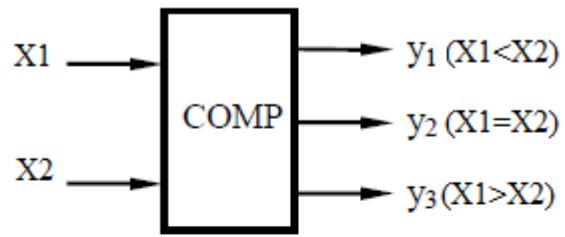
Figura 3.5. Demultiplexor 1:4

3.2.6. Comparatoare

CLC-urile care determină relației dintre două numere sunt *comparatoare numerice*. Acestea au ieșirile reprezentate de *trei funcții* ($<$, $=$, $>$) ce corespund tipului de relație existent între numerele aplicate la intrare.

Elementele definitorii ale unui comparator pe un bit sunt prezentate în figura următoare.

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



$$\begin{aligned}y_1 &= \bar{X}_1 \cdot X_2 \\y_2 &= \bar{X}_1 \cdot \bar{X}_2 + X_1 \cdot X_2 = \bar{X}_1 \oplus X_2 \\y_3 &= X_1 \cdot X_2\end{aligned}$$

Figura 3.6. Comparator pe un bit

3.2.7. Sumatoare

Un *sumator elementar* este un CLC care adună două numere binare x_i, y_i cu un transport de intrare c_i , generând la ieșire doi biți: suma și transportul c_{i+1} către rangul superior, conform tabelului:

x _i	y _i	c _i	s _i	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 relațiile :

$$\begin{aligned}s_i &= \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i \\c_{i+1} &= \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i \bar{c}_i + x_i y_i c_i = c_i (x_i \oplus y_i) + x_i y_i.\end{aligned}$$

Relațiile de mai sugerează obținerea sumatorului elementar din două semisumatoare:

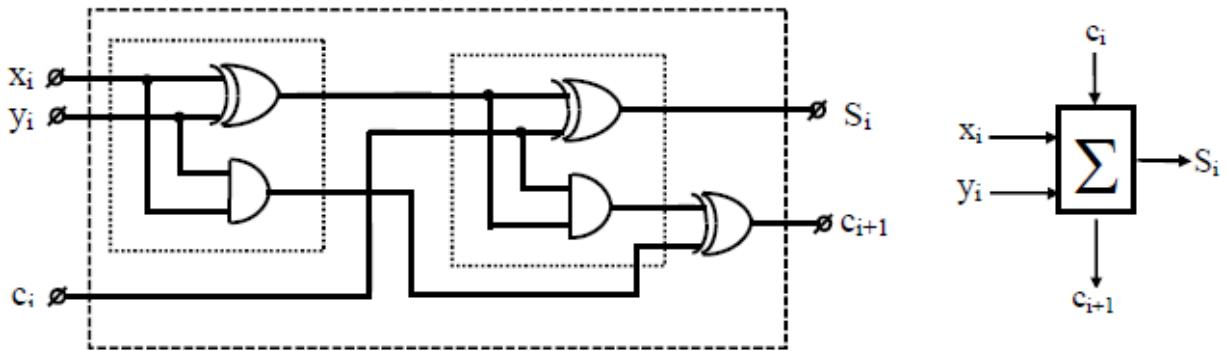


Figura 3.7. Sumatorul elementar

3.3. Mașini cu număr finit de stări

Un **automat finit** (AF) sau o **mașină cu stări finite** este un model de comportament compus din *stări*, *tranzitii* și *acțiuni*.

- O stare reține comportamentul compus din *stări*, *tranzitii* și *acțiuni*. Starea stochează informații despre trecut, adică reflectă schimbările intrării de la inițializarea sistemului până în momentul de față.
- Tranzitia indică o schimbare de stare și este descrisă de o condiție care este nevoie să fie îndeplinită pentru a declanșa tranzitia.
- Acțiunea este o descriere a unei activități ce urmează a fi executată la un anumit moment. Există câteva tipuri de acțiuni:
 - Acțiune *de intrare* executată la intrarea într-o stare.
 - Acțiune *de ieșire* executată la ieșirea dintr-o stare.
 - Acțiune *de intrare de date* executată în funcție de starea prezentă și de datele de intrare.
 - Acțiune *de tranzitie* executată în momentul unei tranzitii.

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

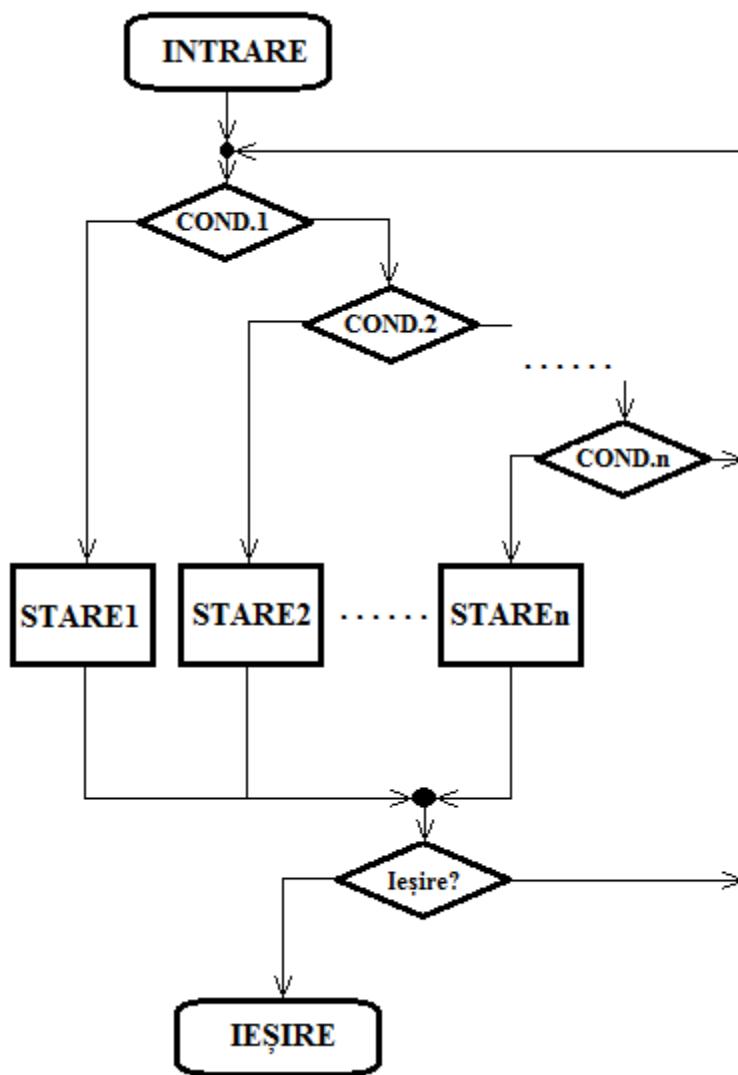


Figura 3.8 Logica automatelor finite

Modelul matematic În funcție de tip, există mai multe definiții. Un automat finit **acceptor** este un cvintuplu $\langle \Sigma, S, s_0, \delta, F \rangle$, unde:

- Σ este alfabetul de intrare (o mulțime finită și nevidă de simboluri).
- S este o mulțime finită și nevidă de stări.
- s_0 este starea inițială, element al lui S . Într-un automat finit nedeterminist, s_0 este o mulțime de stări inițiale.
- δ este funcția de tranziție a stării: $\delta: S \times \Sigma \rightarrow S$.
- F este mulțimea stărilor finale, o submulțime (posibil vidă) a lui S .

Un automat finit **transductor** este un sextuplu $\langle \Sigma, \Gamma, S, s_0, \delta, \omega \rangle$, unde:

- Σ, S și s_0 își păstrează semnificațiile din definiția automatului finit acceptor.
- Γ este alfabetul de ieșire (o mulțime finită și nevidă de simboluri).
- δ este funcția de tranziție a stării $\delta: S \times \Sigma \rightarrow S \times \Gamma$.
- ω este funcția de ieșire.

Dacă funcția de ieșire este o funcție de stare și de alfabetul de intrare ($\omega: S \times \Sigma \rightarrow \Gamma$), atunci această definiție corespunde **modelului Mealy**. Dacă funcția de ieșire depinde doar de stare ($\omega: S \rightarrow \Gamma$), atunci această definiție corespunde **modelului Moore**.

Aplicație: 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 și O reprezintă ieșirea.

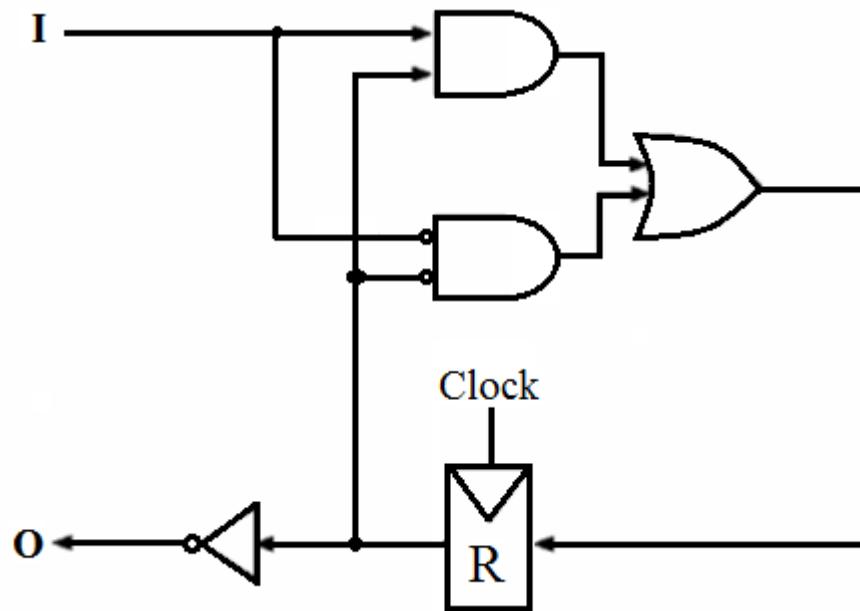


Figura 3.9. Automat cu număr finit de stări.

- Câte stări are automatul A? Desenați diagrama stărilor și a tranzițiilor pentru automatul A.
- Circuitul de mai jos este un automat cu număr finit de stări programabil. Componentele circuitului, sunt etichetate astfel: (M4) - memorie cu patru locații, fiecare conținând un bit ($2^2 \times 1$ -bit), (M2) - memorie cu două locații a către un bit fiecare ($2^1 \times 1$ -bit), (R) - registru pe un bit.

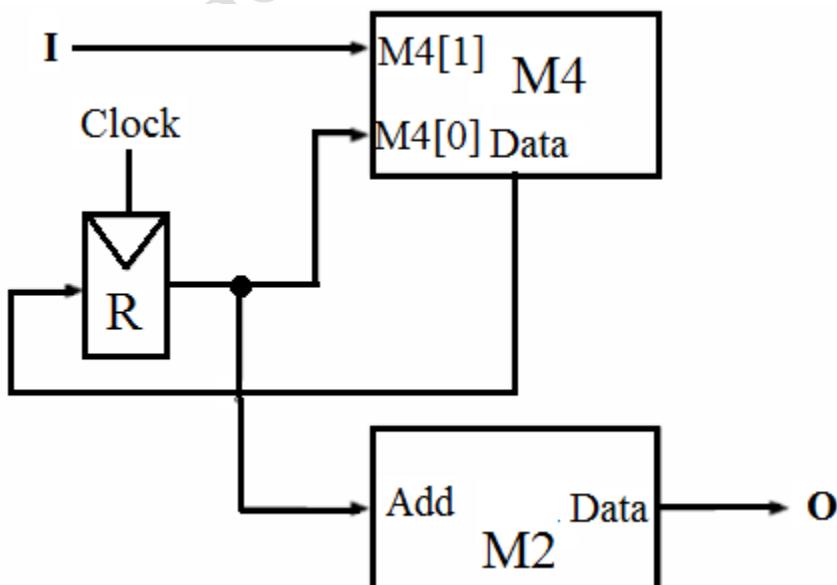


Figura 3.10. Automat programabil cu număr finit de stări.

Ce trebuie să conțină locațiile 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

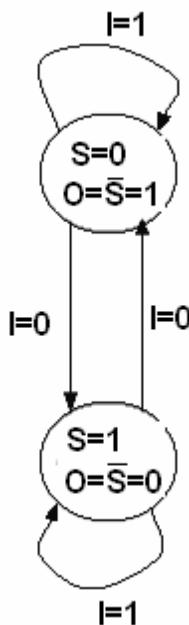
0	0
0	1
1	0
1	1

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 și a tranzițiilor pentru automatul A arată astfel:



ToDo ---->R

Considerăm o stare a automatului ($R=0$) și alta ($R=1$). Din schemă se văd că $O = \text{not } R$. 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 își 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). Înem cont de considerațiile 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 locației, și este negația stării R. În finalul acestui curs, anticipând puțin ce va fi prezentat în cursurile următoare, este ilustrată diagrama fluxului de date din cadrul calculatorului LC-3. Aceasta conține toate structurile logice digitale, combinaționale și secvențiale, care combinate îndeplinesc funcția de procesare a informațiilor, streză starea const

4. Componentele unui sistem de calcul

Structura unui sistem de calcul

Ansamblul de componente hardware (dispozitive fizice) și software (programe) ce soluționează 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 secvențe de biți (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 și mediile de programare. Majoritatea limbajelor de programare dispun de o bibliotecă bogată de funcții prin care se pot accesa resursele calculatorului.
5. Aplicațiile sunt produse program care facilitează accesul la resursele unui calculator și pentru personal nespecializat în domeniul calculatoarelor.

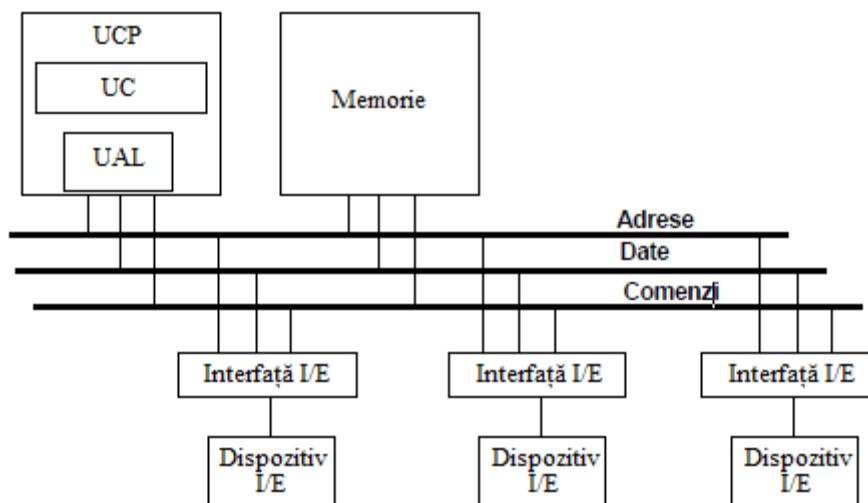


Figura 4.1 Structura fizică a unui calculator

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

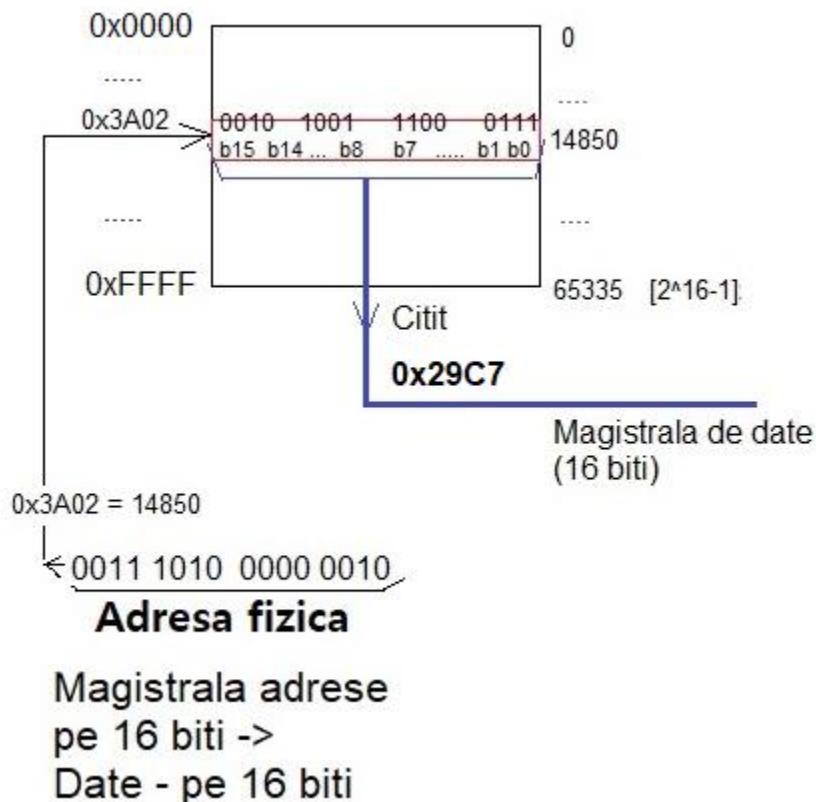
- dispozitive de intrare (ex: tastatură, mouse, interfețe de proces, etc.)
- memorie (interna și externă, volatilă și nevolatilă)
- unitate aritmetico-logică
- unitate de comandă
- dispozitive de ieșire (ex: ecran, imprimantă, etc.).

4.1. Magistrale

Pentru conectarea componentelor sistemului de calcul se folosesc una sau mai multe magistrale. Magistrala se definește 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 secvența de apariție a semnalelor, intercondiționările existente între semnale și relațiile temporare dintre semnale.

Un exemplu de accesare a memoriei pentru citire este dat în figura următoare.



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 alcătuită 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 următoarele tipuri de semnale:

- semnale de date - linii bidirectionale utilizate pentru transferul de date
- semnale de adresa - specifică adresa modulului destinație
- semnale de comandă - specifică direcția de transfer și tipul de modul I/E
- semnale de control - reglarea condițiilor de transferare a datelor
- semnale de întrerupere - apariția 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 magistrelor multimaster.
-

4.1.2. Clasificarea magistrelor

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 directă între 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 arbitrage a magistralei
 - magistrale multimaster: trebuie să conțină semnale de arbitrage 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 efectuează 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 multimea semnalelor magistralei) și deci factorul de creșterea vitezei este egal cu numărul de faze în ale tranzacției (magistrala procesorului Pentium)

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

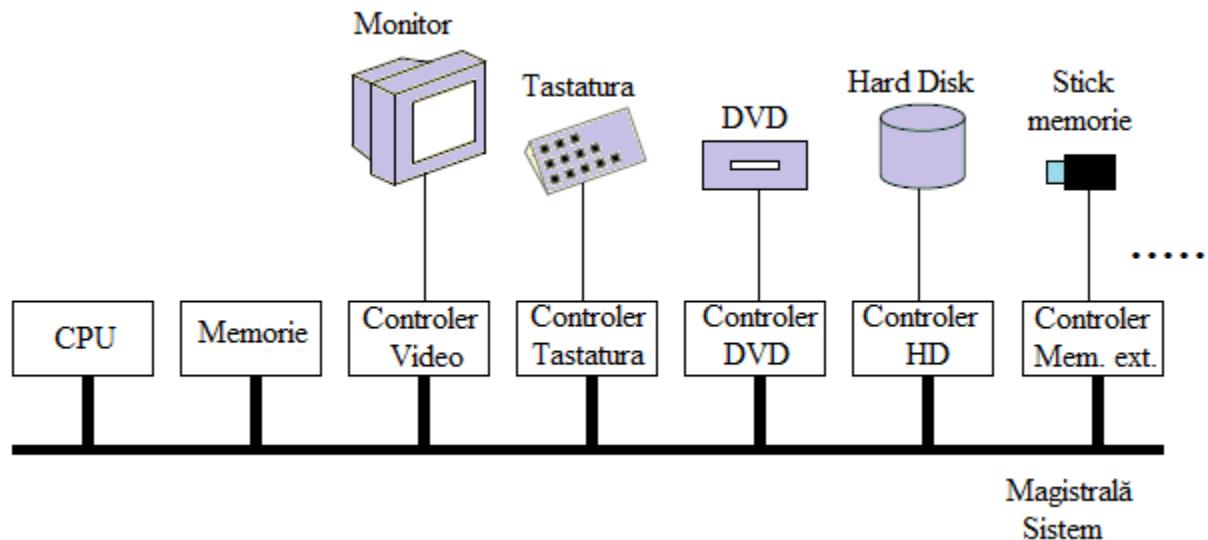


Figura 4.2. Bus sistem tip single

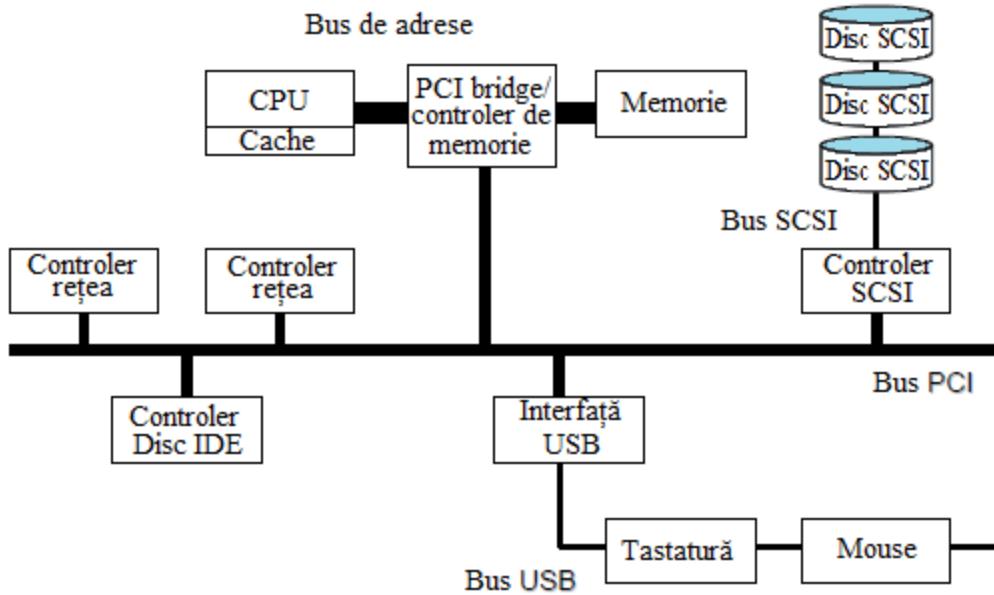


Figura 4.3. Bus sistem tip multiple

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 mari 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 și utilizabil de orice alt producător, 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 componente 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 achiziție de date și instrumentație
696	S100	Microsisteme de dimensiune medie
796	MULTIBUS 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

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 și VAXBI
- familia GPIB (a Hewlett Packard), IEEE 488 și Camac.

Dintre standardele impuse pentru calculatoarele personale se pot aminti:

- ISA (Industrial Standard Architecture) – magistrală de sistem a primelor calculatoare personale compatibile IBM PC și 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 între procesor și 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 externă a datelor (disc, banda magnetică)
- 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 conectate 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 și 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 prezentată diagramele de timp pentru un ciclu de citire memorie și pentru un ciclu de scriere memorie).

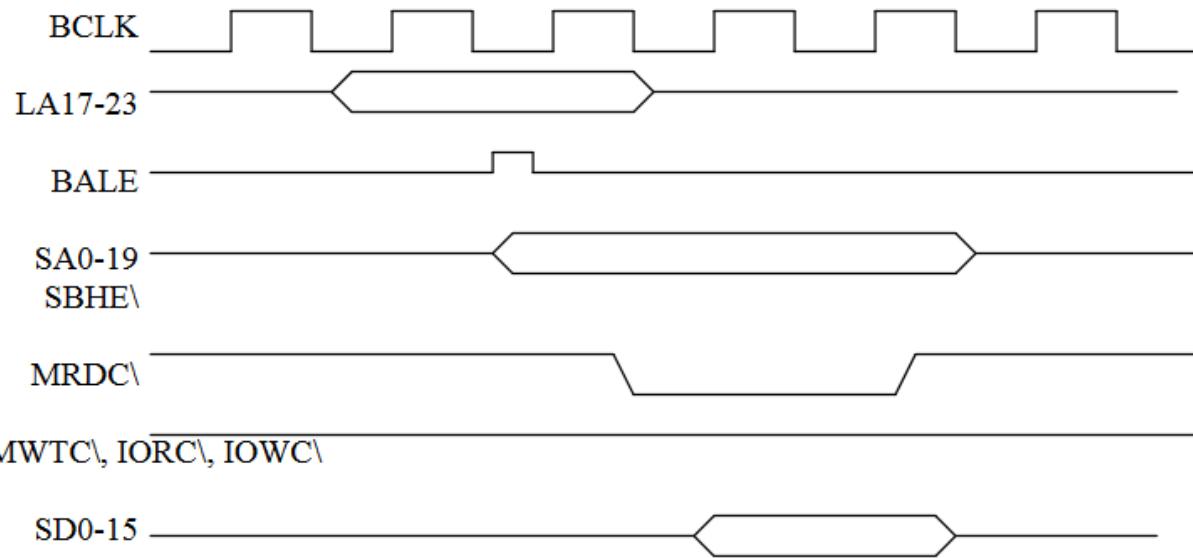


Figura 4.4. Ciclul citire memorie

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 și 8MHz
- *LA_n* = 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 lațimea cuvântului de date care se transferă
- *SD0-15* = System Data lines ; linii bidirectionale pentru transferul de date.

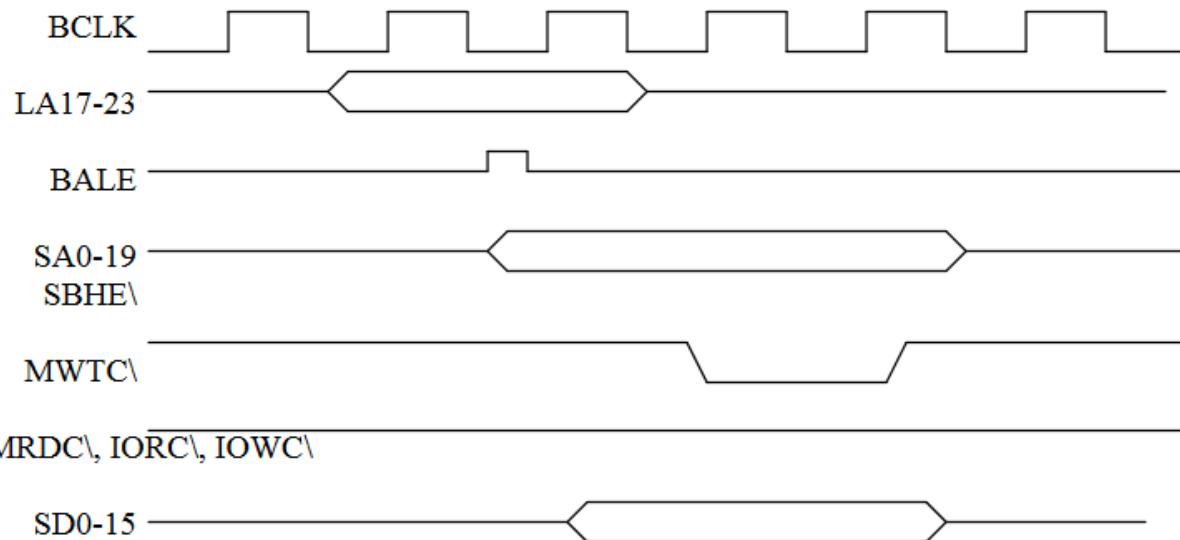


Figura 4.5. Ciclul scriere memorie

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 să fie valide pe toată durata ciclului de transfer.

Ciclurile de citire și 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\ și IOWC\).

Pentru modulele mai lente ciclul de transfer se poate prelungi prin dezactivarea temporară a semnalului CHRDY ; în acest caz ciclul se prelungesc cu un număr întreg de perioade 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 (Macintosh, SUN, etc).

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

În mod ușoar 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 să fie transferate fără validarea fiecărei celule de memorie, acestea fiind consecutive (concatenate). De precizat că nu există limită privind lungimea ciclului de transfer în rafală. Sunt posibile trei tipuri de adrese (spații de adresare) :

- memorie,

- intare/iesire,
- configurare.

La inițializarea sistemului, pentru configurația automată (plug-and-play) a placilor PCI, un dispozitiv PCI este configurație 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 placi pot conține o memorie ROM cu programe pentru inițializarea și 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. Latență 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.

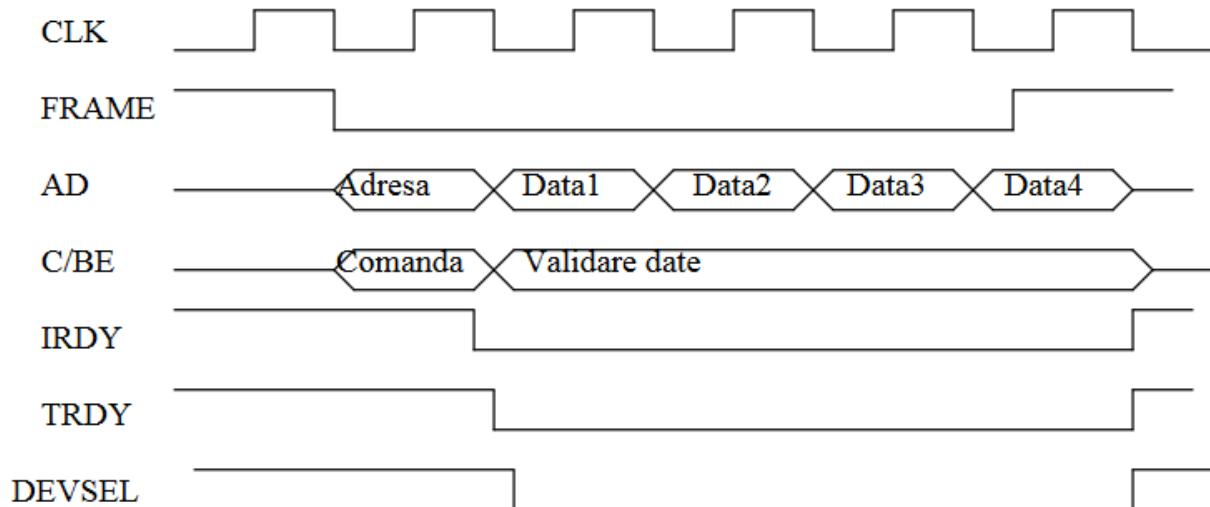


Figura 4.6. Ciclul de transfer în rafală al magistralei PCI

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 pregătită pentru transfer
- *DEVSEL* = Device Select

4.2. Unitatea centrală, procesorul (execuția unei instrucțiuni, tipuri de unități centrale, exemple de microprocesoare)

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

Acesta conține registri interni, unități 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 biți) s-a realizat în anul 1971 la compania *Intel* și 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 biți, cu procesare *pipeline* a instrucțiunilor).

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 și interfețe de intrare/ ieșire. Acestea sunt resursele ce pot fi accesate (scris /citate) de către microprocesor. Așadar, acesta nu “vede” în mod direct perifericele ci interfețele specifice de I/O.

Magistrala (bus) de date este de tip bidirectional TS. Prin intermediul acestui bus microprocesorul aduce din memorie instrucțiunea, respectiv citește 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 dorește să-l scrie în memorie sau într-un port de ieșire. 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, Interfață etc.) pe bus-ul de comenzi.

Pe bus-ul de stări, dispozitivele *slave* (memorii, interfețe) comunică informații referitoare la modul de desfășurare al transferului (ex. semnalul emis spre microprocesor “așteaptă”, având semnificația că transferul de date comandat nu este încă încheiat).

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

- a. Unitatea de Comandă (UC);
- b. Unitatea Aritmetică și Logică (UAL);
- c. Registrele Generale (RG).

a. **Unitatea de Comandă** aduce instrucțiunile din memorie, determină tipul lor după care descompune fiecare instrucțiune într-o secvență 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.

- **Registrul de Instrucțiuni (RI)** păstrează codul instrucțiunii în curs de execuție.
- **Numărătorul de Program (NP)** conține adresa instrucțiunii curente.
- **Registrul de Stare (RS)** este format dintr-un set de bistabile de stare care contin informații legate de modul de execuție a instrucțiunilor (validarea întreruperilor, execuția pas cu pas, etc.), rezultatele operațiilor aritmetice și logice (depasire de capacitate de reprezentare, transport spre rangul superior, etc.), sau informații legate de conținutul anumitor registre (par sau impar, zero, etc.).
- **Blocul Circuitelor de Comandă (BCC)** semnalele de comandă (specifice tipului unei instrucțiuni, fazei curente și informației de stare) necesare executiei instrucțiunilor.
- **Generatorul de Tact (GT)** generează semnalul de tact (a carui frecvență determină viteza de lucru) pentru functionarea sincronă a întregului calculator.

- **Generatorul de Faze (GF)** creează succesiunea specifică de faze care compun o instrucțiune. Faza următoare este determinată de faza curentă, tipul instrucțiunii și informația de stare din UCP.

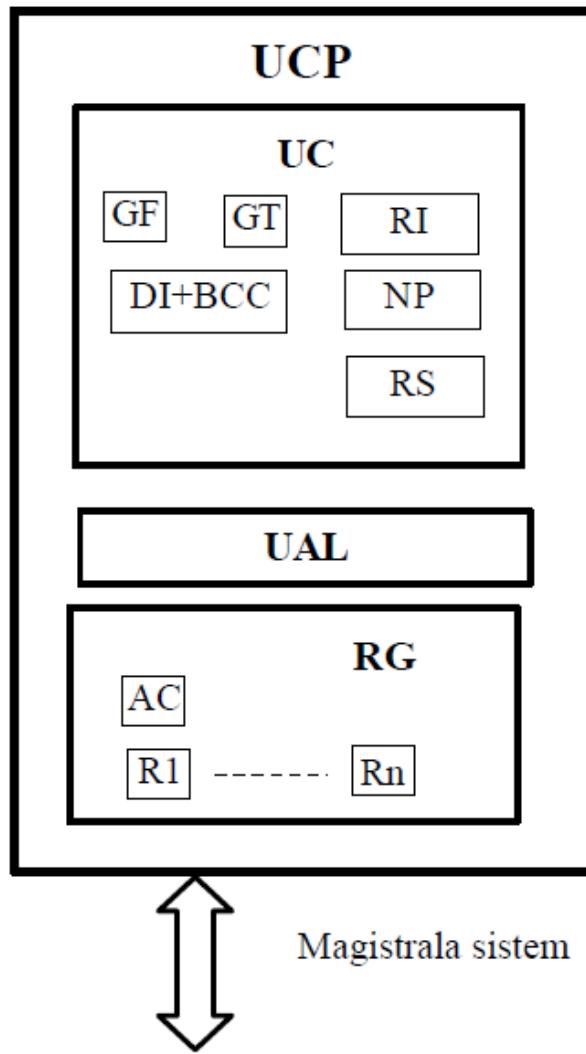


Figura 4.4. Structura UCP

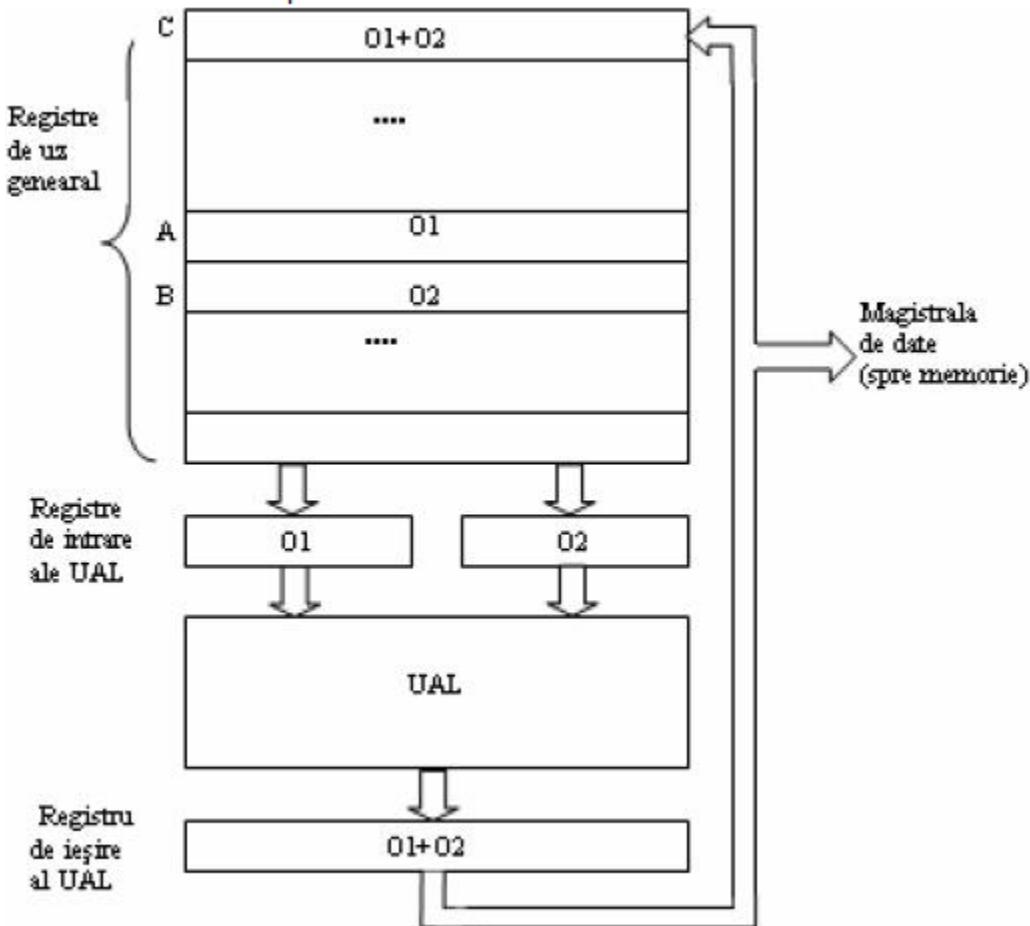
b. Unitatea Aritmetica și Logica (UAL) execută totalitatea operațiilor aritmetice (adunare, scădere, înmulțire, împărțire) și logice (Si, SAU, NU, SAU EXCLUSIV, etc.). O importanță aparte pentru UAL prezintă registrul acumulator AC, care de multe ori este tratat ca făcând parte din aceasta.

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

4.2.1. Structura unui procesor

Unitatea aritmetică și logica. Registre mai importante Procesorul este format în principal din unitatea aritmetică și logica (UAL) și mai multe registre, care sunt legate împreună 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 instrucțiunii care se executa (sau urmează să se

execute), și își modifica conținutul în ritmul executării programului. De asemenea, un alt regisztr important este și regiszrul de instrucțiuni, care păstrează instrucțiunea în curs de execuție. În plus, UCP conține mai multe regiszre de uz general. UAL executa asupra datelor de intrare operații aritmetice (adunari, scaderi, înmulțiri, împărțiri), 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 exemplifică realizarea unei adunări de către aceasta structură.

- Cei doi operanzi, O1 și O2, se află în două regiszre de uz general, A și B.
- Operanzzii sunt transferati în cele două regiszre de intrare ale UAL.
- UAL calculeaza rezultatul adunării, și depune rezultatul în regiszrul de ieșire.
- Rezultatul este mutat în alt regiszr de uz general al UAL, C. Apoi, dacă se dorește, conținutul regiszrului C poate fi transferat în memorie.

Majoritatea instrucțiunilor este de tipul regiszr-regiszr sau regiszr-memorie.

În primul caz, operanzzii (operandul) sunt extrasi din regiszre și adusi în regiszrele de intrare ale UAL, iar rezultatul este depus într-un alt regiszr.

În al doilea caz, operanzzii (operandul) sunt adusi din memorie în regiszre, 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 regiszrul de instrucțiuni (extragă instrucțiunea).

2. Schimbă număratorul de program astfel încât acesta să indice adresa urmatoarei instrucțiuni (conținutul acestuia crește cu un număr egal cu numărul de octeți al instrucțiunii în curs de execuție).
3. Determină tipul instrucțiunii proaspăt extrase (decodifica).
4. Dacă instrucțiunea are nevoie de un operand (cuvânt) din memorie, determină unde se găseste 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 sevență de pași este denumită ciclul extrage-decodifică-execută (fetch-decode-execute).

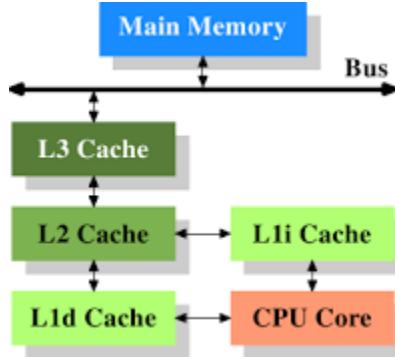
Trebuie precizat că există situații când execuția programului conține ramificări sau salturi, adică instrucțiunile executate nu sunt situate la locații succesive în memorie. În acest caz la punctul 6 de mai sus efectul instrucțiunii este ca în registrul numărator de program este introdusa adresa instrucțiunii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

Tehnica pipeline

În scopul creșterii vitezei de execuție a instrucțiunilor, procesoarele folosesc tehnica numita pipeline (bandă de asamblare sau conductă). Astfel, execuția unei instrucțiuni este împărțita în mai multe parti, de fiecare parte ocupându-se o componentă hardware dedicată (unitate) a procesorului. Toate unitățile hardware pot să funcționeze în paralel. De exemplu la o bandă de asamblare cu 4 unități (numite și segmente) avem: S1 pentru extragere instrucțiune, S2 pentru decodificare, S3 pentru extragere operanzi și S4 pentru execuție. Funcționarea este prezentată 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 operanzi pentru I1; - pe perioada ciclului 4, unitatea S1 extrage instrucțiunea 4 (I4) din memorie, unitatea S2 decodifica I3, unitatea S3 extrage operanzi pentru I2, iar unitatea S4 executa I1. Astfel, după 4 cicluri, instrucțiunea 1 este executată complet, instrucțiunea 2 este executată în proporție 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 îl crește acestora dimensiunile și costul. Memoria intermediara sau cache (de la frantuzescul cacher, a ascunde) este o memorie de capacitate mai mică dar foarte rapidă. Memoria principală a calculatorului (identificată de obicei ca RAM) are o capacitate foarte mare dar este mai lentă. Ideea de bază a memoriei intermediare este urmatoarea: cuvintele de memorie cele mai frecvent utilizate sunt pastrate în memoria intermediara; doar când UCP nu găsește un cuvânt, îl cauta în memoria principală. Orice procesor are un controller de memorie cache care este responsabil cu aducerea datelor din memoria principală în cea de tip cache. O caracteristica a memoriei cache este raportul dintre cache hit (numărul de accesari reușite la memoria cache, adică atunci când datele de care procesorul avea nevoie au fost gasite în memoria cache) și numărul total de accesari.



Termenul de cache miss se referă la cazurile în care datele solicitate de procesor nu au fost încarcate în prealabil în memoria cache, urmând să fie preluate din memoria principală, mai lenta. De exemplu, un procesor Intel Core Duo 6600 are o memorie intermediară pe două niveluri. Există 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 intermediară unificată, situată tot în interiorul capsulei circuitului integrat care gazduiește UCP, legată de UCP printr-o cale de mare viteza (nivelul doi), de 4 MB.

Ambele memorii rulează la frecvența cu care lucrează și procesorul, adică 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. Aceasta memorie este folosită în comun de toate procesoarele.

Modul protejat

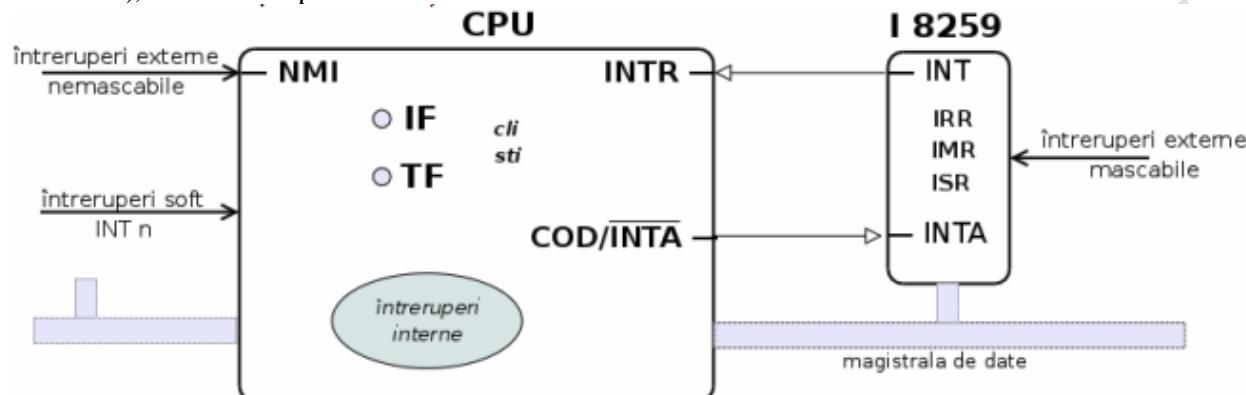
Initial, procesoarele (8088 și 80286) rulau în acest mod real. În acest mod, sistemul de operare (asigură gestionarea resurselor hardware și software ale calculatorului și interfața cu utilizatorul) era de tipul single-tasking, adică numai un program rula la un moment dat. Sistemul de operare folosea instrucțiuni de 16 biți și putea adresa 1 MB de memorie. Nu există protecție incorporată pentru ca un program să nu suprascrăpeste un alt program sau chiar peste sistemul de operare. Astfel, dacă rulau mai multe programe, întregul sistem se putea bloca. O importanță deosebită a procesoarelor, aparută odată cu generația Intel 386, o reprezintă existența modului protejat. Conform acestui mod nu se poate face scrierile la orice adresă din spațiul de memorie. Acest mod este folosit pentru facilitarea unui mod de lucru multitasking foarte bun. Modul multitasking înseamnă că pe un calculator pot rula în același timp mai multe programe aplicative, fără ca ele să interfere unul cu celalalt. Sistemele de operare scrise pentru aceste calculatoare operează cu instrucțiuni pe 32 de biți și pot adresa întregul spațiu de memorie. Un program eronat nu poate deteriora alte programe sau sistemul de operare. Putea fi terminat, în timp ce restul sistemului continua să funcționeze fără a fi afectat.

Întreruperi

Un alt aspect important al procesoarelor este conceptul de întrerupere. În principiu prin întrerupere se înțelege întreruperea executiei normale a unui program datorită 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 cărui adresa depinde de evenimentul care a generat întreruperea. Programul respectiv trebuie să realizeze niște acțiuni în funcție de evenimentul (sursa) care a generat întreruperea. Dupa efectuarea acestui program, procesorul va continua cu rularea programului initial, din punctul în care a fost întrerupt. O alternativă mult mai puțin eficientă este ca UCP să verifice periodic dacă anumite evenimente au avut loc. În acest caz s-ar pierde timp în situațiile în care verificarea se face degeaba. În plus, s-ar putea ca anumite evenimente să apară între două verificări consecutive, iar deservirea lor ulterioară 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 generează întreruperi.

Acestea sunt intreruperi hardware. Intreruperile initiate în urma executiei unui program se numesc intreruperi software (de exemplu la realizarea unei împartiri cu zero).

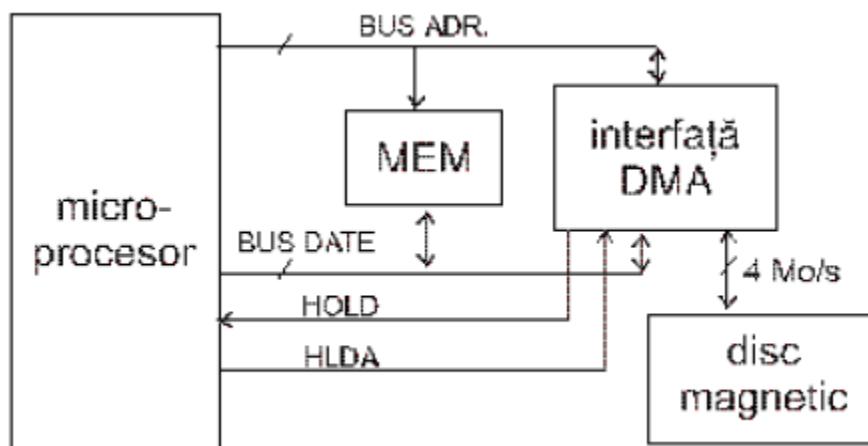
IRQ (Interrupt ReQuest line) reprezintă liniile prin care dispozitivele hardware pot cere intreruperi procesorului. Fiecare dispozitiv (porturi, tastatura, hard disc,...) are alocata o astfel de linie, numita IRQ_i, cu i având valori între 1 și 15. Cele 15 linii sunt conectate la intrările a două circuite speciale numite controlere de intreruperi, legate în cascada, de unde o singura linie este conectată la o intrare specială (pin) al procesorului. Procesoarele din familia Intel acceptă 256 de tipuri de intreruperi (15 hardware), fiecare corespunzând unui anumit eveniment.



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

Tehnica DMA

O alta caracteristica standard a procesoarelor din PC-uri este tehnica DMA. De fiecare dată când are loc un transfer între un dispozitiv de intrare-iesire (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.



Dacă 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, proiectanții 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 spună 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 gasească aici instrucțiuni de executat. Rezulta că la adresa respectiva trebuie să existe memorie ROM, care păstrează informația în permanentă. Deoarece aceasta adresa este cu exact 16 octeți mai jos de sfârșitul 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 mică.

Un **program** reprezintă o lista secvențială de instrucțiuni. Un procesor poate să facă în principal urmatoarele trei tipuri de instrucțiuni:

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

Există mai multe tipuri de limbi de programare:

- Limbi de nivel înalt (C, Pascal, BASIC)
- Limbi de nivel coborât, numite limbi de asamblare.

Orice program trebuie translatat într-un limbaj mașină pe care îl înțelege procesorul. Acest lucru este realizat de programe numite compilatoare, interpretoare sau asambloare.

Exemple de instrucțiuni în limbaj de asamblare:

- LOADA *mem* – încarcă registrul A cu conținutul locației de memorie de la adresa *mem*.
- LOADB *con* – încarcă registrul B cu constanta *con*.
- SAVEB *mem* – salvează registrul B în locația 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 într-o primă abordare ca o stivă de locații binare (cuvinte), fiecare cuvânt fiind caracterizat de o adresă binară unică.

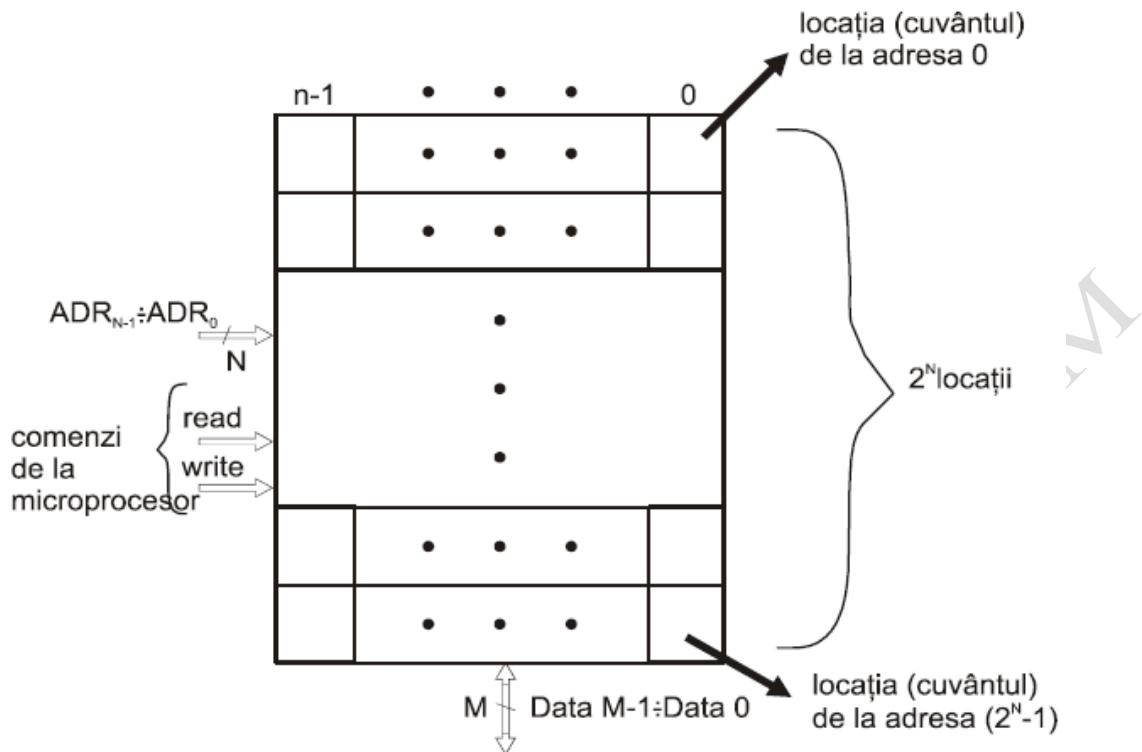


Figura 4.4. Schemă generică de memorie

În general $M=8, 16, 32, 64 \Leftrightarrow$ lărgime bus date microprocesor (microsistem pe M biti)

Memoria este caracterizată prin 2 parametri de bază:

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

Fireste, se doresc capacitați cât mai mari și latente cât mai mici, cerințe în general contradictorii.

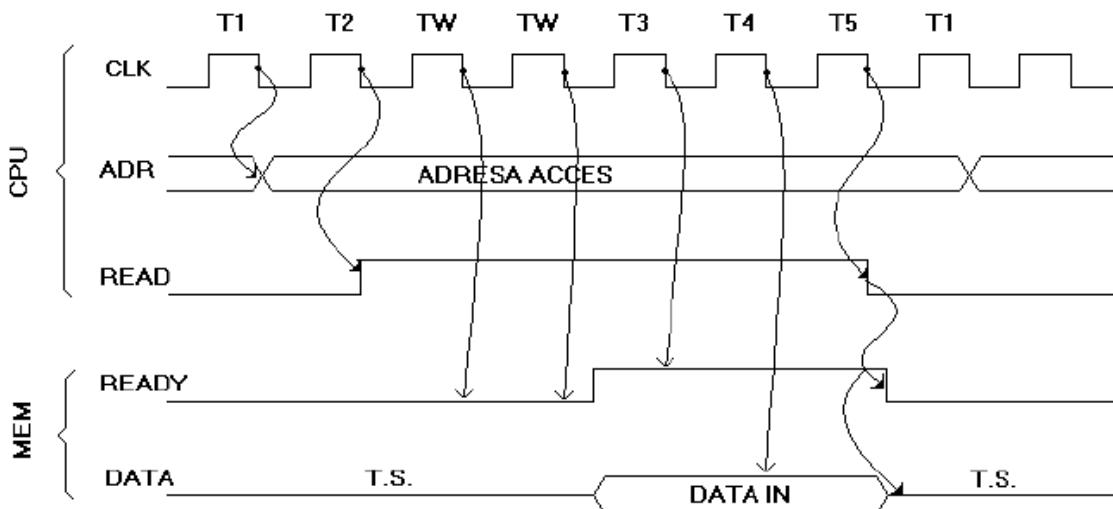


Figura 4.5. Ciclu extern citire memorie

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

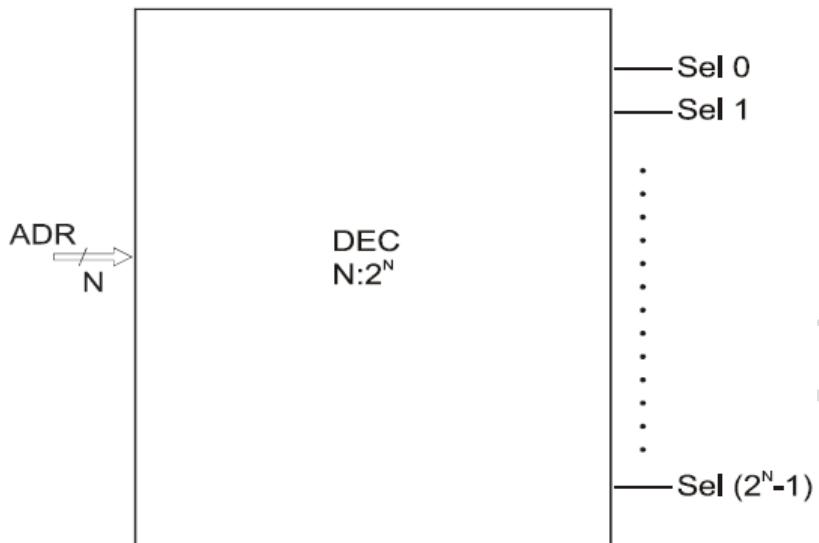


Figura 4.5. Decodificator N:2^N

Cu 10 biți de adrese => 2¹⁰ cuvinte = 1k cuvânt (kilo)

Cu 20 biți de adrese => 2²⁰ cuvinte = 2¹⁰ k cuvânt = 1M cuvânt (mega)

Cu 30 biți de adrese => 2³⁰ cuvinte = 2¹⁰ M cuvânt = 1G cuvânt (giga)

Cu 40 biți de adrese => 2⁴⁰ cuvinte = 2¹⁰ 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 principală

Memoriile EPROM sunt memoriile rezidente care păstrează conținutul și 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 așa 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 memorilor RAM este neprecizabil imediat după alimentare. Prin urmare, imediat după RESET conținutul PC-ului este inițializat și 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 microsistemu, 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 $t_0 = 1 \text{ ns} \div 15 \text{ ns}$, 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 densități mari de integrare (64 Mbiți – 256 Mbiți / chip) și timpi de acces “relativ rezonabili” $t_0 = 30 \text{ ns} \div 60 \text{ ns}$.

Totuși, decalajul între timpul de acces ridicat al acestor memorii și viteza mare de execuție 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 intersecția liniei cu coloana selectată.

2. Necesitatea regenerării memoriei DRAM.

Bitul de informație 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 \Rightarrow deci că periodic el trebuie reîncărcat (*refresh*, regenerare).

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

De exemplu, considerând $N=128$ rânduri (DRAM 4116, 4164) \Rightarrow rata de regenerare $2 \text{ ms}/128 \sim 15,6 \mu\text{s}$.

Prin urmare vom avea accese la DRAM din 2 părți:

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

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

Standardizarea bus-urilor și a protocolelor 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

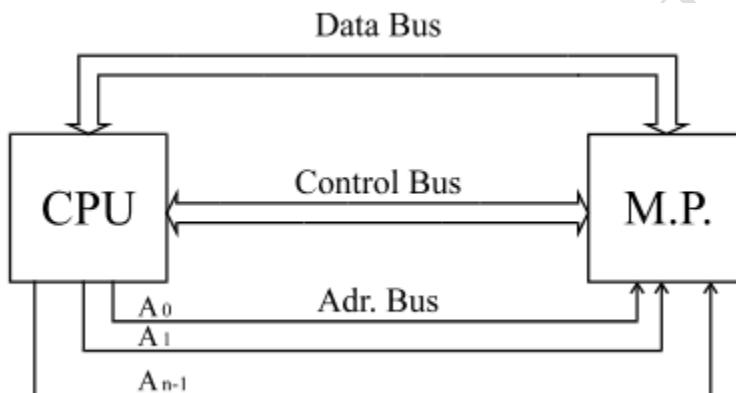
Memoria principală este o memorie operativă adică aici se încarcă programele lansate în execuție și datele aferente.

- 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:

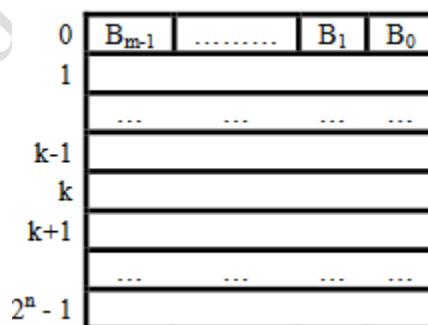
- Memoriile cu acces aleator sunt caracterizate prin faptul că fiecare locație poate fi accesată independent, timpul de acces pentru oricare locație fiind același - independent de poziția locației
- Fiecare celulă conține linii de date (citire sau scriere - numite linii de bit la majoritatea celulelor bipolare sau unipolare) și linii de selecție.

Unitatea de memorie în calculatoarele contemporane se realizează cu ajutorul modulelor integrate. Legătura 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 unidirecțională, cea de date este bidirecțională, iar cea de control conține linii de comandă, dar și linii informație de stare. Memoria este organizată pe locații:



O locație are m ranguri notate B_{m-1}, ..., B₁, B₀.

Citirea și scrierea memoriei se face paralel, deci, se scrie sau se citește un cuvânt pe m biți. 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 definește o anumită adresă. Deci, legătura între conținutul magistralei și o anumită locație se face printr-un proces de decodificare. Dacă de adrese are n linii atunci multimea adreselor este 2^n .

$$A = \{0, 1, \dots, 2^n - 1\}$$

unde A este spațiul adreselor.

Pe de altă parte, dându-se o geometrie a memoriei rezultă mulțimea locațiilor fizice în care sunt memorate cuvinte care conțin m biți.

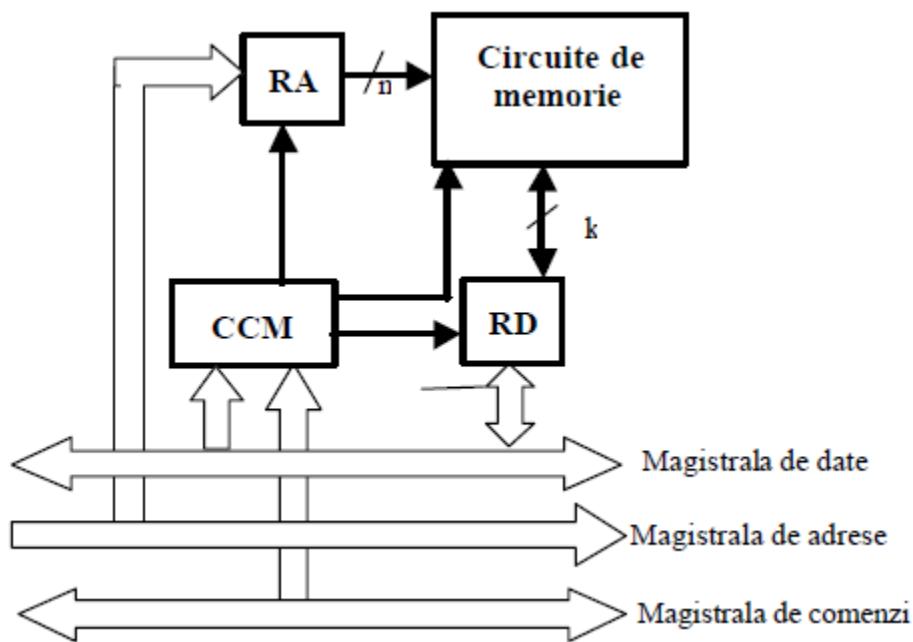
Atunci când lățimea cuvântului memorie nu este suficientă pentru reprezentarea datei, atunci se vor utiliza mai multe locații consecutive. Dacă de exemplu, o dată se reprezintă pe două cuvinte în memorie (UIA), atunci pentru a reprezenta data, se va folosi prima locație pentru reprezentarea cuvântului mai puțin semnificativ (de la adresa j), iar locația următoare (adresa j+1) pentru cuvântul mai semnificativ.

Procesoarele moderne au facilități de gestiune a memoriei. Operațiile de gestiune se referă la implementarea memoriei virtuale, protecția memoriei etc.

Unitatea de gestiune a memoriei (Memory Management Unit - MMU) poate fi încorporată în unitatea de control din CPU sau poate fi externă unității de control. În particular, în cazul microprocesoarelor pot fi încorporate în microprocesoare sau sunt livrate ca un modul separat - modulul MMU).

Memoria principală (MP) conține informația în curs de prelucrare cum ar fi programul în execuție și date cu care operează acesta. Pentru a se realizea o viteza ridicată de transfer MP este conectată direct la magistralele sistemului. Adresa locației referite este pastrată în **Registrul de Adrese (RA)**, iar datele citite/scrisse din/în memorie se păstrează în **Registrul de Date (RD)**. RA și RD vor mai fi întâlnite și la microprocesoare ca registre tampon. Prezența lor este absolut necesară deoarece magistralele nu trebuie reținute pe toată durata transferului în/din memorie.

Circuitele de control ale memoriei (CCM) generează semnalele de selecție 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. Așadar, 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 latența 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 performanță, 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 definește un acces al CPU cu hit în cache, un acces care găsește o copie în cache a datei accesate. Un acces cu miss în cache este unul care nu găsește 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 definește ca parametru de performanță 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 memorilor 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 spațială, există o mare probabilitate ca o dată situată în imediata vecinătate a unei date accesate curent de către CPU, să fie și ea accesată în viitorul apropiat (pe baza acestui principiu statistic se aduce din MP în cache un întreg bloc și nu doar strict cvântul dorit de către CPU). O buclă de program – structură esențială în orice program – exemplifică foarte clar aceste principii și justifică eficiența conceptului de cache.

Din punct de vedere arhitectural, există 3 tipuri distințe de memorii cache în conformitate cu gradul de asociativitate: cu mapare directă, semiasociativă și total asociativă.

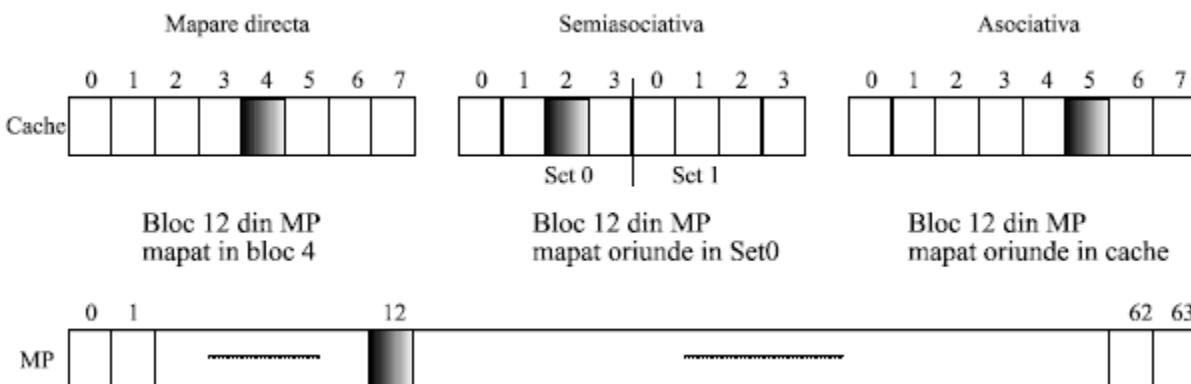


Figura 4.6. Scheme de mapare în cache

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 încărcând toate se mapează pe blocul 4 din cache. Prin urmare, o buclă de program care ar accesa alternativ blocurile 20 și 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 ușor de

implementat), FIFO (sau *round-robin*, se evacuează blocul cel mai vechi din cache. Contorul aferent se încarcă doar la încărcarea blocului în cache și nu la fiecare hit per bloc, ca la algoritmul LRU) și 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 anticipatează pe baze euristice utilitatea de viitor a blocurilor memorate în cache, evacuându-l pe cela mai puțin valoros. Deși acești algoritmi depășesc în mod normal cadrul acestui curs de inițiere în domeniul microprocesoarelor, în continuare se va prezenta totuși unul, integrat în arhitectura numită *Selective Victim Cache*.

Este evident că într-un astfel de cache rata de interferență se reduce odată cu creșterea gradului de asociativitate (N “mare”). Aici, de exemplu, blocurile 12, 20, 28 și 36 pot coexista în setul 0. Prin reducerea posibilelor interferențe ale blocurilor, creșterea gradului de asociativitate determină îmbunătățirea ratei de hit și deci a performanței 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 complicații structurale și deci la creșterea timpului de acces la cache și implicit la diminuarea performanței globale. Optimizarea gradului de asociativitate, a capacitatii cache, a lungimii blocului din cache etc., nu se poate face decât prin laborioase simulări software, variind toti acești parametrii în vederea minimizării ratei globale de procesare a instrucțiunilor [instr./cicli].

În fine, memoriile cache total asociative, implementează practic un singur set permitând maparea blocului practic oriunde în cache. Ele nu se implementează deocamdată în siliciu datorită complexității deosebite și a timpului prohibit de căutare. Reduc însă (practic) total interferențele blocurilor la aceeași locație 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), cercetatorul *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 naștere 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 informații de stare asociate blocurilor din cache. Biții de stare conțin informații 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 același bloc. Această schemă, numită excludere dinamică, reduce miss-urile de conflict în multe cazuri. O predicție greșită implică un acces în nivelul următor al ierarhiei de memorie contrabalanșând eventuale câștiguri în performanță. Schema este mai puțin eficace cu blocuri mari, de capacitate tipice cache-urilor microprocesoarelor curente.

Pentru a reduce numărul de interschimbări dintre cache-ul principal și victim cache, *Stiliadis* și *Varma* au introdus un nou concept numit *selective victim cache* (SVC).

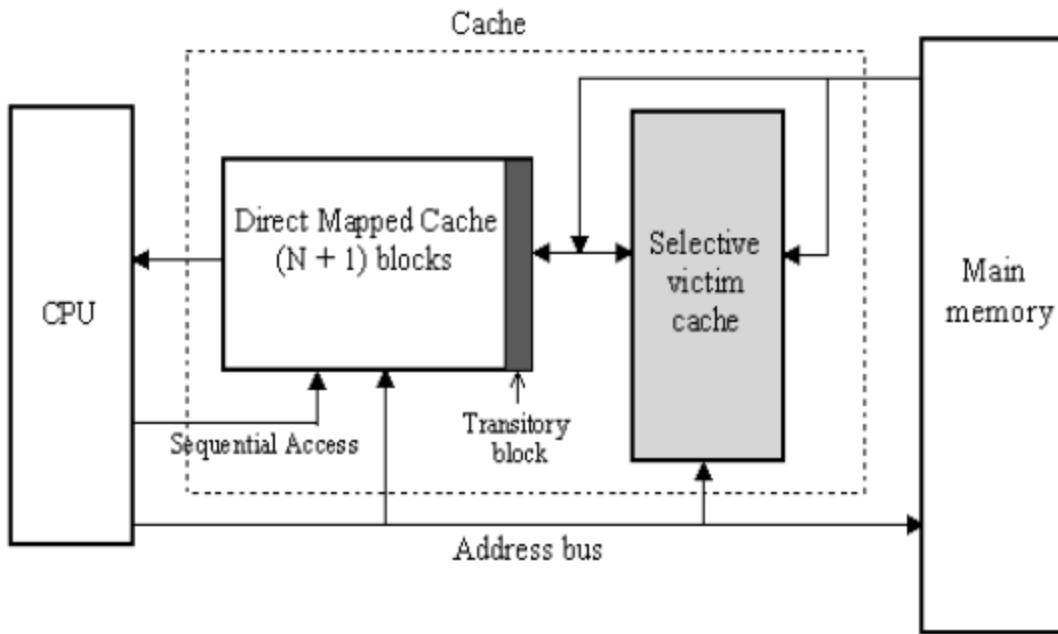


Figura 4.7. Ierarhia de memorie pentru schema SVC

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 predicție euristic bazat pe istoria folosirii sale. Blocurile care sunt mai puțin probabil să fie accesate în viitor sunt plasate în SVC și nu în cache-ul principal. Predicția este de asemenea folosită în cazul unui miss în cache-ul principal pentru a determina 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 performanță

Metricile de performanță 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 referințe 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 și victim cache. Timpul mediu de acces la ierarhia de memorie ia în calcul și aceste penalizări și de aceea este un bun indicator al performanței 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:

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

Unde R este numărul total de referințe generate de programele de tip trace, M_d 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 îmbunătățiri ale ratei de miss precum și ale timpului mediu de acces la memorie, atât pentru cacheuri mici cât și pentru cele mari (4 Kocteți - 128 Kocteți). Simulații făcute pe trace-uri de instrucțiuni a 10 benchmark-uri SPEC'92 arată o îmbunătățire 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 și victim cache s-a redus cu aproximativ 70%.

4.3.4. Memoria virtuală (secundară)

Este descris în capitolul 7.1.

4.3.5. Memoria video

Din punct de vedere constructiv, memoria video este de tip RAM dinamic (DRAM), în mai ultime variante. Memoria RAM dinamica are avantajul densității mari de integrare și deci al raportului capacitate/preț mare, dezavantajul fiind, în schimb, necesitatea reamprospătării periodice a informației conținute [DRAM refresh]. Astfel încât, în intervalul de reămprospătare [duty cycle], nu se pot face accesele normale de citire/scriere la memorie.

Un alt dezavantaj il reprezintă faptul că la un moment dat se poate face doar un singur acces la memorie (citire sau scriere). La nivelul placii grafice au însă loc două tipuri de transfer de date: transferul informației de la unitatea centrală a calculatorului, la memoria video; și transferul datelor din memoria video spre registrii de deplasare, pentru afișarea imaginii pe ecran. Deci capacitatea totală de acces la memorie va trebui împărțita la 2.

La adresabilitatea mari de pixel și la adâncimi mari de culoare, spre monitor vor trebui transferate cantități enorme de date pentru ca afișarea să nu se altereze (rata de reămprospătare a imaginii [monitor refresh rate] va fi mare), ceea ce cu unele tipuri de memorie video nu se reușește.

În încercarea de a elimina dezavantajele enumerate mai sus (și deci de a crește capacitatea totală de acces la memoria video), s-au conceput diverse alte variante de memorie RAM, printre care se remarcă prin performanță, varianta VRAM [Video RAM].

Memoria VRAM este o memorie DRAM de tip dual-port, adică suportă două accese simultane (citire sau scriere), dublând astfel capacitatea totală de acces la ea. Necesitatea reamprospătării informației din memorie se păstrează, fiind tot RAM dinamică. În schimb conține un registru de deplasare propriu, cu încarcare paralela (echivalentul unui ciclu de citire), de capacitate foarte mare (un rand întreg din matricea memoriei). Acest registru poate fi sincronizat separat de accesele normale la memorie, și este utilizat pentru reamprospătarea video. Astfel, dacă la memoria DRAM clasică, reamprospătarea imaginii afișate 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 (Graphics Processing Unit) și circuitele de afișaj, se folosește adesea memorie specială multi-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, de 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 utilizează 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/ieșire (organizare, conectare, interfețe).

Sistemul de intrare/ieșire permite introducerea/extragerea informațiilor și 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 tastură, *mouse*, etc.
- **Dispozitive de ieșire (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 și conectează **UCP** cu memoria **RAM**.
- **Magistrale de Date (MD)** transmit date și instrucțiuni și conectează **UCP**, memoria **RAM** și celelalte componente ale sistemului.

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

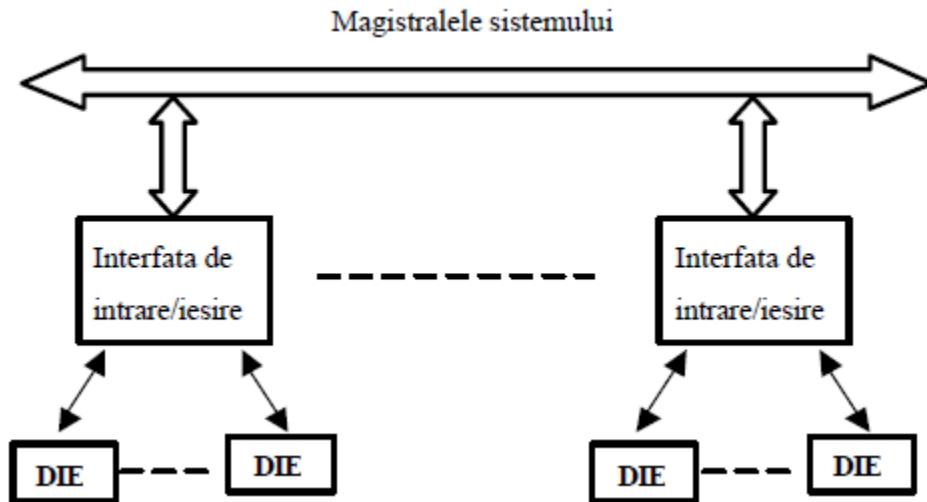
- controller-e de intrare/ieșire
- comunicatia sistemului de operare cu dispozitivele de intrare/ieșire = programare /O(intreruperi/drivere)
- interfetele puse la dispozitia utilizatorului

Un dispozitiv I/O este orice dispozitiv care permite introducerea sau extragerea de informație 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) asigură introducerea și extragerea informației 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 conectează la unitatea centrală prin intermediul *interfețelor de intrare/ieșire*, a caror complexitate variază foarte mult de la simple *registre* până la *controlere inteligente realizate cu microprocesor*.



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

Dispozitive de ieșire.

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

- *dispozitive hardcopy* – exemplu: imprimante cu ace, laser, cu jet de cerneala, termica; plottere cu penita sau electrostatic - creează imaginea fixă, 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 directă (DVST), dispozitive de afisare cu cristale lichide (LCD), ecrane cu plasma, dispozitive de afisare electroluminiscente, etc. – sunt caracterizate de o anumită frecvență de reîmprospătare a imaginii pe un ecran, putându-se obține și efecte dinamice.

Dispozitive de intrare.

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

Execuția unui proces de intrare/ieșire

Viteză de lucru a perifericelor diferă în funcție de performanțele lor tehnologice (foarte mică la dispozitivele exclusiv mecanice) dar este mult mai mică decât a unității centrale, care funcționează 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-iesire, ea așteaptă 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-iesire 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 funcțională a calculatoarelor a apărut odată cu generația a două de calculatoare sub numele de *canal de intrare-iesire*. 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 comunicare și la microcalculatoare – de extensie a magistralei. În timp ce procesorul specializat în operații de intrare-iesire controlează schimbul de date între memorie și periferice, blocul de comandă poate superviza execuția altor operații, dintr-un alt program. Modul de lucru cu o unitate centrală și mai multe programe rezidente în memorie spre execuție se numește *multiprogramare* sau *multitasking*. Programele aflate în memoria internă a unui calculator care funcționează în regim de multitasking se găsesc în diverse stări: pot astepta terminarea unei operații de intrare-iesire, pot aștepta 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 implementați în sistemul de operare.

După terminarea unei operații de intrare-iesire, unitatea de comandă este anunțată, 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

Interfețele și dispozitivele de intrare-iesire ale unui calculator personal sunt adaptate pentru modul de lucru cu un singur utilizator. De exemplu: există o singură intrare de tastatură și se folosește o singură interfață de afișare. Informațiile afișate se păstrează într-o zonă de memorie direct accesibilă procesorului. Astfel pot fi implementate aplicații care necesită o viteza mare de schimbare a informațiilor afișate. În cazul sistemelor multi-utilizator interfața utilizator este asigurată de mai multe dispozitive inteligente de tip display care încorporează o tastatură și un dispozitiv de afișare; legatura cu calculatorul gazdă se realizează prin canale seriale. Viteza de schimbare a informațiilor afișate este limitată de viteza relativ mică a canalului serial.

Configurația tipică de dispozitive și interfețe de intrare-iesire 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, interfață de rețea, sistem audio (interfață soundblaste și boxe).

Tastatura este compusă dintr-o matrice de contacte (taste) și un microsistem bazat pe un microcontrolor. Microsistemul are rolul de a detecta eventualele contacte închise (taste apăsat) și de a transmite această informație sistemului de calcul. La calculatorul IBM-PC comunicația dintre tastatură și calculator se realizează printr-o legătură 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 ușual 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 spațiului de pe disc se poate utiliza pentru extinderea memoriei interne prin tehnica denumită memorie virtuală.

Unitatea de disc optic permite citirea informațiilor digitale înregistrate pe un suport optic. Ca și funcționalitatea această unitate se situează între unitatea hard-disk și cea de disc flexibil: informațiile înregistrate pe discul optic pot fi transportate între calculatoare. Scrierea se face secvențial pe tot discul sau pe o porțiune 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 afișa rezultatele execuției unui program precum și informațiile de operare. Partea "inteligentă" a ieșirii video este interfața video. Există mai multe standarde pentru implementarea interfeței video: MGA, CGA, EGA, VGA și SVGA. Diferențele constau în: rezoluție pe orizontală (320 - 1024 puncte), rezoluție pe verticală (200800 puncte), paleta de culori (2-256 culori) și facilități de citire-scriere a informațiilor grafice. Primele 3 variante de interfață generează semnale digitale, iar ultimele

semnale analogice. De obicei tipul monitorului trebuie să fie în concordanță cu tipul interfeței 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 secvențe specifice de comandă – ESC), sau în mod grafic (caz în care informațiile transmise descriu prin puncte o imagine grafică). Există mai multe standarde pentru limbajul de comunicație între calculator și imprimantă (ex: EPSON, IBM, Hewlett-Packard, etc.). Acestea diferă mai ales prin modul grafic. O imprimantă poate să emuleze (să înțeleagă) mai multe tipuri de limbaje.

Interfața de rețea permite cuplarea calculatorului într-o rețea locală (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 înfășurat (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 și programele aplicative care au o interfață grafică utilizator bazată pe meniu. Mișcările mouse-ului sunt transformate în mișcări ale unui cursor pe ecran cu ajutorul unui program de interfață (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.

Interfața 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. Interfața este alcătuită dintr-o placă de sunet (sound-blaster) și boxe. O ieșire a unității de disc optic este conectată la placa de sunet pentru a permite redarea discurilor de muzică.

5. Nivelul fizic

5.1. Microprocesorul, intreruperile, magistralele de comunicație (magistralele sincrone și asincrone, arbitrajul magistralei)

O magistrală (bus) definește 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 magistrală conține mai multe tipuri de specificații:

- mecanice - conectorii, cablurile, topologia de cuplare;
- electrice - nivelele de tensiune, curentii impedanțe;
- funcționale - descrierea semnalelor.

Unele calculatoare au impus un standard de magistrală 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 comunicație folosită în comun de mai multe blocuri funcționale și este realizată fizic de un set de fire conductoare. O magistrală trebuie să fie versatilă și să aibă un preț scăzut. Datorită versatilității, odată definită o schemă de conexiuni, este foarte ușor să se adauge noi dispozitive în sistem sau acestea să poată fi ușor mutate dintr-un calculator în altul, dacă cele două calculatoare sunt construite cu același tip de magistrală.

Pe de altă parte, folosirea unei magistrale poate fi dezavantajoasă datorită limitărilor de viteză de transfer a datelor (throughput) pe care aceasta le poate crea. Viteza la care poate lucra o magistrală este impusă în primul rând de factori fizici ca: lungimea traseelor și numărul dispozitivelor atașate (acestea din urmă aduc constrângeri atât ca număr efectiv cât și ca diversitate din punct de vedere al ratelor de transfer și a timpilor de întârziere).

O magistrală este organizată în grupe de semnale care au aceeași funcționalitate:

- 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 informației de pe liniile de date. Setul semnalelor de date este folosit pentru a transporta informația între blocurile funcționale ale calculatorului de la sursă la destinație (de la inițiator la țintă). Informația poate fi date, comenzi sau adrese. Adesea magistralele de date sunt despărțite în două seturi pentru a separa informația ce reprezintă adrese. Astfel există și o magistrală de adrese pentru ca datele și adresele să poată fi vehiculate într-o singură tranzacție de la sursă la destinație. După funcționalitatea semnalelor magistralele pot fi magistrale de control, magistrale de date și magistrale de adrese. O clasificare a magistralelor poate fi făcută după 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 scurtă și de mare viteză. După cum îi spune numele, este concepută 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 să o necesitea 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 magistrală procesor memorie, fie o magistrală de fundal.

Magistralele de fundal sunt proiectate astfel încât atât procesorul, memoria cât și dispozitivele I/O să folosească aceeași cale de comunicație. La acest tip de magistrală se face un compromis pentru a satisface atât necesitățile comunicației între procesor și memorie cât și ale comunicației dintre dispozitivele I/O, sacrificând desigur performanța sistemului.

Magistralele I/O și magistralele de fundal sunt definite prin specificații, deseori publice, iar magistralele procesor memorie sunt specifice (proprietatea producătorului).

O altă clasificare a magistralelor se poate face după metoda de comunicație folosită:

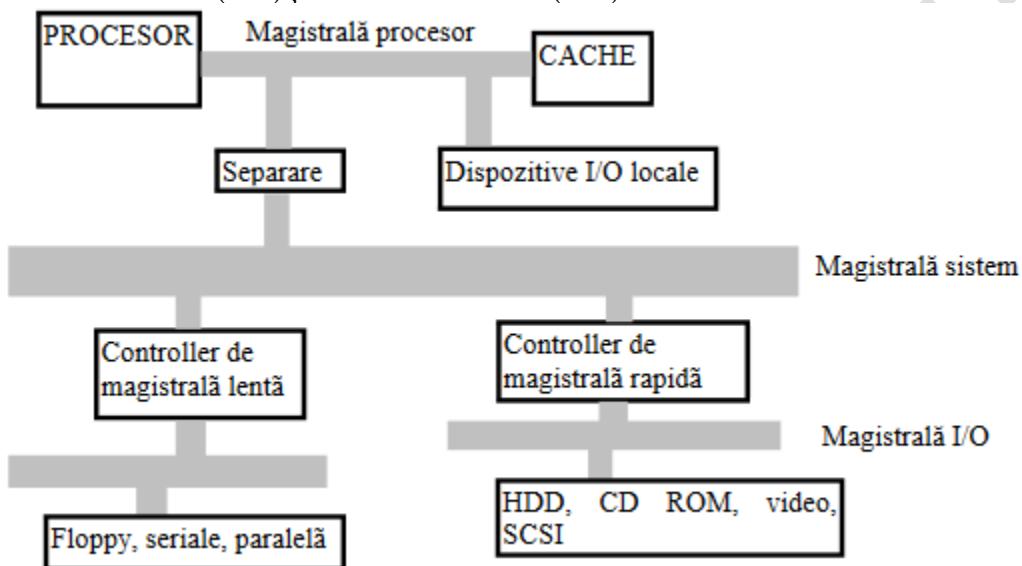
- magistrale sincrone;
- magistrale asincrone.

Magistrale sincrone dispun de o linie de tact în liniile de control, condusă de un oscilator cu quart, între 5-133MHz. Toate transferurile au loc după un protocol fix care este raportat la semnalul de tact, într-un număr întreg de cicluri, numite cicluri de bus. Pentru că protocolul este predeterminat și logica implicată este simplă, magistralele sincrone pot fi foarte rapide. Avantajul magistralelor sincrone este simplitatea deoarece nu există un dialog (protocolele pot fi implementate prin automate finite simple). Dezavantajul este că dacă un transfer este mai scurt el trebuie totuși să dureze un număr întreg de cicluri de bus, ca urmare îmbunătățirile tehnologice ale plăcilor I/O nu duc la mărire vitezei. Într-un cuvânt, dacă avem placă lente și rapide pe o magistrală sincronă, transferurile au loc la viteza celei mai lente. Un alt dezavantaj este că, 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 ușor la o mare varietate de dispozitive conectate. Coordonarea transferurilor de la sursă la destinație se face pe baza unui protocol de dialog conversațional (handshaking). Un astfel de protocol constă într-o serie de pași parcursi de sursă și destinație astfel încât trecerea la pasul următor se face numai cu acordul ambelor părți. Pentru implementarea unui astfel de protocol sunt necesare liniile de control suplimentare (DATA REQUEST, DATA READY etc.).

Așa cum este firesc, cel puțin două blocuri sunt conectate la o magistrală. Pentru a evita haosul pe magistrală, controlul acestora este făcut de un MASTER de magistrală. Un MASTER trebuie să poată să inițieze cereri pentru acces la magistrală și să poată să controleze toate tranzacțiile. 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 tranzacțiile este inefficientă, de aceea există alternativa de magistrală cu mai mulți MASTER. Într-o astfel de magistrală fiecare MASTER este capabil să inițieze un transfer. În aceste condiții trebuie să existe un mecanism de arbitrage a accesului la magistrală 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 magistrală cu mai mulți MASTER conține linii speciale pentru arbitrage. Există mai multe metode de arbitrage. Toate metodele respectă principiul priorității și al corectitudinii (fairness). Pentru cuplarea diferitelor subsisteme la magistrale trebuie să se ia în considerare diafonie, reflexii, impedanțe liniilor etc. Pentru legarea semnalelor la liniile de transmisie se folosesc circuite driver, așa cum sunt de exemplu 74HCxxx sau 74AHCxxx pentru CMOS și 74ABTxxx pentru TTL. Structura tipică a unei unități centrale arată că magistrala este ierarhizată, în funcție de viteză, pe două nivele; unul de viteză mare (PCI) și unul de viteză mică (ISA).



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

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

Legătura între procesor și EP (Echipamente Periferice) se realizează prin canale I/O (de intrare/iesire) prin intermediul magistralei. Evoluția în timp a canalelor I/O este în același timp o evoluție a creșterii complexității și performanțelor. Pot fi enumerate următoarele etape:

1. CPU controlează direct EP;
2. Este adăugat un modul I/O (o interfață serială sau paralelă, programabile). CPU comandă EP prin transfer programat (direct sau prin interogare);
3. Aceeași configurație ca la 2, dar transferul are loc prin întreruperi;
4. Modulul I/O are acces direct la memorie prin DMA. Modulul poate muta informația direct în memorie, accesul CPU fiind necesar doar la începutul și sfârșitul transferului;
5. Modulul I/O folosește 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 intervenție minimă din partea CPU. Memoria locală poate fi folosită și ca buffer de date, realizând astfel o rată de transfer mare.

Evoluția microcontrollerelor integrate a fost paralelă cu cea a procesoarelor. Dacă la procesoare s-a urmărit o creștere a vitezei de prelucrare prin creșterea tactului, creșterea mărimii memoriei CACHE (uzual 256K integrat), lărgimea busului de date și adrese, la microcontrollerere s-a urmărit integrarea de cât mai multe subsisteme utile (memorie EPROM, RAM, convertor AD și DA). Dacă prețul unui procesor nu a scăzut sub câteva zeci de dolari (PII/450MHz), prețul unui microcontroller poate ajunge la câțiva dolari, ceea ce încurajează eforturile proiectanților de a realiza module I/O inteligente cu microcontroller, sau alte aplicații cu microcontroller.

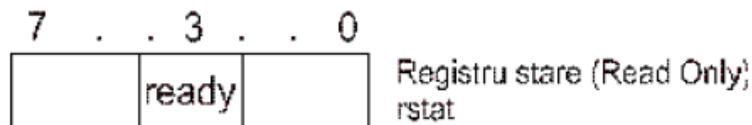
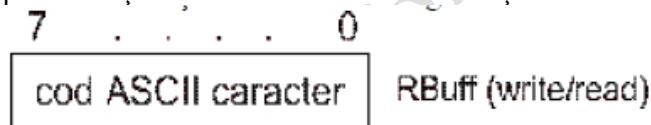
Conexiunea între un modul I/O și unitatea centrală poate fi:

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

Dacă o legătură punct cu punct se numește indiscretabil interfață, o legătură multipunct se poate numi atât interfață cât și magistrală. Interfața 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 inițializa transferul cu perifericul decât în momentul în care bitul de stare semnifică faptul că perifericul este pregătit pentru transfer (nu lucrează la un transfer inițiat anterior). Să considerăm de exemplu interfața cu o tastatură. Această interfață trebuie să conțină minimum 2 registre.



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

Exemple:

$$\text{“A”} = 41\text{h} \Leftrightarrow \underline{0100.0001}$$

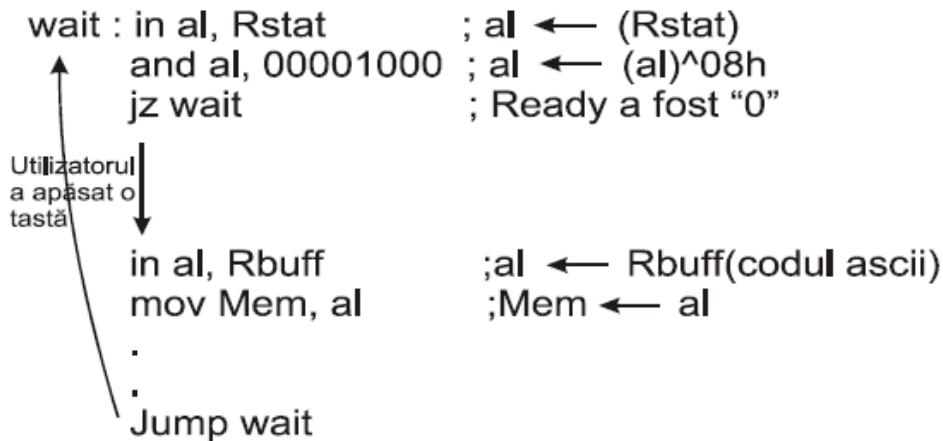
$$\text{“a”} = 61\text{h} \Leftrightarrow \underline{0110.0001}$$

$$\text{“0”} = 30\text{h}$$

$$\text{“ ”} = 20\text{h}$$

Bitul *Ready* din registrul de stare este un bit de tip *Read Only* cu următoarea semnificație: 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 resetă automat odată cu preluarea codului din Rbuff de către microprocesor.

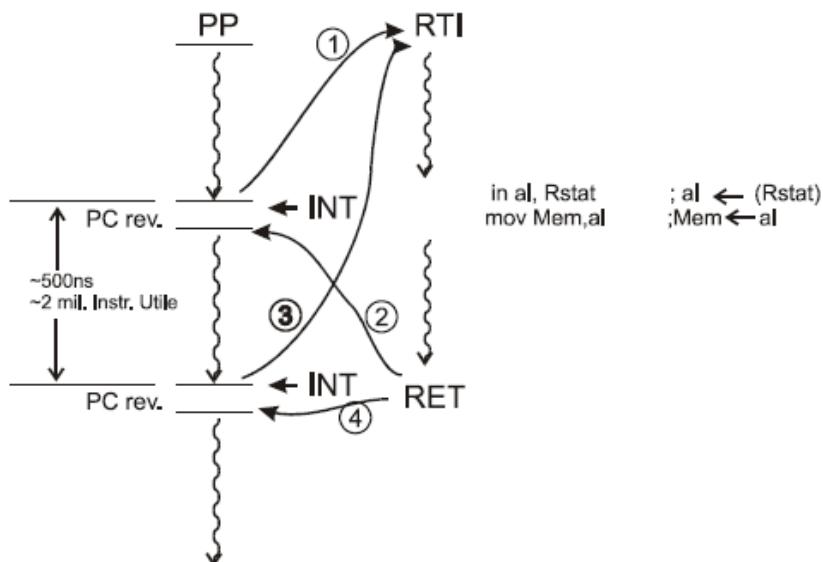
Un program - absolut principal - de gestiune a tastaturii s-ar scrie ca mai jos:



Dezavantajul acestei metode constă în faptul că microprocesorul așteaptă 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 instrucțiune a microprocesorului durează cca. 250 ns (vezi justificarea anterioară) ⇒ că “pierde” $500\text{ ms} / 250\text{ ns} = 2$ milioane instrucțiuni în bucla de așteptare în loc să execute instrucțiuni utile. Acest dezavantaj este eliminat de metoda următoare de comunicare procesor-interfață.

Modul de lucru prin întreruperi hardware

Se bazează pe generarea unui semnal de întrerupere INT de la interfață (port) spre microprocesor ori de câte ori acesta dorește un serviciu de la microprocesor. Ca urmare a recepționării semnalului INT microprocesorul va abandonă programul principal (PP) urmând să intre într-o astfel numită rutină **tratare a întreruperii** în care va satisface cererea interfeței. 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 instrucțiuni din PP executate. În cazul exemplului cu tastatura anterior considerat, interfața va genera întreruperea INT ori de câte ori utilizatorul a apasat o tastă, adică registrul RBuff este “plin”, deci conține codul (ASCII, Unicode etc.) al caracterului tastat.



Așadar RTI după ce execută serviciul necesar perifericului (în cazul acesta preluare și 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 aștepte inutil ca în cazul 1.

Totalitatea acțiunilor executate de către microprocesor din momentul apariției semnalului de întrerupere INT până în momentul procesării primei instrucțiuni din RTI formează așa numitul **protocol hardware de acceptare a întreruperii** (săgețile 1 și 3 din figură). În principiu acest protocol se desfășoară în următoarele etape succesive:

1. Odată sesizată întreruperea INT de către microprocesor acesta își va termina instrucțiunea în curs de execuție după care, dacă anumite condiții 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 aferent instrucțiunii în curs de execuție.

2. Recunoașterea întreruperii: microprocesorul va inițializa așa numitul **ciclu de achitare a întreruperii**. Pe parcursul acestui ciclu extern va genera un semnal de răspuns (achitare) la întrerupere INTACK (*interrupt acknowledge*) spre toate interfețele de intrare - ieșire. Ca urmare a recepționării INTACK interfață care a întrerupt va furniza microprocesorului prin intermediul bus-ului de date un așa 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 și conform unui algoritm care diferă de la microprocesor la microprocesor, acesta va determina adresa de început a RTI, adresă ce va urma să o introducă în PC. Firește, la VI diferenți 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 să la finele RTI unde să revină exact în PP.

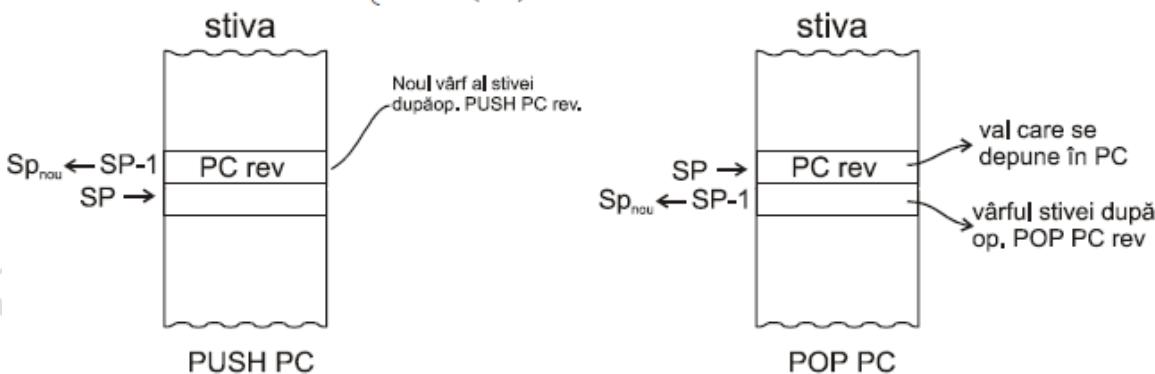
Memoria stivă este o zonă de memorie RAM caracterizată la un moment dat de așa numitul vârf al stivei **adică de ultima locație ocupată din stivă**. Acest vârf al stivei este pointat (adresat) permanent de conținutul unui registru special dedicat, existent în orice microprocesor modern, numit SP (*stack pointer*). În memoria stivă sunt posibile 2 tipuri de operații:

operația PUSH Reg care se desfășoară astfel:

$$\left\{ \begin{array}{l} SP \leftarrow (SP) - 1 \text{ (cuvânt = octet)} \\ (\text{Reg}) \rightarrow \text{Mem | adr. SP} \end{array} \right.$$

operația POP Reg:

$$\left\{ \begin{array}{l} (\text{Reg}) \leftarrow \text{Mem | adr. SP} \\ SP \rightarrow (SP) + 1 \end{array} \right.$$



Stiva este o memorie de tip LIFO (*last in first out*) și care spre deosebire de PC în procesarea secvențială, "crește" (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 și 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 regiștrilor 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 (instrucțiuni POP) acești regiștri.

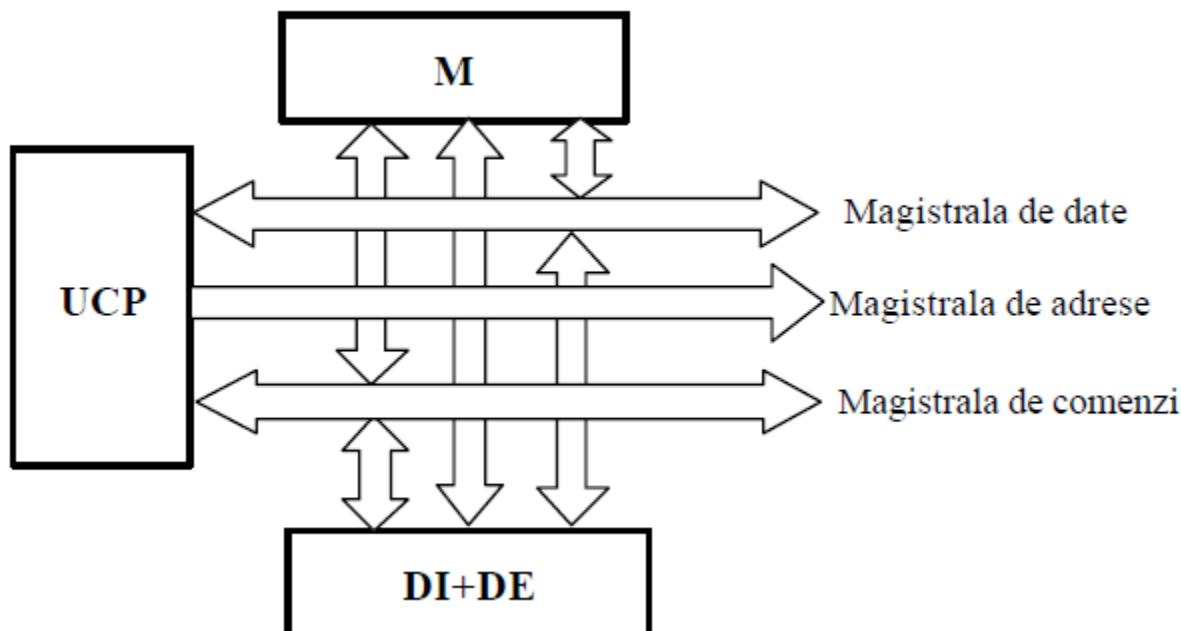
5.2. Magistrala procesorului, magistrala de date, magistrala de adrese, magistrala I/O

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 și RAM, porturi I/O lente, porturi I/O rapide – interfețe DMA, program încărcător - POST, programe BIOS) este prezentată mai jos:



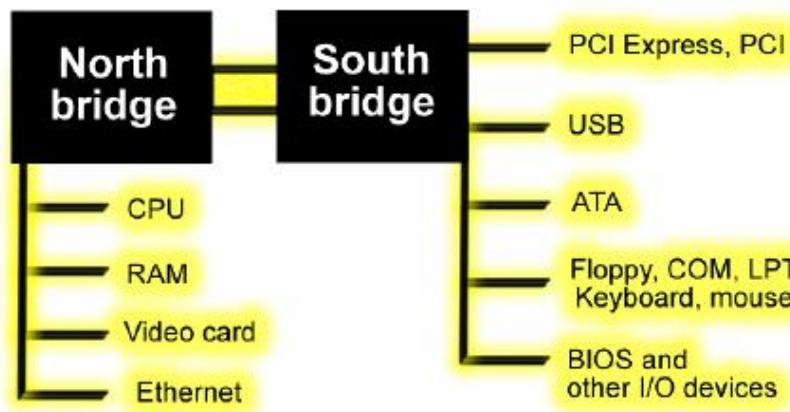
Liniile de comunicație între cele patru module sunt grupate în **magistrale**. Informația vehiculată date, adrese și semnale de comandă, astfel încât în structura calculatorului există trei magistrale, respectiv:

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

5.3. Exemple și comparatii.

Magistrale în calculatoare personale

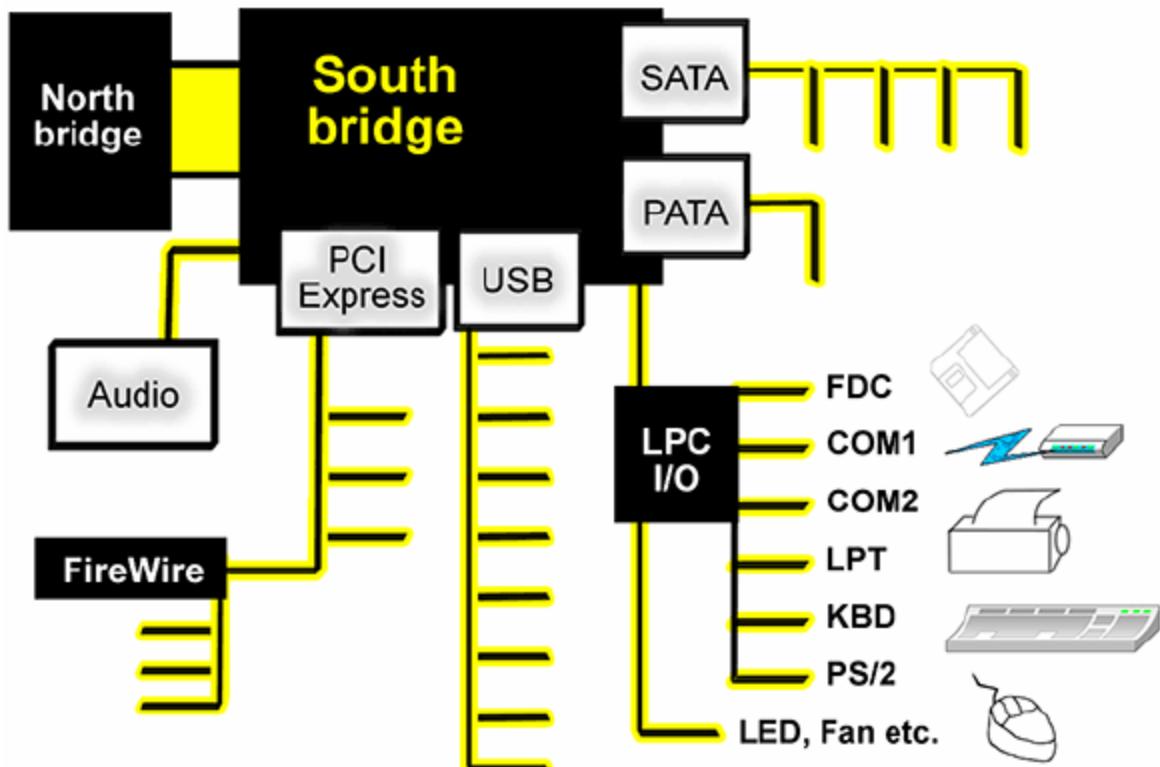
Clasificarea magistrelor dintr-un calculator personal este prezentată mai jos.



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”

Componentă	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 (infraroșu)
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 asamblare

Software pentru generarea de programe în limbaj mașină sunt de regulă compilatoarele și asamblarele. Un **compilator** este un program care transformă programul scris într-un limbaj de programare de nivel înalt în limbaj mașină. Compilatoarele pentru un limbaj de programare vor avea partea „din față” (cea care recunoaște construcțiile din limbajul de programare de nivel înalt) identică, iar „partea din spate” (cea care creează codul mașină) diferită pentru fiecare tip de sistem de calcul. Se poate ca același program compilat cu compilatoare diferite pentru același 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 mașină 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 conținând descrierii într-un limbaj special și prezintă la ieșire un fișier 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 și 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 macroinstructiunile existente și a rezolva, pe cât posibil, blocurile condiționale.

Macrogeneratorul recunoaște și analizează: directivele de control numeric (tipul bazei de numerație), directivele de terminare (.END), directivele de asamblare condițională (.IF, .IFF, .IFT etc.), directivele de definire a macroinstructiunilor (.MACRO, .ENDM), directivele de control (mesaje de eroare), directivele de generare și substituire, directivele de apelare a macrodefinițiilor 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 fișiere sursă în limbaj de asamblare obținute folosind macrogeneratorul sau scrise de utilizator. Ieșirile asamblorului constau dintr-un fișier obiect (.obj, .o, .mmo etc.), un fișier de listare (.lst) și o tabelă de simboluri (.map).

Asamblorul absolut transformă modulele scrise în limbaj de asamblare (ieșire a macrogeneratorului sau scrise de utilizator) 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. Funcția principală a editorului de legături este de a transforma și înlănuiri modulele obiect relocabile rezultate în procesul de asamblare și/sau compilare pentru a obține 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 și semantic,
- generator de cod și
- optimizator.

Toate aceste componente gestionează un tabel de simboluri.

Analizorul lexical realizează o primă traducere a textului programului sursă într-un sir de entități 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 și semantic verifică corectitudinea sintactică a instrucțiunilor și colectează atributele semantice ale categoriilor sintactice din program. Ieșirea analizorului sintactic și semantic o constituie o reprezentare codificată a structurii sintactice și un set de tabele ce conțin atributele semantice ale diferitelor categorii sintactice (simboluri, constante etc.)

Generatorul de cod va traduce ieșirea analizorului sintactic și semantic în format binar relocabil sau în limbaj de asamblare.

Optimizatorul are rolul de a prelucra ieșirea generatorului de cod cu scopul de a minimiza memoria necesară la execuție și a elimina redundanțele din corpul programului. Unele compilatoare efectuează anumite optimizări înainte de generarea codului.

O funcție importantă a compilatorului constă în detectarea erorilor din programul sursă și corectarea sau acoperirea lor. Spre deosebire de compilatoare, interprotoarele efectuează și executarea programului odată cu traducerea programului sursă, furnizând la ieșire rezultatele programului.

Mai precis, diferența față de un compilator este aceea că interpretorul nu produce un program obiect ce urmează să fie executat după interpretare, ci chiar execută acest program.

Din punct de vedere structural, un interpretor se asemănă cu un compilator, dar forma intermedieră obținută nu este folosită la generarea de cod, ci pentru a ușura decodificarea instrucțiunii 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 și/sau șterge caractere, linii sau grupuri de linii din programul sursă sau din orice fișier text.

Editoarele de texte pot lucra în mod linie și/sau mod ecran. Ele pun la dispoziție comenzi privind deplasarea cursorului în text (pe linie, în cadrul unui bloc sau în cadrul întregului fișier), manipularea blocurilor de text (marcare, deplasare, copiere, ștergere), comenzi de formatare a ieșirii etc. Exemplificăm prin Edit (Ms-DOS, Windows), Notepad (Windows), vi sau emacs (UNIX) etc.

Cu toate că majoritatea programelor se scriu în limbi de nivel înalt, programatorii pot scrie programe sau secvențe din unele programe, care trebuie să ruleze foarte repede, în asamblare. În plus, să ar putea să nu existe compilatoare pentru anumite procesoare speciale sau compilatoarele să nu poată să folosesc 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 ţintă. Pe de altă parte, la nivel limbaj mașină, arhitectura este cel mai important aspect de care trebuie să se temă.

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 alți 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 (reghistroare, memorie, porturi). C este un limbaj de programare standardizat, compilat, implementat pe marea majoritate a platformelor de calcul existente azi. Este apreciat pentru eficiența 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*. Instrucțiunile din C care rezolvă o problemă sunt mult mai puține decât cele ale limbajului asamblare aferent calculatorului LC-3 care rezolvă aceeași problemă [Patt03]. De asemenea, LC-3 nu asigură instrucțiuni de înmulțire și împărțire. 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 condiționată. Deși 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 experiență.

6.1.1. Compilare vs. Interpretare

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

- **Interpretare:** Codul sursă HLL este „*interpretat*” de o mașină virtuală (cele mai cunoscute sunt mașinile virtuale Java – JVM) care translatează „*secțiuni*” din codul sursă în limbaj mașină și îl execută direct pe arhitectura gazdă, trecând apoi la următoarea *secțiune* de cod sursă. Dintre avantajele interpretării față de metodele tradiționale de compilare s-ar menționa simplitatea implementării hardware dar și faptul că nu necesită o zonă mare de memorie pentru stocarea programului compilat. Principalul dezavantaj îl reprezintă viteza scăzută de execuție a aplicației, dar și necesitatea existenței în momentul interpretării atât a codului sursă HLL cât și a mașinii virtuale care realizează interpretarea.
- **Compilare:** Compilarea directă translatează codul sursă (poate fi Java, C, C++, Fortran, Pascal) în instrucțiuni mașină, direct executabile pe un procesor țintă, dar nu le execută ca în cazul interpretoarelor. Orice modificare a codului sursă necesită recomplirare. Procesul este realizat static și poate îngloba tehnici de optimizare de tip analiza dependențelor de date dintre instrucțiuni, analiză interprocedurală. 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 același program - obiect, de regulă scris în limbaj de asamblare. În literatura de specialitate, este denumită *fază* a unui compilator o succesiune de operațiuni prin care un program de la intrare suferă anumite modificări. Prin *trecere* apărând unui compilator se înțelege o citire a programului dintr-un fișier, transformarea lui conform unor faze și scrierea rezultatului în alt fișier(de ieșire).

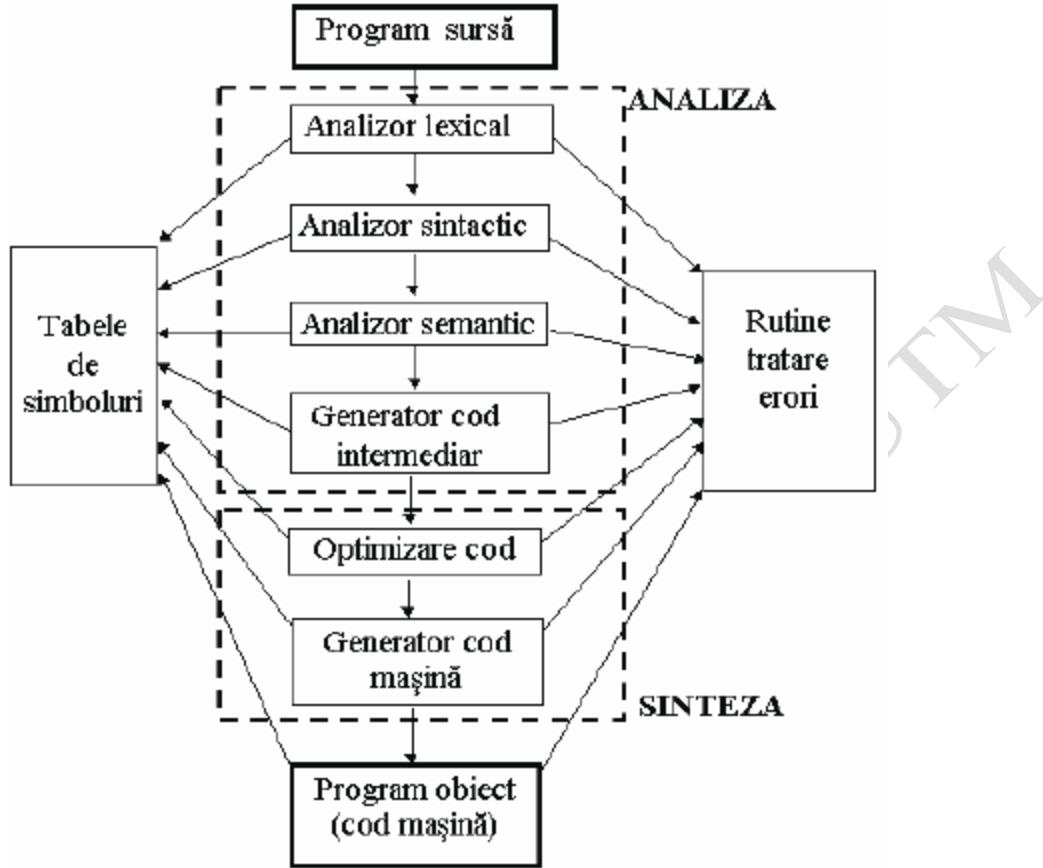


Figura anterioară ilustrează cele două faze majore ale procesului de compilare propriu-zisă:

- **analiza** (de tip “*front end*”) - în care se identifică părțile constituente fundamentale ale programului (variabile, expresii, declarații, definiții, apeluri de funcții) și se construiește o reprezentare internă a programului original, numită „*cod intermediar*” (*analiza lexicală* produce un -> sir de atomi lexicali -> *analiză sintetică* generează -> arborele sintactic -> *analiză semantică* construiește o reprezentare a programului sursă în-> *cod intermediar*). Este dependentă de limbajul de nivel înalt și nu de mașina pe care se va procesa.
- **sinteză** (de tip “*back end*”) - generează cod mașină eventual optimizat. Se disting două etape
 - optimizare cod intermediar (scheduling) pentru o anumită mașină;
 - generare de cod mașină (generare cod într-o gamă variată de formate: *limbaj mașină absolut*, *limbaj mașină 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 destinație (obiect) scris tot într-un limbaj de nivel înalt C. Practic are loc interpretarea *directiveelor de preprocesare* (încep cu # - include, define, if !defined(...)...endif). De exemplu, preprocesorul trebuie să introducă conținutul fișierului <stdio.h> în codul sursă al aplicației create acolo unde apare respectiva directivă de preprocesare, sau, să înlocuiască anumite valori constante declarate ca siruri 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, și care este responsabilă de transformarea programului sursă văzut ca o succesiune de caractere (text) într-o succesiune de atomi lexicali și de attribute ale lor. Conform definiției 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.

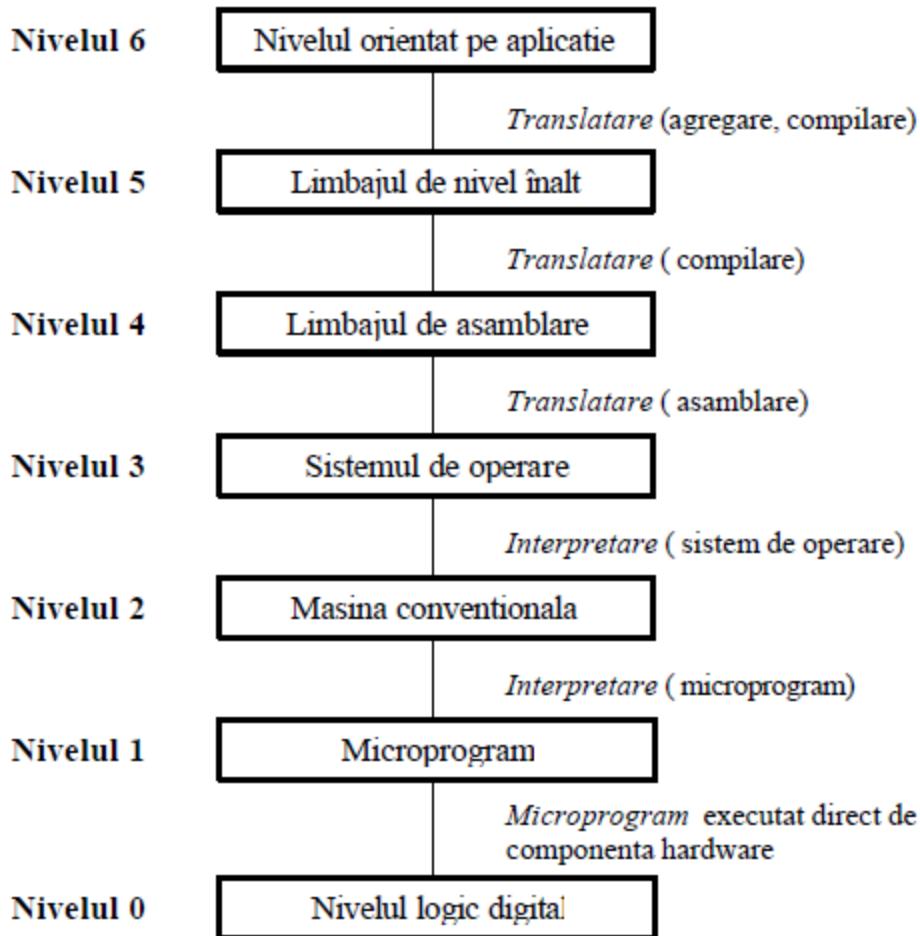
Toate fazele unui compilator dispun de **rutine de tratare a erorilor** și lucrează în comun cu una sau mai multe tabele de simboluri în care se păstrează informații despre cuvintele cheie ale limbajului, identificatorii de variabile (tip, lungime, adresă stocare în memorie etc.), etichete de instrucțiuni, identificatorii de proceduri și funcții și alte informații utile despre programul în curs de compilare. **Tabela de simboluri** este creată în faza de analiză a programului sursă și folosită în scopul validării n umelor 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 față de începutul zonei de date statice sau dinamice, stivă). Compilatorul folosește tabela de simboluri pentru a urmări domeniul de utilizare și valabilitate a unui nume și de a adăuga informații 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 informațiilor existente în mod eficient. Tabela de simboluri este similară celei generate de asamblor însă cuprinde mai multe informații decât aceasta. În asamblare, toți identificatorii (variabilele) erau etichete (*labels*) și informațiile reprezentate de acestea erau adrese. Tabela de simboluri conține informații legate de toate variabilele din program.

Generatorul de cod constituie faza finală a unui compilator. Primește la intrare reprezentarea intermedieră a programului sursă împreună cu informația din tabela de simboluri - folosită la determinarea run-time a adresei obiectelor de date, desemnate de nume în reprezentarea intermedieră, și produce la ieșire un program obiect echivalent. Codul de ieșire trebuie să fie corect și de înaltă calitate, aceasta însemnând că el trebuie să folosească eficient resursele mașinii pentru a putea fi procesat cât mai rapid.

Ieșirile generatorului de cod se găsesc într-o gamă variată de forme: *limbaj de asamblare*, *limbaj mașină relocabil* sau *limbaj mașină absolut*. Producând un program într-un limbaj de asamblare, caracterizat de o mai mare lizibilitate, generarea codului se realizează mult mai ușor. Pot fi generate instrucțiuni simbolice precum și utilizate facilitățile de macro-asamblare ajutătoare la generarea de cod. Producând un program în limbaj mașină absolut, există avantajul că programul poate fi plasat într-o locație fixă a memoriei și poate fi imediat executat. Producând un program în limbaj mașină relocabil, acesta permite subprogramelor să fie compilate separat (similar cu fișierul *Makefile* din cadrul setului de instrumente SimpleScalar 3.0). O mulțime de module-obiect relocate pot fi *legate (linked)* împreună și apoi încărcate pentru execuție de către un *loader*. Câștigul de flexibilitate care dă posibilitatea compilării separate a subrutinelor și apelarea altor programe compilate anterior dintr-un modul obiect este tributar costurilor suplimentare pentru legare și încărcare.

6.3. Programarea în limbaj de asamblare

Limbajul de asamblare permite accesul la unele dintre resursele fizice ale mașinii în condițiile utilizării unor mnemonice mai ușor de manevrat decât codurile numerice asociate instrucțiunilor limbajului mașină 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, și variantele ultime Pentium II, III și IV. Toate aceste procesoare împărtășesc o arhitectură comună și utilizează același set de instrucțiuni de asamblare. Îmbunătățirile arhitecturale aduse în decursul timpului la noile versiuni de procesoare s-au făcut în aşa fel încât să se mențină compatibilitatea cu imaginea inițială a unui procesor I8086. Chiar dacă în interior un procesor din ultima generație diferă semnificativ de structura primului procesor din familie, acest lucru este vizibil în mică măsură pentru un programator. De aceea nu se mai vorbește 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 biți se operează cu imaginea arhitecturală denumită ISA x86 (Instruction Set Architecture). În același mod pentru procesoarele pe 64 de biți s-a introdus arhitectura ISA 64. Această arhitectură diferă semnificativ de cea a procesoarelor pe 16 și 32 de biți, 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 piață a unui procesor pe 64 de biți.

În ceea ce privește arhitectura ISA x86, aceasta a suferit anumite modificări impuse de introducerea, în versiunile mai noi de procesoare, a unor elemente arhitecturale și funcționale noi. Însă aceste elemente

noi nu modifică componente 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 instrucțiunilor. Există registre generale folosite în majoritatea operațiilor aritmetico-logice și registre speciale, care au o destinație specială.

La arhitectura ISA x86 pe 16 biți există 8 registre generale pe 16 biți, regisztre 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 operanziilor, iar următoarele patru mai ales pentru păstrarea și calculul adreselor de operanzi. Primele patru regisztre se pot adresa și pe jumătate, adică la nivel de octet. Astfel fiecare regisztru are o parte superioară "HIGH" (biții D15-D8) și o parte inferioară "LOW" (biții D7-D0). Denumirea lor este AH și AL pentru regisztrul AX, BH și BL pentru regisztrul BX și aşa mai departe.

La procesoarele pe 32 de biți (începând de la '86) aceste regisztre s-au extins la 32 de biți. Astfel regiszrele pot fi adresate pe 32 de biți (cu denumirile EAX, EBX, ECX, ... ESP) sau în mod uzuial pe 16 sau 8 biți. Partea superioară a unui regisztru pe 32 de biți (D31-D16) nu se poate adresa individual.

În principiu regiszrele generale respectă principiul de ortogonalitate, adică pentru majoritatea operațiilor oricare regiszru este utilizabil. La unele procesoare (ex: procesoarele Motorola) acest principiu este respectat strict. În cazul procesoarelor Intel anumite operații speciale impun utilizarea anumitor regiszre, ceea ce înseamnă că există o oarecare specializare între regiszre. Această specializare decurge de multe ori și din denumirea regiszrelor. Astfel:

- regiszrul AX (EAX în varianta pe 32 de biți) – se folosește ca regiszru "acumulator" în majoritatea operațiilor aritmetice și logice, adică păstrează unul dintre operanzi și apoi rezultatul operației; la operațiile de înmulțire și împărțire în mod obligatoriu primul operand și rezultatul se păstrează în acest regiszru
- regiszrul BX (EBX) – se folosește pentru în operații aritmetico-logice sau pentru calculul adresei operandului la "adresarea bazată"
- regiszrul CX (ECX) – se folosește pentru operații aritmetico-logice sau în mod implicit, la anumite instrucțiuni (ex: instrucțiuni de buclare), pe post de contor
- regiszrul DX (EDX) – se folosește pentru operații aritmetico-logice sau pentru păstrarea adresei porturilor la instrucțiunile de intrare/iesire; de asemenea la operațiile de înmulțire și împărțire se folosește ca extensie a regisztrului acumulator
- regiszrele SI și DI (ESI, EDI) – denumite și regiszre index, se folosesc cu precădere pentru calculul adresei operanziilor la "adresarea indexată"; SI adresează elementele șirului sursă (source index), iar DI elementele șirului destinație (destination index)
- regiszrul BP (EBP) – se folosește cu precădere pentru calculul adresei operanziilor la "adresarea bazată", alături de regiszrul BX
- regiszrul SP (ESP) – se folosește aproape în exclusivitate pentru adresarea stivei (stack pointer); conținutul regiszrului se incrementează și se decrementează automat la orice operație cu stiva

În afara regiszrelor generale un procesor mai are și regiszre speciale sau de control. La procesoarele Intel numai o parte din aceste regiszre sunt vizibile și accesibile pentru programator. Aceste regiszre controlează regimul de lucru al procesorului, sau permit efectuarea unor operații speciale de manipulare a spațiului de memorie.

Arhitectura ISA x86 operează cu următoarele regiszre speciale:

- regiszrul PC – "program counter" – păstrează adresa instrucțiunii care urmează; nu este adresabil direct (prin nume) dar conținutul său se poate modifica prin execuția instrucțiunilor de salt
- regiszrul PSW – "program status word" – păstrează indicatorii de stare ai procesorului;

- o parte din indicatori caracterizează rezultatul obținut în urma unei anumite instrucțiuni:
 - ZF – rezultat 0,
 - SF – semnul rezultatului
 - OF – "overflow" indică o depășire de capacitate la ultima operație 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 biți s-a generat un transport
- o parte din indicatori controlează modul de lucru al procesorului:
 - IF – indicator de intrerupere (interrupt), dacă este setat (1 logic) atunci se blochează toate intreruperile mascabile, în caz contrar (0 logic) se validează
 - DF – indicatorul de direcție, arată direcția de parcurs a sirurilor de caractere la instrucțiunile pe siruri (în ordine ascendentă sau descendentă a adreselor)
- registrele de segment – se utilizează pentru calculul adresei operanzilor și a instrucțiunilor; cu ajutorul lor memoria este divizată în segmente; există 4 registre segment în versiunea inițială ISA x86 și 6 registre segment în versiunea pe 32 de biți:
 - registrul CS – registrul segment de cod, folosit pentru adresarea instrucțiunilor (a codurilor); acest regisztr 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 regisztr 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 și care se folosesc pentru adresarea operanzilor

Versiunile mai noi de procesoare conțin și alte registre speciale, dar acestea au o importanță mai mică pentru un programator obișnuit. 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 și executate de procesor. Fiecare instrucțiuni 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 secvență de coduri (instrucțiuni în cod mașină). Un anumit limbaj de asamblare este specific pentru un anumit procesor sau eventual pentru o familie de procesoare. Instrucțiunile sunt în directă corelație cu structura internă a procesorului. Un programator în limbaj de asamblare trebuie să cunoască această structură precum și 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 aplicații pe un alt procesor programul trebuie rescris. În schimb programele scrise în limbaj de asamblare sunt în general mai eficiente atât în ceea ce privește timpul de execuție cât și spațiul de memorie ocupat de program. De asemenea, programarea în limbaj de asamblare dă o mai mare flexibilitate și libertate în utilizarea resurselor unui calculator. Cu toate acestea astăzi utilizarea limbajului de asamblare este mai puțin frecventă deoarece eficiența procesului de programare este mai scăzută, există puține structuri de program și de date care să ușureze munca programatorului, iar programatorul trebuie să cunoască structura procesorului pentru care scrie aplicația. În plus programele nu sunt portabile, adică nu rulează și pe alte procesoare.

6.4. Directive și instrucțiuni ASM.

Un program scris în limbaj de asamblare conține instrucțiuni și directive. Instrucțiunile sunt traduse în coduri executate de procesor; ele se regăsesc în programul executabil generat în urma compilării și a editării de legături. Directivele sunt construcții 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 într-un cod executabil și în consecință NU se execută de către procesor.

6.4.1. Sintaxa unei instrucțiuni în limbaj de asamblare

O instrucțiune 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 instrucțiunea 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 și anumite semne speciale (ex: _), cu restricția 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 și nu pot fi utilizate în alte scopuri
- <parametru_1> - este primul operand al unei instrucțiuni și în același timp și destinația rezultatului; primul parametru poate fi un registru, o adresă, sau o expresie care generează o adresă de operand; adresa operandului se poate exprima și într-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 intențiile programatorului și efectul scontat în urma execuției 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 consideră până la sfârșitul 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 destinația, 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 exhaustivă a tuturor instrucțiunilor cu toate detaliile lor de execuție. Se vor prezenta acele instrucțiuni care se utilizează mai des și au importanță din punct de vedere al structurii și al posibilităților procesorului. Pentru alte detalii se poate consulta documentații complete referitoare la setul de instrucțiuni ISA x86 (ex: cursul AoA – The Art of Assembly Language, accesibil pe Internet).

6.4.2. Instrucțiuni de transfer

Instrucțiunile de transfer realizează transferul de date între registre, între un registru și o locație de memorie sau o constantă se încarcă într-un registru sau locație de memorie. Transferurile de tip memorie-

memorie nu sunt permise (cu excepția instrucțiunilor pe şiruri). La fel nu sunt permise transferurile directe între două registre segment. Ambii parametrii ai unui transfer trebuie să aibă aceeași lungime (număr de biți).

Instrucțiunea 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

Instrucțiunea LEA, LDS și 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

Instrucțiunea XCHG

Această instrucțiune schimbă între ele conținutul celor doi operanzi.

XCHG <parametru_1>,<parametru_2>

Atenție: parametrul 2 nu poate fi o constantă.

Exemple:

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

Instrucțiunile PUSH și POP

Cele două instrucțiuni operează în mod implicit cu vârful stivei. Instrucțiunea 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 poziția curentă a vârfului de stivă. Sintaxa instrucțiunilor este:

```
PUSH <parametru_1>  
POP <parametru_1>
```

Transferul se face numai pe 16 biți. Aceste instrucțiuni sunt utile pentru salvarea temporară și 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. Instrucțiuni aritmetice

Aceste instrucțiuni efectuează cele patru operații aritmetice de bază: adunare, scădere, înmulțire și împărțire. Rezultatul acestor instrucțiuni afectează starea indicatorilor de condiție.

Instrucțiunile ADD și ADC

Aceste instrucțiuni efectuează operația 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 instrucțiune este utilă pentru implementarea unor adunări în care operanzii sunt mai lungi de 32 de biți.

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

Exemple:

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

Instrucțiunile SUB și SBB

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

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

Instrucțiunile MUL și IMUL

Aceste instrucțiuni efectuează operația de înmulțire, MUL pentru întregi fără semn și IMUL pentru întregi cu semn. De remarcat că la operațiile de înmulțire și împărțire trebuie să se tină 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 depășiri de capacitate s-a decis ca rezultatul operației de înmulțire să se păstreze pe o lungime dublă față de lungimea operanzilor. Astfel dacă operanzi 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 și implicit și 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

Instrucțiunile DIV și IDIV

Aceste instrucțiuni efectuează operația de împărțire pe întregi fără semn 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 față 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 împărțirii)
div bx ; AX<=(DX:AX)/BX și DX<=(DX:AX) modulo BX

Instrucțiunile INC și DEC

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

INC <parametru>
DEC <parametru>

Exemple:

inc si ; SI<=SI+1
dec cx ; CX<=CX-1

Instrucțiunea CMP

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

CMP <parametru_1>, <parametru_2>

Exemplu:

```
cmp ax, 50h
```

6.4.4. Instrucțiuni logice

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

Instrucțiunile AND, OR, NOT și XOR

Aceste instrucțiuni implementează cele patru operații de bază: ȘI, SAU, Negatie și 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           ; șterge conținutul lui ax
```

Instrucțiunea TEST

Această instrucțiune efectuează operația ȘI logic fără a memora rezultatul. Scopul operației este de a modifica indicatorii de condiție. Instrucțiunea evită distrugerea conținutului primului operand.

```
TEST <parametru_1>,<parametru_2>
```

Exemple:

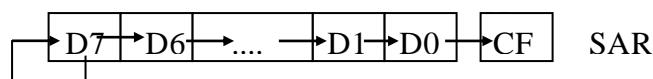
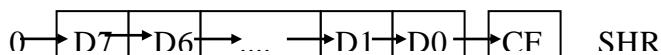
```
test al, 00010000b      ; se verifică dacă bitul D4 din al este zero sau nu
test bl, 0fh              ; se verifica dacă prima cifră hexazecimală din bl este 0
```

6.4.5. Instrucțiuni de deplasare și de rotire

Instrucțiunile SHL, (SAL), SHR și SAR

Aceste instrucțiuni realizează deplasarea (eng. shift) la stânga și respectiv la dreapta a conținutului unui operand. La deplasarea "logică" (SHL, SHR) biți se copiază în locațiile 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 și la dreapta se obține o multiplicare și respectiv o divizare cu puteri ale lui doi (ex: o deplasare la stânga cu 2 poziții binare este echivalentă cu o înmulțire cu 4). La deplasarea la dreapta se dorește menținerea 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 aceeași instrucțiune.

În figura de mai jos s-a reprezentat o deplasare logică și o deplasare aritmetică la dreapta. Se observă că bitul careiese 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>
```

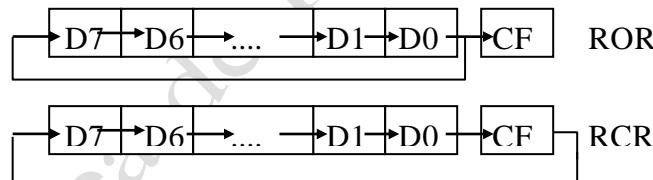
Primul parametru este în concordanță cu definițiile anterioare; al doilea parametru specifică numărul de poziții binare cu care se face deplasare; acest parametru poate fi 1 sau dacă numărul de pași este mai mare atunci se indică prin registrul CL. La variantele mai noi de procesoare se acceptă și forma în care constanta este diferită de 1.

Exemple:

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

Instrucțiunile de rotire ROR, ROL, RCR, RCL

ACESTE INSTRUCȚIUNI REALIZEAZĂ ROTIREA LA DREAPTA SAU LA STÂNGA A OPERANDULUI, CU UN NUMĂR DE POZIȚII BINARE. DIFERENȚA FAȚĂ DE INSTRUCȚIUNILE ANTERIOARE DE DEPLASARE CONSTĂ ÎN FAPTUL CĂ ÎN POZIȚIA ELIBERATĂ PRIN DEPLASARE SE INTRODUC BITUL CARE IESE DIN OPERAND. ROTIREA SE poate FACE CU IMPLICAREA INDICATORULUI DE TRANSPORT (CF) ÎN PROCESUL DE ROTAȚIE (RCR, RCL) SAU FĂRĂ (ROR, ROL). ÎN AMBELE CAZURI BITUL CARE IESE DIN OPERAND SE REGĂSEȘTE ÎN INDICATORUL DE TRANSPORT CF. FIGURA DE MAJ JOA INDICĂ CELE DOUĂ MODURI DE ROTAȚIE PENTRU O ROTAȚIE LA DREAPTA.



Formatul instrucțiunilor:

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

REFERITOR LA PARAMETRI, SE APĂCĂ ACELEAȘI OBSERVAȚII CA ȘI LA INSTRUCȚIUNILE DE DEPLASARE.

6.4.6. Instrucțiuni de salt

INSTRUCȚIUNILE DE SALT SE UTILIZEAZĂ PENTRU CONTROLUL SECVENȚEI 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. Instrucțiunile de apel și revenire din rutine

Utilizarea rutinelor (a procedurilor) permite structurarea programelor scrisă în limbaj de asamblare și 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.

Instrucțiunile CALL și RET

Instrucțiunea 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). Instrucțiunea RET se plasează la sfârșitul 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 secvența rutinei.

Formatul instrucțiunilor:

```
CALL    <adresă>
RET    [<constantă>]
```

unde:

<adresă> - este o etichetă (numele rutinei) sau o expresie evaluabilă ca și o adresă
 <constantă> indică numărul de poziții cu care trebuie să se descarce stiva înainte de revenirea din rutină; în mod uzual acest parametru lipsește

Exemple:

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

Funcție de poziția rutinei față de instrucțiunea 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 instrucțiunea de apel. programatorul poate cere o adresă "near" sau "far" în mod explicit printr-o directivă de declarare a procedurii.

6.4.8. Instrucțiunile de ramificare și ciclare

Acste instrucțiuni modifică secvența de execuție a instrucțiunilor. Există instrucțiuni de salt necondiționat (care se execută în orice condiție) și instrucțiuni de salt condiționat. cele din urmă determină execuția unui salt în măsura în care o anumită condiție este îndeplinită. O condiție poate fi starea unui indicator de condiție sau o combinație a acestora. Formatul instrucțiunilor este:

```
JMP    <adresă>
J<cc>  <adresă>
```

unde:

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

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

Instrucțiunea	Condiția	Instrucțiuni echivalente	Explicații
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

Instrucțiuni de salt condiționat utilizate după compararea a două numere fără semn:

Instrucțiune	Condiție	Indicatori testați	Instrucțiuni echivalente	Explicații
JA	>	CF=0,ZF=0	JNBE	salt la mai mare
JAE	\geq	CF=0	JNB,JNC	salt la mai mare sau egal
JB	<	CF=1	JNAE,JC	salt la mai mic
JBE	\leq	CF=1 sau ZF=1	JNA	salt la mai mic sau egal
JE	=	ZF=1	JZ	salt la egal
JNE	\neq	ZF=0	JNZ	salt la diferit

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

Instrucțiune	Condiție	Indicatori testați	Instrucțiuni echivalente	Explicații
JG	>	SF=OF sau ZF=0	JNLE	salt la mai mare
JGE	\geq	SF=OF	JNL	salt la mai mare sau egal
JL	<	SF!=OF	JNGE	salt la mai mic
JLE	\leq	SF!=OF sau ZF=1	JNG	salt la mai mic sau egal
JE	=	ZF=1	JZ	salt la egal
JNE	\neq	ZF=0	JNZ	salt la diferit

Observație: relațiile de ordine (mai mic, mai mare, etc.) se stabilesc între primul și al doilea parametru al operației 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 depășire de capacitate
  
```

Instrucțiunile de ciclare LOOP, LOOPZ, LOOPNZ

Aceste instrucțiuni 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 condiției de terminare a ciclului și salt la etichetă (la începutul ciclului) în cazul în care condiția nu este îndeplinită.

Sintaxa instrucțiunilor:

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

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

Pentru instrucțiunea LOOP condiția de terminare a ciclului este CX=0, adică contorul ajunge la 0. Pentru instrucțiunea LOOPZ în plus ciclul se încheie și în cazul în care ZF=0. La instrucțiunea 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. Instrucțiuni de intrare/ieșire

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

Instrucțiunile IN și OUT

Instrucțiunea IN se folosește 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 biți sau AL pentru transfer pe 8 biți
<adresa_port> - o adresă exprimabilă pe 8 biți sau registrul DX

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

Exemple:

```
    ïn    al, 20h  
    mov dx, adresă_port  
    out   dx, ax
```

6.4.10. Instrucțiuni pe șiruri

Aceste instrucțiuni s-au introdus cu scopul de a accelera accesul la elementele unei structuri de tip șir sau vector. Instrucțiunile folosesc în mod implicit registrele index SI și DI pentru adresarea elementelor șirului sursă și respectiv destinație. În mod implicit registrul DS păstrează adresa de segment a sursei iar registrul ES adresa de segment a destinației. După efectuarea operației propriu-zise (specificată prin mnemonica instrucțiunii), registrele index sunt incrementate sau decrementate automat pentru a trece la elementele următoare din șir. Indicatorul DF determină direcția de parcurs a șirurilor:

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.

Instrucțiunile MOVSB, MOVSW

Aceste instrucțiuni transferă un element din șirul sursă în șirul destinație. Instrucțiunea 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 unități, deoarece un cuvânt ocupă 2 locații în memorie. Instrucțiunile nu au parametrii; programatorul trebuie să încarce în prealabil adresele șirurilor în registrele SI și DI, și lungimea șirului în CX.

Exemplu:

```
        mov    si, 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
```

Instrucțiunile LODSB, LODSW, STOSB și STOSW

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

Instrucțiunile CMPSB, CMPSW, SCASB și SCASW

Aceste instrucțiuni realizează operații de comparare cu elemente ale unui șir. Primele două instrucțiuni compară între ele elementele a două șiruri, iar ultimele două compara conținutul registrului acumulator cu câte un element al șirului (operație de scanare).

Instrucțiunile REP, REPZ, REPE, REPNZ, REPZN

Aceste instrucțiuni permit execuția multiplă a unei instrucțiuni pe șiruri. Prin amplasarea unei astfel de instrucțiuni în fața unei instrucțiuni pe șiruri obligă procesorul execuția repetată a operației până ce condiția de terminare este satisfăcută. La prima variantă, REP, condiția de terminare este CX=0. La instrucțiunile REPZ și REPE operația se repetă atât timp cât rezultatul este zero sau operanții sunt egali. La REPZN și REPNE operația se repetă atât timp cât rezultatul este diferit de zero sau operanții sunt diferiți.

Exemplu:

```
        mov    si, offset sir_sursa
        mov    di, offset sir_destinatie
        mov    cx, lungime_sir
rep:  movsb                         ; transferă șirul sursă în șirul destinație
```

6.4.11. Instrucțiuni speciale

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

Instrucțiunile CLC, STC, CMC

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

CLC șterge (eng. clear) indicatorul, CF=0

STC setează indicatorul, CF=1

CMC inversează starea indicatorului, CF=NOT CF

Instrucțiunile CLI și STI

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

Instrucțiunile CLD și STD

Aceste instrucțiuni modifica starea indicatorului de direcție DF. Prin acest indicator se controlează modul de parcursere a șirurilor la operațiile pe șiruri: prin incrementare (DF=0) sau prin decrementare (DF=1).

Instrucțiunile NOP, HLT și HIT

Instrucțiunea NOP nu face nimic (eng. no operation). Această instrucțiune se folosește în scopuri de temporizare, pentru întârzierea unor operații sau pentru implementarea unor bucle de așteptare. Instrucțiunea HLT (eng. halt) determină oprirea procesorului. Instrucțiunea HIT determină oprirea temporară a procesorului până la apariția unui semnal de întrerupere sau de inițializare (reset).

Instrucțiunea CPUID

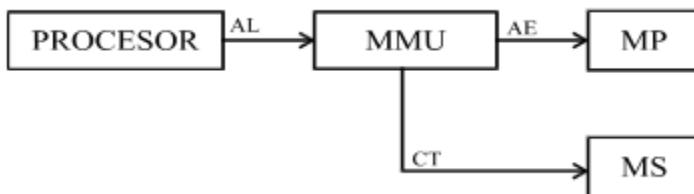
Această instrucțiune permite identificarea tipului de procesor pe care rupeaza programul.

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 spațiu virtual de adresare foarte mare și care, fără ca programatorul să “simtă”, este mapat în memoria fizic disponibilă. Uzual, spațiul virtual de adrese corespunde HDD/SSD, programatorul având iluzia prin mecanismele de memorie virtuală (MV), că deține o memorie (virtuală) de capacitatea hard-discului și nu de capacitatea memoriei fizice preponderentă DRAM (limitată la $64 \div 1024$ 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 mărește probabilitatea ca informația ce se dorește a fi accesată de către CPU din spațiul virtual (disc).

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



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

$$f_T: AL \rightarrow AE.$$

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

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

OPCODE	ADRESĂ
n	1
L	

În câmpul de adresă apar elementele constitutive 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 împărțit în entități de capacitate fixă ($4 \div 64$ Ko actualmente), numite pagini. O pagină poate fi mapată în MP sau pe disc.

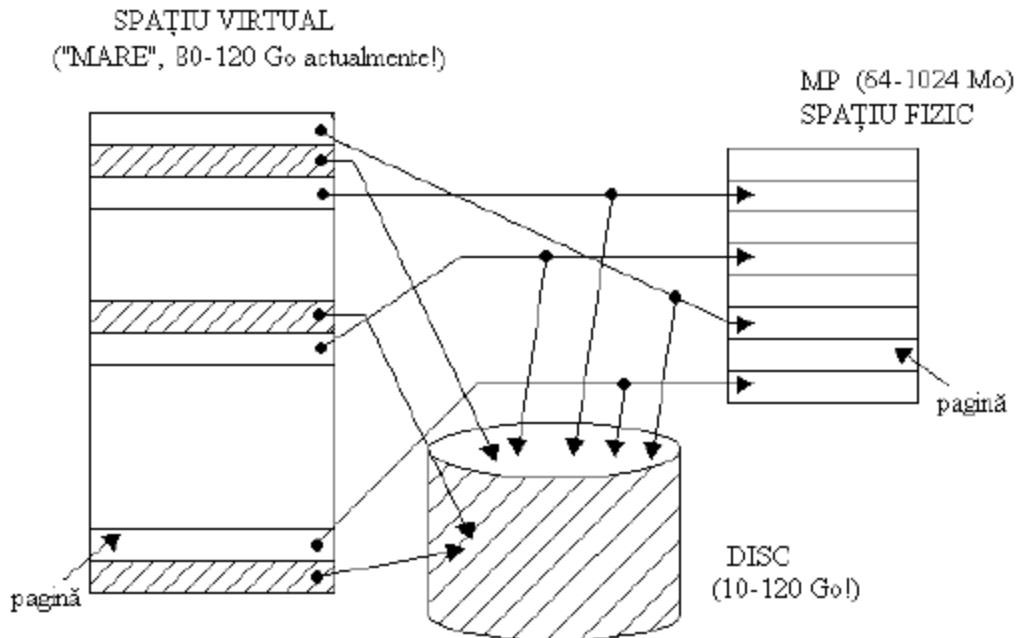


Figura 4.8. Maparea adreselor virtuale în adrese fizice

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 într-o 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 acelaiași program să fie încărcat și să ruleze oriunde ar fi încărcat în MP, modificările de adresa 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 cvasti-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 ($4\text{ ko} \div 16\text{ ko} \dots 64\text{ ko}$) astfel încât să compenseze timpul mare de acces la disc ($9 \div 12\text{ ms}$).
- organizări care să reducă rata de evenimente PF, rezultând un plasament flexibil al paginilor în memorie (MP)
- PF-urile trebuie tratate prin software și nu prin hardware (spre deosebire de miss-urile în cache-uri), timpul de acces al discurilor permitând lejer acest lucru.
- scrisurile în MV se fac după algoritmi tip "Write Back" și nu "Write Through" (ar consuma timp enorm).

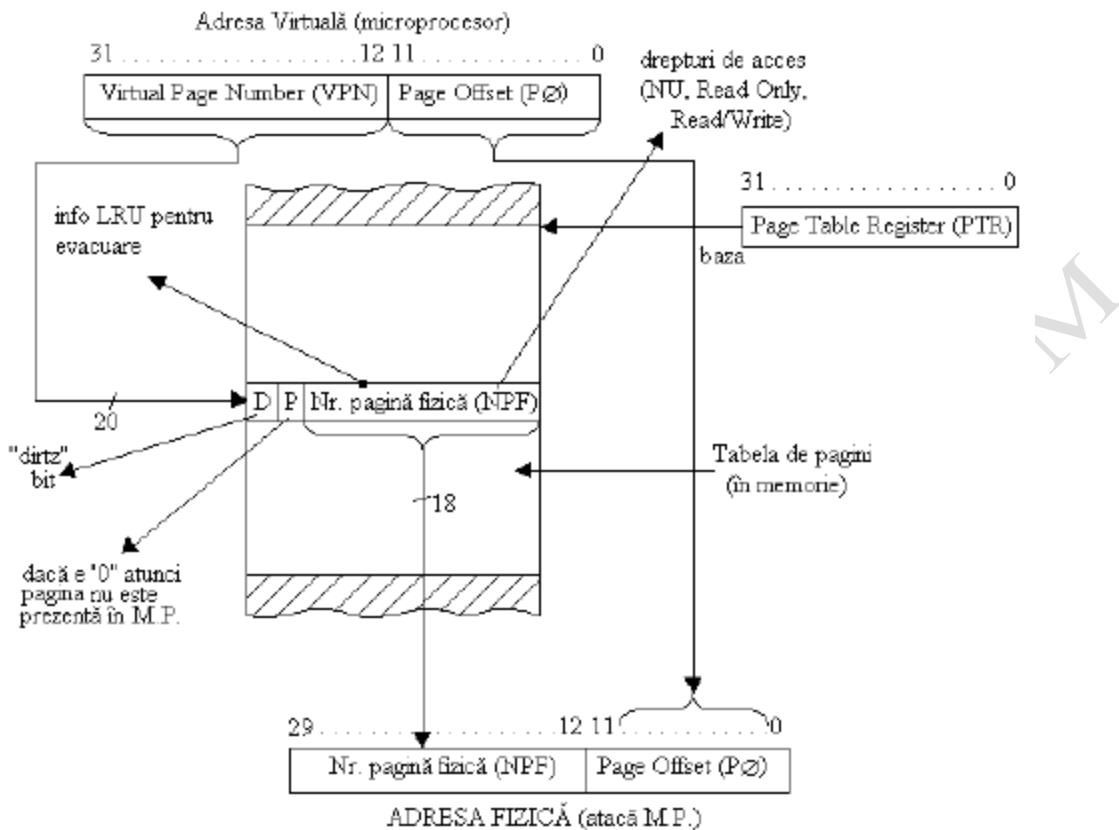


Figura 4.9. Translatare adresă virtuală în adresă fizică

Fiecare program are propria sa tabelă de pagini care mapează spațiul virtual de adresare al programului într-un spațiu 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 și activarea altui proces, inactiv până acum rezultând deci ca fiind necesară salvarea/restaurarea stării proceselor. Desigur, sistemul de operare (S.O.) trebuie doar să reîncarce 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, entități 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 (bază + offset) se încadrează în lungimea adoptată a segmentului. Desigur, segmentarea oferă posibilități de protecție puternice și sofisticate a segmentelor. Pe de altă parte, segmentarea induce și numeroase dezavantaje precum:

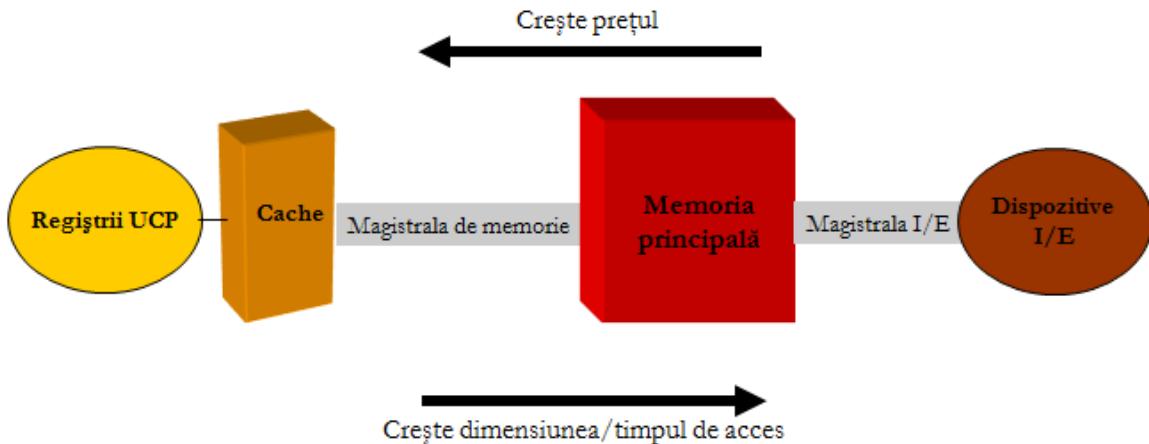
2 cuvinte pentru o adresă virtuală, necesare având în vedere lungimea variabilă a segmentului. Asta complică sarcina compilatoarelor și a programelor

- încărcarea segmentelor variabile în memorie mai dificilă decât la paginare
- fragmentare a memoriei principale (portiuni nefolosite)

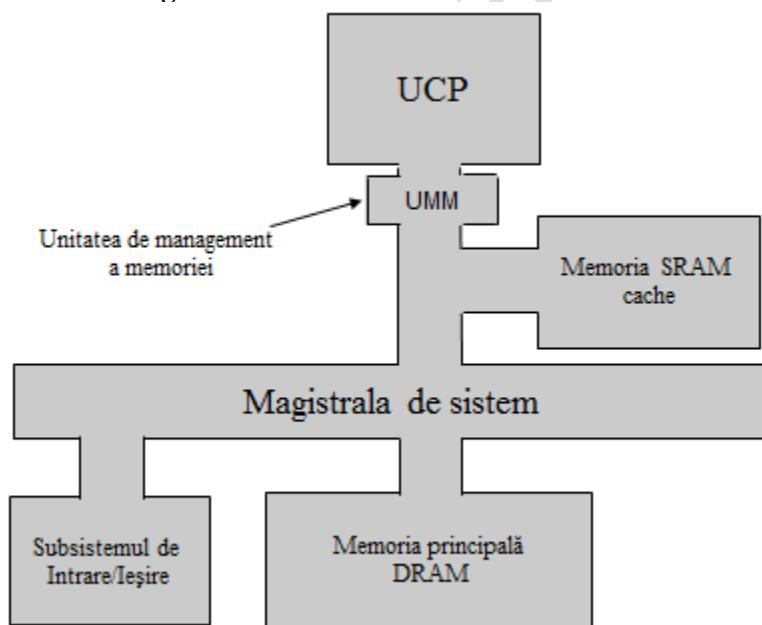
- frecvent, trafic inefficient MP-disc (de exemplu pentru segmente “mici”, transferul cu discul e complet inefficient – accese la nivel de sector = 512 octeți)
- Există în practică și implementări hibride segmentare – paginare

7.2. Exemple de gestionare a memoriei virtuale.

O schema a ierarhiei memoriei este prezentată alăturat.

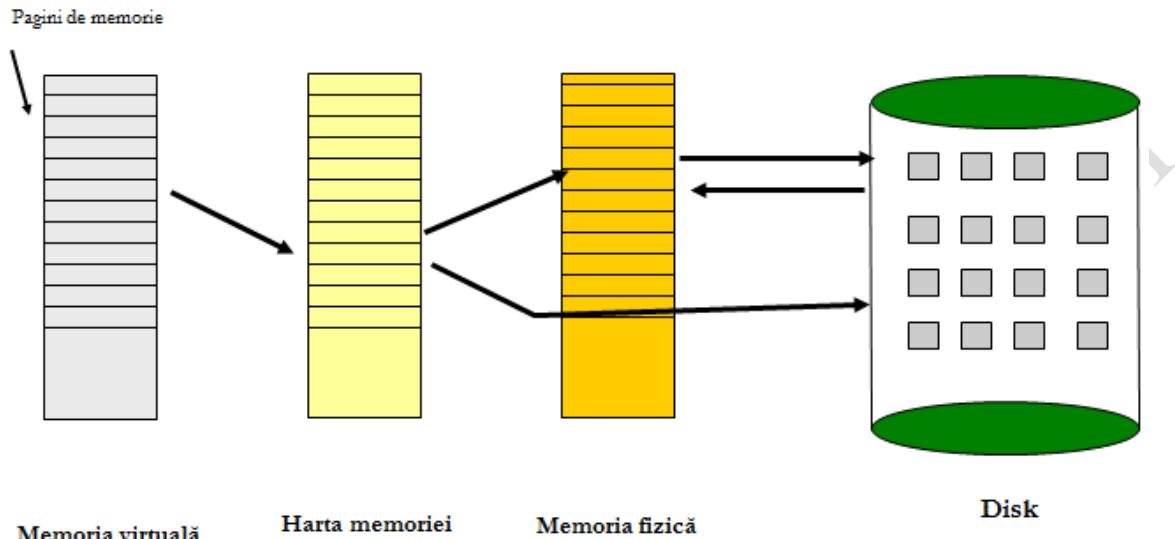


Unitatea de management a memoriei virtuale



7.2.1. Memoria virtuală

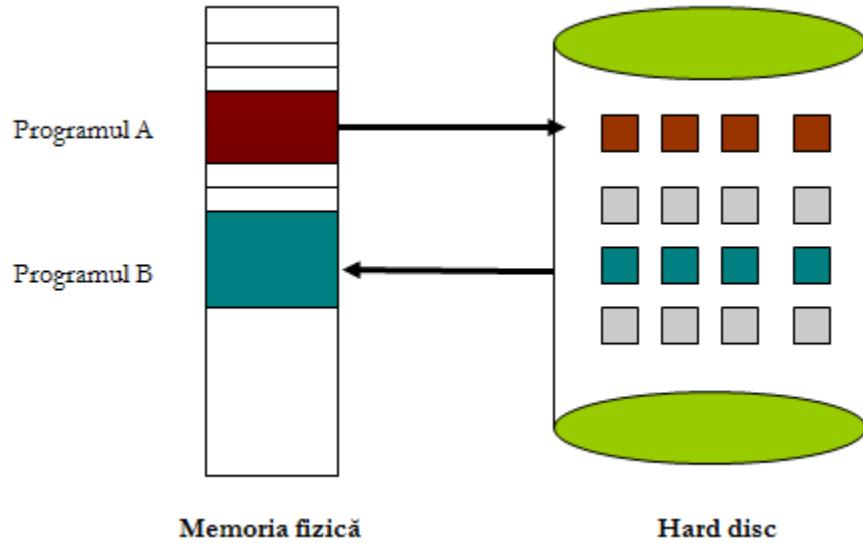
Reprezintă separarea conceptuală a memoriei logice disponibile pentru aplicații față de memoria fizică. În acest mod putem avea o memorie virtuală de dimensiuni mari chiar cu o memorie fizică de dimensiuni reduse :



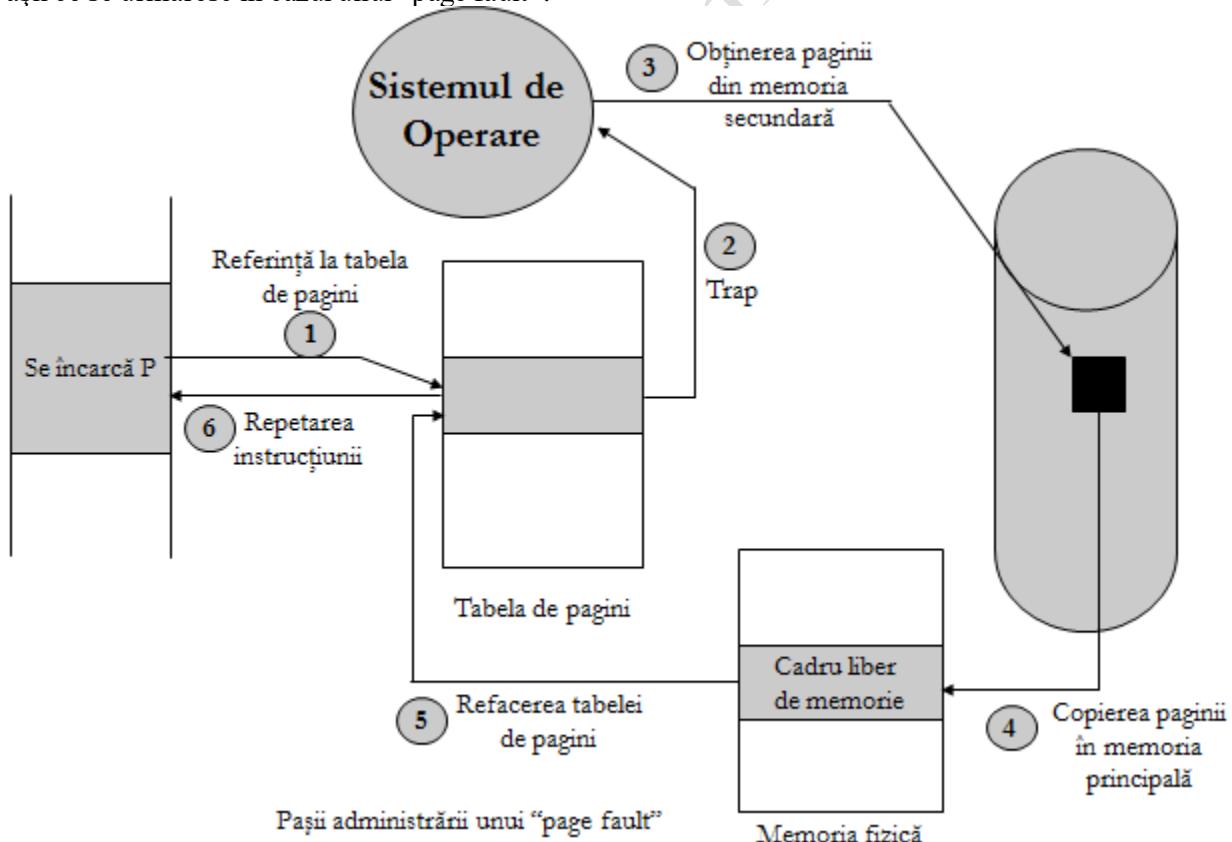
- În același sens, nu toate obiectele (date sau instrucțiuni) pot fi la un moment dat în memoria principală. Dacă avem memorie virtuală, atunci unele dintre obiecte se pot afla pe disc. Spațiul de adresare este de regulă împărțit în blocuri de lungime fixă – pagini.
- La un moment dat, paginile se află 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 și atunci sunt controlate de software și UCP nu face pauză.
- De regulă, UCP comută către alt task atunci când apare un acces la disc. Memoria cache și memoria principală au aceeași relație ca și cea existentă între memoria principală și disc.
- În orice moment, un calculator rulează mai multe procese, fiecare având propriul spațiu de adrese de memorie. Ar fi foarte costisitor să se dedice un întreg spațiu de adresare pentru fiecare proces, având în vedere că multe dintre procese folosesc doar o mică parte a spațiului propriu de adrese. A apărut astfel necesitatea partajării unei părți 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 protecție ce restricționează accesul proceselor la blocuri ce aparțin 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 suprapunerি (overlay).
- Blocurile de memorie în cazul memoriei virtuale se numesc *pagini* sau *segmente*. UCP folosește adrese virtuale ce sunt transluate (hardware 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) și ea nu se află în memorie atunci ea este adusă de pe disc și se re-executează instrucțiunea



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



În cazul supra-alocării memoriei trebuie să renunțăm 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 și se eliberează.

8. Arhitecturi evolute

Arhitectura Setului de Instrucțiuni

Arhitectura setului de instrucțiuni (ISA - Instruction Set Architecture) este folosită pentru a abstractiza funcționarea internă a unui procesor. ISA definește "personalitatea" unui procesor: cum funcționează 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 unitățile de execuție (dacă sunt mai multe), registre și cache. Astfel, o ISA poate fi implementată de diferite microarhitecturi: la ARM, ARMv6 este un exemplu de ISA, și 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 interacțiunea 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 următoarele caracteristice:

- set redus de instrucțiuni: ~100-200; complexitatea care este eliminată din ISA este de fapt transferată programatorului în limbaj de asamblare și compilatoarelor; codul conține mai multe instrucțiuni, și în multe cazuri (mai ales în trecut (anii '70, '80)) acest fapt constituie o problemă
- instrucțiunile sunt de obicei codificate pe lungime fixă; permite o implementare mai simplă a UC
- execuția instrucțiunilor într-un singur ciclu (folosind pipeline)
- **arhitectură load-store** - numai instrucțiunile 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 și 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 și ISA, controlul microprogramului - instrucțiunile sunt implementate în microcod;
- familii de arhitecturi: IA-32, x86-64

Exemplu:

La IA-32:

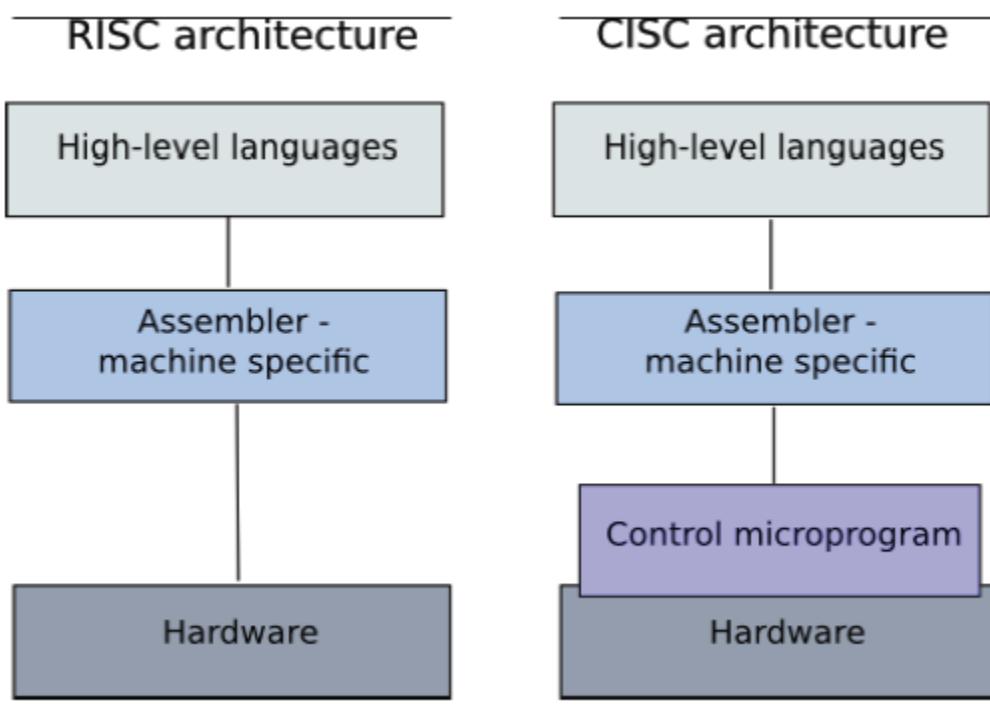
add [ebx], eax (se adună operandul de la adresa data de registrul ebx cu cel din registrul eax și rezultatul se stochează la adresa primului operand)

La o arhitectură 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 arhitectură RISC am avea intai de incarcat operanții din memorie folosind instrucțiuni de tip LOAD și apoi să apelam instrucțiunea pentru inmultire, în final stocând 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 micșora numărul de cicli pe instrucțiune, în timp ce la CISC este invers.

Totuși, lucrurile nu stau atât de simplu. Deși arhitecturile RISC au fost devansate în anii 80 de către CISC datorită lipsei software-ului și al unui segment de piață, precum și datorită prețului 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 apropiie, majoritatea procesoarelor actuale având o arhitectură hibridă, de exemplu Intel Pentium având un interpretor CISC peste un nucleu RISC.

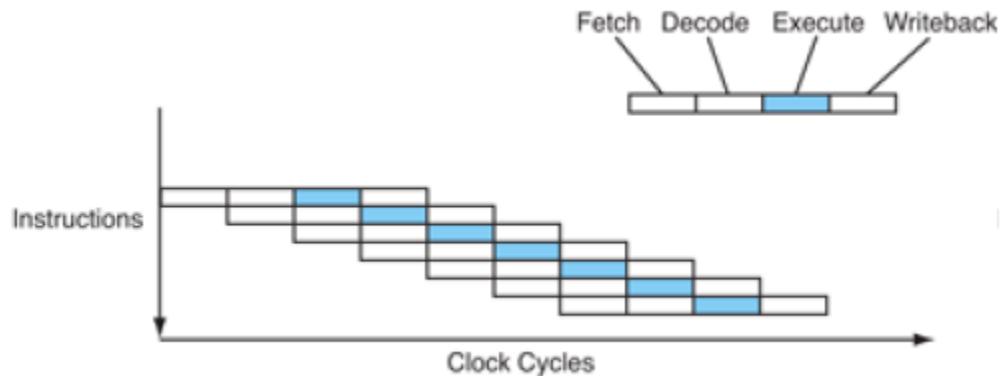
RISC	CISC
costuri mici de fabricație, mai puține tranzistoare	hardware mai complex, costuri mai mari
consum redus de energie	consum mai mare
mai multe registre	mai mult hardware, controlul microprogramului
mai multe linii de cod per program	puțin cod

8.2. Arhitecturi paralele, exemple.

8.2.1. Paralelism - Pipelining

Chiar dacă un program este în mod tradițional interpretat secvențial, instrucțiune cu instrucțiune, 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 arhitectură RISC este cel din 4 etape:

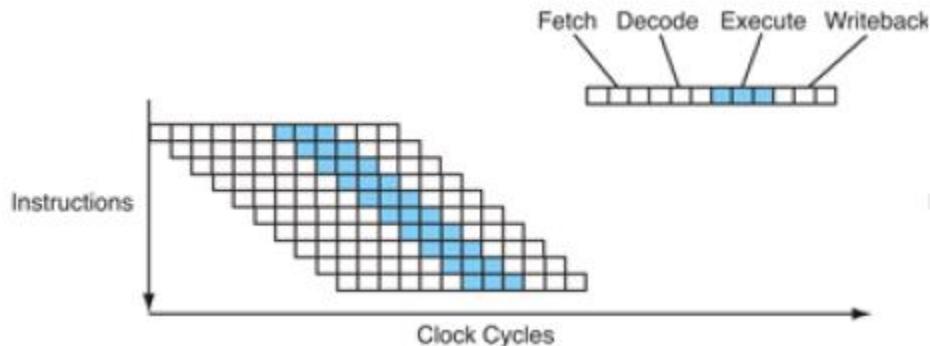
1. Instruction Fetch,
2. Instruction Decode,
3. Execute,
4. Register Write Back.

- În stagiul **Fetch** se aduce instrucțiunea din memorie, de la adresa dată de contorul program (PC - *program counter*).

- În stagiul **Decode** se decodifică instrucțiunea
- în **Execute** se execută efectiv operația de către unitatea aritmetico-logica,
- în **Writeback** se stochează rezultatele în regiștri sau memorie.

Pentru CISC, avem de obicei un număr mai mare de etape (>12).

Banda de asamblare *superpipeline* este mai lungă decât banda obișnuită (cele 4 stagii F, D, E, W sunt împărțite la rândul lor în mai multe stagii).

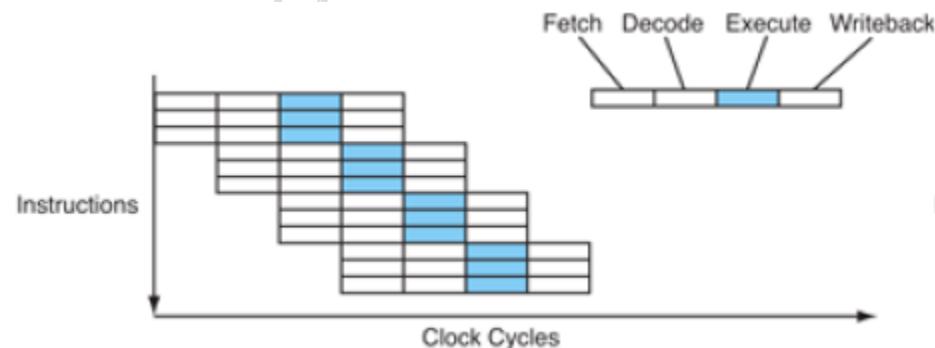


Teoretic cu cât banda de asamblare este mai segmentată, mai multe instrucțiuni pot fi executate în paralel și, fiecare stagiul făcând mai puține operații, se poate mări frecvența ceasului. Însă, alți factori caracteristici ai design-ului pot scădea performanțele și nu întotdeauna un procesor cu superpipelining 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 performanța.

Pentru procesoarele RISC, unde instrucțiunile se execută într-un singur ceas, această durată de execuție este dată de instrucțiunea cea mai lungă.

Banda de asamblare mărește productivitatea (throughput-ul), adică numărul de instrucțiuni executate în unitatea de timp, însă timpul de execuție (latența) a unei instrucțiuni nu se micșorează, ci se poate chiar mări.

Pentru a paraleliza și mai mult execuția instrucțiunilor, **procesoarele superscalare** introduc benzi de asamblare independente ce funcționează în paralel. Se mărește astfel numărul de resurse și complexitatea procesorului, însă mai multe instrucțiuni pot fi aduse, decodificate și executate în același timp, după cum este ilustrat.



8.2.2. Hazard

În cazul unui procesor cu bandă de asamblare, pentru a obține performanță maximă, este necesar ca o instrucțiune să fie prezentă în fiecare stagiul 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, situație care poartă numele de hazard.

8.2.2.1. Tipuri

Există trei tipuri de dependențe care pot conduce la hazard:

1. Structurale: când două instrucțiuni aflate în execuție în pipeline doresc acces la aceeași structură hardware în același timp.

Exemplu:

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

- ambele instrucțiuni 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 același registru sau aceeași locație de memorie și cel puțin una dintre ele este o scriere

• **RAW (Read after Write):** instrucțiunea I_{i+1} citește o locație 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 I_{i+1} scrie o locație de memorie care este citită de instrucțiunea 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 I_{i+1} scrie o locație de memorie care este scrisă și de instrucțiunea I_i

Exemplu:

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

- este necesar ca cele două scrierii 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 instrucțiuni care trebuie citită nu se cunoaște 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, însă 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 trebuie execuțiate, și în schimb trebuie adusă instrucțiunea i4, deci se face ‘flush’ la pipeline și se aduce instrucțiune i4, pierzându-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 execuției instrucțiunilor cu dependințe; aşa numitele *pipeline bubbles* sau *pipeline stalls*.

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

Exemplu:

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

Rezultatul obținut pentru instrucțiunea k în unitatea aritmetică-logică, în stagiul de Execute este forwardat stagiu lui Decode al instrucțiunii k+1.

Out-of-order execution(OOE) este o tehnică de planificare dinamică(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 tehnică 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 însă nu și RAW, fiind folosită și în algoritm Tomásulo 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 11, 12, 13  
mul 15, 12, 14  
add 16, 12, 17
```

În cazul hazardelor de control, putem amâna aducerea instrucțiunilor din branch prin adăugarea de întârzieri în pipeline sau intercalarea altor instrucțiuni independente de acel branch, însă soluția cea mai bună este predicția ramificațiilor (**branch prediction**) care minimizează numărul de cicli pierduți. Această predicție poate fi statică, facută de către compilator prin optimizări de tipul *loop unrolling* (de exemplu în buclă *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 către 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: Larsen and Keller Education; 5 edition (September 30, 2011), ISBN-13: 978-0123838728

2. Ken Spiner, *Computer Architecture* Publisher: Morgan Kaufmann 2020, ISBN: 1641724641
3. Null J., Lobur J., *The essentials of computer organization and architecture*, Jones & Bartlett Learning, 3rd edition (December 17, 2010), ISBN-13: 978-1449600068.
4. P. Chakraborty, *Computer Organisation and Architecture*, Publisher : Chapman and Hall/CRC, 2020, ISBN-13 : 978-0367255732
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.

Anexa I: Procesorul multi-core

Un procesor multi-core este un procesor pe un singur circuit integrat cu două sau mai multe unități de procesare separate, numite nuclee, fiecare dintre ele citește și execută instrucțiuni de program. Instrucțiunile sunt instrucțiuni obișnuite ale CPU (cum ar fi adăugarea, mutarea datelor și ramificarea), dar un singur procesor poate rula instrucțiuni pe nuclee separate în același timp, crescând viteza generală pentru programele care acceptă multithreading sau alte tehnici de calcul paralel. Producătorii integrează de obicei miezurile pe o singură matră de circuit integrat (cunoscut sub numele de cip multiprocesor sau CMP) sau pe mai multe mătrițe într-un singur pachet de cip. Microprocesoarele utilizate în prezent în aproape toate computerele personale sunt multi-core.

Un procesor multi-core implementează multiprocesare într-un singur pachet fizic. Designerii pot cupla strâns sau lejer nucleele într-un dispozitiv cu mai multe nuclee. De exemplu, nucleele pot partaja sau nu memoria cache și pot implementa metode de comunicare între nuclee de transmitere a mesajelor sau de memorie partajată. Topologii comune de rețea utilizate pentru interconectarea nucleelor pot fi de tip magistrală, inel, rețea bidimensională și bară transversală. Sistemele omogene multi-core includ numai kerneluri identice; sistemele multi-core eterogene au nuclee care nu sunt identice (de exemplu, big-LITTLE au nuclee eterogene care împart același set de instrucțiuni, în timp ce unitățile de procesare accelerată AMD au nuclee care nu împart același set de instrucțiuni). La fel ca în cazul sistemelor cu un singur procesor, nucleele din sistemele cu mai multe nuclee pot implementa arhitecturi precum VLIW, superscalare, vectoriale sau multithreading.

VLIW (Very Long Instruction Word) descrie o arhitectură în care un compilator sau un preprocesor descompune instrucțiunile programului în operațiuni de bază care pot fi efectuate de procesor în paralel (adică, în același timp). Aceste operații sunt incluse într-un câvânt de instrucțiuni foarte lung pe care procesorul îl poate apoi dezasambla fără analize suplimentare, dând fiecare operație unei unități funcționale adecvate.

VLIW este uneori văzut ca următorul pas dincolo de arhitectura RISC (reduced instruction set computing), care funcționează, de asemenea, cu un set limitat de instrucțiuni de bază și poate executa, de obicei, mai multe instrucțiuni la un moment dat (o caracteristică denumită SUPERSCALAR). Principalul avantaj al procesoarelor VLIW este mutarea complexității de la hardware la software, ceea ce înseamnă că hardware-ul poate fi mai mic, mai ieftin și necesită mai puțină putere consumată pentru a

funcționa. Provocarea este de a proiecta un compilator sau un pre-procesor care este suficient de intelligent pentru a decide cum să construiască cuvintele de instrucțiuni foarte lungi. Dacă se face preprocessarea dinamică pe măsură ce programul este rulat, performanța poate fi o problemă.

Familia de procesoare Crusoe de la Transmeta folosește cuvinte de instrucțiuni foarte lungi care sunt asamblate de un pre-procesor care se află într-un cip de memorie tip flash (tip de memorie nevolatilă care organizează datele în unități numite blocuri pentru a le sterge). Deoarece procesorul nu trebuie să aibă capacitatea de a descoperi și planifica operațiuni paralele, procesorul conține doar aproximativ o pătrime din tranzistoarele unui procesor obișnuit. Cerința mai mică de energie permite computerelor bazate pe tehnologia Crusoe să fie operate cu baterie aproape intraga și fără a fi reîncărcate. Procesoarele Crusoe emulează setul de instrucțiuni pentru procesorul Intel x86. Teoretic, pre-procesoarele ar putea fi proiectate și pentru a emula alte arhitecturi de procesoare.

Procesorul SUPERSCALAR implementează o formă de paralelism numită paralelism la nivel de instrucțiune într-un singur procesor. Spre deosebire de un procesor scalar, care poate execuția cel mult o singură instrucțiune pe un ciclu de ceas, un procesor superscalar poate execuția mai mult de o instrucțiune în timpul unui ciclu de ceas prin trimitera simultană a mai multor instrucțiuni către diferite unități de execuție de pe socketul procesorului. Prin urmare, acesta permite mai multe fluxuri de procesare (numărul de instrucțiuni care pot fi execuțiate într-o unitate de timp) decât ar fi posibil altfel la o anumită frecvență. Fiecare unitate de execuție nu este un procesor separat (sau un nucleu dacă procesorul este un procesor cu mai multe nuclee), ci o resursă de execuție într-un singur CPU, cum ar fi o unitate logică aritmetică.

În taxonomia tip Flynn, un procesor superscalar cu un singur nucleu este clasificat ca un procesor SISD (single instruction stream, single data stream), deși un procesor superscalar cu un singur nucleu care acceptă operațiuni vectoriale scurte ar putea fi clasificat ca SIMD (single instruction stream, multiple data streams). Un procesor superscalar multi-core este clasificat ca un procesor MIMD (multiple instruction streams, multiple data streams).

*În timp ce un procesor superscalar este considerat de obicei de tip **pipeline**, executarea superscalara și cea **pipeline** (înlănțuită) reprezintă tehnici diferite de îmbunătățire a performanței. Primul execută mai multe instrucțiuni în paralel utilizând mai multe unități de execuție, în timp ce **pipeline** execută mai multe instrucțiuni în aceeași unitate de execuție în paralel, împărțind unitatea de execuție în faze diferite.*

Tehnica superscalară este în mod tradițional asociată cu mai multe caracteristici de identificare (în cadrul unui CPU dat):

- Instrucțiunile sunt emise dintr-un flux de instrucțiuni secvențial
- CPU verifică în mod dinamic dependențele de date între instrucțiuni în timpul rulării (față de verificarea software-ului în timpul compilării)
- CPU poate execuția mai multe instrucțiuni pe ciclu de ceas

Procesoarele VECTORIALE sau matriceale reprezintă CPU care implementează un set de instrucțiuni concepute pentru a funcționa eficient pe matrice mari unidimensionale de date numite vectori. Acest lucru este în contrast cu procesoarele scalare ale căror instrucțiuni funcționează numai pe elemente individuale și în contrast cu unele dintre aceleași procesoare scalare care au unități aritmetice SIMD sau SWAR (SIMD within a register) suplimentare. Procesoarele vectoriale pot îmbunătăți considerabil

performanța în special pentru simularea numerică și alte sarcini similare. Tehnicile de procesare vectorială funcționează și în hardware-ul consolelor de jocuri video dar și în acceleratoarele grafice.

Machinele vectoriale au dominat proiectarea supercomputerelor până în anii 1990 cand ieftinirea modelelor cu microprocesoare convenționale a dus la o scădere drastica a supercalculatoarelor vectoriale.

*În Arhitectura Sistemelor de Calcul, **MULTITHREADING**-ul este capacitatea unei unități centrale de procesare (CPU) (sau a unui singur nucleu într-un procesor cu mai multe nuclee) de a furniza mai multe fire de execuție simultan, susținute de sistemul de operare. Această abordare diferă de multiprocesare. Într-o aplicație cu mai multe fire de execuție, acestea partajează resursele unui singur nucleu sau mai multor nuclee care includ unitățile de calcul, cache-urile CPU și "translation lookaside buffer" (TLB) care este un cache MMU folosit la accelerarea accesului la memoria utilizator.*

Acolo unde sistemele de procesare multiplă includ mai multe unități complete de procesare în unul sau mai multe nuclee, multithreading-ul urmărește să opimeze utilizarea unui singur nucleu prin paralelismului la nivel de thread (fir), precum și a paralelismului la nivel de instrucțiune. Deoarece cele două tehnici sunt complementare, ele sunt combinate în aproape toate arhitecturile de sistem moderne cu procesoare multiple multithreading și cu procesoare cu mai multe nuclee multithreading.

Procesoarele cu mai multe nuclee sunt utilizate pe scară largă în multe domenii de aplicații, inclusiv de uz general, încorporat, de rețea, procesare digitală a semnalului (DSP) și grafică (GPU). Numărul de nuclee crește chiar și până la zeci, iar pentru cipurile specializate peste 10.000, și în supercalculatoare (adică grupuri de cipuri) numărul poate depăși 10 milioane (și într-un caz până la 20 de milioane de elemente de procesare în total, în plus față de gazdă). procesoare).

Îmbunătățirea performanței obținute prin utilizarea unui procesor multi-core depinde foarte mult de algoritmii software utilizati și de implementarea acestora. În special, căștigurile posibile sunt limitate de fracțiunea de software care poate rula în paralel simultan pe mai multe nuclee; acest efect este descris de legea lui Amdahl. În cel mai bun caz, aşa-numitele probleme jenant de paralele pot realiza factori de accelerare aproape de numărul de nuclee sau chiar mai mult dacă problema este împărțită suficient pentru a se potrivi în memoria cache-urilor fiecărui nucleu, evitând utilizarea memoriei sistemului principal mult mai lentă. Majoritatea aplicațiilor, totuși, nu sunt accelerate atât de mult decât dacă programatorii investesc o cantitate prohibitivă de efort în re-factorizarea întregii probleme.

Paralelizarea software-ului este un subiect important de cercetare în curs. Cointegrarea aplicațiilor multiprocesor oferă flexibilitate în proiectarea arhitecturii de rețea. Adaptabilitatea în cadrul modelelor paralele este o caracteristică suplimentară a sistemelor care utilizează aceste protocoale.

Termenii multi-core și dual-core se referă cel mai frecvent la un fel de unitate centrală de procesare (CPU), dar uneori sunt aplicații și procesoarelor de semnal digital (DSP) și sistemului pe un cip (SoC). Termenii sunt în general folosiți numai pentru a se referi la microprocesoare multi-core care sunt fabricate pe aceeași matrice de circuit integrat ; matricele de microprocesor separate din același pachet sunt, în general, denumite printr-un alt nume, cum ar fi modul cu mai multe cipuri . Acest

articol folosește termenii „multi-core” și „dual-core” pentru procesoarele fabricate pe același circuit integrat, dacă nu se specifică altfel.

Spre deosebire de sistemele multi-core, termenul multi-CPU se referă la mai multe unități de procesare separate fizic (care conțin adesea circuite speciale pentru a facilita comunicarea între ele).

Termenii multi-core și masiv multi-core sunt uneori folosiți pentru a descrie arhitecturi multi-core cu un număr deosebit de mărci de nuclee (de la zeci la mii).

Unele sisteme folosesc multe nuclee moi de microprocesor plasate pe un singur FPGA . Fiecare „nucleu” poate fi considerat un „nucleu de proprietate intelectuală a semiconducțoarelor”, precum și un nucleu CPU.

În timp ce tehnologia de fabricație se îmbunătățește, reducând dimensiunea porților individuale, limitele fizice ale microelectronicii bazate pe semiconductori au devenit o preocupare majoră de proiectare. Aceste limitări fizice pot provoca probleme semnificative de disipare a căldurii și de sincronizare a datelor. Sunt folosite diverse alte metode pentru a îmbunătății performanța procesorului. Unele metode de paralelism la nivel de instrucție (ILP), cum ar fi pipeliningul superscalar, sunt potrivite pentru multe aplicații, dar sunt ineficiente pentru altele care conțin cod greu de prezis. Multe aplicații sunt mai potrivite pentru paralelismul la nivel de fir (TLP - thread-level parallelism) și mai multe procesoare independente sunt utilizate în mod obișnuit pentru a crește TLP-ul general al unui sistem. O combinație de spațiu disponibil sporit (datorită proceselor de producție rafinate) și cererea de TLP crescută condus la dezvoltarea procesoarelor multi-core.

Mai multe motive de afaceri determină dezvoltarea arhitecturilor multi-core. Timp de zeci de ani, a fost posibil să se îmbunătățească performanța unui procesor prin micșorarea suprafeței circuitului integrat (IC), ceea ce a redus costul pe dispozitiv pe IC. Alternativ, pentru aceeași zonă de circuit, ar putea fi utilizate mai mulți tranzistori în proiectare, ceea ce a sporit funcționalitatea, în special pentru arhitecturile CISC (complex instruction set computing). Frecvența ceasului a crescut, de asemenea, cu ordine de mărime în deceniile de la sfârșitul secolului al XX-lea, de la câțiva megaherți în anii 1980 la câțiva gigaherți la începutul anilor 2000.

Pe măsură ce rata de îmbunătățire a vitezei de ceas a încetinit, s-a urmărit utilizarea sporită a calculului paralel sub formă de procesoare cu mai multe nuclee pentru a îmbunătății performanța generală de procesare. Pe același cip CPU au fost utilizate mai multe nuclee, ceea ce ar putea duce apoi la vânzări mai bune de cipuri CPU cu două sau mai multe nuclee. De exemplu, Intel a produs un procesor cu 48 de nuclee pentru cercetare în cloud computing; fiecare nucleu are o arhitectură x86 .

Factori tehniči

Deoarece producătorii de computere au implementat de mult timp proiecte de procesare simetrică multiprocesare (SMP) folosind CPU-uri discrete, problemele privind implementarea arhitecturii procesoarelor multi-core și suportarea acestora cu software sunt binecunoscute.

În plus:

Utilizarea unui design de bază de procesare dovedit fără modificări arhitecturale reduce riscul de proiectare în mod semnificativ.

Pentru procesoarele de uz general, o mare parte din motivația pentru procesoarele cu mai multe nuclee provine din câștigurile mult diminuate în performanța procesorului prin creșterea frecvenței de operare.

Acum lucru se datorează a trei factori primari:

- Limitarea memoriei ; decalajul tot mai mare dintre viteza procesorului și a memoriei. Care, de fapt, face ca dimensiunile cache-ului să fie mai mari pentru a masca latența memoriei. Această lucru ajută numai în măsura în care latența de bandă a memoriei nu reprezintă un blocaj în performanță.
- Limitarea ILP ; dificultatea crescândă de a găsi suficient paralelism într-un singur flux de instrucțiuni pentru a menține ocupat un procesor cu un singur nucleu de înaltă performanță.
- Limitarea de putere ; tendința de a consuma energie în creștere exponențială (și astfel de a genera și căldură în creștere exponențială) cu fiecare creștere factorială a frecvenței de operare. Această creștere poate fi atenuată prin „strângerea” procesorului prin utilizarea unor urme mai mici pentru aceeași logică. De perete de putere ipostazele de fabricație, proiectarea sistemului și probleme de implementare , care nu au fost justificate în fața câștigurilor diminuate în performanță datorită peretelui de memorie și de perete ILP .

Pentru a continua să ofere îmbunătățiri regulate de performanță pentru procesoarele de uz general, producători precum Intel și AMD s-au orientat către design-uri multi-core, sacrificând costuri de producție mai mici pentru performanțe mai mari în unele aplicații și sisteme. Arhitecturile multi-core sunt în curs de dezvoltare, dar la fel sunt și alternativele. Un competitor deosebit de puternic pentru piețele stabilite este integrarea în continuare a funcțiilor periferice în cip.

Avantaje

Apropierea mai multor nuclee CPU pe aceeași matrice permite circuitelor de coerență a memoriei cache să funcționeze la o rată de ceas mult mai mare decât ceea ce este posibil dacă semnalele trebuie să călătorescă în afara cipului. Combinarea CPU-urilor echivalente pe o singură matrice îmbunătățește semnificativ performanța operațiunilor cache Snoop (alternativă: Bus snooping). Mai simplu, aceasta înseamnă că semnalele dintre diferențe procesoare parcurg distanțe mai scurte și, prin urmare, acele semnale se degradează mai puțin. Aceste semnale de calitate superioară permit trimiterea mai multor date într-o anumită perioadă de timp, deoarece semnalele individuale pot fi mai scurte și nu trebuie să fie repetate la fel de des.

Presupunând că matricea se poate încadra fizic în pachet, modelele CPU cu mai multe nuclee necesită mult mai puțin spațiu pe placa de circuit imprimat (PCB) decât designurile SMP cu mai multe cipuri. De asemenea, un procesor dual-core folosește puțin mai puțină putere decât două procesoare single-core cuplate, în principal din cauza puterii scăzute necesare pentru a transmite semnale externe cipului. În plus, nucleele partajează anumite circuite, cum ar fi memoria cache L2 și interfața cu magistrala frontală (FSB). În ceea ce privește tehnologiile concurente pentru zona disponibilă a matricei de siliciu, proiectarea cu mai multe nuclee poate utiliza modele dovedite de biblioteci de nuclee ale procesorului. și poate produce un produs cu risc mai mic de eroare de proiectare decât conceperea unui nou design cu nucleu mai larg. De asemenea, adăugarea mai multor cache suferă de scăderea randamentelor.

Cipurile multi-core permit, de asemenea, performanțe mai mari la energie mai mică. Acestea poate fi un factor important în dispozitivele mobile care funcționează cu baterii. Deoarece fiecare nucleu dintr-un procesor multi-core este în general mai eficient din punct de vedere energetic, cipul devine mai eficient decât având un singur nucleu monolit mare. Acest lucru permite performanțe mai mari cu mai puțină energie. O provocare în acest sens este totuși costul suplimentar de scriere a codului paralel.

Dezavantaje

Maximizarea utilizării resurselor de calcul furnizate de procesoarele multi-core necesită ajustări atât la suportul sistemului de operare (OS), cât și la software-ul aplicației existente. De asemenea, capacitatea procesoarelor multi-core de a crește performanța aplicației depinde de utilizarea mai multor fire de execuție în cadrul aplicațiilor.

Integrarea unui cip cu mai multe nuclee poate reduce randamentele producției de cip. Ele sunt, de asemenea, mai dificil de gestionat termic decât modelele cu un singur nucleu cu densitate mai mică. Intel a contracară parțial această primă problemă prin crearea modelelor sale quad-core prin combinarea a două nuclee dual-core pe o singură matrice cu un cache unificat, prin urmare pot fi utilizate orice două matrice dual-core care funcționează, spre deosebire de producerea a patru nuclee pe un o singură matrice și necesită ca toate cele patru să funcționeze pentru a produce un procesor quad-core. Din punct de vedere arhitectural, în cele din urmă, proiectele cu un singur procesor pot folosi mai bine suprafața de siliciu decât nucleele multiprocesare, așa că un angajament de dezvoltare față de această arhitectură poate duce la riscul depășirii. În cele din urmă, puterea brută de procesare nu este singura constrângere asupra performanței sistemului. Două nuclee de procesare care partajează aceeași magistrală de sistem și aceeași lățime de bandă de memorie limitează avantajul de performanță din lumea reală. Într-un raport din 2009, Dr. Jun Ni a arătat că, dacă un singur nucleu este aproape de a fi limitat la lățimea de bandă a memoriei, atunci trecerea la dual-core ar putea aduce o îmbunătățire cu 30% până la 70%; dacă lățimea de bandă a memoriei nu este o problemă, atunci se poate aștepta o îmbunătățire de 90%; în orice caz, Legea lui Amdahl face această afirmație dubioasă. [13] Ar fi posibil ca o aplicație care folosea două procesoare să ajungă să ruleze mai rapid pe unul cu un singur nucleu, dacă comunicarea între procesoare ar fi factorul limitativ, care ar conta ca îmbunătățire de peste 100%.

Hardware

Tendința în dezvoltarea procesoarelor a fost către un număr din ce în ce mai mare de nuclee, deoarece procesoarele cu sute sau chiar mii de nuclee devin teoretic posibile. [14] În plus, cipurile multi-core amestecate cu nuclee multi-threading simultane, memorie pe cip și nuclee „eterogene” (sau asimetrice) cu scop special promit în continuare căștiguri de performanță și eficiență, în special în procesarea aplicațiilor multimedia, de recunoaștere și de rețea. De exemplu, un nucleu big.LITTLE include un nucleu de înaltă performanță (numit „mare”) și un nucleu de putere redusă (numit „MIC”). Există, de asemenea, o tendință de îmbunătățire a eficienței energetice, concentrându-se pe performanța pe watt cu management avansat al puterii cu granulație fină sau

ultrafine și scalarea dinamică a tensiunii și frecvenței (adică computere laptop și playere media portabile).

Cipurile proiectate de la început pentru un număr mare de nuclee (în loc să fi evoluat din designuri cu un singur nucleu) sunt uneori denumite modele cu mai multe nuclee, subliniind diferențele calitative.

Arhitectură

Compoziția și echilibrul nucleelor în arhitectura multi-core arată o mare varietate. Unele arhitecturi folosesc un design de bază repetat în mod consecvent ("omogen"), în timp ce altele folosesc un amestec de nuclee diferite, fiecare optimizat pentru un rol diferit, "eterogen".

Modul în care sunt implementate și integrate mai multe nuclee afectează în mod semnificativ atât abilitățile de programare ale dezvoltatorului, cât și așteptările consumatorului cu privire la aplicații și interactivitate față de dispozitiv. Un dispozitiv anunțat ca fiind octa-core va avea nuclee independente numai dacă este anunțat ca True Octa-core sau stil similar, spre deosebire de a fi doar două seturi de quad-core, fiecare cu viteze fixe de ceas.

Articolul „CPU Designers debate multi-core future” de Rick Merritt, EE Times 2008, include următoarele comentarii:

Chuck Moore [...] a sugerat că computerele ar trebui să fie ca telefoanele mobile, folosind o varietate de nuclee de specialitate pentru a rula software modular programat de o interfață de programare a aplicațiilor de nivel înalt.

[...] Atsushi Hasegawa, un inginer șef senior la Renesas, a fost în general de acord. El a sugerat că utilizarea de către telefonul mobil a multor nuclee de specialitate care lucrează în concert este un model bun pentru viitoarele proiecte multi-core.

[...] Anant Agarwal, fondatorul și directorul executiv al startup-ului Tilera, a fost de părere opusă. El a spus că cipurile multi-core trebuie să fie colecții omogene de nuclee de uz general pentru a menține modelul software simplu.

Efecte software

O versiune învechită a unei aplicații antivirus poate crea un fir nou pentru un proces de scanare, în timp ce firul său GUI așteaptă comenzi de la utilizator (de exemplu, anulează scanarea). În astfel de cazuri, o arhitectură cu mai multe nuclee prezintă puține beneficii pentru aplicația în sine, din cauza unui singur fir care face toate sarcinile grele și a incapacității de a echilibra munca uniform pe mai multe nuclee. Programarea unui cod cu adevărat multithread necesită adesea o coordonare complexă a firelor de execuție și poate introduce cu ușurință erori subtile și greu de găsit din cauza împetrite a procesării datelor partajate între fire (vezi siguranța firelor). În consecință, un astfel de cod este mult mai dificil de depanat decât codul cu un singur thread atunci când se rupe. S-a perceput o lipsă de motivație pentru scrierea aplicațiilor threaded la nivel de consumator din cauza rarității relative a cererii la nivel de consumator pentru utilizarea maximă a hardware-ului computerului. De asemenea,

sarcinile seriale, cum ar fi decodarea algoritmilor de codificare a entropiei utilizată în codurile video, sunt imposibil de paralelizat, deoarece fiecare rezultat generat este folosit pentru a ajuta la crearea următorului algoritm de decodare a entropiei.

Având în vedere accentul tot mai mare pus pe designul de cipuri cu mai multe nuclee, ca urmare a problemelor grave de consum termic și de energie cauzate de orice creștere semnificativă suplimentară a vitezei de ceas a procesorului, măsura în care software-ul poate fi multithread pentru a profita de aceste noi cipuri este probabil să fie singura cea mai mare constrângere asupra performanței computerului în viitor. Dacă dezvoltatorii nu sunt capabili să proiecteze software pentru a exploata pe deplin resursele oferite de mai multe nuclee, atunci ei vor atinge în cele din urmă un plafon de performanță de netrecut.

Piața telecomunicațiilor a fost una dintre primele care au avut nevoie de un nou design de procesare paralelă a pachetelor de căi de date, deoarece a existat o adoptare foarte rapidă a acestor procesoare cu mai multe nuclee pentru calea de date și planul de control. Aceste MPU-uri vor înlocui procesoarele de rețea tradiționale care se bazau pe microcod sau picocode proprietar.

Tehnicile de programare paralelă pot beneficia direct de mai multe nuclee. Unele modele de programare paralelă existente, cum ar fi Cilk Plus, OpenMP, OpenHMPP, FastFlow, Skandium, MPI și Erlang pot fi utilizate pe platforme multi-core. Intel a introdus o nouă abstractizare pentru paralelismul C++ numită TBB. Alte eforturi de cercetare includ Codeplay Sieve System, Cray's Chapel, Sun's Fortress și IBM X10.

Procesarea multi-core a afectat, de asemenea, capacitatea de dezvoltare a software-ului de calcul modern. Dezvoltatorii care programează în limbi mai noi ar putea descoperi că limbile lor moderne nu acceptă funcționalitatea multi-core. Acest lucru necesită apoi utilizarea bibliotecilor numerice pentru a accesa codul scris în limbaje precum C și Fortran, care efectuează calcule matematice mai rapid decât limbaje mai noi precum C#. MKL de la Intel și ACML de la AMD sunt scrise în aceste limbi native și profită de procesarea multi-core. Echilibrarea volumului de lucru al aplicațiilor între procesoare poate fi problematică, mai ales dacă acestea au caracteristici de performanță diferite. Există diferite modele conceptuale pentru a trata problema, de exemplu, folosind un limbaj de coordonare și blocuri de construcție a programelor (biblioteci de programare sau funcții de ordin superior). Fiecare bloc poate avea o implementare nativă diferită pentru fiecare tip de procesor. Utilizatorii programează pur și simplu folosind aceste abstracții și un compilator intelligent alege cea mai bună implementare în funcție de context.

Gestionarea concurenței capătă un rol central în dezvoltarea aplicațiilor paralele. Pașii de bază în proiectarea aplicațiilor paralele sunt:

Compartimentare

Etapa de partiziionare a unui proiect are scopul de a expune oportunități de execuție paralelă. Prin urmare, accentul se pune pe definirea unui număr mare de sarcini mici pentru a produce ceea ce se numește o descompunere fină a unei probleme.

Comunicare

Sarcinile generate de o partitie sunt destinate sa fie executate concomitent, dar, in general, nu pot fi executate independent. Calculul care trebuie efectuat intr-o sarcina va necesita de obicei date asociate cu o alta sarcina. Datele trebuie apoi transferate intre sarcini pentru a permite calculului sa continue. Acest flux de informatie este specificat in faza de comunicare a unui proiect.

Aglomerare

In a treia etapa, dezvoltarea trece de la abstract la concret. Dezvoltatorii revizuiesc deciziile luate in fazele de partitie si comunicare in vederea obtinerii unui algoritm care sa se execute eficient pe o anumita clasa de computere paralele. In special, dezvoltatorii iau in considerare dacă este utila combinarea, sau aglomerarea, sarcinilor identificate prin faza de partitie, astfel incat sa ofere un numar mai mic de sarcini, fiecare de dimensiune mai mare. Ei determina, de asemenea, dacă merită replicarea datelor și calculelor.

Cartografiere

In a patra și ultima etapa a proiectarii algoritmilor paraleli, dezvoltatorii specifică unde urmează să fie executată fiecare sarcină. Această problemă de mapare nu apare pe uniprocesoare sau pe computere cu memorie partajată care oferă programare automată a sarcinilor.

Pe de altă parte, pe partea de server, procesoarele multi-core sunt ideale deoarece permit multor utilizatori să se conecteze la un site simultan și să aibă fire independente de execuție. Acest lucru permite servere Web și servere de aplicații care au un randament mult mai bun.

Licențiere

Furnizorii pot licenta anumite programe „pe procesor”. Acest lucru poate da naștere la ambiguitate, deoarece un „procesor” poate consta fie dintr-un singur nucleu, fie dintr-o combinație de nuclee.

Initial, pentru unele dintre software-ul său de întreprindere, Microsoft a continuat să folosească un sistem de licențiere per-socket. Cu toate acestea, pentru unele software, cum ar fi BizTalk Server 2013, SQL Server 2014 și Windows Server 2016, Microsoft a trecut la licențierea per-core. Oracle Corporation numără un procesor AMD X2 sau un procesor Intel dual-core ca un singur procesor [necesită citare], dar utilizează alte valori pentru alte tipuri, în special pentru procesoarele cu mai mult de două nuclee.

Aplicații încorporate

Un sistem încorporat pe un card plug-in cu procesor, memorie, sursă de alimentare și interfețe externe

Calculul încorporat operează într-o zonă a tehnologiei procesoarelor distinctă de cea a computerelor „mainstream”. Aceleași impulsuri tehnologice către multi-core se aplică și aici. Într-adevăr, în multe cazuri, aplicația este o potrivire „naturală” pentru tehnologiile multi-core, dacă sarcina poate fi împărțită cu ușurință între diferitele procesoare.

În plus, software-ul încorporat este dezvoltat în mod obișnuit pentru o versiune hardware specifică, ceea ce face ca problemele de portabilitate a software-ului, codul moștenit sau sprijinirea dezvoltatorilor independenți să fie mai puțin critice decât este cazul pentru computere sau

computere de întreprindere. Ca rezultat, este mai ușor pentru dezvoltatorii să adopte noi tehnologii și, ca urmare, există o varietate mai mare de arhitecturi și furnizori de procesare multi-core.

Procesoare de rețea

Începând cu 2010, procesoarele de rețea multi-core au devenit mainstream, companii precum Freescale Semiconductor, Cavium Networks, Wintegra și Broadcom producând toate produse cu opt procesoare. Pentru dezvoltatorul de sistem, o provocare cheie este cum să exploateze toate nucleele din aceste dispozitive pentru a obține performanță maximă în rețea la nivel de sistem, în ciuda limitărilor de performanță inerente unui sistem de operare cu multiprocesare simetrică (SMP). Companii precum 6WIND furnizează software portabil de procesare a pachetelor conceput astfel încât planul de date de rețea să ruleze într-un mediu de cale rapidă în afara sistemului de operare al dispozitivului de rețea.

Procesare digitală a semnalului

În procesarea semnalului digital se aplică aceeași tendință: Texas Instruments are TMS320C6488 cu trei nuclee și TMS320C5441 cu patru nuclee, Freescale MSC8144 cu patru nuclee și MSC8156 cu șase nuclee (și ambele au declarat că lucrează la succesorii cu opt nuclee). Intrările mai noi includ familia Storm-1 de la Stream Processors, Inc cu 40 și 80 de ALU de uz general per cip, toate programabile în C ca motor SIMD și Picochip cu 300 de procesoare pe o singură matrice, axate pe aplicații de comunicare.

Sisteme eterogene

În calculul eterogen, în care un sistem folosește mai mult de un tip de procesor sau nuclee, soluțiile multi-core devin din ce în ce mai frecvente: Xilinx Zynq UltraScale+ MPSoC are un quad-core ARM Cortex-A53 și dual-core ARM Cortex-R5. Soluțiile software precum OpenAMP sunt folosite pentru a ajuta la comunicarea între procesoare.

Dispozitivele mobile pot folosi arhitectura ARM big-LITTLE.

Anexa II: Infineon TriCore

Designer:	Infineon Technologies AG
Biți	pe 32 de biți
Introdus	1999
Versiune	2
Proiecta	RISC
Tip	Încărcare – Stocare
Endianess	Puțin pentru memoria de date și registrele CPU
Deschis	Nu
Registre	16 AdrReg, 16 DataReg, PCXI, PSW, PC

TriCore este o arhitectură de microcontroler pe 32 de biți de la Infineon . Acesta reunește elementele unui nucleu de procesor RISC , un microcontroler și un DSP într-un cip .

În 1999, Infineon a lansat prima generație de AUDO (procesor unificat pentru automobile), care se bazează pe ceea ce compania descrie ca un „ nucleu de microcontroler unificat RISC / MCU / DSP ” pe 32 de biți , numit TriCore, care se află în 2011 la al patrulea său. generație, numită AUDO MAX (versiunea 1.6).

TriCore este o arhitectură dual core eterogenă, asimetrică, cu un procesor de control periferic care permite modurile utilizatorului și protecția sistemului de bază.

Familiile AUDO [1] ale Infineon vizează unitățile de control al motoarelor pe benzină și diesel (ECU), aplicații în vehicule hibride și electrice , precum și aplicații de transmisie, siguranță activă și pasivă și șasiu. De asemenea, vizează aplicații industriale, de exemplu aplicații optimizate de control al motoarelor și procesarea semnalului.

Diferite modele oferă diferite combinații de memorii, seturi periferice, frecvențe, temperaturi și ambalaje. Infineon oferă, de asemenea, software despre care se spune că ajută producătorii să îndeplinească standardele de siguranță SIL/ASIL [2] . Toți membrii familiei AUDO sunt compatibile cu binar și au aceleași instrumente de dezvoltare. Este disponibilă și o bibliotecă AUTOSAR care permite integrarea codului existent.

Siguranță

Portofoliul Infineon include microcontrolere cu caracteristici hardware suplimentare, precum și software-ul de siguranță SafeTcore și un IC watchdog. [3]

Familiile AUDO acoperă aplicații de siguranță, inclusiv suspensii active și sisteme de asistență pentru șofer, precum și controlul domeniului EPS și șasiu. Unele caracteristici ale portofoliului de produse sunt protecția memoriei, periferice redundante, unități MemCheck cu CRC integrate, ECC pe memorii, funcționalitate integrată de testare și depanare și FlexRay.

Anexa III - Estimarea grosiera a performanței procesorului folosind legea lui Amdahl

Estimarea prin comparare a performanțelor procesoarelor cu arhitectura similară, folosind legea lui Amdahl

Dacă software-ul dvs. care ruleaza pe un procesor folosește doar un singur nucleu, frecvența este un indicator decent al performanțelor procesorului. Cu toate acestea, dacă software-ul dvs. este capabil să utilizeze mai multe nuclee CPU, devine foarte dificil să estimăm performanța diferitelor modele de CPU, deoarece aproape niciun program nu va fi 100% eficient în utilizarea acestor nuclee. Problema

este să determinați exact cât de eficient este programul dvs. în utilizarea mai multor nuclee CPU (eficiența paralelizării) și să utilizați acea valoare pentru a estima performanța diferitelor modele de procesor.

Ce este legea lui Amdahl?

La cel mai elementar nivel, Legea lui Amdahl este o modalitate de a arăta că, dacă un program (sau o parte a unui program) nu este 100% eficient în utilizarea mai multor nuclee CPU, veți primi din ce în ce mai puține beneficii prin adăugarea mai multor nuclee. De la un anumit punct - care poate fi calculat matematic odată ce cunoașteți eficiența paralelizării - veți avea performanțe mai bune utilizând mai puține nuclee care rulează la o frecvență mai mare decât folosind mai multe nuclee care rulează la o frecvență mai mică.

Legea lui Amdahl:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)(B + \frac{1}{n}(1 - B))}$$

Unde:

$S(n)$ este accelerarea teoretică

$T(n)$ este timpul necesar unui algoritm pentru a se termina atunci când rulează n fire

B este fracția algoritmului care este strict în serie (deci $1 - B$ este cât de mult din program poate fi rulat în parallel)

Dacă nu vă ocupați în mod regulat de ecuații complexe, aceasta poate fi puțin descurajantă pentru o ecuație. Cu toate acestea, deoarece ne preocupă în primul rând viteza maximă care poate fi atinsă prin creșterea numărului de nuclee CPU, această ecuație se poate simplifica puțin în următoarele:

Formula de paralelizare:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

Unde:

$S(n)$ este accelerarea teoretică

P este fracția algoritmului care poate fi făcută paralelă

n este numărul de fire CPU

Ceea ce se spune în principiu este că cantitatea de accelerare pe care o va vedea un program prin utilizarea nucleelor se bazează pe cât de mult din program este serial (poate fi rulat doar pe un singur nucleu CPU) și cât de mult este paralel (poate fi împărțit). între mai multe nuclee CPU).

Pentru a utiliza această ecuație, mai întâi trebuie să determinați eficiența de paralelizare a programului dvs. Cu acest număr, puteți utiliza legea lui Amdahl și frecvența unui procesor pentru a estima destul de precis performanța aproape a oricărui procesor care utilizează o arhitectură similară

procesorului pe care l-ați folosit pentru testare. Deși sunteți cu siguranță invitat să urmați acest ghid în întregime, dacă sunteți mai preocupat de estimarea efectivă a performanței unui procesor decât de toate calculele din spatele acestuia, nu ezitați să treceți la secțiunea „Mod Ușor: Utilizarea unei foi de calcul Google Doc”.

Limitările legii lui Amdahl

Deși metoda descrisă mai sus este excelentă pentru a determina cât de mult dintr-un program poate fi rulat în paralel, aceasta (și Legea lui Amdahl în general) are câteva limitări:

Nu fiecare acțiune efectuată într-un program va avea aceeași cantitate de paralelizare. Dacă te uiți la rezultatele articolului nostru recent: Adobe Photoshop CC CPU Multi-threading Performance , vei observa că câte nuclee CPU poate folosi Photoshop variază foarte mult în funcție de ceea ce faci de fapt. Puteți atenua oarecum această limitare testând diverse sarcini și calculând eficiența paralelizării pentru fiecare sarcină în mod individual (care este ceea ce am făcut), dar, în funcție de program, este posibil să nu fie fezabil să testați fiecare acțiune posibilă.

Legea lui Amdahl se aplică numai dacă CPU reprezintă blocajul. Dacă ceea ce faci nu este limitat de procesor, vei descoperi că după un anumit număr de nuclee nu mai vezi vreun câștig de performanță. Dacă performanța plăcii video, a memoriei RAM sau a hard diskului/SSD împiedică programul să ruleze mai rapid, adăugarea mai multor nuclee CPU nu va ajuta niciodată, chiar dacă programul este 100% paralel. De asemenea, rețineți că, dacă ajungeți să achiziționați un procesor mai rapid decât cel cu care ați testat, este absolut posibil ca noul procesor să fie suficient de rapid încât să se poată face altceva în sistem (RAM, HD, GPU etc.) apoi deveniți blocajul și limitați performanța noului procesor mai rapid.

Multe programe sunt codificate pentru a utiliza un anumit număr de nuclee. Chiar dacă este posibil ca un program să încearcă să folosească mai multe nuclee, multe programe au un anumit număr de nuclee CPU care pot fi utilizate. De fapt, marea majoritate a software-ului disponibil astăzi utilizează încă doar un singur nucleu CPU! Acest lucru se face dintr-o varietate de motive, de la natura a ceea ce face programul, ceea ce face ca acesta să nu fie propice utilizării mai multor nuclee CPU până la pur și simplu mai ușor de programat pentru un număr fix de nuclee.

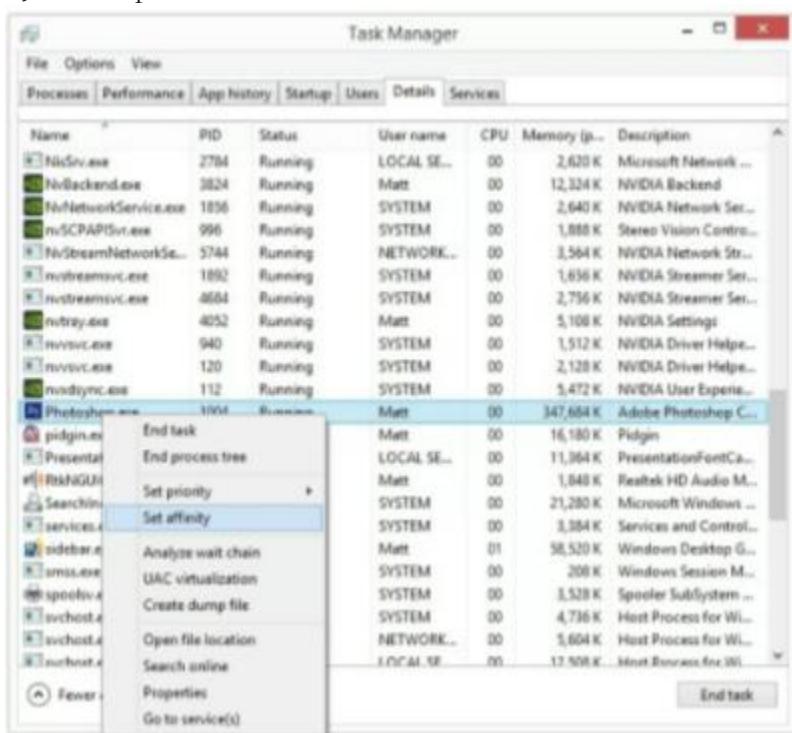
Estimarea performanței unui procesor va fi precisa numai pentru procesoarele bazate pe o arhitectură similară. Dacă procesorul pe care l-ați folosit pentru a determina eficiența paralelă a unui program este foarte diferit de procesorul pe care intenționați să îl cumpărați, este posibil să nu puteți estima cu exactitate performanța unui procesor. Chiar și în afara procesoarelor AMD vs Intel, dacă procesorul pe care l-ați testat are mai mult de o generație sau două, performanța reală a unui procesor mai nou poate fi foarte diferită (și de obicei mai rapidă) decât ceea ce va estima Legea lui Amdahl. Puteți calcula cu exactitate eficiența paralelă și o puteți utiliza pentru a compara performanța relativă a două sau mai multe procesoare care utilizează aceeași arhitectură, dar nu veți putea determina mai mult decât o idee generală a performanței reale.

Pasul 1: Testează-ți programul cu un număr diferit de nuclee CPU

Determinarea eficienței de paralelizare a unui program nu este ceva pe care îl puteți găsi doar căutând într-un fișier ReadMe.txt. Trebuie pur și simplu să rulați programul și să stabiliți timpul

necesar pentru a finaliza o sarcină cu numărul de nuclee CPU pe care le poate folosi limitat în mod artificial. Din fericire, nu trebuie să-ți schimbi procesorul de multe ori pentru a face acest lucru. În schimb, putești seta pur și simplu afinitatea programului prin Task Manager din Windows. Acest lucru nu este la fel de bun ca dezactivarea completă a nucleelor CPU prin BIOS - ceea ce este posibil pe unele plăci de bază - dar pare ca este mult mai precis decât v-ați aștepta.

Pentru a seta afinitatea, pur și simplu lansați programul pe care dorîți să îl testați, deschideți Task Manager, faceți clic dreapta pe lista programului sub Detalii, selectați „Set Affinity” și alegeti firele pe care dorîți să le permită programului să le folosească. Rețineți că, dacă procesorul dvs. acceptă Hyperthreading, vor exista de fapt de două ori mai multe fire de execuție enumerate decât are CPU-ul dvs. de fapt nuclee. Puteți fie să dezactivați Hyperthreading în BIOS înainte de a efectua testarea, fie pur și simplu să selectați două fire pentru fiecare nucleu al procesorului pe care dorîți să îl testați. Firele de execuție hyperthreading sunt întotdeauna listate imediat după nucleul fizic în Windows, așa că ar trebui să selectați două fire pentru fiecare nucleu CPU pe care dorîți să-l folosească programul. Cu alte cuvinte, selectarea firelor de execuție 1 și 2 va permite programului să folosească doar un singur nucleu CPU, selectarea firelor 1-4 va permite programului să folosească două nuclee CPU și așa mai departe.





....

Rețineți că setarea afinității durează doar până când programul este închis. Dacă viitoare când rulați programul, va trebui să resetați din nou afinitatea. Cu toate acestea, dacă dorîți să testați rapid o singură acțiune folosind diferite numere de nuclee CPU, nu trebuie să închideți programul înainte de a schimba afinitatea - doar faceți clic pe „Set Affinity” și schimbați-l din mers. Cu toate acestea, veți obține rezultate mai precise prin închiderea programului între rulări, deoarece aceasta va curăța RAM care este deja alocată programului.

Cu capacitatea de a seta câte nuclee CPU poate folosi un program, tot ce trebuie să faceți este să efectuați o acțiune repetabilă folosind o varietate de nuclee CPU. De exemplu, puteți cronometra cât de mult durează pentru a finaliza o randare în AutoCAD sau exportați imagini în Lightroom folosind o varietate de nuclee CPU. Cu cât este mai mare varietatea numărului de nuclee pe care le testați, cu atât mai bine, dar trebuie să testați cel puțin cu un singur nucleu CPU și toate nucleele CPU posibile. Dacă este posibil, vă recomandăm să testați cu cât mai multe combinații posibil (deci dacă aveți un procesor cu opt nuclee, testați cu 1,2,3,4,5,6,7 și 8 nuclee).

Pasul 2: Determinarea fracției de paralelizare

În acest moment, ar trebui să aveți o listă care să indice cât de mult a durat programul dvs. pentru a finaliza o acțiune folosind diferite numere de nuclee CPU. Doar pentru a avea un exemplu, să presupunem că rezultatele arată ca cele din coloana „Timp de acțiune (secunde)” din graficul de mai jos:

# de nuclee	Timp de acțiune (secunde)	Accelerare reală	Amdahl's Law Speedup (97% eficient)
-------------	---------------------------	------------------	-------------------------------------

1	645,4	1	1
2	328,3	1,97	1,95
3	230	2.8	2.8
4	172	3,75	3,67
.....			

Cea mai ușoară modalitate de a folosi aceste rezultate pentru a determina eficiența paralelizării unui program este să stabilim mai întâi cât de repede programul a finalizat sarcina cu N nuclee față de cât timp a durat cu un singur nucleu. Pentru a afla acest lucru, trebuie pur și simplu să împărțiți cât a durat acțiunea cu un singur nucleu la cât a durat cu N nucleu. În exemplul nostru, pentru două nuclee accelerarea este $645,4/328,3$, ceea ce este egal cu 1,97. Completăți acest lucru pentru fiecare rând și putem folosi aceste numere pentru a determina fracția de paralelizare a programului.

Există o modalitate matematică complexă de a folosi numerele reale de accelerare pentru a găsi direct fracția de paralelizare folosind potrivirea neliniară a curbei celor mai mici pătrate, dar cel mai simplu mod pe care l-am găsit este să ghicim pur și simplu fracția, să vedem cât de aproape sunt rezultatele, apoi ajustați-l până când viteza reală este aproape de accelerarea calculată folosind legea lui Amdahl. Folosirea unui program precum Excel face acest lucru mult mai ușor, dar o puteți face doar cu un calculator și un bloc de hârtie dacă doriti să o faceți manual și aveți ore de făcut.

Pentru a găsi fracția de paralelizare, trebuie să utilizați ecuația de paralelizare pe care am enumerat-o mai devreme și să introduceți diferite valori pentru P:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

Un loc bun pentru a începe ar putea fi să încercați $P=.8$ (sau 80% eficient în paralel) și să efectuați acest calcul pentru fiecare # de nuclee. De exemplu, pentru 4 nuclee ecuația ar fi:

$$S(n) = \frac{1}{(1 - .8) + \frac{.8}{4}}$$

Un loc bun pentru a începe ar putea fi să încercați $P=.8$ (sau 80% eficient în paralel) și să efectuați acest calcul pentru fiecare # de nuclee. De exemplu, pentru 4 nuclee ecuația ar fi:

Aceasta este egală cu 2,5. Comparați acest lucru cu accelerarea noastră reală din exemplul nostru (care a fost 3,75) și veți vedea că programul nostru exemplu este de fapt mai mult de 80% eficient, așa că trebuie să creștem fracția de paralelizare la ceva mai mare. În cazul nostru, fracția reală a fost .97 (97%), ceea ce este destul de decent. Veți observa că rezultatele nu se aliniază perfect de fiecare dată, deoarece există o anumită marjă de eroare care există întotdeauna atunci când rulați benchmark-uri - trebuie pur și simplu să faceți o medie și să o apropiați cât mai aproape. Având acest lucru într-o foaie de calcul în care puteți reprezenta grafic ambele serii de date, este mult mai ușor.

Pasul 3: Estimați performanța procesorului folosind raportul de paralelizare

Odată ce aveți raportul de paralelizare, îl puteți utiliza pentru a estima performanța oricărui alt procesor care utilizează aceeași arhitectură sau o arhitectură similară. Dacă sunteți interesat de un procesor care utilizează o arhitectură complet diferită, puteți utiliza totuși această metodă pentru a determina diferența relativă de performanță între un număr de modele diferite de procesor din aceeași familie, dar probabil că nu va fi o reprezentare exactă a performanței reale.

Pentru a estima performanța unui procesor, trebuie să cunoașteți frecvența de operare și câte nuclee are atât procesorul cu care ați folosit pentru comparație, cât și procesorul care vă interesează. Cu aceste specificații în mână, mai întâi trebuie să calculați câte nuclee efective au ambele procesoare, ceea ce se face folosind ecuația:

$$EffectiveCores = \frac{1}{(1 - P) + \frac{P}{CPUCores}}$$

Practic, aceasta folosește aceeași ecuație de paralelizare pe care am folosit-o mai devreme doar folosind numărul real de nuclee pe care le are CPU. Acest lucru ne oferă numărul efectiv de nuclee CPU pe care le are CPU atunci când rulează programul, dacă programul a fost efectiv 100% eficient. Din aceasta, putem înmulții numărul de nuclee efective cu frecvența de operare a fiecărui procesor pentru a obține în esență câte operații pe secundă este capabil să completeze procesorul (sau GFLOPs):

$$GFLOPs = CPUFrequency * EffectiveCores$$

Cu aceasta, ar trebui să ajungeți cu o estimare a cât timp i-ar lua unui procesor să finalizeze acțiunea pe care ați evaluat-o.

Utilizarea unei tabele de calcul

Dacă dorîți să estimați performanța unui procesor folosind legea lui Amdahl va trebui să utilizati un tabel de calcul (ex. Estimare_CPU_Performante-Amdahl.xlsx).

Pentru a utiliza tabelul, procedați în felul următor:

- Finalizați Pasul 1: testați programul cu un număr diferit de nuclee CPU .
- După ce ați testat aplicația dvs. cu diferite numere de nuclee CPU active, introduceți rezultatele în tabel
- Ajustați raportul de eficiență paralelă (celula galbenă) până când cele două linii de pe grafic sunt similare. Dacă nu puteți alinia cele două linii, este posibil ca programul dvs. să nu fie limitat de procesor (consultați secțiunea Limitări ale legii Amdahl)

- Schimbați celulele albastru deschis pentru a reflecta nucleele și frecvența procesorului pe care l-ați folosit pentru testare (rândul 28) și procesorului(e) pentru care sunteți interesat(e) să estimați performanța (rândul 29-30)
- Ar trebui să vedeți o estimare a cât timp ar trebui să ia fiecare procesor pentru a efectua acțiunea pe care ați evaluat-o în celulele verzi

Acest lucru este mult mai ușor decât încercarea de a urmări toate ecuațiile diferite, deși înțelegem că există unii oameni cărora le place în mod ciudat să facă matematică.



Concluzie

Indiferent dacă ați urmat instrucțiunile pas cu pas sau pur și simplu ați folosit documentul Google pe care l-am conectat, acum ar trebui să aveți resursele și informațiile necesare pentru a estima performanța unui procesor pentru programul și aplicația dvs. exacte. Deși acesta nu este cel mai ușor proces din lume, poate fi de neștiut atunci când încercați să decideți ce procesor să utilizați în noul dumneavoastră computer.

Să presupunem că doriti să cumpărați un sistem nou, dar trebuie să alegeti între două modele de procesoare care au același pret, dar foarte diferite în ceea ce privește frecvența și numărul de nuclee. De exemplu, să folosim un Xeon E5-2667 V3 și un Xeon E5-2690 V3. Folosind datele din exemplul din Pasul 2 și presupunând că procesorul nostru de testare a fost un Xeon E5-2660 V3 2,6GHz Ten Core, putem estima performanța acestor două procesoare ea fiind:

Model CPU	~ MSRP	Timp de acțiune estimate
Intel Xeon E5-2660 V3 la 2,6 GHz cu zece nuclee (procesor de testare)	1450 USD	85,3 secunde
Intel Xeon E5-2667 V3 3,2 GHz cu opt nuclee	2057 USD	82,5 secunde
Intel Xeon E5-2690 V3 la 2,6 GHz cu două nuclee	2090 USD	74,4 secunde

În acest exemplu, rularea pe un E5-2667 V3 ar trebui să dureze aproximativ 82,5 secunde pentru a finaliza acțiunea pe care am evaluat-o, în timp ce pe un E5-2690 V3 ar trebui să dureze doar aproximativ 74,4 secunde. Deoarece cele două procesoare sunt la doar 33 USD una de cealaltă, acest lucru ne face o idee că E5-2690 V3 este cea mai bună alegere în acest caz.

Rețineți că acest lucru se aplică numai procesoarelor care au o arhitectură similară cu cea pe care ați folosit-o pentru testare și numai pentru acțiunea pe care ați evaluat-o. Orice lucru diferit (chiar și în cadrul aceluiași program) poate avea rezultate drastic diferite. Cu toate acestea, dacă tot așteptați să finalizați o randare, un export de finalizat sau orice altă sarcină specifică, vă puteți limita testarea doar la acele sarcini. Fiecare poate avea o eficiență de paralelizare diferită, dar dacă determinați

eficiență pentru fiecare sarcină și le acordați o anumită pondere (probabil în funcție de cât de des așteptați ca fiecare să termine), puteți lua o decizie mult mai corectă cu privire la CPU care este potrivit.

Facultatea de Informatică - UTM