

Cuprins

INTRODUCERE	5
UNITATEA DE ÎNVĂȚARE 1	11
1. Stocarea datelor	11
1.1. Stocarea biților	11
1.1.1. Porți logice și circuite basculante bistabile	11
1.1.2. Tehnici de stocare	12
1.1.3. Sistemul de notație hexazecimal	13
1.2. Memoria principală	13
1.2.1. Organizarea memoriei principale	14
1.2.2. Organizarea unei celule de memorie	15
1.3. Dispozitive de stocare de masă	19
1.3.1. Discuri magnetice	21
1.3.2. Discuri compacte	25
1.3.3. Benzi magnetice	26
1.3.4. Înregistrări logice și fizice	26
1.4. Codificarea utilizată pentru stocarea informațiilor	27
1.4.1. Reprezentarea simbolurilor	27
1.4.2. Reprezentarea valorilor numerice	28
1.4.3. Reprezentarea altor tipuri de date	30
1.5. Sistemul binar de numerație	31
1.5.1. Adunarea în binar	31
1.5.2. Reprezentarea fracțiilor în sistemul binar	32
1.6. Stocarea numerelor întregi	34
1.6.1. Notația în exces	34
1.6.2. Notația în complement față de doi	35
1.6.3. Adunarea numerelor reprezentate în complement față de doi	38
1.6.4. Problema depășirii superioare	39
1.7. Stocarea numerelor fracționare	40
1.7.1. Notația în virgulă mobilă	40
1.7.2. Erori de rotunjire	42
1.8. Erori de comunicație	43
1.8.1. Biți de paritate	43
1.8.2. Coduri corectoare de erori	45
TEST AUTOEVALUARE 1 (Stocarea datelor)	47
UNITATEA DE ÎNVĂȚARE 2	52
2. Manipularea datelor	52
2.1. Unitatea centrală de prelucrare	52
2.2. Execuția instrucțiunilor	55
2.3. Execuția programelor	60

2.4.	Alte instrucțiuni	61
2.5.	Principii de proiectare pentru calculatoarele actuale	63
2.6.	Prelucrare simultană	64
2.7.	Instrucțiuni aritmetice și logice	65
2.8.	Comunicația între unitatea centrală și controlere	67
2.9.	Comunicația serială și paralelă	69
TEST AUTOEVALUARE 2 (Manipularea datelor)		73
UNITATEA DE ÎNVĂȚARE 3.....		75
3.	Sistemele de operare.....	75
3.1.	Evoluția sistemelor de operare	75
3.2.	Arhitectura unui sistem de operare	77
3.3.	Coordonarea activităților desfășurate de calculator.....	81
3.4.	Gestionarea proceselor concurente	85
TEST AUTOEVALUARE 3 (Sisteme de operare).....		88
UNITATEA DE ÎNVĂȚARE 4.....		92
4.	Algoritmii	92
4.1.	Conceptul de algoritm	92
4.2.	Reprezentarea algoritmilor	93
4.3.	Dezvoltarea algoritmilor.....	96
4.4.	Structuri iterative	98
4.5.	Structuri recursive.....	102
4.6.	Eficiență și corectitudine	103
TEST AUTOEVALUARE 4 (Algoritmii)		107
UNITATEA DE ÎNVĂȚARE 5.....		111
5.	Limbaje de programare	111
5.1.	Perspective istorice	111
5.2.	Conceptele programării clasice	114
5.3.	Module de program	121
5.4.	Implementarea limbajelor.....	123
5.5.	Programarea declarativă	128
TEST AUTOEVALUARE 5 (Limbaje de programare).....		132
UNITATEA DE ÎNVĂȚARE 6.....		139
6.	Structuri de date	139
6.1.	Vectori	139
6.1.1.	Vectori unidimensionali	139
6.1.2.	Vectori multidimensionali	140

6.2. Liste	141
6.3. Arbori.....	150
TEST AUTOEVALUARE 6 (Structuri de date)	154
UNITATATEA DE ÎNVĂȚARE 7.....	157
7. Structuri de fișiere	157
7.1. Fișierele secvențiale	157
7.2. Fișiere de text.....	159
7.3. Fișiere indexate	160
7.4. Fișiere dispersate (hashed files).....	161
7.5. Rolul sistemului de operare	162
TEST AUTOEVALUARE 7 (Structuri de fișiere).....	164
UNITATEA DE INVATARE 8	166
8. Structuri de Baze de date	166
8.1. Considerații generale	166
8.2. Implementarea stratificată a bazelor de date.....	167
8.3. Modelul relațional.....	169
TEST AUTOEVALUARE 8 (Structura bazelor de date).....	181
UNITATEA DE ÎNVĂȚARE 9	183
9. Sistem informatic și sistem informațional	183
9.1 Conceptul de informație.....	183
9.2 Noțiunea de sistem.....	185
9.3 Sistem informatic	190
TEST AUTOEVALUARE 9 (Sistem Informațional și Sistem Informatic de fișiere).....	196
BIBLIOGRAFIE	198

INTRODUCERE

Informatica este o disciplină care construiește baza științifică pentru mai multe domenii, ca:

- proiectarea și programarea calculatoarelor;
- prelucrarea informațiilor;
- rezolvarea algoritmică a problemelor;
- procesul algoritmic propriu-zis.

Prin urmare, nu putem reduce studiul informaticii la deprinderea modului de utilizare a calculatoarelor actuale (eventual a PC-urilor), ci trebuie să înțelegem atât aria de cuprindere, cât și dinamica unui mare număr de domenii înrudite.

a) Studiul algoritmilor

***Algoritm** = set de pași prin care se definește modul în care poate fi dusă la îndeplinire o anumită sarcină.*

În domeniul calculatoarelor **algoritmii** sunt reprezentați prin *programe*. Aceste programe formează software-ul, spre deosebire de calculatoarelor propriu-zise care poartă numele de hardware.

Pentru ca un calculator să poată duce la îndeplinire o anumită sarcină, trebuie (ca mai întâi) să se descopere și să se reprezinte sub formă de program un algoritm pentru efectuarea sarcinii respective.

Algoritmii ocupă un rol central în informatică. Principalul obiectiv al acestor eforturi era descoperirea unui set mic de instrucțiuni care să descrie modul de rezolvare a oricărei probleme dintr-o anumită categorie. Unul dintre cele mai cunoscute rezultate obținute în acest domeniu este algoritmul lui Euclid (descoperit în antichitate de matematicianul precizat).

Determinarea celui mai mare divizor comun a 2 numere întregi pozitive.

Descriere: *Algoritmul primește ca intrare 2 numere întregi pozitive și calculează cel mai mare divizor comun al celor 2 numere.*

Procedura:

Pasul 1 : Se notează cu M cea mai mare valoare și cu N cea mai mică valoare de intrare.

Pasul 2 : Se împarte M la N și se notează cu R restul împărțirii.

Pasul 3 : Dacă $R \neq 0$, se atribuie lui M valoarea N și lui N valoarea R , apoi se revine la pasul 2: cel mai mare divizor comun al celor 2 numere este valoarea notată cu N .

După descoperirea algoritmului (în cazul nostru al lui Euclid), într-un fel, putem spune că toate informațiile necesare pentru efectuarea [activității] operației respective sunt incluse într-o formă codificată în algoritm.

Dezvoltarea algoritmilor este o preocupare majoră în domeniul efectuării calculelor și principalele probleme legate de această acțiune sunt :

- descoperirea unui algoritm care să rezolve o problemă (găsirea soluției pentru o problemă). Pentru anumite domenii (ex. contabilitate), algoritmii se găsesc exprimați în legi și metodologii (norme) specifice.
- reprezentarea algoritmului într-o formă în care poate fi comunicată unei mașini sau altor oameni. Adică scrierea unui algoritm trebuie transcris din forma conceptuală într-un set clar de instrucțiuni lipsit de ambiguitate. Există o mulțime de variante de scheme de reprezentare a algoritmilor numite *limbaje de programare*.

În prezent, în acest domeniu s-au realizat sisteme automate de organizare și planificare care permit și nespecialiștilor să dezvolte, fără ajutor din afară, sistemele de care au nevoie. Exprimarea curentă a acestei activități este ingineria *software*.

Scopul acestor prelegeri nu este prezentarea în detaliu a modului în care arhitectura calculatoarelor este implementată prin intermediul circuitelor electronice (este vorba de un alt domeniu electronic). Ar fi de dorit ca arhitectura calculatoarelor să reflecte numai cunoștințele legate de procesele algoritmice și să nu fie limitată de tehnică.

Arhitectura calculatoarelor poate fi studiată și în alt context decât acela al stocării și regăsirii datelor. În acest domeniu, caracteristicile interne ale calculatorului se reflectă adesea în caracteristicile externe. Proiectarea sistemelor de calcul este strâns legată de interfața lor cu lumea exterioară. De exemplu, se pune problema modului în care algoritmul poate fi furnizat calculatorului și a modului de precizare în care algoritm să fie executat.

În contextul în care calculatorul îndeplinește diferite sarcini, trebuie rezolvate multe probleme legate de coordonarea activităților și alocarea resurselor .

b) Dezvoltarea mașinilor algoritmice

Unul din primele dispozitive de calcul utilizate de oameni a fost *abacul*. Prezența lui a fost atestată în civilizațiile antice (greacă și romană), el fiind folosit și astăzi. Bilele poziționate pe sârme în cadrul abacului semnifică anumite valori.

Abacul reprezintă și stochează valori prin poziția bilelor așezate de operatorul uman (intrările) care, de asemenea, observă pozițiile finale ocupate de bile, ceea ce reprezintă rezultatul (ieșirile). Deci abacul este o mașină de stocare a datelor și devine o mașină algoritmică numai împreună cu omul care-l utilizează.

Proiectarea mașinilor de calcul s-a bazat într-o vreme pe tehnologia roților dințate. Printre inventatorii acestui tip de mașini de calcul s-a aflat și Blaise Pascal (1623—1662), Gottfried Leibniz (1646—1716) și Charles Babbage (1792—1871). Mașinile lor reprezentau datele prin poziționarea unor roți dințate, datele de intrare erau furnizate mecanic prin stabilirea poziției particulare a acestor roți.

În cazul mașinilor lui Pascal și Leibniz, valorile de ieșire erau obținute prin observarea poziției finale a roților (similar citirii numerelor pe indicatorul de kilometraj al automobilului). La mașina lui Babbage rezultatele se tipăreau pe hârtie, evitând erorile de transcriere.

În ceea ce privește abilitatea de a urma algoritm putem spune că:

- mașina lui Pascal efectua numai adunări;
- mașina lui Leibniz avea algoritmi (operațiile aritmetice) înglobați, dar operatorul trebuia să-l selecteze pe cel dorit;
- mașina lui Babbage era astfel proiectată încât secvența de pași trebuia executată să poată fi comunicată acesteia prin perforarea într-un anumit mod a unor cartele de hârtie.

Mai târziu, Herman Hollerith (1860—1929) a folosit ideea reprezentării informațiilor sub forma unor perforații în cartele de carton, pentru îmbunătățirea procesului de înregistrare în cadrul recensământului efectuat în 1890 în Statele Unite. Lucrările sale au stat la baza creării companiei I B M.

Anul de naștere al informaticii tehnice este 1946 când a fost creat computerul electronic de integrare numerică aritmetică construit cu tuburi electronice și cântărind sute de tone. John von Neumann a construit în 1951 calculatorul EDVAC (Electronic Discrete Variable Automatic Computer), iar în 1949 la Universitatea Cambridge (SUA) a fost realizat EDSAC

(Electronic Delay Storage Automatic Computer), primul calculator care dispunea de sistem de operare.

c) Evoluția informaticii

Capacitatea-limită de stocare a datelor, ca și procedurile de programare consumatoare de timp au reprezentat restricții care au redus complexitatea algoritmilor executați de primele generații de calculatoare.

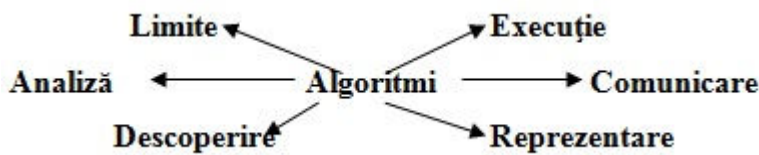
Pe măsură ce aceste limite au început să fie depășite, sistemele de calcul au început să fie folosite pentru sarcini de mai mare anvergură, din ce în ce mai complexe. Eforturile s-au îndreptat din ce în ce mai mult către studiul algoritmilor și al procesului de programare. Aceste eforturi se pot aduna, fără a exagera, în știința algoritmilor. Aria de cuprindere a acestei științe este foarte largă, incluzând subiecte de matematică, inginerie, psihologie, management, lingvistică etc.

Ceea ce ne propunem să prezentăm în cadrul acestor prelegeri (lecții) sunt :

- ideea centrală a fiecărui domeniu;
- direcțiile curente de cercetare;
- tehnicile utilizate în domeniul respectiv.

De exemplu, în domeniul programării ne vom concentra asupra principiilor de bază ale instrumentelor de programare și asupra evoluției lor, și nu asupra dezvoltării abilităților de programare. Pentru a nu pierde din vedere imaginea de ansamblu, vom încerca să identificăm câteva întrebări fundamentale pentru informatică :

1. Ce probleme se pot rezolva prin procesele algoritmice?
2. Cum se pot descoperi algoritmii?
3. Cum se pot îmbunătăți tehnicile de reprezentare și comunicare a algoritmilor?
4. Cum pot fi aplicate cunoștințele dobândite cu privire la algoritmi (tehnologia în vederea obținerii unor mașini algoritmice îmbunătățite) ?
5. Cum pot fi analizate și comparate caracteristicile diversilor algoritmi?



Rolul central detinut de algoritm in informatica

d) Rolul abstractizării

Pentru sistemele de o complexitate care depășește capacitatea noastră de înțelegere se poate privi sistemul respectiv ca un ansamblu de componente ale căror caracteristici interne pot fi ignorate, ceea ce ne permite să ne concentrăm asupra modului în care componentele respective interacționează unele cu altele și asupra modului cum sunt utilizate pentru construirea unor componente de nivel mai înalt.

Această distincție care se face între proprietățile externe ale unei componente și detaliile interne care țin de construcția componentei respective poartă numele de ***abstractizare***. În orice progres obținut, o mică parte a societății noastre se specializează în implementarea lui, ceilalți folosesc rezultatul ca pe un instrument abstract, deci o unealtă a cărei implementare internă nu suntem obligați să o înțelegem.

e) Implicații etice, sociale și juridice

Informatica estompează multe distincții care serveau în trecut ca reguli pe care oamenii își întemeiau deciziile și pune sub semnul întrebării o parte din principiile care stau la baza societății, ca de exemplu :

- Care este diferența între un comportament inteligent și inteligența propriu-zisă?
- În ce condiții putem vorbi despre existența vieții?
- Care este diferența între plante și animale?

Informatica generează asemenea întrebări, punând în discuție principiile și bazele pe care este construită cunoașterea noastră.

În context juridic, se pune întrebarea în ce măsură cineva poate deține în proprietate un produs *software* și care sunt drepturile și obligațiile care decurg din această proprietate. În domeniul eticii, suntem confrunțați cu numeroase opțiuni care pun la îndoială principiile uzuale de comportament. În context politic, apare întrebarea dacă și în ce măsură tehnologia calculatoarelor și aplicațiile acestora trebuie controlate de stat.

Rezolvarea acestor probleme necesită existența unor cunoștințe organizate sub forma metodelor, tehnicilor, normelor pentru fiecare știință sau tehnologie pusă în discuție.

Pentru a putea să luăm decizii logice cu privire la depozitarea deșeurilor radioactive, trebuie să înțelegem care sunt efectele radiațiilor, care sunt măsurile de protecție care trebuie luate și să putem evalua durata de risc.

Pentru a acorda sau nu guvernelor sau companiilor dreptul de a construi baze de date referitoare la cetățeni sau clienți, membrii societăți trebuie să înțeleagă care sunt posibilitățile, limitele și implicațiile tehnologiei bazelor de date. Ceea ce va fi prezentat în cursul pe care-l propunem vă furnizează aceste cunoștințe elementare în domeniul a ceea ce numim **i n f o r m a t i c ă**.

Cu toate acestea, știința nu reușește întotdeauna să răspundă clar și hotărât tuturor problemelor sociale care-i sunt ridicate. Nu întotdeauna există un singur răspuns corect; de multe ori așa-zisele soluții sunt în fapt compromisuri între mai multe puncte de vedere.

Este nevoie de capacitatea de a asculta și de a înțelege și alte puncte de vedere, de a purta discuții raționale și de a-ți schimba opiniile în funcție de datele obținute în urma dialogului. Pentru a încuraja un astfel de comportament, fiecare capitol al cursului se va finaliza cu un set de *probleme de etică*, pe care le considerăm, mai ales în contextul profesiei pe care ați ales-o, la fel de importante ca și aspectele tehnice propuse spre dezbateri.

UNITATEA DE ÎNVĂȚARE 1

1. Stocarea datelor

1.1. Stocarea biților

Calculatoarele utilizate în prezent reprezintă informațiile sub forma șiruri de biți. Un **bit** (*binary digit* = cifră binară) simbolizează una din cifrele 0 sau 1.

Stocarea unui bit într-un calculator necesită un dispozitiv care să poată să se afle într-una din cele două stări. De exemplu:

- un întrerupător (**pornit/oprit**);
- un releu (**deschis/închis**);
- un steag de start (**ridicat/coborât**).

Una din stări este utilizată pentru reprezentarea simbolului *0*, iar cealaltă pentru reprezentarea simbolului *1*.

1.1.1. Porți logice și circuite basculante bistabile

Trebuie introduse operațiile **AND** (= și), **OR** (= sau) și **XOR** (= sau exclusiv) care sunt prezentate mai jos :

0	0	1	1
<u>AND 0</u>	<u>AND 1</u>	<u>AND 0</u>	<u>AND 1</u>
0	0	0	0

0	0	1	1
<u>OR 0</u>	<u>OR 1</u>	<u>OR 0</u>	<u>OR 1</u>
0	1	1	1

0	0	1	1
<u>XOR 0</u>	<u>XOR 1</u>	<u>XOR 0</u>	<u>XOR 1</u>
0	1	1	0

Figura 1.1 - Operațiile AND, OR și XOR

Aceste operații lucrează cu valorile **adevăr (true)/fals (false)** și sunt denumite *operații booleene*.

Intrările operației AND sunt reprezentate de adevărul sau falsitatea componentelor expresiei; ieșirea reprezintă adevărul sau falsitatea expresiei compuse.

Expresia $P \text{ AND } Q$ este adevărată numai când ambele sale componente sunt adevărate. Toate variantele operației AND se pot vedea în prima linie a figurii 1.1. Similar, operația OR se bazează pe o expresie compusă sub forma $P \text{ OR } Q$. Asemenea expresie compusă este adevărată când cel puțin una dintre componente este adevărată, ceea ce coincide cu reprezentarea OR din linia a doua a figurii 1.1.

În limbajul curent nu există o conjuncție care să poată exprima semnificația operației XOR. XOR produce ca rezultat o ieșire cu valoarea 1 atunci când una dintre intrările sale este 1 și cealaltă 0 (vezi linia a treia din fig. 1.1.). Operația NOT este o altă operație booleană de AND, OR și XOR, prin faptul că are o singură intrare și o singură ieșire. Ieșirea reprezintă opusul intrării; dacă intrarea operației NOT este adevărată, ieșirea este falsă, și invers.

1.1.2. Tehnici de stocare

Calculatoarele anilor '60 conțineau inele de material magnetic de dimensiuni mici, denumite **miezuri (cores)**, pe care erau înfășurate fire electrice, [mini]bobine. La trecerea curentului electric prin bobine, miezul poate fi magnetizat într-una din cele două direcții. Direcția câmpului magnetic poate fi detectată observându-se efectul asupra unui curent electric ce trece prin centrul miezului.

Astfel un miez reprezintă o posibilitate de memorare a unui bit. Datorită dimensiunilor lor mari și a necesarului ridicat de putere (electrică), aceste sisteme sunt depășite astăzi. Calculatoarele înregistrează datele încă utilizând tehnologia magnetică, dar într-un mod foarte apropiat de acela în care se fac înregistrările pe bandă magnetică. În unele sisteme de calcul se folosește, de asemenea, și tehnica laserului pentru înregistrarea bitilor.

Diferențele dintre circuitele basculante bistabile electronice și dispozitivele de stocare magnetice (sau laser) reprezintă argumente pro și contra în ceea ce privește aplicațiile lor.

Circuite basculante bistabile electronice care pot fi acționate electronic sunt mai rapide decât cele magnetice și, de aceea, sunt utilizate pentru stocarea datelor în circuitele interne ale calculatorului. Însă un circuit basculant bistabil electronic pierde informația stocată în el atunci când sursa de alimentare este oprită.

În schimb, dispozitivele de stocare magnetice sau cu laser păstrează datele, ceea ce le recomandă pentru realizarea de sisteme de stocare cu longevitate mare.

1.1.3. Sistemul de notație hexazecimal

Când ne referim la activitățile din interiorul unui calculator lucrăm cu șiruri de biți, care pot fi uneori foarte lungi. Pentru a simplifica reprezentarea șirurilor de biți, se va analiza o notație prescurtată, denumită **notația hexazecimală** (hexadecimal notation).

Notația hexazecimală utilizează un singur simbol pentru reprezentarea a patru biți.

Sistemul de codificare hexazecimal:

Cod binar	0000	0001	0010	0011	0100	0101	0110
Reprezentare hexazecimală	0	1	2	3	4	5	6

0111	1000	1001	1010	1011	1100	1101	1110	1111
7	8	9	A	B	C	D	E	F

Figura 1.2 - Sistemul de notație hexazecimal

1.2. Memoria principală

În scopul stocării datelor, un calculator conține un mare număr de circuite, fiecare dintre ele fiind capabile să stocheze biți. Acest "rezervor" de biți este cunoscut sub numele de **memorie principală** (*main memory*).

1.2.1. Organizarea memoriei principale

Celulele de stocare din memorie sunt organizate în unități, (cuvinte) cu dimensiunea uzuală de 8 biți. Acest șir de 8 biți a impus cuvântul *octet* și, de asemenea, a impus pentru cuvântul *byte* - șir de biți cu aceeași lungime.

Dimensiunea memoriei este adesea măsurată în multipli de 1.048.576 celule ($1.048.576 = 2^{20}$, fiind mai natural pentru a fi utilizat decât 1.000.000 care este o putere a lui 10). Pentru a desemna această unitate de măsură se folosește termenul *mega*.

Deci: 1 *mega byte* (1 *MB*) $\equiv$ 1.048.576 *byte*.

Alte unități de măsură a dimensiunilor de memorie sunt **kilooctetul** $\equiv$ *kilobyte* (*KB*) care este egal cu 1024 octeți (2^{10} octeți) și **gigaoctetul** $\equiv$ *gigabyte* (*GB*) care este egal cu 1024 *MB* (2^{30} octeți).

Pentru identificarea celulelor individuale din memoria principală a unui calculator, fiecare are atribuit un nume unic, denumit *adresă*. Adresele utilizate în tehnica de calcul sunt în întregime numerice. Pentru exemplificare putem considera toate celulele de memorie plasate pe un singur rând și numerotate în ordine crescătoare pornind de la 0.

Acest sistem permite identificarea unică a unei celule și, de asemenea, o relație de ordonare între celule permite referiri de tipul "celula precedentă" sau "celula următoare". Celulele din memoria principală care stochează biți sunt combinate cu circuite necesare pentru a permite, și altor circuite să stocheze și să recupereze datele din celulele memoriei principale.

Există astfel și alte celule (circuite) care pot prelua datele din memorie solicitând informații despre conținutul unei anumite adrese (operație de citire) sau pot înregistra informații în memorie solicitând ca un anumit șir de biți să fie plasat în celula aflată la o anumită adresă (operație de scriere).

Acest sistem de identificare a celulelor memoriei principale permite apelarea, cercetarea și modificarea individuală a acestora. Astfel, o celulă a memoriei principale cu o adresă mică este la fel de accesibilă ca și una cu o adresă foarte mare, deci datele stocate în memoria principală a calculatorului pot fi prelucrate în orice ordine. Din aceste motive memoria principală a unui calculator se mai numește și **memorie cu acces aleator** (*random access memory* — *RAM*).

Acest *acces aleator* la mici unități de date (celula) se deosebește radical de sistemele de stocare de masă (care manipulează șiruri lungi de biți sub forma de blocuri).

1.2.2. Organizarea unei celule de memorie

Considerăm biții dintr-o celulă de memorie ca fiind aranjați pe un rând. Vom denumi capetele acestui rând *marginea superioară*, iar celălalt capăt *marginea inferioară*.

Considera deasemenea biții aranjați într-un rând orientat de la stânga la dreapta cu marginea superioară plasată la stânga.

Bitul de la capătul din stânga este denumit *cel mai semnificativ bit*.

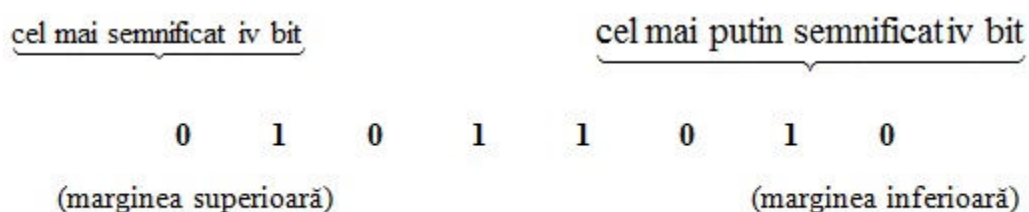


Figura 1.3 - Organizarea unei celule de memorie cu dimensiune de un octet

Memoria primara

Memoria (*Memory*) păstrează programele și datele. Fără o memorie, din care procesoarele să poată citi și în care să poată scrie informații, nu poate exista nici un calculator numeric cu program memorat.

Biți

Unitatea elementară a memoriei este cifra binară, numită **bit**. Un bit poate conține un 0 sau un 1. Sistemul de numerotare binar necesită numai două valori distincte, reprezentând cea mai sigură metodă pentru codificarea informației numerice.

Ordinea octeților

Octeții dintr-un cuvânt pot fi numerotați de la stânga la dreapta, sau de la dreapta la stânga. Vom reprezenta în figură o parte a memoriei unui calculator pe 32 biți.

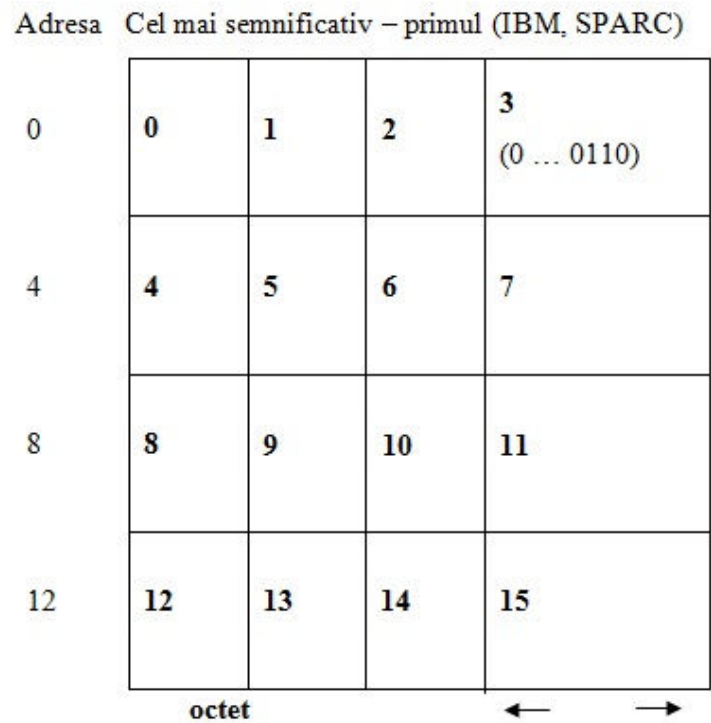


Figura 1.4 a – Biți numerotați de la stânga la dreapta

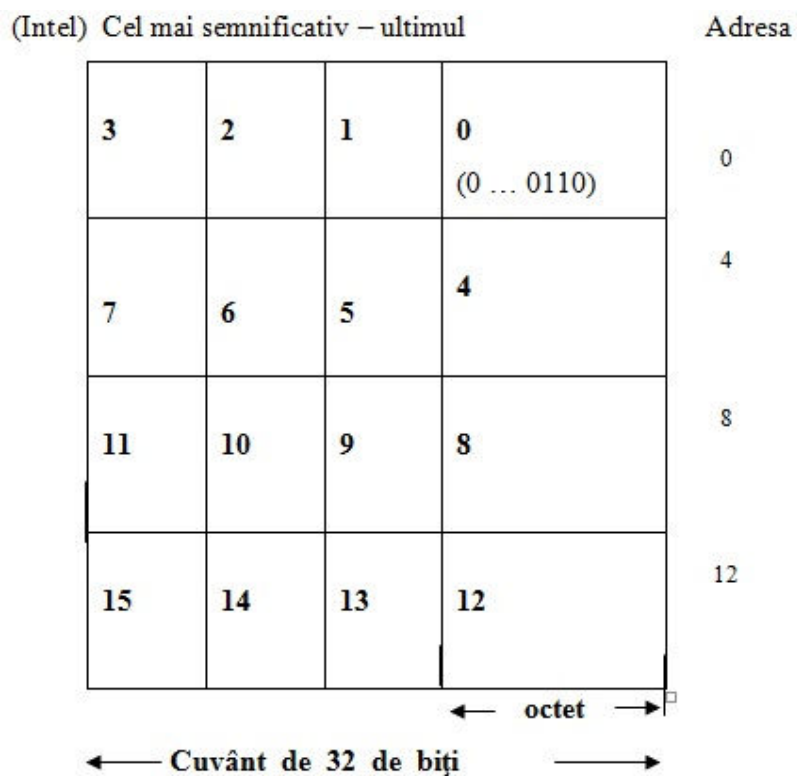


Figura 1.4 b – Biți numerotați de la dreapta la stânga

Să precizăm cum vom reprezenta valoarea numerică: 6 este reprezentată de biții 110 în partea cea mai din dreapta a unui cuvânt și de zerouri pe ceilalți 29 de biți din stânga.

Majoritatea aplicațiilor au nevoie de un amestec de numere întregi, șiruri de caractere și alte tipuri de date. Să considerăm o fișă simplă de personal: nume angajat, vârsta și codul departamentului (Paul Sandu, 21 ani, Departament = 260 = $1 \times 256 + 4$) și să o reprezentăm în cele două moduri precizate mai sus.

Ad.					Ad.				
0	P	A	U	L	L	U	A	P	0
4	S	A	N	D	D	N	A	S	4
8	U	0	0	0	0	0	0	U	8
12	0	0	0	21	0	0	0	21	12
16	0	0	1	4	0	0	1	4	16

Figura 1.5 – Reprezentarea unei fișe de personal

Reprezentările sunt bune și consistente intern, problema apare la transmiterea fișei, prin rețea de la mașina cea mai semnificativă către cealaltă mașină. Rezultatul transferului propus arată astfel:

L	U	A	P
D	N	A	S
0	0	0	U
21	0	0	0
4	1	0	0

Figura 1.6 – Rezultatul obținut

Mașina a inversat în timpul transmisiei:

- ordinea caracterelor din cuvânt (ceea ce este corect),
- octeții dintr-un întreg (ceea ce este incorect).

O soluție este de a inversa octeții prin software (după crearea unei copii), ceea ce va conduce la :

P	A	U	L
S	A	N	D
"U"	0	0	0
0	0	0	21
0	0	1	4

Figura 1.7 – Soluția de inversare prin software a octeților

Problema este poziția caracterului "U" poziționat prost. Soluții de îndreptare nu există la transferul între mașini diferite ca organizare a memoriei. Apare evident necesitatea unui standard pentru stabilirea ordinii octeților în cadrul organizării memoriei.

Memoria intermediară

Întotdeauna CPU-urile au fost mai rapide decât memoriile. Proiectanții CPU-ului sunt preocupați de sporirea vitezei acestora, pe când proiectanții memoriilor sunt preocupați de sporirea capacității, și nu de sporirea vitezei. Datorită acestor abordări, situația CPU-urilor și a memoriilor, vis-à-vis de viteză, continuă să se înrăutățească în timp. Practic, acest dezechilibru are următoarea metamorfoză: după ce CPU-ul a lansat o cerere către memorie, nu va primi cuvântul așteptat timp de mai multe cicluri CPU; cu cât memoria este mai lentă, cu atât CPU-ul are mai mult de așteptat.

În acest moment, inginerii știu cum să construiască memorii la fel de rapide ca și CPU-urile, dar ar trebui încorporate în cip-ul CPU, crescându-i dimensiunile și costul. Cip-ul CPU este limitat la anumite dimensiuni. În acest sens, se cunosc tehnici de combinare a unei cantități mici de memorie rapidă cu o cantitate mare de memorie lentă, la un preț moderat.

Memoria mică, rapidă, se numește **memorie intermediară** (*cache*). Ideea de bază a unei memorii intermediare : cuvintele de memorie cele mai frecvent utilizate sunt păstrate în memoria intermediară.

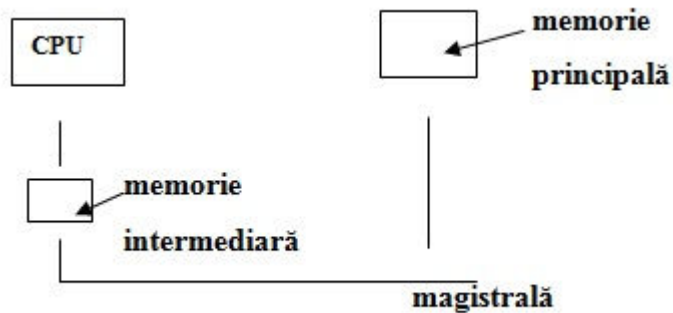


Figura 1.8 – Plasarea memoriei intermediare

1.3. Dispozitive de stocare de masă

Limitele tehnologice precum și necesitatea stocării unor copii de siguranță ale datelor vitale, au făcut ca memoria principală a unui calculator să nu satisfacă cerințele impuse de diversele aplicații. Din aceste motive, calculatoarele sunt echipate, pe lângă memoria principală, cu sisteme de stocare de masă (*mass storage system*), denumite și **memorie secundară**. Stocarea datelor pe aceste sisteme se face în unități de mari dimensiuni denumite **fișiere** (*files*).

Unul din principalele dezavantaje ale sistemelor de stocare de masă este acela că, în general, ele necesită o mișcare suplimentară mecanică, astfel fiind mai lente la stocarea și recuperarea datelor în comparație cu memoria principală a calculatorului.

Principalul avantaj al dispozitivelor de stocare de masă este acela că, în multe situații, sunt mai ieftine decât memoria principală, iar suportul pe care se înregistrează datele poate fi extras din calculator și depozitat într-un loc sigur în scopul recuperării ulterioare a datelor.

Cu privire la dispozitivele care pot fi cuplate sau decuplate de la calculator, se folosesc termenii *on-line* și *off-line*.

Termenul *On-line* indica că dispozitivul sau informațiile sunt conectate și pot fi folosite de calculator fără intervenție umană. Termenul *Off-line* indica că este necesară intervenția umană înainte ca dispozitivul sau informațiile să poată fi utilizate de calculator; dispozitivul trebuie pornit sau informațiile trebuie introduse într-un anumit mecanism.

Memoria secundară

Oricât de mare ar fi memoria principală, întotdeauna ea va fi prea mică pentru cerințele utilizatorilor.

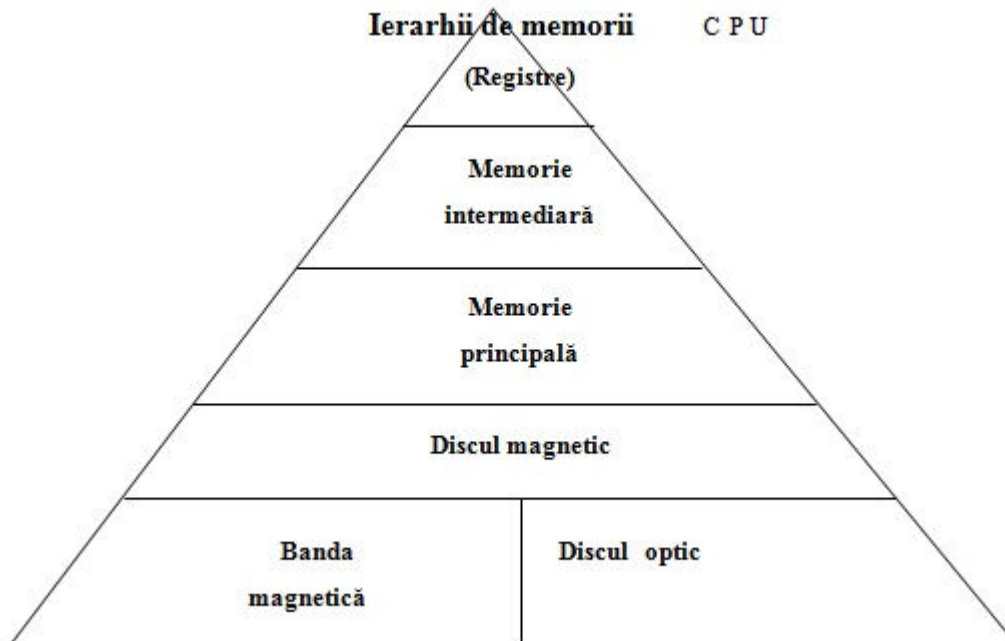


Figura 1.9 - Ierarhie de memorii cu cinci niveluri

Mărimea memoriei intermediare are dimensiuni de la 32 KB până la zeci de MB. Memoria principală are dimensiuni de la 16 MB până la zeci de GB.

Discul magnetic este suportul actual pentru păstrarea permanentă (80 GB, 100 GB, 120 GB, 200 GB) și stă la baza ierarhiei de memorii, iar banda magnetică și discul optic sunt destinate păstrării arhivelor. Există următorii parametri importanți de caracteristici specifice, pe măsură ce ne deplasăm spre baza ierarhiei:

1. Timpul de acces se mărește (crește).

- Registrele CPU pot fi accesate în **câteva monosecunde**;
- Memoriile intermediare pot fi accesate într-un **multiplu apropiat de timpul de acces al registrelor**;
- Timpii de acces la memoria principală au valori tipice de câteva **zeci de nanosecunde** ;
- Timpii de acces la discul magnetic ≥ 10 msec;

- Timpul de acces la banda magnetică și discul optic sunt de mărimea secundelor (inclusiv timpul de extragere și inserare în dispozitivul de intrare/ieșire).

2. Capacitatea de stocare crește.

- Registrele CPU pot stoca 128 octeți;
- Memoriile intermediare pot stoca câțiva megaocteți (MB);
- Memoriile principale: zeci ÷ mii MB;
- Discurile magnetice: zeci ÷ sute GB;
- Benzile magnetice și discurile optice nu sunt tot timpul utilizate, astfel încât capacitatea lor este limitată de bugetul proprietarului.

3. Numărul de biți primit pe dolar crește. Prețurile pe componente se măsoară astfel:

- Memoria principală, în dolari/MB;
- Stocarea pe discul magnetic în centime/pennies/MB;
- Stocarea pe banda magnetică, în dolari/GB.

1.3.1. Discuri magnetice

Un disc magnetic este alcătuit din unul sau mai multe platane de aluminiu, cu un înveliș magnetizabil. Inițial, diametrele acestor discuri erau de 50 cm, 12 cm, dar acum (2005) au în general 3 cm sau mai puțin. Un cap de citire/scriere, care conține o bobină de inducție ce se deplasează foarte aproape de suprafața platanului, „sprijinindu-se” pe o pernă de aer (cu excepția discurilor flexibile/dischetelor unde atinge suprafața). La trecerea unui curent negativ/pozitiv prin bobina capului, acesta magnetizează suprafața de dedesubtul capului, aliniind particulele magnetice la dreapta sau la stânga, funcție de polaritatea curentului. Astfel, se realizează scrierea, citirea realizându-se în modul precizat în paragraful următor. La trecerea capului de citire peste o suprafață magnetizată, un curent pozitiv/negativ este indus în bobina capului. În continuare este prezentată geometria unei piste specifice discului magnetic.

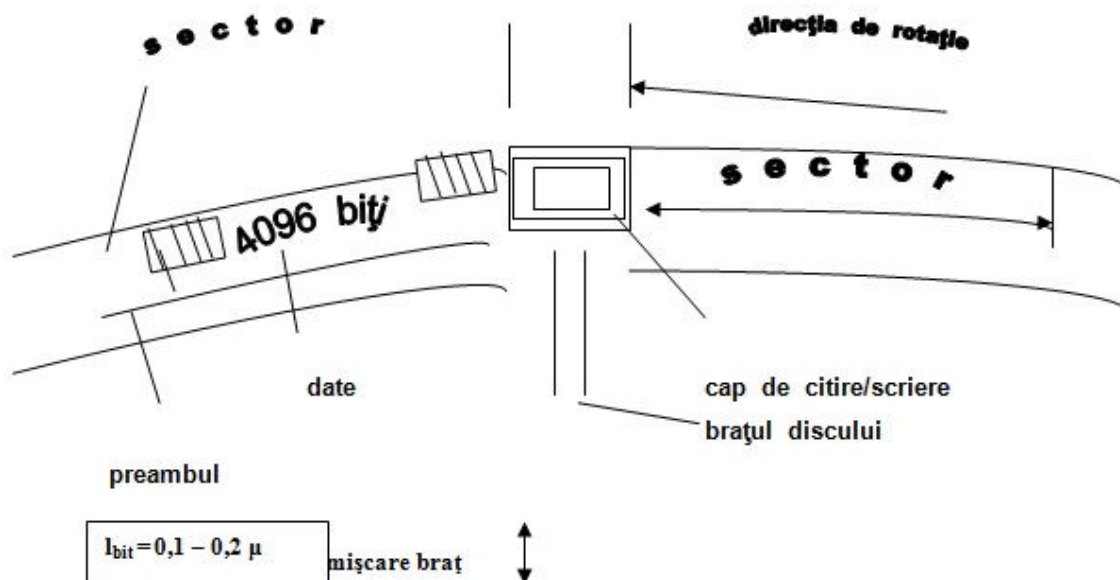


Figura 1.10 - O porțiune a unei piste a discului (sunt precizate două sectoare)

Secvența circulară de biți scrisă la rotația completă a discuției se numește pistă (*track*). Fiecare pistă este împărțită în sectoare de lungime fixă, de obicei de câte 512 octeți de date, precedați de un preambul care permite capului să se sincronizeze înainte de citire sau scriere. După zona de date urmează un cod corector de erori (ECC – Error Correcting Code), fie un cod Hamming, fie un cod care poate corecta erori multiple numit Cod Reed Solomon.

Sectoarele consecutive sunt separate de un spațiu între sectoare (*Intersector gap*). Capacitatea discului formatat este cu cca. 15% mai mică decât cea a discului neformatat. La fiecare distanță radială poate fi scrisă o altă pistă. Piste (coroane circulare) sunt cercuri concentrice față de axul discului.

Cu tehnologia actuală, discurile au între 800 și 2000 de piste pe centimetru, lărgimea pistei fiind între 5-10 microni. Discurile actuale ajung de la densități de 50000 până la 100000 biți/cm. Astfel de discuri se numesc **discuri Winchester**. Majoritatea discurilor sunt constituite din mai multe platan suprapuse pe verticală.

Fiecare platan dispune de propriul braț și cap. Brațele sunt sudate între ele; la o deplasare într-o nouă poziție radială sunt mutate toate odată. Setul de piste dintr-o poziție radială se numește cilindru.

Factorii care influențează performanțele discurilor sunt:

- deplasare în poziția radială corectă, căutare (*seek*);
- timpii medii de căutare (între piste alese aleatoriu) se situează în intervalul 5-15 msec.;
- latența de rotație (*rotational latency*) necesară rotirii platanului astfel încât sectorul dorit să ajungă sub capul de citire; întârzierea medie: $4 \div 8$ msec. Majoritatea discurilor se rotesc la 5400, 7200, 10800 rotații/minut;
- timpul de transfer depinde de densitatea lineară și de viteza de rotație și are valori de $25 \div 100$ microsecunde (la data de transfer $5 \div 20$ MB/sec., pentru un sector de 512 octeți).

Evident, timpul de căutare și latența sunt predominante în timpul de transfer. Fiecărui disc îi este asociat un controlor de date (*disk controller*), un cip care controlează dispozitivul.

Printre sarcinile controlerului sunt :

- acceptarea comenzilor de la software, de tipul READ, WRITE, FORMAT;
- controlul mișcării brațului;
- detecția și corecția erorilor;
- conversia datelor citite din memorie;
- când controlorul descoperă un sector defect, îl înlocuiește cu unul din sectoarele libere rezervate în acest scop în cadrul fiecărui cilindru.

Cea mai obișnuită formă de stocare de masă utilizată în prezent o reprezintă stocarea pe disc. În această structură, stocarea datelor se face pe un disc subțire care se învâртеște și care este acoperit cu o peliculă subțire de material magnetic.

Deasupra și/sau dedesubtul discului sunt plasate capete de **citire/scriere**, astfel încât, pe măsură ce discul se rotește, fiecare cap parcurge un cerc denumit **pistă** (*track*) care se află pe suprafața de sus sau jos a discului. Deoarece fiecare pistă conține mai multe informații decât vom dori să prelucrăm la un moment dat, pistele sunt împărțite în arce de cerc denumite **sectoare**, unde informațiile sunt înregistrate ca un șir continuu de biți.

Prin repoziționarea capetelor de **citire/scriere** se obține acces la sectoare situate pe diferite piste concentrice. Pistele sunt compuse din mai multe sectoare individuale, fiecare dintre ele putând fi tratate ca un șir de biți independent. Numărul de piste de pe suprafața discului, precum și numărul de sectoare de pe o pistă, diferă de la un sistem la altul. Dimensiunile sectoarelor sunt în general fie de 512 octeți, fie de 1024 octeți.

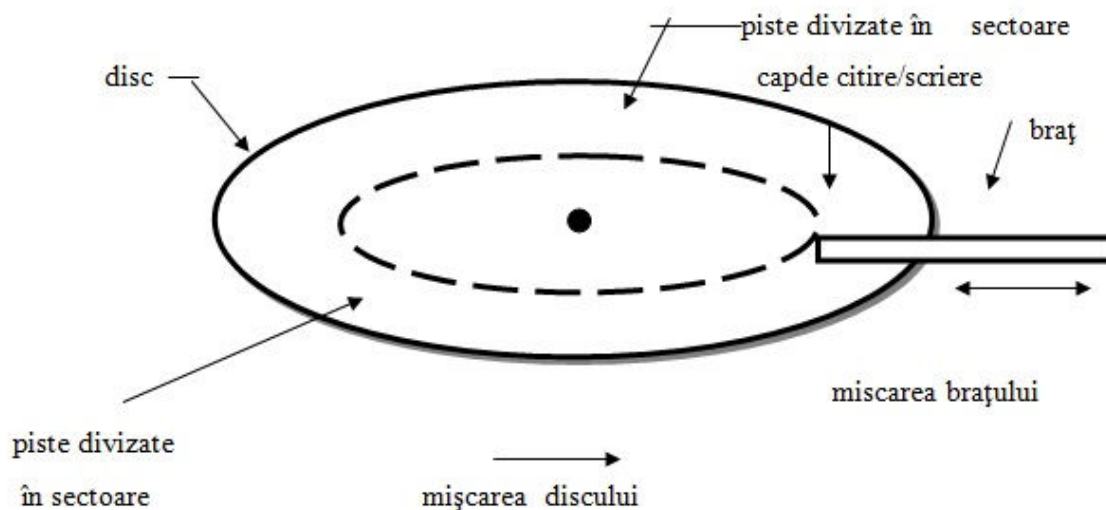


Figura 1. 11 - Structura unui sistem de stocare pe disc

Localizarea pistelor și sectoarelor nu reprezintă ceva permanent în structura fizică a discului, ele sunt marcate magnetic printr-un proces denumit **formatare** (sau **inițializare**) a discului. Cele mai multe sisteme de calcul pot reformata discurile atunci când formatul acestuia nu este compatibil cu cel propriu. Reformatarea unui disc distruge toate informațiile stocate anterior pe el.

Sistemele cu capacitate mică utilizează un singur disc, denumit și **dischetă/disc flexibil** (*floppy disk*). Sunt disponibile în comerț astfel de dischete. Dischetele au diametrul de $3\frac{1}{2}$ țoli, și sunt introduse într-o carcasă rigidă de plastic. Deși capacitatea dischetelor este limitată la câțiva megaocteți (1,44 MB), ele au avantajul că se introduc și se scot ușor în unitatea citire/scriere și sunt ușor de păstrat. Dischetele reprezintă o soluție bună pentru stocarea *off-line* a informațiilor.

Discurile de mare capacitate, care pot stoca mai mulți gigaocteți de date, constau dintr-un număr de cinci până la zece discuri fixe, montate în paralel pe un ax comun, cu suficient spațiu între ele încât să permită accesul capetelor de **citire/scriere**. Deoarece aceste discuri sunt rigide, ele sunt cunoscute sub numele de **hard-disc**.

În cazul acestor sisteme (hard-disc) capetele de citire/scriere nu ating suprafața discului, ceea ce-i permite acestuia viteze mari de rotație. Distanța dintre capetele de citire/scriere și suprafața dischetei este foarte mică, astfel încât o particulă de praf se poate bloca între cap și

suprafața discului deteriorându-le pe amândouă. Pentru prevenirea acestui fenomen hard-discul este închis într-o carcasă etanșă.

Pentru evaluarea performanței discurilor se folosesc mai multe criterii (parametrii):

- timpul de căutare (*seek time*) = timpul necesar deplasării capetelor de citire/scriere de la o pistă la alta;
- timpul de întârziere (*rotation delay/latency time*) = jumătate din timpul necesar pentru ca discul să efectueze o rotație completă, respectiv timpul mediu în care datele respective ajung în poziția capului de citire/scriere după ce acesta a fost adus la pista dorită;
- timpul de acces (*access time*) = suma dintre timpul de căutare și timpul de întârziere;
- rata de transfer (*transfer rate*) a datelor către sau de la disc.

Deoarece capetele de citire/scriere nu ating suprafața discului, ele pot avea viteze de rotație de cca. 5000—7000 rotații/minut, în timp ce în cazul discurilor flexibile aceste viteze nu depășesc 300 rotații/minut. Rata de transfer a discurilor fixe se măsoară în megabytes/secundă, față de cea a dischetelor care se măsoară în kilobytes/ secundă.

Dacă timpul de întârziere se măsoară, în cazul circuitelor electronice în nanosecunde (miliardimi de secundă) și chiar mai mici, timpii de căutare, întârziere și acces în cazul discurilor se măsoară în milisecunde (miimi de secundă).

1.3.2. Discuri compacte

Discul compact (CD) este compatibil cu cel utilizat în domeniul înregistrărilor muzicale, cu diferența că, pentru a obține rate ridicate de transfer al datelor, cititoarele de CD-uri din calculatoare, rotesc mult mai rapid discul. Aceste discuri, cu diametrul de cca. 5 țoli, sunt confecționate dintr-un material reflectorizant, acoperit cu o peliculă protectoare transparentă.

Înregistrarea informațiilor pe CD se face prin creare de striții în adâncimea suprafeței reflectorizante. Informațiile sunt stocate pe o singură pistă care are formă de spirală. Una din cele mai răspândite forme actuale de stocare a datelor pe compact-disc este reprezentată de dispozitivele ce pot fi numai citite, denumite CD-ROM (*Compact disk read only memory*). Capacitatea de stocare a acestor CD-ROM-uri este de 600 MB.

Sunt, de asemenea, dispozitive și sisteme CD care permit și modificarea datelor stocate. Există de asemenea sisteme care utilizează dispozitive magneto-optice pentru înregistrarea

informațiilor, topind suprafața reflexivă de pe CD cu ajutorul unei raze laser și apoi rearanjând-o prin intermediul unor câmpuri magnetice înainte ca aceasta să se răcească.

1.3.3. Benzi magnetice

Dispozitivele de stocare mai vechi utilizează banda magnetică. Informațiile se înregistrează pe o peliculă magnetică depusă pe o bandă de material plastic, care este stocată pe un sir de role.

Pentru acces la date, banda magnetică este introdusă într-o unitate de bandă care poate derula banda, citi și scrie informația sub controlul calculatorului. Există și unități de bandă magnetică cu cartuș care utilizează casete. Depinzând de formatul utilizat, pe unele benzi magnetice pot fi stocate volume de ordinul gigabytes (*GB*).

1.3.4. Înregistrări logice și fizice

Datele în memoria principală a unui calculator pot fi apelate la nivelul celulelor de memorie de dimensiunea unui octet. Proprietățile fizice ale dispozitivelor de stocare de masă impun ca manipularea datelor stocate să se facă utilizând unități cu dimensiuni mai mari de un octet.

Un bloc de date corespunzător caracteristicilor fizice ale unui dispozitiv de stocare este denumit **înregistrare fizică** (*physical record*). În afara împărțirii datelor în înregistrări fizice determinate de caracteristicile dispozitivului de stocare, fișierele stocate posedă o diviziune naturală, legată de structura aplicației din care fac parte.

De exemplu, un fișier care conține informații referitoare la angajații unei firme este alcătuit de obicei din blocuri de informații referitoare la fiecare angajat. Aceste blocuri de date care apar în mod natural sunt denumite **înregistrări logice** (*logical records*).

Dimensiunile înregistrărilor logice se potrivesc cu totul întâmplător ca lungime (mărime) cu dimensiunile înregistrărilor fizice impuse (ca mărime) de un dispozitiv de stocare. În consecință este posibil să existe mai multe înregistrări logice stocate ca o singură înregistrare fizică sau o înregistrare logică stocată pe mai multe înregistrări fizice. În acest spirit, pentru

recuperarea datelor de pe sistemele de stocare de masă este necesară o anumită activitate de decodificare.

1.4. Codificarea utilizată pentru stocarea informațiilor

În continuare vor fi studiate mai în amănunt tehnicile utilizate pentru reprezentarea informațiilor sub forma unor șiruri de biți.

1.4.1. Reprezentarea simbolurilor

Reprezentarea datelor într-un calculator se realizează prin proiectarea unui cod în care diferite simboluri (litere alfabetice, semne de punctuație etc.) au asociate șabloane (modele) de biți unice pentru ca informația să poată fi stocată sub forma unor propoziții codificate pe suportul magnetic. Astfel au fost create și folosite diverse coduri pentru diferite echipamente, acest lucru având ca efect apariția și proliferarea problemelor de comunicație.

Pentru a remedia această situație, Institutul American pentru Standarde (American National Standards Institute — ANSI) a adoptat codul *American Standard Code form Information Interchange (ASCII)*.

Acest cod utilizează modele cu o lungime de șapte biți pentru reprezentarea literelor mari și mici ale alfabetului englez, semnelor de punctuație, cifrelor de la 0 la 9, precum și a anumitor caractere de control (trecere rândul următor — *line feed*, revenire la marginea din stânga a rândului — *carriage return*, tabulator — *tab*).

În prezent codul *ASCII* este extins la un format de opt biți pe simbol, prin adăugarea unui zero pe poziția celui mai semnificativ bit în fața fiecărui șablon al vechiului cod pe șapte biți.

Există și alte coduri mai puternice capabile să reprezinte documente scrise într-o varietate de limbaje. Ele utilizează șabloane (modele) pe 16 biți (65.536 șabloane diferite) sau chiar pe 32 biți (peste 17 milioane de șabloane). [*Unicode*, cod dezvoltat de I S O (International Standard Organisation).]

1.4.2. Reprezentarea valorilor numerice

Metodele de stocare a informațiilor sunt inefficiente atunci când informațiile ce trebuie memorate sunt de natură numerică. Pentru înțelegere, să presupunem că vrem să stocăm nr. 99 sub forma unor simboluri *ASCII*; sunt necesari 16 biți, dar observăm că acesta este cel mai mare număr stocabil pe 16 biți.

O abordare mult mai eficientă este stocarea valorii reprezentate în baza doi (binar).

În notația binară (baza doi) poziția fiecărei cifre este asociată cu o anumită pondere, numai că ponderea asociată fiecărei poziții este de două ori mai mare decât ponderea poziției din dreapta. Mai exact, cifra cea mai din dreapta a unei reprezentări binare are asociată ponderea 1 (2^0), următoarea poziție în stânga cu 2 (2^1) și așa mai departe.

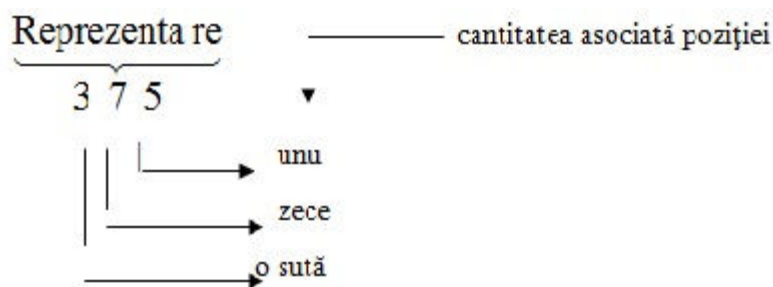


Figura 1.12 - Sistemul zecimal de numerație

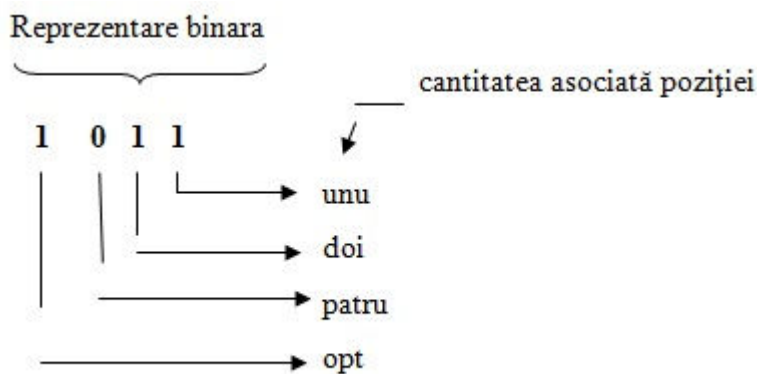


Figura 1.13 - Sistemul binar de numerație

Pentru a afla valoarea unei reprezentări în binar, vom înmulți valoarea fiecărei cifre cu ponderea asociată poziției sale și vom aduna rezultatele.

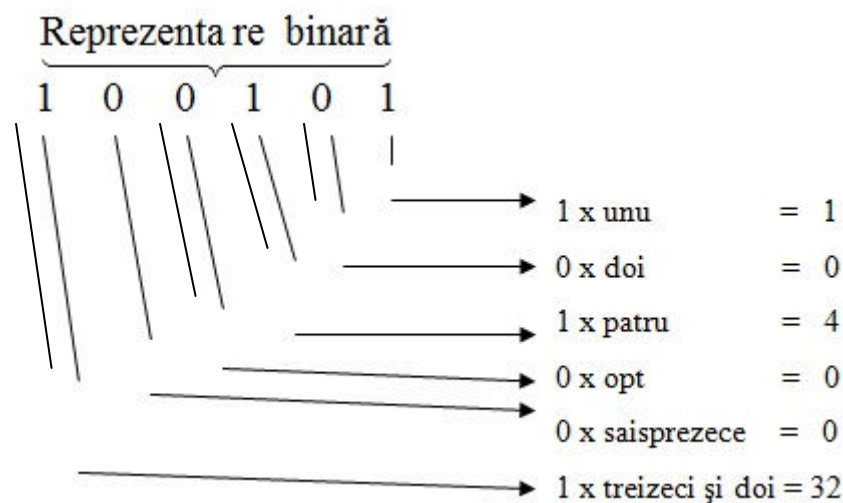


Figura 1.14 - Decodificarea reprezentării binare 100101

Pentru a afla reprezentarea în binar există un algoritm clasic.

- Pasul 1.** Se împarte valoarea la doi și se memorează restul.
- Pasul 2.** Cât timp câtul obținut diferă de zero, se continuă împărțirea noului cât la doi, memorându-se restul.
- Pasul 3.** Când s-a obținut un cât egal cu zero, reprezentarea în binar a valorii inițiale constă din resturile împărțirilor afișate de la dreapta la stânga în ordinea în care au fost memorate.

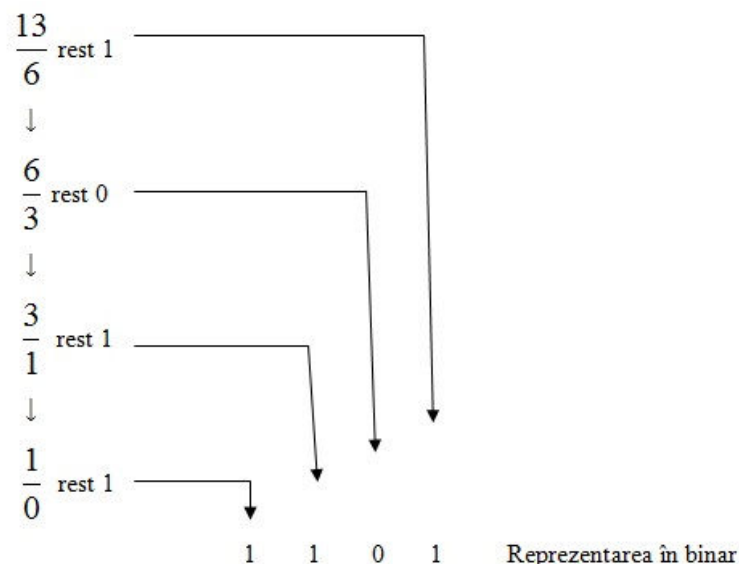


Figura 1.15 - Reprezentarea în binar a numărului treisprezece

1.4.3. Reprezentarea altor tipuri de date

Aplicațiile utilizate în informatica implica și folosirea de imagini, sunete, secvențe video. Tehnicile pentru reprezentarea acestor categorii de date nu sunt încă global standardizate la nivel mondial.

Una din metodele utilizate pentru stocarea imaginilor este considerarea imaginii ca o colecție (suma) de puncte (pixel $\leftrightarrow$ picture element). În forma cea mai simplă, o imagine alb-negru poate fi codificată ca un lung șir de biți ce reprezintă liniile de pixeli, unde fiecare bit are valoarea 1 sau 0, funcție de culoarea alb sau negru.

Reprezentările de acest tip sunt cunoscute sub numele de hărți de biți (*bit maps*), deci șirul de biți nu este altceva decât o hartă a imaginii reprezentate. Sistemele "populare" de hărți de biți includ fișiere de tip *TIFF* (*Tag Image Format File*) și *GIF* (*Graphic Interchange Format*). Fotografiiile sunt prezentate ca hărți de biți prin fișiere în format *JPEG* (*Joint Photographic Experts Group*).

În cadrul acestor reprezentări acestor fișier imaginea nu poate fi scalată la o dimensiune arbitrară. Există posibilitatea scalării imaginii prin memorare ca un set de directive care precizează modul de desenare. Acest mod de precizare a modului de desenare furnizează o

descriere compatibilă cu orice mărime a unităților sistemului de coordonate care ar putea fi specificat atunci când se afișează imaginea.

De asemenea, tot metode de reprezentare a datelor sunt și *MPEG (Motion Picture Experts Group)* — tehnică pentru date video și audio, și *D X F (Drowing Interchange Format)* — utilizat la sisteme de proiectare asistată de calculator, unde imaginile trebuie rotite și redimensionate pe ecranul monitorului.

1.5. Sistemul binar de numerație

Pentru studierea tehnicilor de stocare a valorilor numerice utilizate la calculatoarele de astăzi, trebuie mai întâi cunoscut sistemul de reprezentare binar.

1.5.1. Adunarea în binar

Pentru a aduna două valori reprezentate în notația binară, se procedează astfel:

Se adună cifrele din coloana cea mai din dreapta, se scrie cifra cea mai puțin semnificativă a acestei sume sub coloană, se transportă cea mai semnificativă cifră a sumei (dacă există) în următoarea coloană din stânga și se adună apoi coloana respectivă.

$$\begin{array}{r} \text{Exemplu :} \quad 00111010 \\ + \quad 00011011 \\ \hline \end{array}$$

Se adună cifrele cele mai din dreapta 0 și 1 și obținem cifra 1, pe care o scriem sub coloana respectivă.

$$\begin{array}{r} 00111010 \\ + 00011011 \\ \hline 1 \end{array}$$

Se adună apoi 1 și 1 din coloana următoare, obținând rezultatul 10. Vom scrie 0 sub coloană și vom transfera cifra 1 deasupra coloanei următoare.

$$\begin{array}{r}
1 \\
00111010 \\
+ 00011011 \\
\hline
101
\end{array}$$

Se adună cifrele 1, 0 și 0 din coloana următoare, obținând rezultatul 1, și vom scrie 1 sub coloană.

Cifrele din coloana următoare 1 și 1 dau ca rezultat al adunării 10; se va scrie cifra 0 sub coloana respectivă și vom transfera cifra 1 deasupra coloanei următoare.

$$\begin{array}{r}
1 \\
00111010 \\
+ 00011011 \\
\hline
0101
\end{array}$$

Adunăm cifrele 1, 1 și 1 din această coloană și obținem 11; se va scrie cifra 1 sub coloana respectivă și vom transfera cifra 1 deasupra coloanei următoare.

$$\begin{array}{r}
1 \\
00111010 \\
+ 00011011 \\
\hline
10101
\end{array}$$

Se va continua în același fel, obținând în final:

$$\begin{array}{r}
00111010 \\
+ 00011011 \\
\hline
01010101
\end{array}$$

1.5.2. Reprezentarea fracțiilor în sistemul binar

Pentru extinderea notației binare, pentru a fi adecvată reprezentării valorilor fracționare, se va utiliza notația în virgulă fixă (*radix point*). Aceasta înseamnă că cifrele de la stânga virgulei mobile (punctului) reprezintă partea întreagă a valorii și sunt interpretate ca în sistemul

binar, iar cifrele din dreapta punctului reprezintă partea fracționară a valorii și sunt interpretate într-o manieră similară, pozițiile lor având asociate ponderi fracționare.

Astfel prima poziție din dreapta punctului are atribuită ponderea $1/2$, următoarea $1/4$, apoi $1/8$ și așa mai departe. Rezultă că regula aplicată anterior rămâne valabilă, fiecare poziție are o pondere alocată de două ori mai mare decât cea a poziției din dreapta sa.

Exemplu :

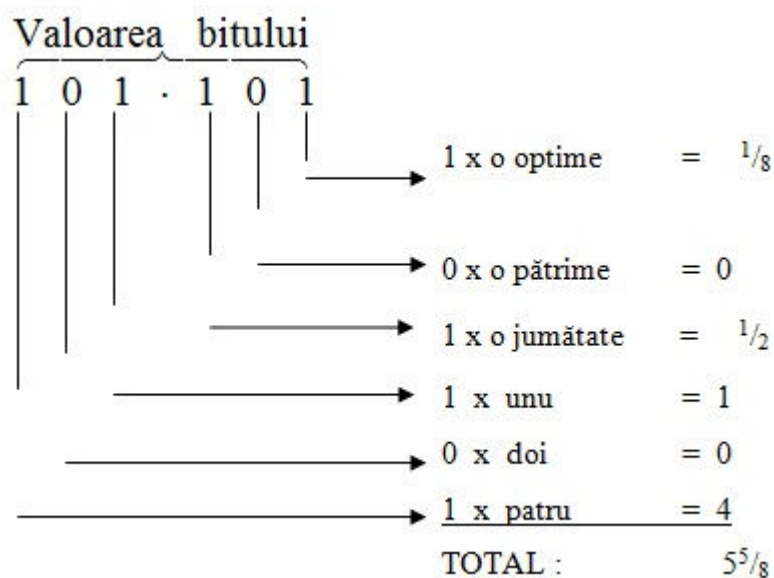


Figura 1.16 - Decodificarea reprezentării binare 101.101

Pentru a aduna două reprezentări binare în virgulă fixă, se vor alinia unul sub altul punctele de separare între partea întreagă și cea fracționară și se va aplica același proces de adunare ca și cel prezentat anterior.

Exemplu :

$$\begin{array}{r}
 10.011 \\
 + 100.11 \\
 \hline
 111.001 \\
 2
 \end{array}$$

1.6. Stocarea numerelor întregi

Adesea avem nevoie să memorăm atât valori pozitive cât și valori negative, deci este nevoie de un sistem de notare care să reprezinte ambele categorii de numere (pozitive și negative).

Sistemele de notație pentru reprezentarea ambelor categorii de numere pot folosi circuite electronice realizate și utilizate pe scară largă în cadrul echipamentelor de calcul. Vom prezenta în continuare două astfel de sisteme de notație: **notația în exces** (*excess notation*) și **notația în complement față de doi** (*two's complement notation*).

1.6.1. Notația în exces

Valorile din sistemul de reprezentare în exces sunt codificate utilizând cuvinte binare de aceeași lungime. Pentru realizarea unui sistem de notație în exces se stabilește mai întâi lungimea cuvintelor binare utilizate, apoi scriem una sub alta toate combinațiile posibile în ordinea în care ar apărea dacă am număra în binar.

În această înșiruire observăm că primul cuvânt binar care are pe poziția bițului cel mai semnificativ valoarea 1 survine aproximativ la jumătatea listei.

- se va considera că acest model reprezintă valoarea 0.
- cuvintele binare care urmează reprezintă valorile 1, 2, 3,...
- cuvintele binare care-l preced sunt utilizate pentru codificarea valorilor negative -1, -2, -3,...

Cuvânt <u>binar</u>	Valoarea <u>reprezentată</u>
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

Figura 1.17 - Tabel conversie pentru sistemul de notație în exces cu opt

Cuvânt <u>binar</u>	Valoarea <u>reprezentată</u>
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

Figura 1.18 - Sistemul de notație în exces care utilizeaza cuvinte cu lungimea de trei biți

1.6.2. Notația în complement față de doi

Cel mai folosit sistem de reprezentare a numerelor întregi în informatica este **notația în complement față de doi** (*two's complement notation*). Acest sistem utilizează un număr fix de biți pentru reprezentarea valorilor din cadrul sistemului. În figurile 1.13.1 si 1.13.2. sunt prezentate sistemele în complement față de doi bazate pe cuvinte de cod cu lungimea de trei și

patru. Pentru construcția unui astfel de sistem se începe cu un șir de biți cu valoarea 0 de lungime adecvată și apoi se numără în binar până când se ajunge la un șir de biți care începe cu un bit 0, iar toți ceilalți biți sunt 1. Cuvintele obținute astfel reprezintă valorile 0, 1, 2, 3,...

Utilizând cuvinte binare de lungime trei biți :

Cuvânt	Valoarea
<u>binar</u>	<u>reprezentată</u>
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

Figura 1.19 - Sisteme de notație în complement față de doi (trei biți)

Utilizând cuvinte binare de lungime patru biți :

Cuvânt	Valoarea	Cuvant	Valoarea
<u>binar</u>	<u>reprezentată</u>	<u>binar</u>	<u>reprezentata</u>
0111	7	1111	-1
0110	6	1110	-2
0101	5	1101	-3
0100	4	1100	-4
0011	3	1011	-5
0010	2	1010	-6
0001	1	1001	-7
0000	0	1000	-8

Figura 1.20 - Sisteme de notație în complement față de doi (patru biți)

Cuvintele care reprezintă valorile negative se obțin începând cu un șir de biți cu valoarea 1 și numărul în sens descrescător, până la atingerea cuvântului care începe cu un biț 1 și are toți ceilalți biți 0. Cuvintele astfel obținute reprezintă valorile -1, -2, -3,...

Se observă că bitul de semn, în notația în complement față de doi, este 1 pentru valorile negative și 0 pentru valorile pozitive. Complementul unui cuvânt binar este cuvântul obținut prin schimbarea tuturor biților de 1 în valoarea 0, respectiv a tuturor biților 0 în 1; cuvântul 0110 este complementul lui 1001 și invers.

În sistemul de codificare în complement față de doi pe 4 biți din figura 1.13.2., cuvintele care reprezintă valorile 2 și -2 se termină amândouă în 10, cuvântul corespunzător valorii 1 începe cu 00, iar reprezentarea valorii -2 începe cu 11. Această observație conduce la realizarea unui algoritm care să permită obținerea reprezentării binare pentru valori de semne contrare, dar care au același modul.

Vom copia cuvântul original începând de la dreapta până după apariția unui bit cu valoarea 1, apoi vom face complementul biților rămași (vom schimba toți biții 1 rămași în 0 și toți biții 0 în 1) pe măsură ce-i copiem.

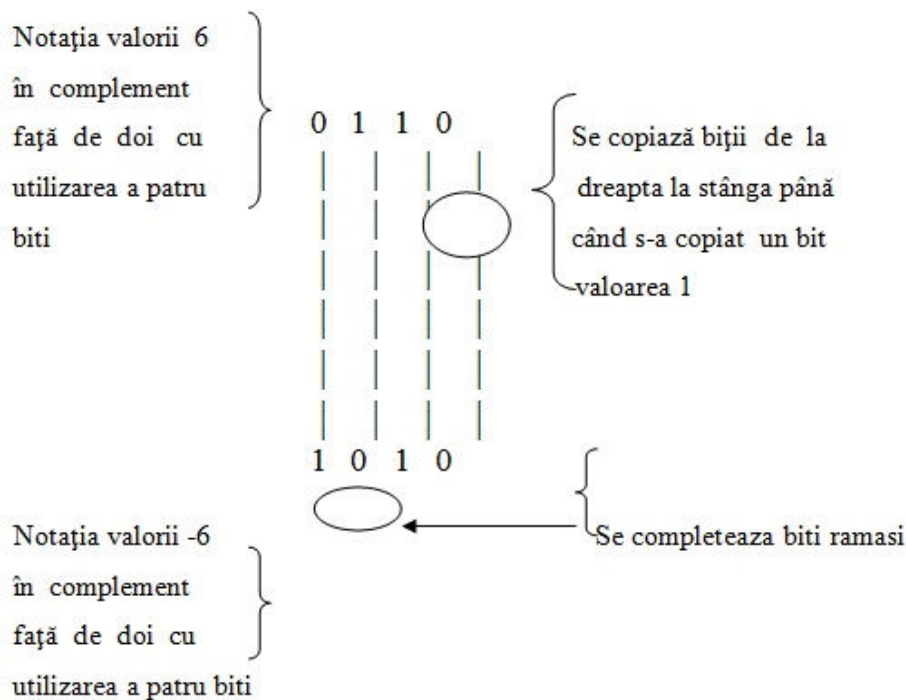


Figura 1.21 - Codificarea valorii -6 în complement față de doi utilizând 4 biți

Aceste proprietăți elementare ale sistemelor în complement față de doi conduc la obținerea unui algoritm pentru decodificarea reprezentărilor în complement față de doi.

Dacă modelul (șablonul) ce trebuie decodificat are bitul de semn 0, se va citi direct valoarea ca și cum cuvântul ar fi o reprezentare binară (ex. 0110 reprezintă valoarea binară 110, adică 6). Dacă șablonul care trebuie decodificat are bitul de semn 1, se va înțelege că valoarea reprezentată este negativă și trebuie găsit modulul acestei valori. Modulul căutat se va determina copiind cuvântul original de la dreapta la stânga, până la primul bit egal cu 1 (inclusiv acesta), apoi vom complementa biții rămași și îi vom scrie în continuare tot de la dreapta la stânga (lângă biții deja copiați), iar în final vom decodifica cuvântul obținut ca și când ar fi o reprezentare binară normală (ex. decodificăm 1010; bitul de semn este 1, deci valoarea numărului este negativă; convertim cuvântul și obținem 0110, ceea ce corespunde valorii 6, deci șablonul original reprezintă valoarea -6).

1.6.3. Adunarea numerelor reprezentate în complement față de doi

Pentru adunarea valorilor reprezentate în complement față de doi se aplică același algoritm ca și pentru adunarea în binar, cu excepția faptului că toate cuvintele binare, inclusiv rezultatul, au aceeași lungime. Altfel spus, la adunarea în complement față de doi, orice bit suplimentar generat la stânga răspunsului de către un transport final va fi eliminat.

$$\begin{array}{rcl}
 \text{De exemplu:} & 0\ 1\ 0\ 1 & 0\ 1\ 1\ 1 \\
 & \underline{0\ 0\ 1\ 0} & \underline{1\ 0\ 1\ 1} \\
 & 0\ 1\ 1\ 1 & 0\ 0\ 1\ 0
 \end{array}$$

Acceptând această convenție, să considerăm următoarele trei probleme de adunare:

$$\begin{array}{rcl}
 3 \rightarrow & 0\ 0\ 1\ 1 & (-3) \rightarrow 1\ 1\ 0\ 1 \\
 +2 \rightarrow & \underline{+0\ 0\ 1\ 0} & +(-2) \rightarrow \underline{+1\ 1\ 1\ 0} \\
 & 0\ 1\ 0\ 1 \rightarrow 5 & 1\ 0\ 1\ 1 \rightarrow -5
 \end{array}$$

$$\begin{array}{r}
 7 \rightarrow \quad 0 \ 1 \ 1 \ 1 \\
 +(-5) \rightarrow \underline{+1 \ 0 \ 1 \ 1} \\
 \hline
 0 \ 0 \ 1 \ 0 \rightarrow 2
 \end{array}$$

Figura 1.22 - Probleme de adunare utilizând notația în complement față de 2

În fiecare caz în parte, vom codifica valorile utilizând notația în complement față de doi (pe patru biți), vom efectua adunarea procedând așa cum s-a descris anterior și vom decodifica rezultatul înapoi la notația zecimală.

Se observă că, prin trecerea la notația în complement față de doi, putem calcula răspunsul corect aplicând în toate cazurile același algoritm de adunare. Dacă am folosi tehnicile uzuale, a treia problemă ar fi de fapt un proces complet diferit de primele două; este vorba de operația de scădere. Deci, în calculatorul care utilizează notația în complement față de doi se realizează numai adunare și negarea biților.

În consecință, dacă în calculator se dorește scăderea lui 5 (0101) din 7 (0111), mai întâi va fi schimbat în -5 (1011) și apoi se va efectua adunarea $0111 + 1011 = 0010$, ceea ce reprezintă valoarea 2.

În ceea ce privește înmulțirea, ea este o adunare repetată, iar împărțirea o scădere repetată (ex. $6:2$ reprezintă de fapt de câte ori poate fi scăzut 2 din 6 fără să se obțină un rezultat negativ).

Astfel putem efectua toate cele patru operații aritmetice standard (adunarea, scăderea, înmulțirea și împărțirea) utilizând un circuit pentru adunare combinat cu un circuit pentru negarea unei valori.

1.6.4. Problema depășirii superioare

În oricare din sistemele de numerație pe care le-am prezentat există o limită privind mărimea pe care pot să o reprezinte valoric. La utilizarea notației în complement față de doi cu cuvinte de patru biți, valoarea 9 nu are asociat un model (șablon), deci nu putem efectua corect adunarea $5+4$.

O problemă similară apare la utilizarea de cuvinte de cinci biți; de ex. să încercăm să reprezentăm valoarea 17, apar erori. Aceste erori se numesc **depășiri superioare** (*overflow*). Depășirea superioară, la utilizarea notației în complement față de doi, poate apare la adunarea a două valori negative sau pozitive.

Depășirea poate fi detectată prin verificarea bitului de semn al răspunsului. Altfel spus, dacă rezultatul adunării a două valori pozitive/ negative apare ca fiind negativ/pozitiv, este semnalată depășirea superioară.

Deoarece calculatorul manipulează cuvinte mult mai lungi decât cele precizate mai sus (3 biți/4 biți), valorile mari pot fi prelucrate fără să apară o valoare de depășire (de exemplu, pentru calculatoarele care utilizează cuvinte de 32 de biți pentru stocare, în notația în complement față de doi, este posibil lucrul cu valori de până la 2.147.483.647, fără apariția depășirii superioare).

Dacă sunt necesare valori și mai mari, se folosește metoda denumită **dubla precizie** (*double precision*). Această metodă permite ca lungimea cuvintelor utilizate să fie mărită față de cea utilizată de obicei de către calculator. Există și alte soluții, cum ar fi schimbarea unității de măsură cu păstrarea preciziei impuse.

1.7. Stocarea numerelor fracționare

În acest caz trebuie memorată și poziția virgulei zecimale nu numai modelele (șabloanele) de 0 și 1. Metoda uzuală pentru a face acest lucru se numește **notația în virgulă mobilă** (*floating point notation*).

1.7.1. Notația în virgulă mobilă

Vom explica notația în virgulă mobilă printr-un exemplu care utilizează numai un octet pentru efectuarea stocării.

Cel mai semnificativ bit din cadrul octetului este bitul de semn. Dacă bitul de semn este 0, valoarea stocată este pozitivă; iar dacă este 1, valoarea stocată este negativă. Împărțim cei șapte biți rămași în două grupuri/câmpuri: **câmpul exponentului** (*exponent field*) și **câmpul mantisei** (*mantissa field*). Considerăm trei biți, care urmează după bitul de semn, ca fiind câmpul exponentului, iar cei patru biți rămași ca fiind câmpul mantisei :

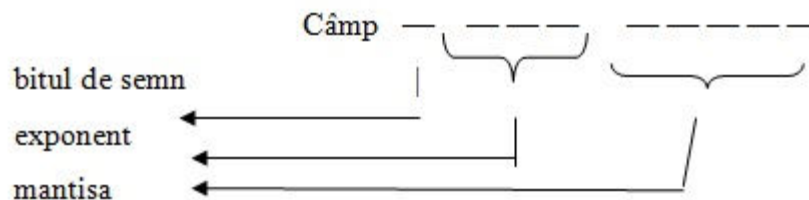


Figura 1.23 - Elemente ale notației în virgulă mobilă

Se poate explica semnificația acestor câmpuri considerând următorul exemplu. Octetul conține șirul de biți 01101011. Deci, bitul de semn = 0, exponentul = 110, mantisa = 1011.

Pentru a decodifica octetul, extragem mai întâi mantisa și plasăm în stânga ei o virgulă zecimală, obținând $\cdot 1011$.

Extragem apoi conținutul câmpului exponentului (110) și-l interpretăm ca pe un întreg stocat utilizând metode în exces pe trei biți, în cazul nostru reprezintă valoarea pozitivă 2. Acest fapt ne precizează că trebuie să mutăm virgula la dreapta cu doi biți (un exponent negativ înseamnă că virgula trebuie deplasată la stânga). În cazul nostru obținem : $10 \cdot 11$ ceea ce înseamnă $2^{3/4}$.

Observăm că bitul de semn din exemplul considerat este 0, deci valoarea reprezentată este pozitivă. Vom trage concluzia că 01101011 reprezintă valoarea $2^{3/4}$.

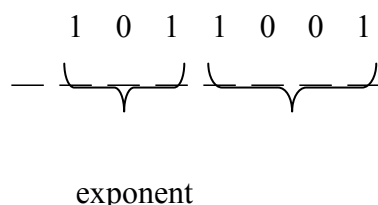
Pentru a stoca o valoare utilizând notația în virgulă mobilă, vom proceda în sens invers decât în prezentarea de mai sus. De exemplu, pentru a codifica valoarea $1^{1/8}$, o vom exprima mai întâi în notația binară 1.001 vom copia apoi cuvântul binar în câmpul rezervat mantisei de la stânga la dreapta, începând cu primul bit diferit de zero din reprezentarea binară.

Octetul arată astfel :

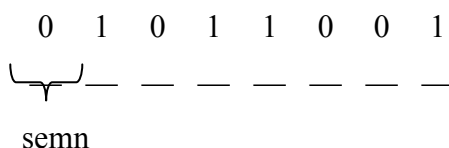
1 0 0 1
— — — — — — — —

Acum trebuie completat câmpul exponentului. Ne imaginăm conținutul câmpului mantisei având o virgulă zecimală la stânga și vom determina numărul de biți și direcția în care trebuie să fie deplasată virgula pentru a se obține numărul binar inițial.

În exemplul nostru, observăm că virgula din $.1001$ trebuie deplasată cu un bit la dreapta pentru a obține 1.001 . Din această cauză, exponentul trebuie să aibă valoarea pozitivă 1; vom plasa combinația 101 (care este reprezentarea valorii pozitive 1, în exces cu patru) în câmpul exponentului:



În final vom înscrie 0 în bitul de semn, deoarece valoarea stocată este pozitivă; la sfârșit octetul arată astfel :



1.7.2. Erori de rotunjire

Să studiem încercarea stocării valorii $2^{5/8}$ utilizând sistemul în virgulă mobilă pe un octet (prezentat anterior). Scriem mai întâi valoarea $2^{5/8}$ în binar : 10.101 . La copierea acestui rezultat în câmpul mantisei, vom descoperi că nu avem suficient spațiu și, ca urmare, ultimul 1 (cel care orespunde ultimului $1/8$) se va pierde.

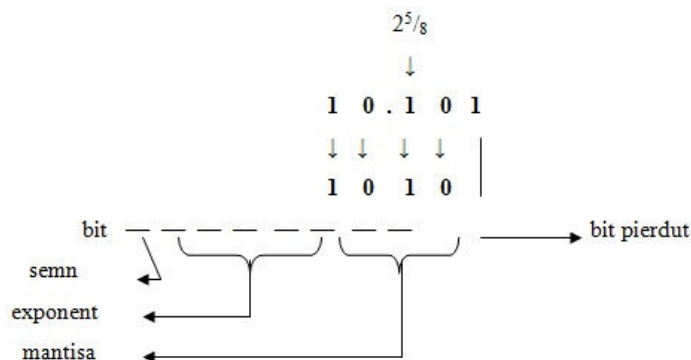
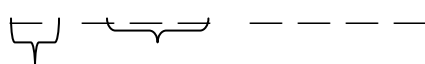


Figura 1.24 - Codificarea valorii $2^{5/8}$

Dacă ignorăm această problemă și continuăm completarea câmpului exponentului și a bitului de semn, obținem cuvântul:

0 1 1 0 1 0 1 0



bit de exponentul 2 pentru notația în exces
semn pentru exprimare pe 4 biți

Acest cuvânt însă reprezintă valoarea $2^{1/2}$.

Ceea ce s-a întâmplat poartă numele de **eroare de rotunjire** (*round - off error*), cauzată în acest caz de lungimea prea mică a câmpului de primire a mantisei.

Soluția pentru rezolvarea acestei probleme este creșterea dimensiunii câmpului mantisei, ceea ce se întâmplă în cazul calculatoarelor reale. De obicei, pentru stocarea valorilor în virgulă mobilă se utilizează cel puțin 32 biți; dacă nici în acest mod nu se rezolvă problema, se poate aplica conceptul de dublă precizie.

1.8. Erori de comunicație

La transferarea informațiilor între diverse componente ale calculatorului sau în cazul stocării datelor, există posibilitatea ca șirul de biți primit înapoi să nu fie identic cu cel original.

Pentru rezolvarea unor asemenea probleme, au fost dezvoltate diferite tehnici de codificare care permit detectarea și corectarea erorilor. În prezent, datorită faptului că aceste tehnici de codificare sunt implementate pe scară largă în componentele inteme ale sistemelor de calcul, ele sunt invizibile pentru cei care utilizează calculatoarele, dar pe ele se bazează fiabilitatea echipamentelor actuale.

1.8.1. Biți de paritate

O metodă simplă pentru detectarea erorilor se bazează pe următoarea regulă : dacă fiecare cuvânt binar manipulat are un număr impar de biți de 1, apariția unui cuvânt cu un număr par de

biți de 1 semnaleză o eroare. Pentru a folosi această regulă, avem nevoie de un sistem în care fiecare cuvânt binar să conțină un număr impar de biți 1, ceea ce se obține ușor prin adăugarea unui bit suplimentar, **bitul de paritate** (*parity bit*).

Bitul de paritate se plasează pe poziția bitului cel mai semnificativ, deci codul de opt biți devine un cod de nouă biți.

Bitul de paritate va lua valoare 0 sau 1, astfel încât cuvântul rezultat să aibe un număr impar de 1.

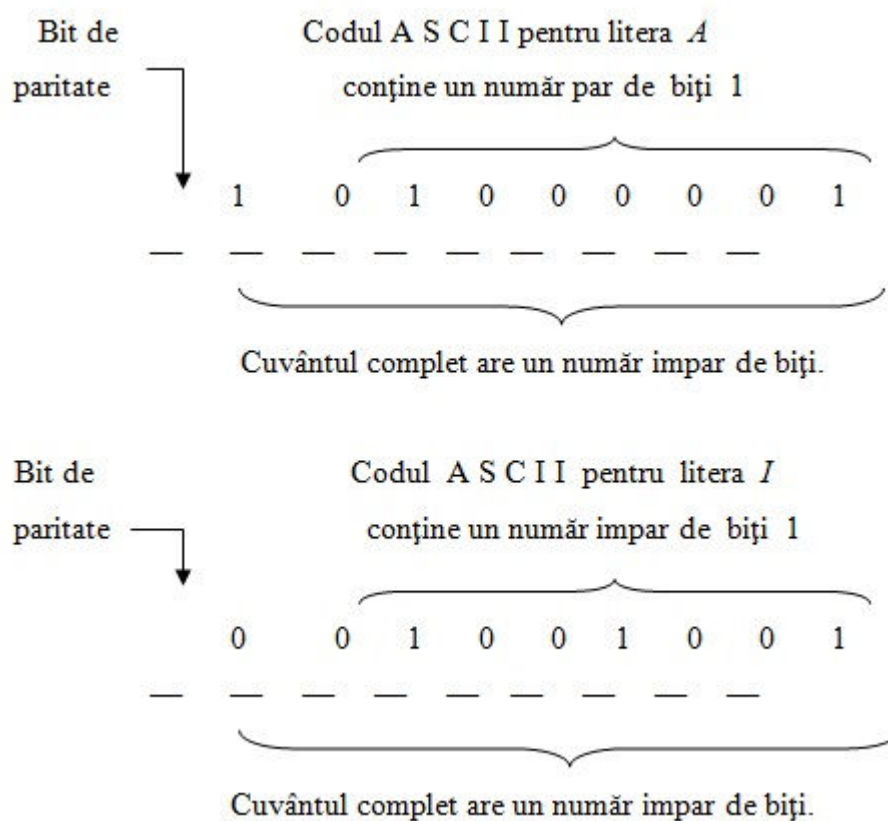


Figura 1.25 - Modificarea codurilor ASCII pentru caracterele *A* și *I*, astfel încât să aibe paritatea impară

După această modificare (precizată în figura de mai sus pentru caracterele *A* și *I*), ambele cuvinte vor avea nouă biți și conțin un număr impar de biți 1. După această modificare a sistemului de codificare, un cuvânt cu un număr par de biți 1 semnaleză faptul că s-a produs o eroare și deci cuvântul respectiv este incorect. Sistemul de paritate descris poartă numele de **paritate impară** (*odd parity*), deoarece fiecare cuvânt conține un număr impar de biți 1.

O altă tehnică utilizează **paritatea pară** (*even parity*). În această tehnică, fiecare cuvânt trebuie să conțină un număr par de biți 1, iar prezența unei erori este semnalată de apariția unui cuvânt cu un număr impar de biți 1. Șirurile lungi de biți sunt însoțite adesea de un grup de biți de paritate dispuși într-un **octet de control** (*checkbyte*).

Printre variantele principiului de verificare cu octet de control, se numără schemele de detecție a erorilor numite **semne de control** (*checksums*) și **control de coduri ciclice** (*cyclic redundancy check* — CRC).

1.8.2. Coduri corectoare de erori

Bitul de paritate permite detectarea unei erori singulare, dar nu furnizează informația necesară pentru corectarea erorii. Pot fi, însă, concepute **coduri corectoare de erori** (*error correcting codes*) care nu numai că detectează erorile, dar le și corectează.

Intuitiv, se poate crede că nu poate fi corectată informația dintr-un mesaj, decât dacă se cunoaște informația conținută de mesaj. Contrarul acestei afirmații se va demonstra în continuare. Pentru a înțelege modul de funcționare a codului corector de erori, vom defini distanța Hamming dintre două numere binare. *Distanța Hamming între două numere binare este numărul de biți prin care diferă cele două cuvinte.*

<u>Simbol</u>	<u>C o d</u>
A →	0 0 0 0 0 0
B →	0 0 1 1 1 1
C →	0 1 0 0 1 1
D →	0 1 1 1 0 0
E →	1 0 0 1 1 0
F →	1 0 1 0 0 1
G →	1 1 0 1 0 1
H →	1 1 1 0 1 0

Figura 1.26 – Exemplu de cod corector de erori

Exemplu: în figura de mai sus, distanța Hemming dintre simbolurile A și B în codul prezentat este patru, dintre B și C este trei.

Caracteristica importantă a codului prezentat este că oricare două cuvinte de cod sunt separate de o distanță Hemming de cel puțin trei biți. Altfel spus, trebuie să modificăm cel puțin patru biți în cod pentru a apare un alt cuvânt din lista propusă de coduri.

Să presupunem că recepționăm 010100. Dacă comparăm acest cuvânt binar cu lista de coduri propusă în fig. 1.26, obținem distanțele din figura 1.27. Vom putea trage concluzia că a fost transmis caracterul D, acesta fiind cel mai apropiat de codul recepționat.

Cu cât distanța Hemming dintre două coduri utilizate este mai mare, cu atât de pot detecta și corecta mai multe erori.

<u>Caracter</u>	<u>Distanța dintre cuvântul recepționat și caracterele codului propus</u>
A	2
B	4
C	3
D	1
E	3
F	5
G	2
H	4

Figura 1.27 - Decodificarea cuvântului 010100 utilizând codul din figura 1.26

TEST AUTOEVALUARE 1 (Stocarea datelor)

1. Care din urmatoarele reprezentari zecimale este egala cu valoarea binara : 111.01001
 1. $7 \frac{5}{32}$
 2. $7 \frac{9}{32}$
 3. $7 \frac{7}{16}$
2. Care din urmatoarele reprezentari zecimale este egala cu valoarea octala : 71.15
 1. $58 \frac{11}{64}$
 2. $57 \frac{13}{64}$
 3. $59 \frac{17}{64}$
3. Care dintre urmatoarele afirmatii despre memoria interna (RAM) este falsa :
 1. poate fi doar citita
 2. de regula are componente electromecanice in miscare
 3. informatia dispare la scoaterea de sub tensiune
4. Care din urmatoarele reprezentari zecimale este egala cu valoarea hexazecimala 41A.0CA:
 1. $1050 \frac{202}{4096}$
 2. $1060 \frac{198}{4096}$
 3. $1030 \frac{214}{4096}$
5. Convertiti urmatoarea fractie ordinara : $-1 \frac{3}{4}_{(10)}$ in informatii stocate intr-un octet, folosind notatia in virgula mobila.

Se va utiliza notatia in exces pe trei biti.

111.....	3
110.....	2
101.....	1
100.....	0
011.....	-1
010.....	-2
001.....	-3
000.....	-4

6. Convertiti in baza opt urmatoarele siruri de biti :

1. 1011011. 1011 011
2. 11101011. 1010101010101

7. Care sunt sirurile de biti reprezentate de urmatoarele numere hexazecimale :

1. BAF61C. 6A11
2. 11010.1101111

8. Converteți fiecare dintre următoarele reprezentări octale în forma zecimala (baza 10) echivalentă :

1. 6714. 1062
2. 314281. 217

9. Converteți fiecare dintre următoarele reprezentări binare în forma intreg si fractie ordinara echivalentă :

1. 1 0 0 1 . 0 1 0 1
2. 0 . 1 0 1 1 0 1

10. Exista urmatoarele informatii stocate intr-un octet, folosind notatia in virgula mobila :

1. 1 010 1 0 0 1

Calculati valoarea in baza 10 a sirului de biti de mai sus.

Pentru rezolvare utilizati urmatoarea notatie in exces pe trei biti

111.....	3
110.....	2
101.....	1
100.....	0
011.....	-1
010.....	-2
001.....	-3
000.....	-4

11. Care din urmatoarele reprezentari zecimale este egala cu valoarea binara : 1010.0111

1. $10 \frac{3}{4}$
2. $10 \frac{7}{16}$
3. $10 \frac{11}{64}$

12. Care dintre urmatoarele afirmatii despre memoria interna (RAM) este adevarata :

1. poate fi doar citita
2. de regula are componente electromecanice in miscare
3. informatia dispare la scoaterea de sub tensiune

13. Convertiti urmatoarea fractie ordinara : $-2 \frac{3}{4}_{(10)}$ in informatii stocate in mantisa unui octet, folosind notatia in virgula mobila. Se va utiliza notatia in exces pe trei biti.

111.....	3
110.....	2
101.....	1
100.....	0
011.....	-1
010.....	-2
001.....	-3
000.....	-4

14. Utilizati notatia hexazecimala pentru a reprezenta urmatoarele siruri de biti :

1. 1101101.110 011
2. 11101001011.1010101010101

15. Convertiti in baza opt urmatoarele siruri de biti :

1. 1101101.110 011
2. 1110100.10111010101

16. Exista urmatoarele informatii stocate intr-un octet, folosind notatia in virgula mobila :

1. 1 1 1 1 1 0 0 1

Convertiti sirurul de biti de mai sus in valoarea lor in baza 10

Pentru rezolvare utilizati urmatoarea notatie in exces pe trei biti

111.....	3
110.....	2

101.....	1
100.....	0
011.....	-1
010.....	-2
001.....	-3
000.....	-4

17. Utilizati notatia hexazecimala pentru a reprezenta urmatoarele siruri de biti :

1. 10110110.1110 011

2. 1110100101.110101010101

18. Se considera urmatoarea reprezentare a numerelor binare in complement fata de doi:

0111.....	7
0110.....	6
0101.....	5
0100.....	4
0011.....	3
0010.....	2
0001.....	1
0000.....	0
1111.....	-1
1110.....	-2
1101.....	-3
1100.....	-4
1011.....	-5
1010.....	-6
1001.....	-7
1000.....	-8

Efectuati adunarile $7+4$; si scaderea $6-2$.

Detaliati modul de rezolvare a fiecarei operatii solicitate.

19. Care din sirurile de mai jos ar putea reprezenta un numar scris in baza hexa (16) :

1. 10010
2. 101011
3. 10111

20. Care dintre urmatoarele afirmatii despre memoria externa (CD-W, HDD, DVD-W, discheta) este adevarata

1. poate fi doar citita
2. informatia dispare la scoaterea de sub tensiune
3. de regula are componente electromecanice in miscare

22. Care dintre urmatoarele afirmatii despre memoria externa (CD-W, HDD, DVD-W, discheta) este falsa :

1. poate fi doar citita
2. informatia dispare la scoaterea de sub tensiune
3. de regula are componente electromecanice in miscare

UNITATEA DE ÎNVĂȚARE 2

2. Manipularea datelor

2.1. Unitatea centrală de prelucrare

Circuitele care realizează diferite operații asupra datelor nu sunt conectate direct la celulele memoriei principale. Aceste circuite sunt grupate în **unitatea centrală de prelucrare** (*central processing unit* — *CPU*). **Unitatea centrală de prelucrare (CPU)** se compune din:

- unitatea aritmetico-logică (*arithmetic/logic unit*) conține circuitele care realizează manipularea datelor ;
- unitatea de comandă (*control unit*) conține circuitele utilizate pentru coordonarea activității calculatorului.
- registri (registrele)

În figura de mai jos este prezentată organizarea unui calculator [simplu] construit în jurul unei magistrale. Unitatea Centrală de Prelucrare (UCP/CPU – Control Processing Unit), „creierul” calculatorului, are rolul de a executa programele păstrate în memoria principală prin extragerea instrucțiunilor componente ale unui program, examinarea lor și execuția lor secvențială (una după alta).

Componentele sunt conectate printr-o *magistrală (bus)*, formată dintr-o mulțime de căi paralele prin care sunt transmise adrese, date și semnale de control. Magistralele se pot afla atât în exteriorul UCP, conectând-o cu memoria și dispozitivele de intrare/ieșire, cât și în interiorul UCP. Aceasta este alcătuită din mai multe componente :

- Unitatea de control care răspunde de extragerea instrucțiunilor din memoria principală și de determinarea tipului [lor] fiecăruia.
- Unitatea aritmetică și logică execută operații necesare pentru îndeplinirea instrucțiunilor (SI logic, SAU logic, ...).

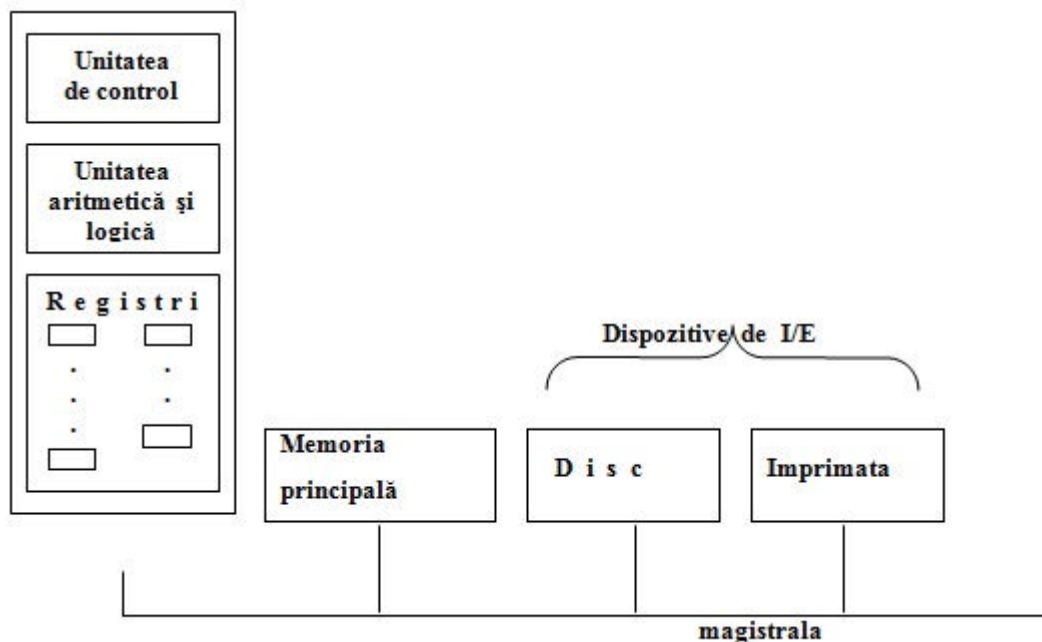


Figura 2.1 - Organizarea unui calculator simplu

UCP conține de asemenea și o memorie specială *foarte rapidă*, folosită pentru depozitarea rezultatelor temporare, precum și a unor informații de control. Această memorie este formată dintr-un număr de registre, fiecare cu o anumită dimensiune și o anumită funcțiune.

Principalele registre sunt:

- **contorul de program** (*PC – Program Counter*) – el indică următoarea instrucțiune care va fi extrasă pentru execuție;
- **registrul de instrucțiuni** (*Instruction Register – IR*), în care se păstrează instrucțiunea în execuție.

Interfața CPU / Memorie

Transferarea cuvintelor binare între unitatea centrală a unui calculator și memoria principală se realizează prin conectarea acestora printr-un grup de fire denumite **magistrală**. Prin intermediul magistralei, unitatea centrală poate să extragă (să citească)/să plaseze (să scrie) date din/în memoria principală, furnizând adresa celei de memorie dorite, împreună cu un semnal de citire. Analizând acest mecanism, observăm că presupune atât implicarea unității de comandă cât și a unității aritmetico-logice.

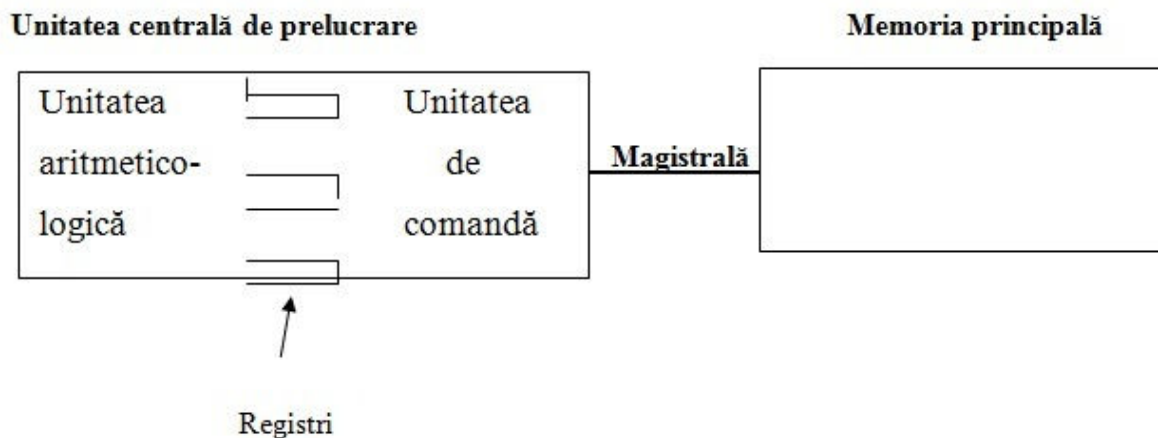


Figura 2.2 - Arhitectura unitate centrală de prelucrare/memorie principală

Organizarea CPU-ului Von Neumann (tipică)

Această parte se numește calea de date (*data path*) și include până la 32 de registre, Unitatea aritmetică și logică (UAL) și mai multe magistrale de legătură. Registrele trimit datele în cele două registre de intrare ale UAL, notate cu A și B, care le păstrează în timp ce UAL face calculele.

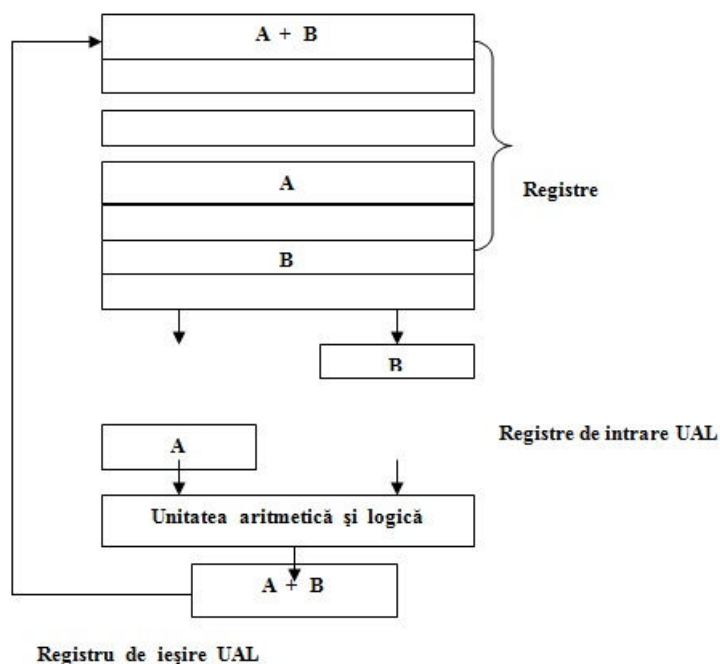


Figura 2.3 - Calea de date a unei mașini Von Neumann (adunarea)

După ce UAL execută cu datele de intrare, adunări, scăderi și alte operații simple, transmite rezultatul în registrul de ieșire. Majoritatea instrucțiunilor executate de UAP sunt de tip: registru-memorie, registru-registru. Instrucțiunile registru-memorie permit cuvintelor din memorie să fie încărcate în registre, de unde sunt folosite ca date de intrare pentru UAL în instrucțiunile următoare.

Instrucțiunile registru-registru extrag doi operanzi din registre, îi aduc în registrele de intrare UAL, execută o operație oarecare asupra lor (adunare, SI logic, ...) și depun rezultatul înapoi într-unul din registre. Acest ultim proces se numește ciclul căii de date (*Data path cycle*) și reprezintă „inima” celor mai multe UCP-uri. Cu cât acest ciclu este mai rapid, cu atât mașina merge mai repede.

2.2. Execuția instrucțiunilor

UCP execută fiecare instrucțiune printr-o serie de pași mici, după cum urmează :

- a. transformarea instrucțiunii următoare din memorie în registrul de instrucțiuni;
- b. schimbarea contorului de program, pentru a indica instrucțiunea următoare;
- c. determinarea tipului instrucțiunii proaspăt extrase;
- d. dacă instrucțiunea are nevoie de un cuvânt din [spațiul] de memorie, determinarea locului unde se găsește acesta;
- e. extragerea cuvântului într-unul din registrele UCP, dacă este cazul;
- f. executarea instrucțiunii;
- g. reluarea pasului *a* pentru a începe execuția instrucțiunii următoare.

Această secvență [de pași] se mai numește ciclul **extragere–decodificare–execuție** (*fetch–decode–execute*). În fig. 2.4 este prezentat în detaliu procesul adunării a două valori stocate în memorie.

Instrucțiuni în cod mașină

Instrucțiunile prezentate în fig. 2.4 reprezintă instrucțiuni executabile de către unitatea centrală de prelucrare și poartă denumirea instrucțiunii **în cod mașină** (*machine instructions*).

Când ne referim la instrucțiunile cunoscute de un calculator, observăm că ele pot fi clasificate în trei categorii (grupe): *instrucțiuni de transfer de date, instrucțiuni aritmetico-logice, instrucțiuni de control*.

- Pasul 1.** Se citește din memorie una din valorile care trebuie adunate și se plasează într-un registru.
- Pasul 2.** Se citește din memorie cealaltă valoare care trebuie adunată și se plasează într-un alt registru.
- Pasul 3.** Se activează circuitul de adunare, având ca intrări registrii utilizați în pașii 1 și 2.
- Pasul 4.** Se stochează rezultatul în memorie.
- Pasul 5.** Stop.

Figura 2.4 - Adunarea unor valori stocate în memorie

Instrucțiuni de transfer de date

Instrucțiuni care realizează deplasarea datelor dintr-un loc în altul, dar fără dispariția lor din poziția inițială. Pașii 1, 2 și 4 din fig. 2.4 intră în această categorie. Termenii **transfer** sau **mutare** sunt mai puțin adecvați; mai exacți ar fi **copiere** sau **clonare**.

Cererea de încărcare a unui registru de uz general cu conținutul unei celule de memorie este desemnată de o instrucțiune LOAD, iar transferul conținutului unui registru într-o celulă de memorie se face prin intermediul unei instrucțiuni STORE.

În fig. 2.4 pașii 1 și 2 reprezintă instrucțiuni LOAD, iar pasul 4 este o instrucțiune STORE.

O parte importantă a instrucțiunilor de transfer se referă la operații (comenzi) între dispozitive în afara CPU și memoria internă. Aceste instrucțiuni se ocupă de operațiile de intrare/ieșire (*input/ output - I/O*) din calculator și uneori plasate într-un grup distinct de instrucțiuni.

Instrucțiuni aritmetice și logice

Instrucțiunile care indică unități de comandă să solicite unități aritmetico-logice efectuarea unei anumite operații. Pasul 3 din fig. 2.4 face parte din această categorie de instrucțiuni.

Operațiile logice posibile de efectuat sunt: AND, OR și XOR. Operații care realizează deplasarea la dreapta sau la stânga a conținutului registrilor: SHIFT, ROTATE. Vor fi detaliate în subcapitolul 2.7.

Instrucțiuni de control

Instrucțiuni care nu manipulează date, ci dirijează modul de execuție al programelor. Pasul 5 din fig. 2.4 face parte din această categorie ca un caz elementar.

Această familie de instrucțiuni conține și instrucțiunile de salt (JUMP, BRANCH) care realizează acțiune ca unitatea de comandă să execute altă instrucțiune decât cea care urmează. Există două variante de instrucțiuni de salt: **salt necondiționat** și **salt condiționat**. Ca exemplu, pentru **saltul condiționat** prezentăm secvența următoare:

Pasul 1. Se încarcă (LOAD) un registru cu o valoare din memorie.

Pasul 2. Se încarcă (LOAD) alt registru cu altă valoare din memorie.

Pasul 3. Dacă a doua valoare este zero salt (JUMP) la pasul 6.

Pasul 4. Se împarte conținutul primului registru la conținutul celui de-al doilea registru și se depune rezultatul în al treilea registru.

Pasul 5. Se stochează (STORE) conținutul celui de-al doilea registru în memorie.

Pasul 6. Stop.

Figura 2.5 - Împărțirea a două valori stocate în memorie

Saltul condiționat se utilizează când se dorește îndeplinirea unei anumite condiții. Primele calculatoare erau foarte puțin flexibile, deoarece programul executat de fiecare dispozitiv era cablat în unitatea de comandă, ca o parte a sistemului.

Una din soluțiile utilizate în primele calculatoare electronice pentru a dobândi mai multă flexibilitate a constituit-o proiectarea unităților de control, astfel încât diversele blocuri să poată fi reconectate după nevoie. Acest lucru se poate realiza utilizând o placă de conexiuni realizate pe principiul plăcilor de comutare (utilizate în centralele telefonice).

Instrucțiunile ca șiruri de biți

Un pas înainte s-a făcut odată cu înțelegerea faptului că, în mod similar datelor, programele pot fi codificate și stocate în memoria principală a calculatorului. Programul unui calculator poate fi schimbat prin modificarea conținutului memoriei, în loc să se reconecteze blocurile unității de comandă a calculatorului.

Conceptul de program stocat în memoria calculatorului a devenit în prezent situația-standard de lucru. Pentru a-l putea aplica, calculatorul e proiectat astfel încât să recunoască anumite modele de biți ca reprezentând diferite instrucțiuni. Această colecție de instrucțiuni, împreună cu sistemul de codificare, poartă numele de **limbaj-mașină** (*machine language*) și definește modul de comunicare al algoritmului pe care calculatorul trebuie să-l execute.

O instrucțiune-mașină, din punctul de vedere al codificării, constă, de obicei, din două părți: **câmpul codului de operație** (*operation code - op.code*) și **câmpul operandului** (*operand code*). Șirul de biți care apare în câmpul "op-code"-ului specifică operația elementară (STORE, SHIFT, XOR, JUMP) a cărei execuție e solicitată de instrucțiune. Modelul de biți pentru codul operandului oferă detalii asupra operației respective (Ex.: pentru operația de tip STORE, informația din câmpul operandului precizează registrul care conține datele ce trebuie stocate precum și precizarea celulei de memorie în care se vor stoca ele).

Un limbaj tipic mașină

În continuare vom preciza cum ar trebui codificate instrucțiunile unui calculator obișnuit. Pentru aceasta propunem un calculator prezentat schematic mai jos.

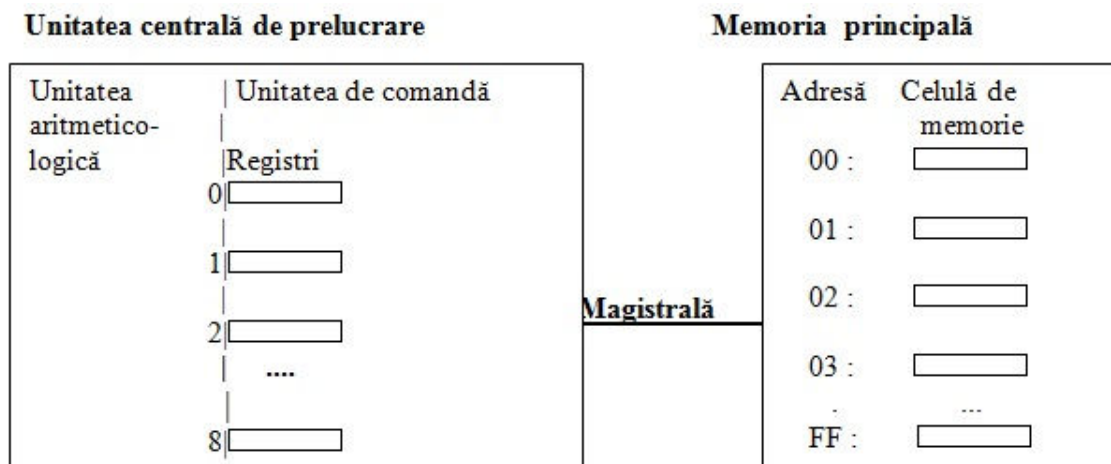


Figura 2.6 - Arhitectura calculatorului model

Calculatorul are 16 regiștri de uz general, numerotați de la 0 la F în hexazecimal, iar memoria sa conține 256 celule. Fiecare celulă de memorie este desemnată individual (identificată) printr-un număr întreg între 0 și 255. Vom considera că atât registrii cât și celulele de memorie au mărimea de opt biți.

Coduri de operație

Fiecare instrucțiune este codificată pe un număr de 16 biți, reprezentați cu ajutorul a patru cifre hexazecimale (vezi figura următoare).

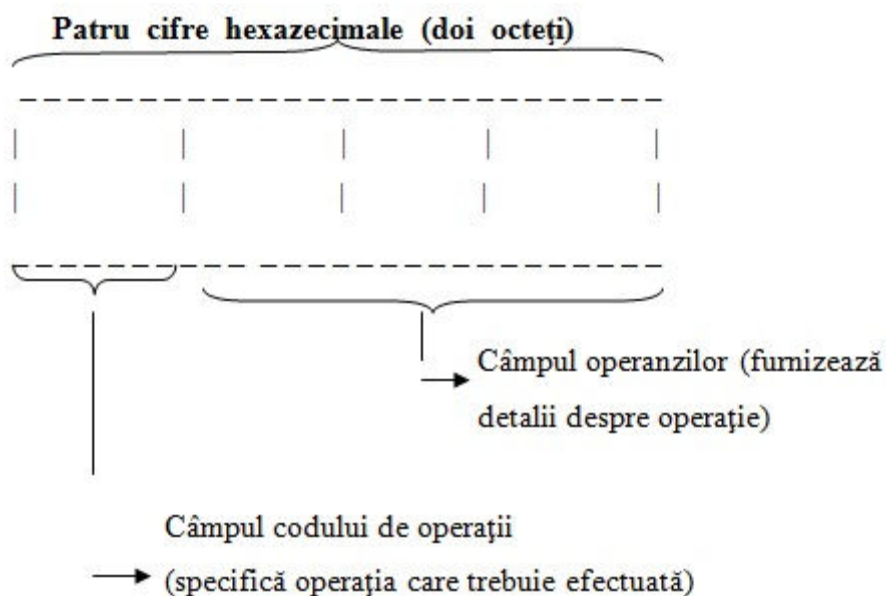


Figura 2.7 - Formatul unei instrucțiuni mașină pentru limbajul restrâns

Codul de operație al fiecărei instrucțiuni este reprezentat de primii patru biți sau de prima cifră hexazecimală.

Calculatorul dispune de două instrucțiuni de adunare ADD: una pentru adunarea reprezentărilor în complement față de doi și una pentru adunarea reprezentărilor în virgulă mobilă. Tratarea este diferită datorită faptului că adunarea cuvintelor binare care reprezintă valori codificate cu notația în complement față de doi necesită executarea unor acțiuni diferite față de cele de la adunarea valorilor reprezentate în virgulă mobilă.

2.3. Execuția programelor

Un program este "urmărit" de calculator prin copierea instrucțiunilor din memorie, în unitatea de comandă, pe măsură ce are nevoie de ele. Fiecare instrucțiune ajunsă în unitatea de comandă este decodificată și executată. Ordinea de extragere a instrucțiunilor din memorie corespunde ordinii în care sunt stocate, cu excepția când o instrucțiune de salt (JUMP) specifică altfel. Pentru înțelegerea executării unui program trebuie studiată amănunțit unitatea de comandă din interiorul CPU. Aceasta (UC) conține doi registri cu destinație specială: **contorul programului** (*program counter*) și **registrul de instrucțiuni** (*instruction register*), conform fig. 2.4. Registrul contorului programului conține adresa următoarei instrucțiuni care trebuie executată, permițând calculatorului să urmărească locul în care se află în program. Registrul de instrucțiuni este utilizat pentru stocarea instrucțiunilor în curs de execuție.

Unitatea de comandă își realizează sarcinile repetând continuu un algoritm, denumit **ciclul mașinii** (*machinecycle*), care constă în 3 pași: **extragere** (*fetch*), **decodificare** și **execuție** (fig. 2.8).

(1) **Extrage** următoarea instrucțiune din memorie (conform contorului programului) și apoi incrementează controlul programului.

(2) **Decodifică** șirul de biți din registrul de instrucțiuni

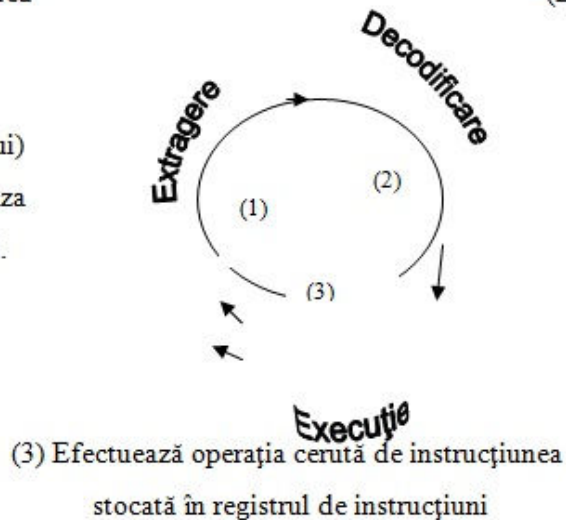


Figura 2.8 - Ciclul mașină

În timpul pasului de extragere, unitatea de comandă solicită memoriei principale să-i furnizeze următoarea instrucțiune care va fi executată. Unitatea știe unde se află următoarea

instrucțiune în memorie, deoarece adresa acesteia se află în registrul contorului programului. Unitatea de comandă plasează instrucțiunea recepționată din memorie în registrul său de instrucțiuni și incrementează apoi contorul programului astfel încât acesta să conțină adresa următoarei instrucțiuni.

Având astfel instrucțiunea în registrul de instrucțiuni, unitatea de comandă începe faza de decodificare din ciclul mașinii. În acest moment, ea analizează câmpurile codului de operație și operanzilor pentru determinarea acțiunilor ce trebuie efectuate.

După decodificarea instrucțiunii, unitatea de comandă intră în faza de execuție. De exemplu, dacă instrucțiunea se referă la încărcarea datelor din memorie, unitatea de comandă realizează (efectuează) operația de încărcare; dacă instrucțiunea se referă la o operație aritmetico-logică, unitatea de comandă activează unitatea aritmetico-logică, având ca intrări registrii corespunzători.

După execuția oricărei instrucțiuni, unitatea de comandă începe un nou ciclu al mașinii cu faza de extragere. Deoarece contorul programului a fost incrementat la sfârșitul fazei precedente de extragere, el furnizează din nou unității de comandă adresa corectă a instrucțiunii.

Programe și date

Mai multe programe pot fi stocate simultan în memoria principală a unui calculator, atât timp cât ocupă zone de memorie diferite, iar prin setarea adecvată a contorului programului se poate determina care program se va executa la pornirea calculatorului.

Deoarece datele sunt stocate de asemenea în memorie și sunt codificate tot cu cifre binare (0 și 1), calculatorul nu poate face distincția între date și programe.

Existența unui aspect comun pentru programe și date permite unui program să manipuleze alte programe (sau chiar pe el însuși) ca pe niște date.

2.4. Alte instrucțiuni

Pentru a avea o perspectivă mai largă, să studiem alte alternative la arhitectura de calculator prezentată.

Arhitecturi CISC și RISC

Proiectarea unui limbaj-mașină implică luarea multor decizii, una dintre ele fiind să construim:

- masina cu structură complexă — care să poată decodifica și executa o largă varietate de instrucțiuni;
- o mașină mai simplă care să dispună de un set limitat de instrucțiuni.

Prima structură se numește **calculator cu set complex de instrucțiuni** (*complex instruction set computer - CISC*), iar a doua opțiune conduce la realizarea unui **calculator cu set restrâns de instrucțiuni** (*reduced instruction set computer - RISC*).

Cu cât structura procesorului este mai complexă cu atât mai simplă este programarea, în timp ce în cazul calculatorului mai simplu aceeași operație ar necesita o secvență de mai multe instrucțiuni. Pe de altă parte, structurile complexe sunt mai greu și mai scump de realizat, utilizarea lor fiind mai costisitoare.

În prezent pe piață există atât procesoare CISC cât și RISC. Procesul Pentium (Intel Corporation) reprezintă un exemplu de arhitectură CISC; seriile de procesoare Power PC (dezvoltate de Apple Computer IBM și Motorola) urmează arhitectura RISC.

RISC vs. CISC

În anii '70, proiectanții de calculatoare au încercat acoperirea „spațiului semantic” dintre posibilitățile calculatoarelor și necesitățile limbajelor de programare de nivel înalt. Cu greu s-ar fi gândit atunci cineva să proiecteze mașini mai simple.

În 1980 s-a început proiectarea cip-urilor. VLSI pentru UCP fără interpretator, apărând termenul de RISC pentru cip-ul de unitate centrală. Aceste noi procesoare erau mult mai diferite decât procesoarele de până atunci, în sensul că aceste noi UCP-uri nu trebuiau să păstreze compatibilitatea cu produsele existente.

În această situație, proiectanții erau liberi să aleagă noi seturi de instrucțiuni care maximizau performanțele globale ale mașinii. Conta mai puțin cât dura o instrucțiune, dar conta numărul de instrucțiuni care puteau fi lansate în execuție într-o secundă. Caracteristica pregnantă a acestor procesoare a fost numărul relativ mic de instrucțiuni disponibile (cca. 50) comparativ cu numărul mult mai mare de instrucțiuni ale altor procesoare de calculatoare ale firmelor IBM, DBC (cca. 250-300).

Acronimul folosit pentru procesoarele calculatoarelor cu set redus de instrucțiuni este RISC (Calculator cu Set Redus de Instrucțiuni – *Reduced Instruction Set Computer*). S-ar putea crede că datorită unor avantaje oferite de tehnologia RISC, mașinile RISC ar fi trebuit să detroneze mașinile CISC, dar nu s-a întâmplat așa.

Care au fost motivele care au generat această situație:

- compatibilitatea cu modelele anterioare;
- dezvoltarea unor componente software dedicate gamei de procesoare Intel;
- procesoarele Intel au încorporat instrucțiuni RISC (un nucleu RISC) într-o arhitectură CISC. Chiar dacă această abordare hibridă nu este la fel de rapidă ca și cea RISC „pura”, le permite obținerea unei performanțe globale competitive și rularea vechiului software nemodificat.

2.5. Principii de proiectare pentru calculatoarele actuale

(principiile proiectării RISC – *RISC design principles*)

1) Toate instrucțiunile sunt executate direct de către hardware

Instrucțiunile uzuale nu sunt interpretate prin microinstrucțiuni. Eliminarea unui nivel de interpretare conduce la creșterea majorității instrucțiunilor. Pentru seturile de instrucțiuni CISC, instrucțiunile mai complicate pot fi „sparte” în părți separate, acțiune care încetinește mașina, dar pentru instrucțiuni care apar mai rar.

2) Maximizează rata de lansare în execuție a instrucțiunilor

În calculatoarele moderne, lansarea în execuție a cât mai multe instrucțiuni pe secundă este problema principală. Procesorul MIPS (*Millions of Instructions per Second*) realizează paralelismul execuției instrucțiunilor, ceea ce conduce la îmbunătățirea performanțelor.

3) Instrucțiunile trebuie să fie ușor de decodificat

Decodificarea instrucțiunilor individuale, pentru a determina resursele de care au nevoie, este un factor critic care limitează rata de lansare a instrucțiunilor. Cu cât se folosesc mai puține formate diferite pentru instrucțiuni, acestea sunt mai eficiente. De asemenea, numărul mic de câmpuri cu dimensiune prestabilită este o altă componentă care poate eficientiza execuția instrucțiunii.

4) Numai instrucțiunile LOAD și STORE trebuie să acceseze memoria

Unul din modurile simple de a descompune operațiile în pași separați este impunerea condiției ca operanzii majorității instrucțiunilor să fie transferați în/din registre.

Deoarece accesul la memorie poate dura mult, cel mai bine ar fi să se suprapună execuția acestor instrucțiuni cu a altora, dacă ele nu fac altceva decât să mute operanzi între registre și memorie.

5) Furnizează registre suficiente

Datorită accesului la memorie relativ lent, sunt necesare multe registre (≥ 32). Trebuie evitat să se intre în criză de registre și să fim obligați să le salvăm în memorie.

2.6. Prelucrare simultană

Există o limită în ceea ce privește dezvoltarea calculatoarelor foarte rapide, semnalele electrice se propagă prin circuite cu maximum viteza luminii. Chiar și viitoarele "calculatoare optice" sunt afectate de această limitare.

Deoarece lumina (unda electromagnetică) parcurge o distanță de aproximativ 30 de cm. într-o nanosecundă (o miliardime de secundă), rezultă că sporirea vitezei de execuție (lucru) a unui calculator devine în ultimă instanță o problemă de miniaturizare. Într-un efort de rezolvare a acestei probleme, cercetarea și-a îndreptat atenția asupra conceptului de **capacitate de transfer** (*throughput*).

Capacitatea de transfer se referă la cantitatea totală de operații pe care le poate efectua calculatorul într-un anumit timp. Îmbunătățirea capacității de transfer a unui calculator (fără creșterea vitezei de execuție) este tangibilă prin tehnica de **prelucrare simultană** (*pipelining*). Această metodă se referă la posibilitatea ca în orice moment, în conductă (*pipe*) să se afle mai multe instrucțiuni "în lucru". O instrucțiune este executată, alta este decodificată și încă o alta este extrasă din memorie.

Datorită "prelucrării" în același timp a 3 instrucțiuni, capacitatea de transfer a calculatorului crește de 3 ori.

Calculatoare -multiprocesor

Alte soluții pentru creșterea capacității de transfer face parte din categoria **prelucrării paralele** (*parallel processing*), în care se utilizează mai multe procesoare pentru executarea operației curente. Argumentul în favoarea acestei abordări îl reprezintă creierul uman.

Susținătorii prelucrării paralele se pronunță în favoarea calculatoarelor-multiprocesor, care conțin, în opinia lor, configurații cu un factor de utilizare mult mai ridicat.

2.7. Instrucțiuni aritmetice și logice

Grupul **operațiile aritmetice și logice** conține instrucțiuni care solicită operații aritmetice, logice și de deplasare.

Operații logice

Operațiile logice *AND* (ȘI), *OR* (SAU), *XOR* (SAU echivalent) se pot extinde la operații care combină două șiruri de biți pentru a produce o ieșire de forma unui șir de biți, aplicând operația elementară bit cu bit.

Exemplu :

	1 0 0 1 1 0 1 0	
AND	<u>1 1 0 0 1 0 0 1</u>	
	1 0 0 0 1 0 0 0	
	1 0 0 1 1 0 1 0	1 0 0 1 1 0 1 0
OR	<u>1 1 0 0 1 0 0 1</u>	<u>XOR 1 1 0 0 1 0 0 1</u>
	1 1 0 1 1 0 1 1	0 1 0 1 0 0 1 1

Operația AND este utilizată la **maskare** (*masking*).

Exemplu :

	0 0 0 0 1 1 1 1	← operand maskă (<i>mask</i>)
AND	<u>1 0 1 0 1 0 1 0</u>	

În acest caz operandul denumit **maskă** (*mask*) determină care parte a celui alt operand va afecta rezultatul. Deci operația AND permite copierea unei părți a unui șir de biți, plasându-se 0 în partea neduplicată.

Operația OR poate fi utilizată și ea pentru a copia o parte a unui șir de biți, plasându-se însă pe pozițiile neduplicate. Una din utilizările principale ale operației XOR o reprezintă compunerea complementului unui șir de biți, astfel:

$$\begin{array}{r}
 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\
 \text{XOR } 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0 \\
 \hline
 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1
 \end{array}$$

Operații de rotire și deplasare la nivel de bit

Operațiile de rotire și deplasare furnizează posibilitatea de deplasare a biților dintr-un registru și sunt folosite la rezolvarea problemelor de aliniere. Aceste operații sunt clasificate după direcția de mișcare (stânga/dreapta), ținându-se cont și dacă procesul este circular.

Dacă se face deplasarea către un capăt al șirului de biți, bitul de la capătul spre care se face deplasarea dispare, iar la celălalt capăt al șirului apare un spațiu liber. Operațiunile de deplasare se diferențiază tocmai prin ceea ce se întâmplă cu acest bit suplimentar în urma deplasării.

Una din tehnicile de deplasare este să se plaseze bitul suplimentar în spațiul liber de la celălalt capăt. Rezultatul este o deplasare circulară, denumită **rotație**.

Altă tehnică de deplasare elimină bitul de la capătul șirului spre care se face deplasarea și completează cu 0 spațiul liber apărut la celălalt capăt, adică realizează o **deplasare logică** (*logical shift*).

Se întâlnesc adesea deplasări la dreapta care completează întotdeauna spațiul liber, cu valoarea bitului de semn; acestea se numesc **deplasări aritmetice** (*arithmetic shift*).

Operații aritmetice (precizări suplimentare)

Aceste operații pot fi adesea efectuate utilizând doar operația de adunare, alături de negarea logică.

În cazul adunării:

- dacă valorile care trebuie adunate sunt stocate utilizându-se notația în complement față de doi, operația de adunare trebuie realizată ca o operație de adunare în binar;

- dacă operanzii sunt stocați utilizându-se notația în virgulă mobilă pentru adunare, trebuie mai întâi efectuată extragerea mantiselor operanzilor, deplasarea acestora la stânga sau la dreapta în funcție de valoarea câmpurilor exponenților, verificarea biților de semn, adunarea propriu-zisă și apoi convertirea rezultatului în virgulă mobilă.

În cazul celor 2 operații de adunare, din punctul de vedere al calculatorului între ele nu există nici o similitudine.

2.8. Comunicația între unitatea centrală și controlere

Comunicația între unitatea centrală de prelucrare și un controler este controlată la fel ca și comunicația dintre CPU și memoria principală.

Faptic, controlerul este reprezentat de un bloc de celule din memoria principală. Atunci când CPU scrie un șir de biți într-o celulă de memorie din cadrul blocului de memorie (ex. instrucțiunea STORE), șablonul e transferat de fapt controlerului și nu memoriei. Similar, atunci când CPU încearcă să citească date dintr-una din celulele de memorie (instrucțiune LOAD), ea primește un șir de biți de la controler. Acest sistem de comunicație, denumit **mapare în memorie a operațiilor de intrare/ieșire** (*memory mapped I/O*) este reprezentată de fig. 2.9.

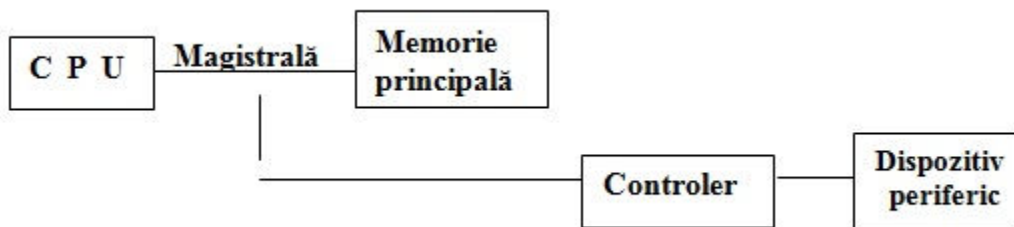


Figura 2.9 - Reprezentarea principală a mapării în memorie a operațiilor de I / O

Blocul de adrese asociate unui controler este denumit **port**, el reprezintă "poarta" prin care informațiile intră sau ies din calculator.

Între controler și dispozitivul periferic pe care-l controlează are loc o comunicare în ambele sensuri. Dacă n-ar exista o cale de comunicație în ambele sensuri între calculator și imprimantă (de exemplu), imprimanta ar rămâne foarte repede în urmă.

Controlere

Comunicația dintre unitatea centrală de prelucrare a unui calculator și un dispozitiv periferic este controlată de un dispozitiv intermediar, denumit **controler** (*controller*). Fiecare controler gestionează comunicația cu un anumit tip de dispozitiv periferic. Un **controler** corespunde fizic unei plăci cu circuite electrice.

Controler-ul convertește mesajele și datele la forme compatibile cu caracteristicile interne ale calculatorului respectiv la cele ale dispozitivelor periferice atașate controlerului.

Controlerele sunt atașate la aceeași magistrală care conectează unitatea centrală la memoria principală (fig. 2.10).

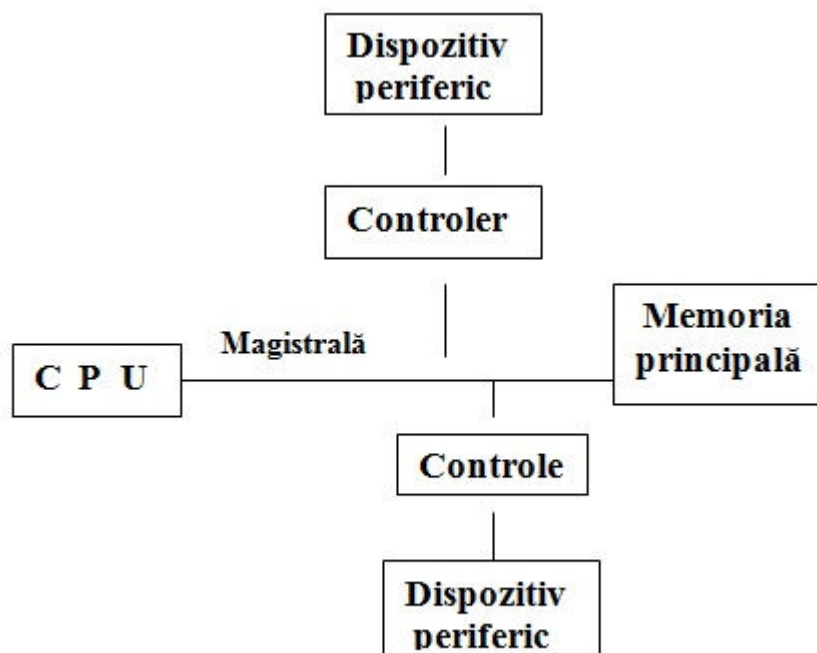


Figura 2.10 - Conectarea controlerelor la magistrala unui calculator

Fiecare controler monitorizează semnalele transmise de unitatea centrală de prelucrare și răspundere atunci când îi este adresat un semnal. Abilitatea (facilitatea) unui controler de a accede la memoria principală a calculatorului poartă numele de **acces direct la memorie** (*direct*

memory access - DMA). Unitatea centrală poate trimite controlerului cereri codificate prin care să-i ceară să citească un anumit sector de pe disc și să plaseze aceste date într-o anumită zonă de memorie precizată.

Apoi CPU poate continua execuția altor operații în timp ce controlerul efectuează cererea solicitată. După terminarea sarcinii atribuite, controlerul transmite prin magistrala calculatorului un anumit semnal către CPU (astfel de semnale iau forma de întreruperi și vor fi studiate în cap. Sistemul de operare).

Un bloc de memorie utilizat pentru transferul datelor spre și dinspre dispozitivele periferice poartă numele de **zonă-tampon** (*buffer*). Atașarea controlerelor în magistrala unui calculator mărește semnificativ complexitatea operațiilor de control al comunicației de-a lungul acestei căi principale de comunicație.

Chiar în cazul unei proiectări foarte bune, magistrala principală poate deveni un punct critic, cunoscut sub numele de **gâtuirea von Neumann** (*von Neumann bottleneck*), se datorează concurenței pentru accesul la magistrală între unitatea centrală de prelucrare și controlere.

2.9. Comunicația serială și paralelă

Comunicația dintre diferite părți ale unui sistem de calcul se efectuează într-una dintre cele două forme elementare paralelă sau serială. Este vorba de modul de transfer al șirurilor de biți. În cazul **comunicației paralele** (*parallel communication*), toți biții dintr-un șir sunt transferați simultan, fiecare pe o linie separată. În acest mod se realizează transferul rapid al datelor, dar este necesară o linie de comunicație cu un număr mare de cabluri electrice.

Comunicația serială (*serial communication*) se bazează pe transmiterea șirului bit cu bit. Această metodă este mai lentă, dar linia de comunicație este mai simplă.

Un exemplu obișnuit îl reprezintă liniile telefonice, informațiile digitale fiind convertite în semnale audio cu ajutorul unui dispozitiv numit **modem** (**modulator-demulator**). Datorită limitărilor impuse de caracteristicile sistemului telefonic existent, o astfel de comunicație nu se poate realiza prin tehnicile de comunicație paralelă.

Viteza comunicației seriale se măsoară în biți pe secundă (bps), iar domeniul de variație se situează între câteva sute de biți pe secundă și milioane de biți pe secundă. O altă unitate

uzitată este **rata band** (*band rate*); ea se referă la viteza cu care se schimbă starea liniei pe care are loc comunicația.

O altă metodă de creștere a eficienței transferurilor de date (inclusiv stocarea datelor) este **compresia de date** (*data compression*), adică reducerea numărului de biți necesar pentru reprezentarea informațiilor.

În cazul reprezentării unor șiruri de caractere se poate recurge la un cod Huffman (cod dependent de frecvență). În cadrul acestui cod, lungimea unui șir de biți care reprezintă un caracter să fie invers proporțională cu frecvența de utilizare a caracterului.

În acest fel se obține o reprezentare mai scurtă a textului decât dacă am utiliza un cod de lungime uniformă (codul ASCII). Eforturile de standardizare a tehnicilor de compresie a datelor au dus la includerea acestora în multe din modemurile existente în prezent pe piață. Atunci când 2 modemuri care utilizează scheme de compresie compatibile comunică între ele, modemul emițător comprimă datele înainte de a efectua transmisia, iar modemul receptor decomprimă datele după recepționarea lor.

Folosind asemenea soluții, modemurile pot obține rate de transfer echivalente cu 56700 bps, chiar dacă de fapt sunt transmiși numai 14400 biți pe secundă, la o rată band de 1200.

Limbajul - mașină

Fiecare instrucțiune/mașină are o lungime de doi octeți. Primii patru biți alcătuiesc câmpul codului de operație, iar următorii doisprezece biți formează câmpul operandului.

În tabelul următor sunt prezentate instrucțiunile/mașină, în notație hexazecimală, însoțite de o scurtă descriere. Literele R, S și T sunt utilizate în locul cifrelor hexazecimale pentru a reprezenta identificatorul de registri. Literele X și Y sunt utilizate în locul cifrelor hexazecimale în diferite câmpuri care nu reprezintă registrii.

C o d operație	Operand	D e s c r i e r e
1	R X Y	Încarcă (LOAD) registrul R cu valoarea găsită în celula de memorie a cărei adresă este X Y. <u>Ex.</u> 1 4 A 3 va avea ca rezultat plasarea conținutului celulei de memorie cu adresa A 3 în registrul 4.
2	R X Y	Încarcă (LOAD) registrul R cu valoarea reprezentată de șirul de biți X Y. <u>Ex.</u> 2 0 A 3 va avea ca rezultat înscrierea valorii A 3 în registrul 0.
3	R X Y	Stochează (STORE) valoarea registrului R în celula de memorie a cărei adresă este X Y. <u>Ex.</u> 3 5 B 1 va avea ca rezultat plasarea conținutului registrului 5 în celula de memorie cu adresa B 1.
4	O R S	Mută (MOVE) conținutul registrului R în registrul S. <u>Ex.</u> 4 0 A 4 va avea ca rezultat copierea conținutului registrului A în registrul 4.
5	R S T	Adună (ADD) șirurile de biți din registrii S și T, ca și cum ar fi reprezentări în complement față de doi, și depune rezultatul în registrul R. <u>Ex.</u> 5 7 2 6 are ca rezultat adunarea valorilor din registrii 2 și 6 și plasarea rezultatului în registrul 7.
6	R S T	Adună (ADD) șirurile de biți din registrii S și T ca și cum ar fi reprezentați în virgulă mobilă și depune rezultatul în registrul R. <u>Ex.</u> 6 3 4 E are ca rezultat adunarea valorilor din registrii 4 și E ca valori reprezentate în virgulă mobilă și plasarea rezultatului în registrul 3.

C o d operație	Operand	D e s c r i e r e
7	R S T	Execută un <i>sau</i> logic (OR) între șirurile de biți din registrii S și T și depune rezultatul în registrul R. <u>Ex.</u> 7 C B 4 are ca rezultat executarea unui <i>sau</i> logic între conținutul registrilor B și 4 și plasarea rezultatului în registrul C.
8	R S T	Execută un <i>și</i> logic (AND) între șirurile de biți din registrii S și T și depune rezultatul în registrul R. <u>Ex.</u> 8 0 4 5 are ca rezultat executarea unui <i>și</i> logic între conținuturile registrilor 4 și 5 și plasarea rezultatului în registrul 0.
9	R S T	Execută un <i>sau</i> exclusiv (XOR) între șirurile de biți din registrii S și T și depune rezultatul în registrul R. <u>Ex.</u> 9 5 F 3 are ca rezultat executarea unui <i>sau</i> exclusiv între conținutul registrilor F și 3 și plasarea rezultatului în registrul 5.
A	R O X	Rotește (ROTATE) șirul de biți din registrul R cu un bit la dreapta de X ori. La fiecare rotație, bitul cel mai puțin semnificativ este mutat pe poziția bitului cel mai semnificativ. <u>Ex.</u> A 4 0 3 are ca rezultat mutarea circulară la dreapta a conținutului registrului 4 de 3 ori.
B	R X Y	Salt (JUMP) la instrucțiunea plasată în celula de memorie cu adresa X Y dacă conținutul registrului R este egal cu conținutul registrului 0. Altfel se continuă execuția normală a secvenței de instrucțiuni. <u>Ex.</u> B 4 3 C se compară mai întâi conținutul registrului 4 cu conținutul registrului 0. Dacă cei doi registri sunt identici, secvența de execuție va fi modificată astfel încât următoarea instrucțiune care se va executa să fie cea aflată la adresa de memorie 3 C. Altfel, execuția progra-mului va continua în mod normal.
C	0 0 0	Oprirea (HALT) execuției. <u>Ex.</u> C 0 0 0 are ca rezultat oprirea execuției programului.

TEST AUTOEVALUARE 2 (Manipularea datelor)

1. Precizati care este ordinea corecta a celor trei pasi in care se executa o instructiune :
 1. extragere, decodificare, executie
 2. decodificare, extragere, executie
 3. extragere, executie, recodificare
2. Care din afirmatiile despre instructiunile de transfer de date este adevarata :
 1. manipuleaza date
 2. dirijeaza modul de executie a programelor
 3. realizeaza operatii de intrare / iesire
3. Se da urmatoarea secventa de program (adunarea a doua valori stocate in memorie)

Pasul 1	Se incarca un registru cu o valoare din memorie.	
Pasul 2	Se incarca alt registru cu o alta valoare din memorie.	
Pasul 3	Daca a doua valoare este zero salt la pasul 6	
Pasul 4	Se imparte continutul primului registru la continutul celui de al doilea registru si se depune rezultatul in al treilea registru	
Pasul 5	Se stocheaza continutul celui de al treilea registru	
Pasul 6	Stop.	

Precizati pentru fiecare pas, in coloana treia, categoria/tipul instructiunii implicate.

4. Care dintre următoarele afirmații despre prelucrarea interactivă (interactive processing) este falsă:
1. nu permite executarea programelor ce poartă un dialog cu utilizatorul
 2. prelucrarea interactivă permite prelucrarea în timp real (real time processing)
 3. lucrările stocate în vederea execuției în memoria de masă sunt deservite după principiul FIFO (primul intrat, primul ieșit)
5. Care din afirmațiile despre instrucțiunile de transfer de date este falsă :
1. manipulează date
 2. dirijează modul de execuție a programelor
 3. realizează operații de intrare / ieșire, respectiv in / din calculator

UNITATEA DE ÎNVĂȚARE 3

3. Sistemele de operare

3.1. Evoluția sistemelor de operare

Sisteme cu un singur procesor

Pentru primele sisteme de operare s-a acționat asupra simplificării încărcării programelor și reducerea perioadei de tranziție dintre lucrări. A fost stabilit un singur operator care efectua toate operațiile cu calculatorul.

Operatorul încărca toate materialele pe suportul de stocare de masă al calculatorului, unde sistemul de operare avea acces la ele pentru a le executa. Acesta a fost începutul **prelucrării pe loturi** (*batch procession*) - executarea lucrărilor prin plasarea lor într-un grup unic și apoi reluarea lor fără o altă interacțiune cu utilizatorul. În vederea execuției, lucrările stocate în memoria de masă erau plasate într-o **coadă de lucrări** (*job queue*) (fig. 3.1).

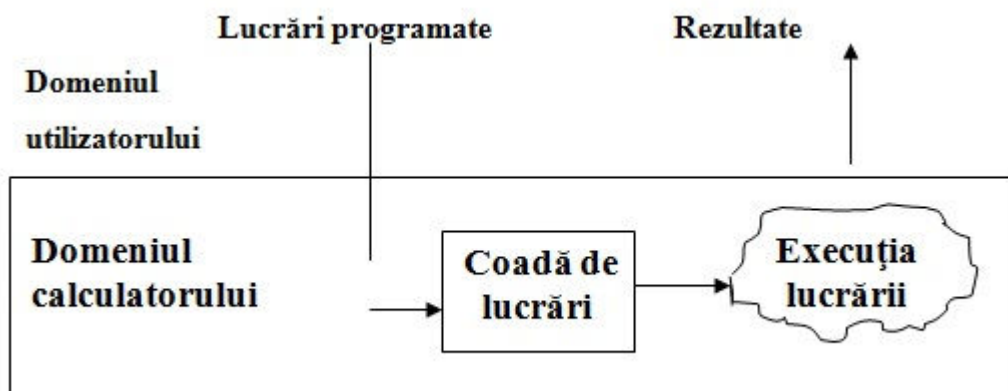


Figura 3.1 - Prelucrare pe loturi

Coadă reprezintă o structură de stocare în care obiectele (lucrările) sunt organizate după principiul primul intrat, primul ieșit (FIFO – First Input First Output).

Un dezavantaj al prelucrării utilizând administrator de sistem este acela că, după ce lucrarea a fost transmisă în coada de lucrări, utilizatorul nu mai poate interveni asupra programului.

Pentru a răspunde la aceste cerințe au fost dezvoltate noi sisteme de operare care permit **prelucrarea interactivă** (*interactive processing*) (fig. 3.2.).

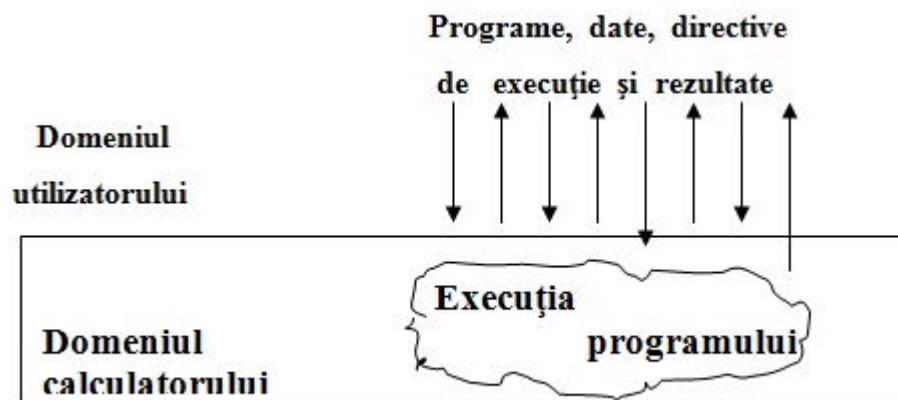


Figura 3.2 - Prelucrarea interactivă

Aceste sisteme permit executarea unui program care poartă un dialog cu utilizatorul prin intermediul terminalelor de control la distanță sau al stațiilor de lucru. Sistemele interactive au dat naștere conceptului de **prelucrare în timp real** (*real time processing*).

Dacă sistemele interactive ar fi putut să se adreseze unui singur utilizator la un moment dat, prelucrarea în timp real n-ar fi pus nici o problemă. Datorită prețului ridicat al calculatoarelor este necesar ca fiecare mașină să deservească mai mulți utilizatori.

O soluție la această problemă este proiectarea și realizarea sistemului de operare în așa fel încât să parcurgă pe rând diferitele activități ce trebuie executate printr-un proces numit **partajarea timpului**.

Mai exact, **partajarea timpului** (*time sharing*) se referă la tehnica de împărțire a timpului în intervale, denumite **felii de timp** (*time slices*), cu restricția executării unei activități numai într-o felie de timp la un moment dat. La sfârșitul fiecărei felii de timp, activitatea curentă este trecută în repaus și o altă activitate primește dreptul de execuție în următoarea felie de timp. Prin baleierea rapidă a activităților în acest mod se creează iluzia executării simultane a mai multor

procese. În prezent partajarea timpului este utilizată atât în sistemele cu un singur procesor, cât și în sistemele multiprocesor, dar în cazul primelor este denumită de obicei multitasking (iluzia că mai multe activități sunt desfășurate simultan).

Sisteme multiprocesor

Nevoia de a partaja informațiile și resursele între diferite calculatoare a condus la ideea conectării calculatoarelor pentru schimbul de informații.

A apărut conceptul de structură de mai multe calculatoare mici, conectate într-o **rețea** (*net work*), prin care utilizatorii partajează resursele. **Software**-ul pentru controlul unei rețele de calculatoare poate fi privit ca un sistem de operare în rețea.

3.2. Arhitectura unui sistem de operare

Pentru a înțelege un sistem de operare tipic, propunem o clasificare a categoriilor de componente.

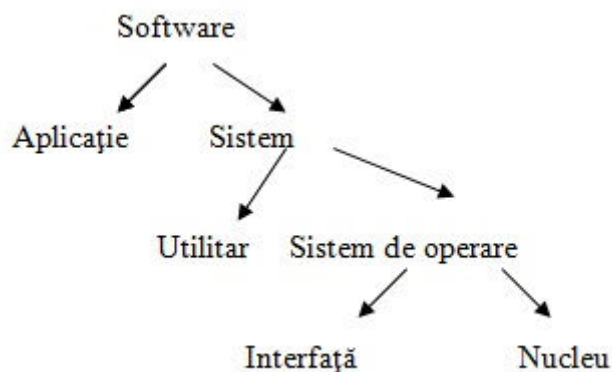


Figura 3.3 – Clasificarea *software*-ului

Există două categorii de *software* distincte : **software de aplicații** (*application software*) și **software de sistem** (*system software*). **Software-ul de aplicații** conține programele care efectuează anumite activități particulare specifice beneficiarului (*end user*-ului).

Spre deosebire de software-ul de aplicații, **software-ul de sistem** efectuează acele activități care sunt comune sistemelor de calcul în general.

Clasa software-ului de sistem se împarte în două categorii: sistemul de operare propriu-zis și module software numite software utilitar. Într-un anumit sens, software-ul utilitar constă în

unități de software care extind caracteristicile sistemului de operare (abilitatea de formatare a unui disc sau de copiere a unui fișier).

Unii utilizatori de calculatoare includ în clasa de software utilitar orice software livrat odată cu sistemul de operare.

Interfața

Partea dintr-un sistem de operare care definește modul de interacțiune dintre sistemul de operare și utilizatorii săi poartă numele de **interfață** (*shell*). Sarcina interfeței este aceea de a permite comunicarea cu utilizatorul (sau utilizatorii) calculatorului.

Interfețele moderne realizează acest lucru folosind o **interfeța grafică cu utilizatorul** (*graphical user interface - G U I*), în care obiectele manipulate (fișiere și programe) sunt reprezentate grafic pe ecran prin **pictograme** (*incons*). Astfel de sisteme permit utilizatorului să execute comenzi prin selectarea și deplasarea pictogramelor pe ecran cu un dispozitiv denumit "mouse". Vechile interfețe comunicau cu utilizatorul numai prin mesaje de tip text (utilizând tastatura și ecranul). Interfața realizează legătura între un utilizator și "inima" sistemului de operare (vezi fig. 3.4).

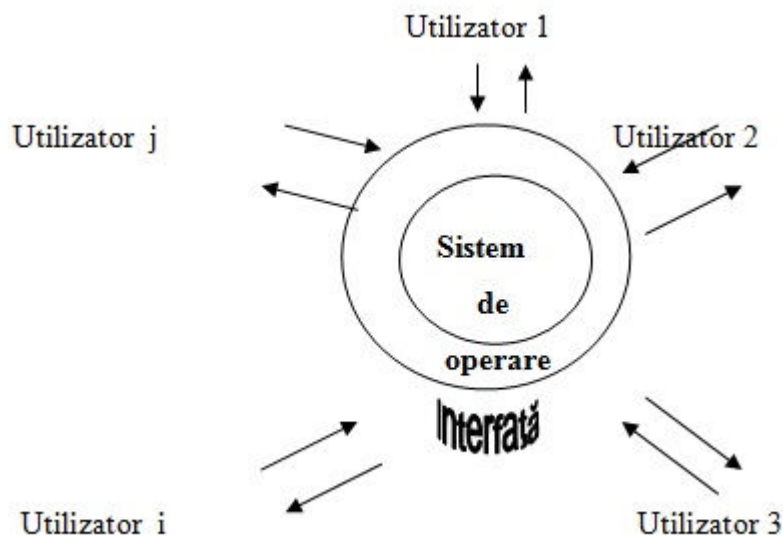


Figura 3.4 - Interfața dintre utilizatori și sistemul de operare

Nucleul

Partea din interiorul unui sistem de operare este adesea denumită **nucleu** (*kernel*). Nucleul conține acele componente software care efectuează operațiile primare necesare pentru funcționarea calculatorului.

Una din aceste componente este **administratorul de fișiere** (*file manager*), având sarcina să coordoneze utilizarea facilităților oferite de memoria masă a calculatorului.

Pentru simplificarea utilizării calculatorului, sistemele de administrare a fișierului permit gruparea lor în unități denumite **directoare** (*directory*) sau **dosare** (*folder*). Astfel este posibil ca utilizatorul să-și organizeze fișierele conform scopului propus, permițând ca directoarele să conțină alte directoare, denumite **subdirectoare**, realizându-se astfel o organizare ierarhizată a informațiilor. Secvența de directoare care indică drumul până la un anumit subdirector sau fișier se numește **cale** (*path*).

Orice acces la un fișier se obține prin intermediul administratorului de fișiere, care solicită deschiderea fișierului. Dacă administratorul de fișiere acceptă cererea de acces, el furnizează informațiile necesare pentru găsirea și manipularea fișierului. Informațiile sunt stocate într-o zonă din memoria principală care poartă numele de **descriptor de fișier** (*file descriptor*).

O altă componentă a nucleului constă dintr-o colecție de "drivere" de dispozitiv (*device drivers*) - module software care comunică cu controlerele (sau uneori direct cu dispozitivele periferice) pentru efectuarea operațiilor de către periferice.

Fiecare "driver" de dispozitiv este proiectat în mod individual pentru un anumit tip de controler sau dispozitiv (imprimantă, disc, monitor...) și traduce cererile formulate în termeni generali într-o secvență de instrucțiuni specifice controler-ului sau dispozitivului atașat celui driver.

Altă componentă a nucleului sistemului de operare este administratorul de memorie (*memory manager*), însărcinat cu activitatea de coordonare a utilizării memoriei principale a calculatorului. Această componentă este foarte utilizată în mediile multiutilizator sau multitasking.

Sarcina administratorului de memorie se complică atunci când cantitatea totală de memorie solicitată depășește dimensiunea memoriei disponibile. În acest caz, administratorul de memorie poate crea iluzia unui spațiu suplimentar de memorie rotind programele și datele între

memoria principală și disc. Această memorie iluzorie este denumită **memorie virtuală** (*virtual memory*).

Pornirea calculatorului

Lansarea în execuție a sistemului de operare se realizează prin intermediul unei proceduri cunoscute sub numele de **încărcarea sistemului de operare** (*boot strap*) ; **încărcare** (*booting*).

Zona de memorie în care se așteaptă să se găsească programul pentru execuție se numește **memorie permanentă** (*read-only memory — R O M*). Cea mai mare parte din memoria internă a unui calculator de uz general este memoria volatilă, conținutul memoriei pierzându-se la oprirea calculatorului.

În scopul încărcării sistemului de operare al unui calculator de uz general, memoria ROM conține un program de mici dimensiuni, denumit **bootstrap**. La pornirea calculatorului acest program se execută automat și în multe cazuri obiectul transferului (de pe suportul de stocare de masă în memoria principală a calculatorului - vezi fig. 3.5) este chiar sistemul de operare. După plasarea în memorie a sistemului de operare, programul *bootstrap* instruește VC să sară la zona de memorie care-l conține.

Din acest moment sistemul de operare devine activ și preia controlul calculatorului.

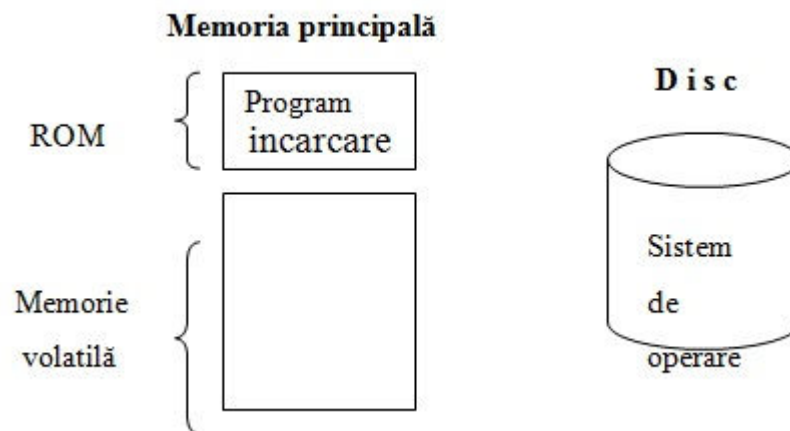


Figura 3.5.a - Procesul de încărcare a sistemului de operare

(**Pasul 1**) Calculatorul execută programul de încărcare aflat în memorie. Sistemul de operare se află pe dispozitivul de stocare de masă.

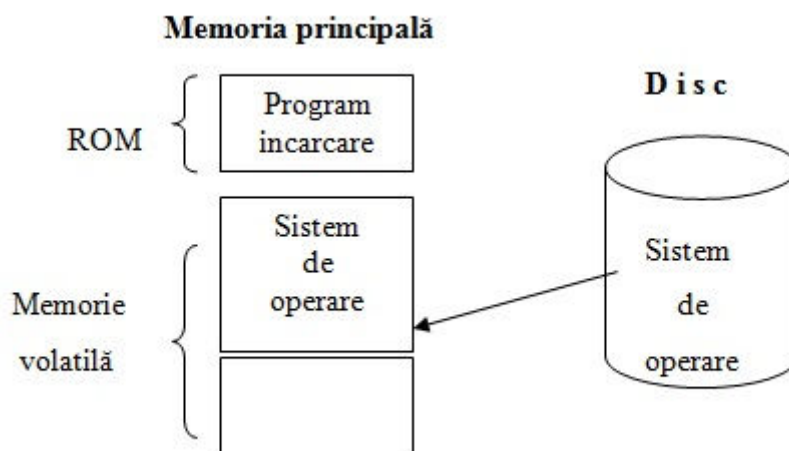


Figura 3.5.b. - Procesul de încărcare a sistemului de operare

(**Pasul 2**) Programul de încărcare realizează transferul sistemului de operare în memoria principală și îi cedează apoi controlul.

3.3. Coordonarea activităților desfășurate de calculator

În continuare se va prezenta modul în care sistemul de operare coordonează execuția *software*-ului de aplicație și utilitar, cât și pe cea a propriilor sale module.

Conceptul de proces

Unul din cele mai importante concepte, în cadrul sistemelor de operare, este deosebirea dintre un program și acțiunea de execuție a acestuia.

Programul reprezintă un set static de directive, iar execuția lui este o activitate dinamică, ale cărei proprietăți se modifică în timp. Această activitate poartă numele de **proces**. Procesul este caracterizat de starea procesului. Starea procesului reprezintă un instantaneu al functionării calculatorului la un moment dat. Un singur program poate fi asociat în același timp mai multor procese. (Ex.: Doi utilizatori pot edita simultan documente diferite, într-un sistem multiutilizator cu partajarea timpului, sistemul de operare utilizând o singură copie a programului de editare).

Sarcina sistemului de operare este să coordoneze mai multe procese care concurează pentru utilizarea feliilor de timp.

Coordonarea implică alocarea resurselor necesare fiecărui proces (dispozitive periferice, spațiu în memoria principală, acces la date și acces la unitatea centrală de prelucrare), împiedicarea interferenței proceselor independente și asigurarea schimbului de informații între procese care trebuie să realizeze acest lucru. Pentru acest tip de comunicație se folosește numele de **comunicație între procese** (*interprocess communication*).

Administrarea proceselor

Operațiile asociate coordonării proceselor sunt efectuate de către **secvențiator** (*scheduler*) și **executor** (*dispatcher*), din nucleul sistemului de operare.

Secvențiatorul memorează o înregistrare a proceselor prezente în sistemul de calcul, introduce noi procese și le elimină pe cele care s-au terminat. Pentru a putea urmări toate procesele, secvențiatorul le înregistrează într-un bloc de informații denumit **tabel de procese** (*process table*), în memoria principală. În acest tabel sunt memorate informații, ca: zona de memorie alocată procesului, prioritatea procesului, dacă procesul este în așteptare etc.

Executorul este componenta nucleului care asigură execuția proceselor active, programate de secvențiator.

Într-un sistem cu partajarea timpului, execuția programelor se realizează prin împartirea timpului în intervale scurte, fiecare purtând numele de **feliu de timp** (*time slice*) și având o durată de cca 50 de milisecunde. Procedura de trecere de la un proces la altul poartă numele de **comutare între procese** (*process switch*).

La primirea unui semnal de întrerupere, unitatea centrală de prelucrare completează ciclul curent de extragere-decodificare- execuție salvează poziția din procesul curent și începe execuția unui **program de tratare a întreruperilor** (*interrupt handler*), care este stocat la o locație predeterminată din memoria principală. Programul de tratare a întreruperilor este o componentă a executorului. Efectul semnalului de întrerupere este de suspendare a procesului curent și de transfer a controlului către executor.

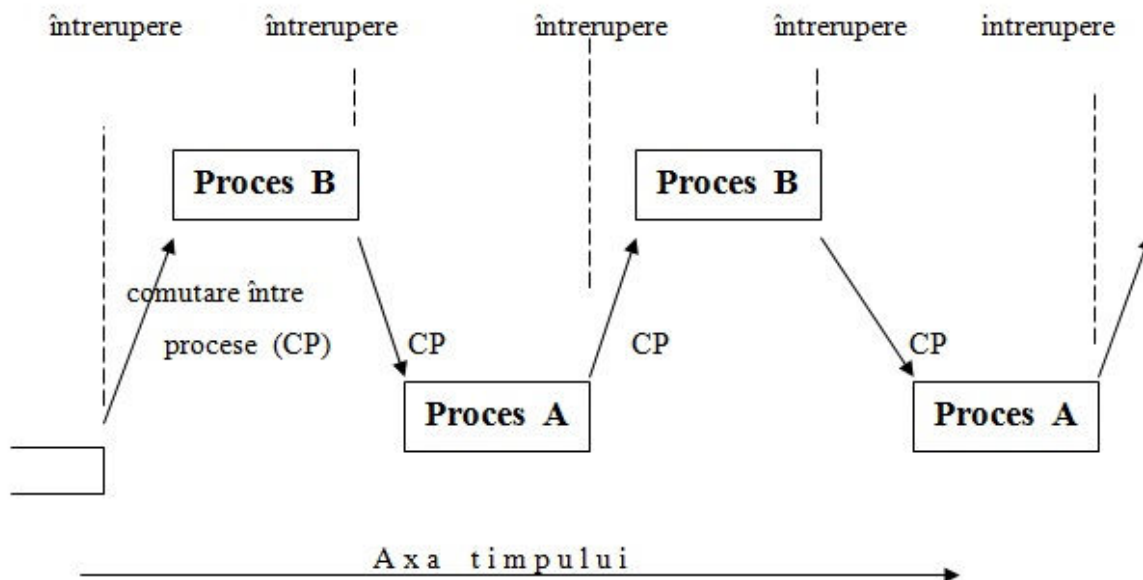


Figura 3.6 - Partajarea timpului între procesele A și B

În acest punct executorul permite secvențiatorului să actualizeze tabelul de procese, apoi executorul selectează procesul care are cea mai mare prioritate dintre procesele gata de continuare din tabel și permite procesului selectat să-și înceapă „felia” de timp.

Abilitatea de oprire și de repornire ulterioară a unui proces este de importanță vitală pentru succesul unui sistem cu partajarea timpului. Calculatoarele proiectate pentru sisteme de operare cu partajarea timpului includ acțiunea de salvare a valorii contorului de program, precum și conținutul regiștrilor și al celulelor de memorie asociate, ca parte a reacției unității centrale de prelucrare la semnalul de întrerupere. Ele dispun de obicei de instrucțiuni de limbaj mașină pentru reîncărcarea unei stări salvate anterior.

Modelul client/server

Modulele sistemului de operare (într-un sistem cu partajarea timpului) concurează între ele sub controlul executorului pentru „feliile” (cuantele) de timp.

Pentru a dobândi accesul la un fișier aflat pe un dispozitiv de stocare de masă, orice proces trebuie să obțină mai întâi informațiile necesare de la administratorul de fișiere.

Pentru simplificarea comunicației între procese, componentele unui sistem de operare sunt adesea proiectate în conformitate cu modelul **client/server**:

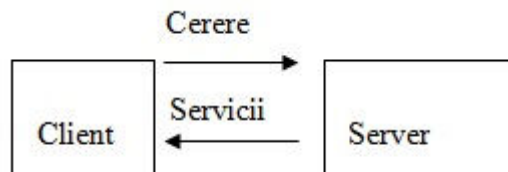


Figura 3.7 - Modelul client/server

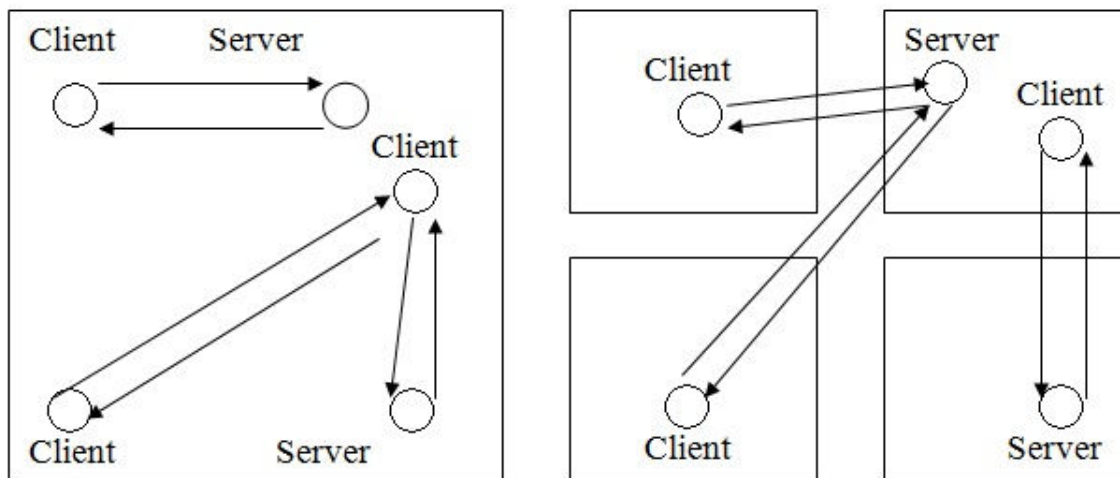


Figura 3.8 - Structuri de comunicație între clienți și servere, care operează pe un calculator sau sunt distribuite pe mai multe calculatoare

Acest model definește cele două roluri fundamentale pe care le pot juca diferitele componente: **client** (emite cereri către unități, respectiv **server** (satisfacă cererile emise de clienți). Aplicarea modelului **client/server** în proiectarea *software*-ului conduce la uniformizarea tipurilor de comunicații care au loc în sistem.

Dacă componentele unui sistem de operare sunt proiectat ca servere și clienți, comunicația între componentele din cadrul aceluiași calculator sau între componente ale unor calculatoare aflate la mare distanță unele de altele - figura 3.8., are aceeași formă.

Atât timp cât o rețea de calculatoare permite trimiterea de cereri și răspunsuri între calculatoare, mai mulți clienți și mai multe servere se pot distribui în orice configurație convenabilă, pentru rețeaua respectivă.

3.4. Gestionarea proceselor concurente

Componentele nucleului unui sistem de operare se ocupă, în principal, cu alocarea resurselor calculatorului către procesele ce se desfășoară în sistem. Atribuim, în acest caz, termenului **resurse** atât dispozitivele periferice ale calculatorului, cât și funcțiile de care dispune calculatorul. (Ex.: Administratorul de fișiere alocă atât accesul la fișierele existente, cât și spațiul pe disc pentru crearea de noi fișiere).

Deoarece un calculator „nu gândește” independent, ci doar execută instrucțiuni, pentru ca sistemul de operare să funcționeze fiabil, s-au dezvoltat algoritmi care să acopere orice problemă identificată ca posibilă.

Semafoare

Să luăm în discuție un sistem de operare cu partajarea timpului și la care este conectată o singură imprimantă. Dacă un proces este în situația de a-și tipări rezultatele, el trebuie să solicite sistemului de operare accesul la programul *driver* al imprimantei. În acest moment, sistemul de operare trebuie „să decidă” dacă satisface această cerere, verificând dacă imprimanta nu este cumva utilizată de alt proces. Dacă imprimanta este liberă, sistemul de operare trebuie să acorde permisiunea utilizării ei și să permită procesului să continue.

Dacă două procese ar căpăta simultan acces la imprimantă, rezultatul n-ar prezenta nici o utilitate pentru nici unul dintre ele. Soluția ar reprezenta-o utilizarea unui indicator (*flag*) – un bit în memoria ale cărei stări sunt: 1 (**setat**) și 0 (**șters**).

Indicatorul **șters** arată că imprimanta este liberă, iar **setat** indică faptul că imprimanta este deja alocată. Deși această soluție pare bună la prima vedere, există totuși o problemă. Operația de testare și eventual de setare a indicatorului necesită mai mulți pași-mașină. Nu este exclus ca operația să fie întreruptă după detectarea unui indicator nul, dar înainte ca indicatorul să fie setat, ceea ce face posibilă apariția situației următoare:

Presupunem că imprimanta este disponibilă și un proces cere utilizarea sa. Indicatorul care corespunde imprimantei este verificat și găsit ca fiind șters, ceea ce indică că imprimanta este disponibilă. Dar în acel moment procesul este întrerupt și alt proces își începe felia de timp, și acest nou proces solicită utilizarea imprimantei. Indicatorul imprimantei este verificat din nou și găsit tot șters, deoarece procesul precedent a fost întrerupt înainte de a-l putea seta. În consecință,

sistemul de operare permite celui de al doilea proces să înceapă utilizarea imprimantei. Se ajunge deci la o situație în care două procese „utilizează” simultan aceeași imprimantă.

Problema este că operația de testare și (eventual) de setare a indicatorului trebuie să fie terminată fără a fi întreruptă prin utilizarea instrucțiunilor de invalidare și validare a întreruperilor, disponibile în limbajele-mașină.

O altă metodă o reprezintă utilizarea **instrucțiunii de testare și setare** (*test-and-set*), care este disponibilă în multe limbaje-mașină. Unității centrale de prelucrare i se cere să citească valoarea unui indicator, să o memoreze și apoi să seteze indicatorul, toate acestea printr-o singură instrucțiune-mașină. Avantajul acestei metode este că, de vreme ce unitatea centrală termină întotdeauna de executat o instrucțiune înainte de a sesiza o întrerupere, operația de testare și setare a indicatorului, implementată cu o singură instrucțiune, nu poate fi întreruptă.

Un indicator implementat corect, ca mai sus, poartă numele de **semafor** (*semaphore*). Prin similitudine o secvență de instrucțiuni care trebuie executată fără întreruperi corespunde unei porțiuni de cale ferată pe care poate trece, la un moment dat, un singur tren. O asemenea secvență de instrucțiuni poartă numele de **zonă critică** (*critical region*). Înainte de a intra într-o zonă critică, un proces trebuie să găsească noul semafor asociat ei și să-l seteze, apoi trebuie să șteargă semaforul după ieșirea din zona critică.

Interblocarea

O altă problemă care poate apărea în timpul alocării resurselor este **interblocarea** (*deadlock*) — situația în care desfășurarea a două sau mai multe procese este blocată, deoarece fiecare dintre ele așteaptă acces la resursele alocate celuilalt.

Un alt exemplu apare în situația în care procesele creează noi procese pentru efectuarea unor operații mai simple. Dacă secvențiatorul nu are destul spațiu în tabelul de procese și fiecare proces din sistem trebuie să creeze un proces suplimentar înainte de a-și termina treaba, atunci nici un proces nu va putea continua.

La o analiză mai atentă, constatăm că poate să apară o interblocare numai dacă sunt satisfăcute simultan următoarele trei condiții:

1. Există o competiție pentru resurse care nu pot fi partajate;
2. Resursele sunt solicitate în mod parțial, ceea ce înseamnă că, dispunând de anumite resurse, un proces va reveni ulterior solicitând mai multe;

3. O dată ce o resursă a fost alocată, nu se poate forța eliberarea sa.

Concluzia care rezultă este că eliminarea interblocării se poate face prin eliminarea uneia dintre cele trei condiții de mai sus.

Așa cum am văzut, pentru ca un calculator să poată îndeplini o anumită sarcină, trebuie să-i furnizăm un algoritm care să-i spună exact ce are de făcut. În acest sens, în continuare vom prezenta câteva concepte fundamentale, ca:

- dezvoltarea și reprezentarea algoritmilor
- controlul iterativității și recursivității algoritmilor.

De asemenea vor fi descriși în continuare câțiva algoritmi foarte cunoscuți de căutare și sortare.

TEST AUTOEVALUARE 3 (Sisteme de operare)

1. Un *job queue* poate fi definit ca:
 - a. locul unde lucrările sunt stocate în memoria de masă
 - b. executarea lucrărilor prin plasarea lor într-un grup unic
 - c. o coadă de lucrări
2. O coadă poate reprezenta:
 - a. O structură de stocare
 - b. O coadă de lucrări (job queue)
 - c. Primul intrat, primul ieșit (FIFO)
3. Principiul după care o coadă poate funcționa, este:
 - a. First Input First Output (FIFO)
 - b. Last Input First Output (LIFO)
 - c. First Input Last Out (FILO)
 - d. First Come First Served (FCFS)
 - e. Garbage In Garbage Out (GIGO)
4. Partajarea timpului (time sharing) se refera la tehnica de:
 - a. felii de timp (time slices)
 - b. Împărțirea timpului în interval
 - c. Executare a unei activități într-o felie de timp la un moment dat
5. Partajarea timpului (time sharing) este utilizată în:
 - a. Programarea algoritmilor și stabilirea gradului de complexitate
 - b. Sistemele cu un singur procesor
 - c. Sistemele multiprocesor
6. Arhitectura unui sistem de operare este compusă din:
 - a. Software
 - b. Aplicație
 - c. Interfață
 - d. Algoritmi

7. Software-ul de system se împarte în:
 - a. Sistem de aplicații pentru sistemul de operare
 - b. Sistemul de operare propriu-zis
 - c. Module software
 - d. Software utilitar
8. Interfața unui sistem de operare definește:
 - a. Modul de interacțiune dintre sistemul de operare și utilizatorii săi
 - b. Comunicarea cu procesele unui sistem de operare
 - c. Comunicarea cu utilizatorul
9. Partea din interiorul unui sistem de operare poartă numele de:
 - a. Nucleu (kernel)
 - b. Director (folder)
 - c. Cale (path)
10. Secvența de directoare indică:
 - a. Poziția unde mă aflu pe hard-disk
 - b. Drumul până la un anumit subdirector sau fișier
 - c. Operațiile primare necesare pentru funcționarea calculatorului
11. O componentă a nucleului sistemului de operare este:
 - a. Administratorul de memorie (memoriz manager)
 - b. Administratorul de procese
 - c. Driver
12. Memoria iluzorie a unui sistem de operare:
 - a. Memorie fizică
 - b. Memoria virtuală
 - c. Memoria RAM
13. *Boot straping* se referă la:
 - a. Zona de memorie în care este încărcat sistemul de operare
 - b. Lansarea în execuție a sistemului de operare
 - c. Memoria permanentă

14. Memoria în care se găsește programul pentru execuție este:
- a. ROM (Read-only memory)
 - b. RAM (Random Access Memory)
 - c. HDD (Hard Disk Drive)
15. Un program reprezintă:
- a. Un set static de directive
 - b. O listă de instrucțiuni
 - c. O activitate dinamică
16. Sarcina unui sistem de operare este să coordoneze:
- a. Un proces sau mai multe procese
 - b. Fișierele de pe hard disk
 - c. Alocarea resurselor necesare
17. Coordonarea proceselor sunt efectuate de :
- a. Secvențiator (scheduler)
 - b. Executor (dispatcher)
 - c. Interprocess communication
18. Secvențiatorul memorează (scheduler) se referă la:
- a. Memorarea înregistrării proceselor prezente în sistemul de calcul
 - b. Eliminarea proceselor ce se află în memorie
 - c. Creerea de noi procese ce sunt memorate în managerul de procese al sistemului de operare
19. Procesele active sunt programate de:
- a. Secvențiator
 - b. Executor
 - c. Managerul de task-uri
20. Executorul (dispatcher) se referă la:
- a. Execuția proceselor active
 - b. Este component a nucleului
 - c. Programarea secvențială a proceselor
21. Definiți modelul client/server prin precizarea principalelor componente importante precum și rolul legăturii dintre ele.

22. Modelul client/server se referă la:

- a. Proiectarea software-ului
- b. Proiectarea echipamentelor hardware necesare rulării unui software
- c. Comunicația dintre procesele unui sistem de operare în vederea rulării unui software în parametrii optima

23. Termenul de resurse se referă la:

- a. Dispositive periferice
- b. Administratorul de fișiere
- c. Procesele calculatorului

24. Definiți conceptul de semafor si exemplificați funcționarea acestuia printr-un exemplu.

25. Interblocarea poate apărea în condițiile:

- a. Există o competiție pentru resurse care nu pot fi partajate
- b. O data ce o resursă a fost alocată, nu se poate forța eliberarea sa
- c. Resursele sunt solicitate în mod parțial, ceea ce înseamnă că, dispunând de anumite resurse, un process va reveni ulterior

UNITATEA DE ÎNVĂȚARE 4

4. Algoritmii

4.1. Conceptul de algoritm

Algoritmul este abstract și trebuie diferențiat de reprezentarea sa. În acest context, al distincției dintre algoritm și reprezentarea sa, se poate lămuri și diferența dintre două concepte înrudite - program și proces. Programul este reprezentarea unui algoritm (reprezentarea formală a unui algoritm proiectat în vederea unei aplicații informatice).

Procesul este definit ca fiind activitatea de execuție a programului, dar putem defini procesul ca activitatea de execuție a unui algoritm. Deci, programele, procesele și algoritmii sunt entități înrudite, dar distincte.

Un algoritm constă dintr-o mulțime ordonată de pași executabili, descriși fără echivoc, care definesc un proces finit.

Analizând definiția algoritmului, ne vom opri asupra cerinței ca pașii care descriu algoritmul să fie ordonați, deci trebuie să aibe o ordine clară în execuția pașilor. Există și algoritmi paraleli care conțin mai multe secvențe de pași, fiecare secvență urmând să fie executată de alt procesor în cadrul unei mașini multiprocesor (deci nu este vorba numai de un fir de execuție).

În continuare vom analiza cerința ca algoritmul să fie compus din pași executabili. Fie instrucțiunile:

Pasul 1: *Construiți o listă cu toate numerele întregii pozitive.*

Pasul 2: *Sortați lista în ordine descrescătoare.*

Pasul 3: *Extrageți primul număr întreg din lista rezultată.*

Aceste instrucțiuni nu descriu un algoritm, deoarece pasul 1 și 2 sunt imposibil de executat (nu se poate construi o listă cu toți întregii pozitivi și întregii pozitivi nu pot fi așezați în ordine descrescătoare).

Altă cerință, descrierea fără echivoc a pașilor algoritmului înseamnă ca, în timpul execuției programului, informațiile din fiecare stare a procesului trebuie să fie suficiente pentru a determina unic și complet acțiunile în fiecare pas.

Execuția algoritmului nu necesită creativitate, ci doar capacitatea de a urma instrucțiunile. Descrierea algoritmului ca un proces finit înseamnă că execuția algoritmului trebuie să aibe sfârșit. Trebuie realizată o delimitare între procesele care se termină cu un răspuns- rezultat și cele care se execută la infinit, fără a ajunge la vreun rezultat.

Termenul de algoritm este folosit adesea într-o accepțiune mai puțin riguroasă pentru a descrie procese care nu sunt neapărat finite. (Ex. Algoritm de împărțire fără rest a numerelor întregi, împărțirea lui 1 la 3).

4.2. Reprezentarea algoritmilor

Primitive

Reprezentarea unui algoritm nu se poate face în absența unui limbaj. În cazul oamenilor acest limbaj poate fi limbajul natural uzual sau limbajul imaginilor. Aceste canale naturale de comunicare duc la neînțelegeri, termenii folosiți pot avea mai multe sensuri.

Informatica abordează aceste probleme stabilind o mulțime bine definită de blocuri elementare care stau la baza reprezentării algoritmilor. Aceste blocuri elementare poartă numele de *primitive*. Definirea precisă a primitivelor elimină ambiguitatea și permite un grad uniform de detaliere.

O mulțime de reguli care guvernează modul în care aceste primitive pot fi combinate pentru reprezentarea ideilor mai complexe constituie un **limbaj de programare**. Fiecare primitivă constă din două elemente: sintaxa și semantica. *Sintaxa* se referă la reprezentarea simbolică a primitivei, în timp ce *semantica* se referă la conceptul reprezentat (semnificația primitivei respective).

Pentru a reprezenta algoritmi destinați calculatorului utilizând un ansamblu de primitive, se poate porni de la instrucțiunile pe care calculatorul știe să le execute. Detalierea algoritmilor la acest nivel este incomodă, în mod normal se utilizează primitive de nivel mai înalt, construite pe baza primitivelor de nivel scăzut oferite de limbajul-mașină. Rezultă astfel un limbaj de programare formal, în care algoritmi pot fi exprimați într-o formă conceptual mai înaltă decât în limbajul-mașină.

Pseudocodul

Vom prezenta în continuare un sistem de notare mai intuitiv, cunoscut sub numele de **pseudocod**. O metodă de obținere a pseudocodului este de a relaxa pur și simplu regulile limbajului formal în care va fi exprimată varianta finală a algoritmului.

Ceea ce ne propunem este să discutăm despre dezvoltarea și prezentarea algoritmilor fără să ne oprim asupra unui limbaj de programare. Ne propunem să reprezentăm anumite structuri care pot deveni primitive. Printre structurile mai des întâlnite remarcăm:

1. Selectarea între două activități în funcție de îndeplinirea unei condiții. Fiecare dintre aceste instrucțiuni poate fi rescrisă astfel încât să respecte structura:

if(condiție) *then* (activitate)
else (activitate)

, unde am folosit cuvintele-cheie *if* (dacă), *then* (atunci) *else* (astfel) pentru a introduce substructurile din cadrul structurii principale și paranteze pentru delimitarea granițelor dintre aceste substructuri.

Adoptând pentru **pseudocod** această structură sintactică, am obținut o metodă uniformă de exprimare a unei structuri semantice des întâlnite. În cazurile în care activitatea alternativă (*else*) lipsește, vom utiliza sintaxa mai scurtă:

if (condiție) *then* (activitate).

2. altă structură algoritmică [întâlnită] este executarea repetată a unei instrucțiuni sau a unei secvențe de instrucțiuni atâta timp cât o anumită condiție este adevărată. Modelul pseudocodului este:

while (condiție) *do* (activitate).

O astfel de instrucțiune înseamnă verificarea condiției, executarea activității și revenirea la verificarea condiției. Când condiția nu mai este îndeplinită, se trece la executarea instrucțiunii care urmează. Uneori ne referim la anumite valori prin intermediul unor nume sugestive. Pentru astfel de asocieri se utilizează:

assign (nume) *the value* (expresie)

, unde nume este un nume descriptiv, iar expresie indică valoarea care este asociată numelui respectiv.

Lizibilitatea programului poate fi îmbunătățită prin indentare (adică scriere poziționată pe condiții logice).

Exemplu: *if* (articolul este impozabil)

then [*if* (preț > limită)

then (plătește *x*)

else (plătește *y*)]

else (plătește *z*).

Vom folosi pseudocodul pentru a descrie activități care să fie utilizate ca instrumente abstracte în alte aplicații, aceste unități de program le vom numi în continuare **procedură** și vom folosi cuvântul *procedure* pentru atribuirea fiecărui modul de pseudocod un nume.

procedure *nume*.

Această instrucțiune introductivă va fi urmată de instrucțiunile care descriu acțiunea modulului respectiv. În figura 4.1 este prezentat pseudocodul unei proceduri numită **Salutări**, care tipărește de trei ori mesajul „Hello”:

procedure Salutări

assign Count the value 3

while Count 0 ***do***

(tipărește mesajul „Hello” și

assign Count the value Count 1).

Figura 4.1 - Procedura Salutări exprimată în pseudocod

Atunci când este nevoie de efectuarea procedurii altundeva în cadrul secvenței pseudocodului, procedura va fi apelată prin nume. Procedurile trebuie să fie cât mai generale posibil. Astfel o procedură care sortează orice listă de nume, trebuie să fie scrisă în așa fel încât lista respectivă să nu fie specifică procedurii, ci să fie desemnată în reprezentarea acesteia sub un nume generic. Vom adopta convenția de a scrie aceste nume generice între paranteze pe același rând cu numele procedurii.

(Ex.) ***procedure*** Sort (*List*).

Atunci când avem nevoie de serviciile procedurii *Sort*, va fi identificată lista care se substituie listei *List*. (Ex.):

- se aplică procedura *Sort* asupra listei cu membrii organizației;

- se aplică procedura *Sort* asupra listei cu elevii clasei.

Nu trebuie pierdut din vedere faptul că scopul în care folosim pseudocodul este să creionăm algoritmi, nu să scriem un program.

4.3. Dezvoltarea algoritmilor

Dezvoltarea unui program constă din două activități:

- dezvoltarea algoritmului;
- reprezentarea algoritmului ca program.

Până acum ne-a preocupat problema reprezentării algoritmilor, dar stabilirea algoritmului constituie, în general, cea mai incitantă etapă a dezvoltării *software*-ului.

Teoria rezolvării problemelor

Capacitatea de a rezolva diferite probleme rămâne mai degrabă un talent artistic care trebuie dezvoltat. Etapele pentru rezolvarea problemelor, definite în linii mari de matematicianul Polya (1945), rămân valabile și astăzi ca principii de bază:

- **Faza 1.** Înțelegerea problemei;
- **Faza 2.** Conceperea unui plan de rezolvare a problemei;
- **Faza 3.** Punerea în practică a planului respectiv;
- **Faza 4.** Evaluarea soluției din punctul de vedere al corectitudinii și ca potențial instrument pentru rezolvarea altor probleme.

În contextul dezvoltării programelor, aceste etape devin:

- **Faza 1.** Înțelegerea problemei;
- **Faza 2.** Conceperea unei metode de rezolvare a problemei prin procedură algoritmică;
- **Faza 3.** Formularea algoritmului și reprezentarea lui ca program;
- **Faza 4.** Evaluarea programului din punctul de vedere al corectitudinii și ca potențial instrument pentru rezolvarea altor probleme.

Importanța primului pas

Vom identifica pe scurt câteva metode de rezolvare a problemelor. Toate aceste metode au ceva în comun și anume: important este primul pas.

Acest prim pas se poate realiza prin mai multe metode:

1. Una dintre aceste metode ar fi să procedăm invers, anume: dacă se dorește a se găsi o metodă de a obține o anumită ieșire pe baza unei intrări date, se pornește de la valoarea de ieșire, încercând să se ajungă la valoarea de intrare.
2. O altă metodă este aceea de a se căuta o problemă înrudită cu cea care trebuie rezolvată mai simplu sau care are deja o soluție. Se va încerca aplicarea soluției problemei înrudite și asupra problemei inițiale. Această metodă este foarte utilă în contextul dezvoltării programelor pentru care dificultatea constă în găsirea algoritmului general care să permită rezolvarea tuturor cazurilor.
3. În final avem metoda rafinării pas cu pas. Această tehnică pornește de la împărțirea problemei în mai multe subprobleme. Metoda permite abordarea problemei generale în mai multe etape, în ideea că fiecare în parte este mai ușor de rezolvat decât problema în ansamblul ei. În continuare subproblemele vor fi la rândul lor descompuse în mai multe etape, până când se ajunge la o serie de subprobleme ușor de rezolvat.

Rafinarea pas cu pas este o metodologie descendentă (*top-down*), care merge de la general către particular. Există, de asemenea, și metodologia ascendentă (*bottom-up*) care pornește de la cazurile particulare către cazul general. Soluțiile obținute prin metoda rafinării pas cu pas au în mod natural o structură modulară, motiv pentru care această metodă este apreciată.

Un algoritm cu o structură modulară poate fi ușor adoptat la o reprezentare modulară, ceea ce simplifică mult gestionarea activității de dezvoltare a programului propriu-zis. De asemenea, modulele rezultate din aplicarea metodei de rafinare pas cu pas sunt pe deplin compatibile cu ideea de lucru în echipă, modul cel mai răspândit de dezvoltare a unui produs *software*.

Dezvoltarea multor proiecte *software* pentru prelucrarea datelor presupune o dezvoltare a unei componente organizatorice. Problema nu presupune descoperirea unui algoritm uluitor, ci organizarea coerentă a sarcinilor care trebuie duse la îndeplinire.

În concluzie, dezvoltarea algoritmilor rămâne mai degrabă un talent care se dezvoltă în timp, decât o știință foarte exactă cu metodologii bine stabilite.

4.4. Structuri iterative

În cadrul **structurilor iterative** unele instrucțiuni se repetă ciclic. Vom prezenta în continuare câțiva algoritmi „celebri”: căutarea secvențială și binară.

Algoritmul de căutare secvențială

Scopul acestui algoritm este determinarea unei valori dacă există sau nu în cadrul unei liste. Presupunem că lista a fost sortată pe baza unei reguli oarecare, funcție de entitățile componente (o listă de nume respectă ordinea alfabetică, o listă de numere respectă ordinea crescătoare/descrescătoare). Pentru exemplificare, propunem următorul algoritm: căutăm într-o listă atâta timp cât mai sunt nume și numele-țintă este mai mic decât numele curent.

În pseudocod, acest proces se reprezintă astfel:

se selectează primul articol din listă ca articol de test

while (valoare-țintă > articol-test și mai sunt articole de
luat în considerare)

do (selectează următorul articol din listă ca
articol de test).

La ieșirea din această structură, articolul de test va fi mai mare sau egal cu numele-țintă, fie va fi ultimul nume din listă. În oricare din aceste cazuri putem determina dacă procesul s-a încheiat cu succes prin compararea valorii articolului de test cu valoarea-țintă. Pentru aceasta se adaugă la valoarea rutinei în pseudocod (deja scrisă) următoarea instrucțiune:

if (valoare-țintă = valoare articol-test)
then (declară căutarea un succes)
else (declară căutarea un eșec).

Am făcut ipoteza (presupunerea) că lista conține cel puțin un articol. Totuși vom plasa rutina pe ramura *else* a instrucțiunii:

if (lista este goală)
then (declară căutarea un eșec)
else (...).

În acest mod vom obține procedura din fig. 4.2.

```
procedure Search (list target value)  
if (lista este goală)  
    then (declară căutarea un eșec)  
    else [(selectează primul articol din list ca articol de test)  
        while(target value = articolul de test și mai sunt articole  
            de luat în considerare)  
            do (selectează următorul articol din list ca articol de  
                test)  
        if (target value = articolul de test)  
            then (declară selectarea un succes)  
            else (declară căutarea un eșec)]
```

Figura 4.2 - Algoritm de căutare secvențială reprezentat în pseudocod
(*sequential search algorithm*)

Acest algoritm este folosit pentru listele de mici dimensiuni sau când utilizarea lui este dictată de alte considerente.

Controlul ciclurilor

Utilizarea repetată a unei instrucțiuni sau a unei secvențe de instrucțiuni este unul din conceptele algoritmice cele mai importante. O metodă de implementare a acestui concept este structura iterativă denumită **ciclu/bucă** (*loop*).

În cadrul buclei o serie de instrucțiuni, numită **corpul buclei**, este executat în mod repetat sub supravegherea unui proces de control.

Ex. *while* (condiție) *do* (corpul ciclului).

Instrucțiunea *while* de mai sus reprezintă o structură ciclică deoarece execuția ei conduce la parcurgerea ciclică a secvenței: testează condiția; execută corpul ciclului; testează condiția;

execută corpul ciclului, ... , testează condiția, până când condiția este îndeplinită. Să analizăm mai îndeaproape și instrucțiunea de control. Controlul unui ciclu (bucle) constă în trei activități: inițializare, testare, modificare, așa cum se prezintă și în fig. 4.3:

inițializare	=	stabilește o stare inițială care poate fi modificată în vederea atingerii condiției de încheiere
testare	=	compară condiția curentă cu condiția de încheiere și, în caz de egalitate, încheie procesul de repetare
modificare	=	schimbă starea astfel încât aceasta să avanseze către condiția de încheiere

Figura 4.3 - Componentele controlului repetitiv

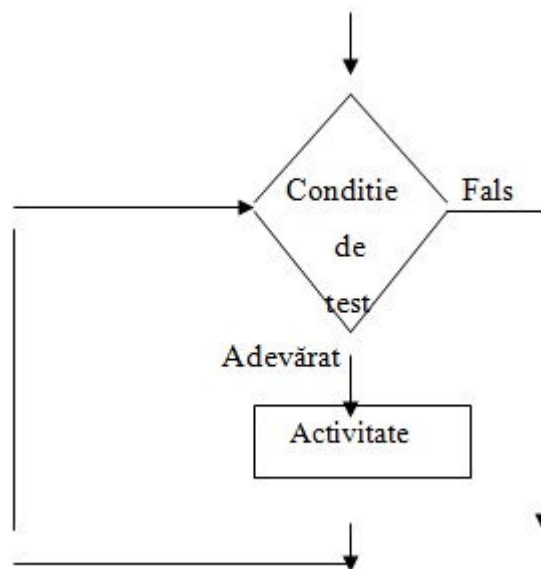


Figura 4.4 - Structura ciclului de tip *while*

Există două structuri de tip buclă foarte răspândite și anume: *while* (condiție) *do* (activitate) care se reprezintă prin schema logică (*flowchart*) din fig. 4.4. În cazul structurii *while* testarea condiției de încheiere se face înaintea executării corpului ciclului.

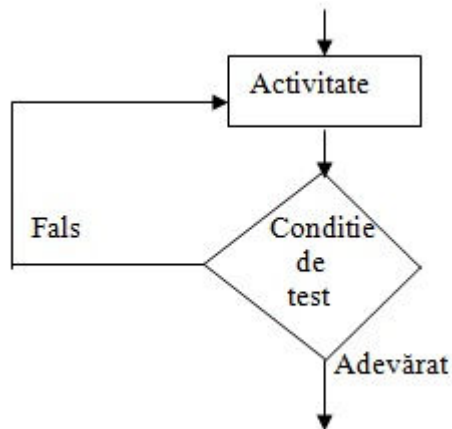


Figura 4.5 - Structura ciclului de tip *repeat*

A doua structură tip buclă, reprezentată în fig 4.5., solicită să se execute mai întâi corpul ciclului și după aceea să se testeze condiția de încheiere. Pentru reprezentarea schemei din fig. 4.5. în pseudocod se utilizează următoarea formulare :

repeat (activitate) *until* (condiție).

Ca exemplu suplimentar de utilizare a structurilor interactive vom prezenta algoritmul de sortare prin inserare. Algoritmul de sortare prin inserare sortează o listă extrăgând în mod repetat câte un articol și inserându-l în poziția corespunzătoare ordinii dorite.

procedure Sort (lista)

if (lista conține două sau mai multe articole) **then**

[(umbrește porțiunea din lista de la al doilea până la ultimul articol;

repeat (elimină umbra din primul articol al porțiunii umbrite din

listă și identifică acest articol drept pivot; mută pivotul

într-o locație temporară lăsând în listă un spațiu gol);

while (există un nume deasupra spațiului și acest nume este mai mare decât pivotul)

do (mută numele de deasupra spațiului în spațiul gol,

lăsând un spațiu deasupra numelui respectiv;

mută pivotul în spațiul gol din listă)

until (orice umbră din listă a fost eliminată).

Figura 4.6 - Sortarea prin inserare

4.5. Structuri recursive

Structurile recursive reprezintă o alternativă de realizare a proceselor repetitive fără a utiliza cicluri. Pentru a înțelege această tehnică vom discuta **algoritmul de căutare binară** (*binary search*).

Algoritmul de căutare binară

Vom relua problema căutării unui articol într-o listă sortată abordând metoda pe care o folosim când căutăm un cuvânt în dicționar. La căutarea într-un dicționar începem prin a deschide dicționarul la o pagină din zona în care credem că se află articolul căutat. S-ar putea să găsim cuvântul chiar în pagina respectivă, dacă nu, trebuie să continuăm căutarea. Numai că am restrâns (deja) zona de căutare, putându-ne rezuma fie la zona dinaintea poziției curente, fie la cea de după poziția curentă. Fig. 4.7 este reprezentarea nucleului căutării binare în pseudocod a acestui mod de căutare aplicat asupra unei liste generice.

În această situație nu avem avantajul de a ști cu aproximație în ce zonă se află articolul căutat, așa că instrucțiunile din figură ne spun să începem prin a deschide lista la articolul din „mijloc”. În cazul când lista are un număr par de elemente, prin articol de mijloc vom înțelege primul articol din cea de a doua jumătate a listei. Dacă articolul astfel selectat nu este cel căutat, rutina oferă două alternative, ambele oferind o căutare efectuată cu ajutorul unei proceduri numită *search*. Selectează „mijlocul” listei *list* ca articol de test; execută unul dintre următoarele blocuri de instrucțiuni, în funcție de situarea valorii-țintă *Target/Value* față de articolul de test.

```
Target Value = articol de test (declară căutarea încheiată cu succes)
Target Value < articolul de test [(aplică procedura search pentru a vedea dacă Target
Value este în porțiunea din list de deasupra articolului de test) și
    if (acea căutare este încununată de succes)
    then (declară această căutare încununată de succes)
    else (declară această căutare încheiată cu eșec)]
Target Value > articolul de test [aplică procedura search pentru a vedea dacă Target
Value este în porțiunea din list de sub articolul de test și
    if (acea căutare este încununată de succes)
    then (declară această căutare încheiată cu succes)
    else (declară această căutare încheiată cu eșec)].
```

Figura 4.7 - Nucleul căutării binare

Se observă că procedura permite și rezolvarea cazului căutării într-o listă goală dacă va fi completată, obținându-se pentru cautarea binara programul în pseudocod prezentat în fig. 4.8.

```
procedure search (lista, Target Value
if (lista este goală)
    then (declară căutarea încheiată cu eșec)
    else [selectează „mijlocul” listei cu articol de test; execută una dintre următoarele
        blocuri de instrucțiuni, în funcție de situarea valorii-țintă Target Value față de
        articolul de test
        Target Value = articolul de test;
            (declară căutarea încheiată cu succes)
        Target Value < articolul de test
            (aplică procedura search pentru a vedea dacă Target Value este în porțiunea din lista
            de deasupra articolului de test, și
            if (acea căutare este încununată de succes)
                then (declară această căutare încheiată cu succes)
                else (declară această căutare încheiată cu eșec)]
        Target Value > articolul de test
            [aplică procedura search pentru a vedea dacă Target Value este în porțiunea din lista
            de sub articolul de test și
            if (acea căutare este încununată de succes)
                then (declară această căutare încheiată cu succes)
                else (declară această căutare încheiată cu eșec)].
```

Figura 4.8 - Algoritmul căutării binare reprezentată în pseudocod

4.6. Eficiență și corectitudine

Două dintre subiectele legate de algoritmi sunt importante și vor fi abordate în continuare: *eficiența* și *corectitudinea* acestora.

Eficiența algoritmilor

Deși calculatoarele actuale execută milioane de instrucțiuni pe secundă, eficiența proiectării algoritmilor rămâne o preocupare importantă, deoarece adesea diferența dintre un algoritm eficient și unul ineficient constituie diferența între o soluție practică și una inaplicabilă.

Să analizăm o problemă practică pentru a înțelege mai bine eficiența algoritmilor. Problema constă în regăsirea înregistrării unui student din mulțimea înregistrărilor studenților (cca 30.000 studenți) din cadrul unei universități. Pentru început vom analiza modul de regăsire folosind algoritmul de regăsire secvențială.

Acest algoritm parcurge lista de la început, comparând fiecare articol cu numărul căutat. Necunoscând valoarea-țintă, nu putem spune cât de departe va merge în listă această căutare. Statistic ne așteptăm ca media să fie pe la jumătatea listei. Prin urmare, în medie algoritmul secvențial va investiga cca 15.000 de înregistrări pentru fiecare căutare. Dacă regăsirea și verificarea unei înregistrări durează o milisecundă, o astfel de căutare durează cca 15 secunde. O zi de lucru are 28.800 secunde; dacă s-ar lucra continuu, algoritmul de căutare secvențială va permite să se efectueze cca $1.900 \div 2.000$ de căutări pe zi. Deci regăsirea celor 30.000 de studenți va dura cca. 15 zile.

Acum a venit rândul să analizăm căutarea binară. Această căutare începe prin a compara valoarea-țintă cu articolul situat în mijlocul listei. Este posibil să nu găsească articolul căutat, dar se reduce aria căutării la jumătate din lista inițială. Deci după prima interogare, căutarea binară mai are de luat în calcul cel mult 15.000 de înregistrări. După a doua interogare rămân cel mult 7.500 și așa mai departe, până la cea de a 15-a interogare când se găsește articolul căutat. Dacă o interogare durează o milisecundă, atunci regăsirea unei anumite înregistrări se va face în maximum 0,015 secunde. Aceasta înseamnă că pentru a găsi înregistrările corespunzătoare ale celor 30.000 de studenți, este nevoie de cel mult 7,5 minute.

Cele două rezultate arată clar o îmbunătățire substanțială a timpului de căutare folosind algoritmul de căutare binară.

Verificarea produselor software

Verificarea software-ului este o sarcină dintre cele mai importante, iar căutarea unor tehnici eficiente de verificare constituie unul dintre cele mai active domenii de cercetare.

Una din direcțiile de cercetare este încercarea de a demonstra corectitudinea programelor prin aplicarea tehnicilor logicii formale. Deci, ideea de bază este reducerea verificării la o procedură de logică formală care să nu fie afectată de concluziile care pot rezulta din aplicarea unor argumente intuitive. Pentru demonstrarea corectitudinii unui program vom porni de la presupunerea că la începutul execuției acestuia sunt îndeplinite un număr de condiții, numite **precondiții**.

Pasul următor constă în evaluarea modului în care consecințele acestor precondiții se propagă prin program. Cercetătorii au analizat diverse structuri de programe pentru a vedea cum este afectată execuția lor de o anumită afirmație despre care se știe că este adevărată înainte de execuția structurii respective. Dacă, de exemplu, se știe că afirmația este adevărată înainte de execuția structurii *if* (condiție)

then (instrucțiunea 1)

else (instrucțiunea 2)

, atunci imediat înainte de execuția instrucțiunii 1 putem spune că atât afirmația respectivă, cât și condiția de test sunt adevărate, în timp ce dacă se execută instrucțiunea 2 știm că sunt adevărate afirmația respectivă și negarea condiției de test. Să luăm ca exemplu ciclul *repeat* din fig 4.9:

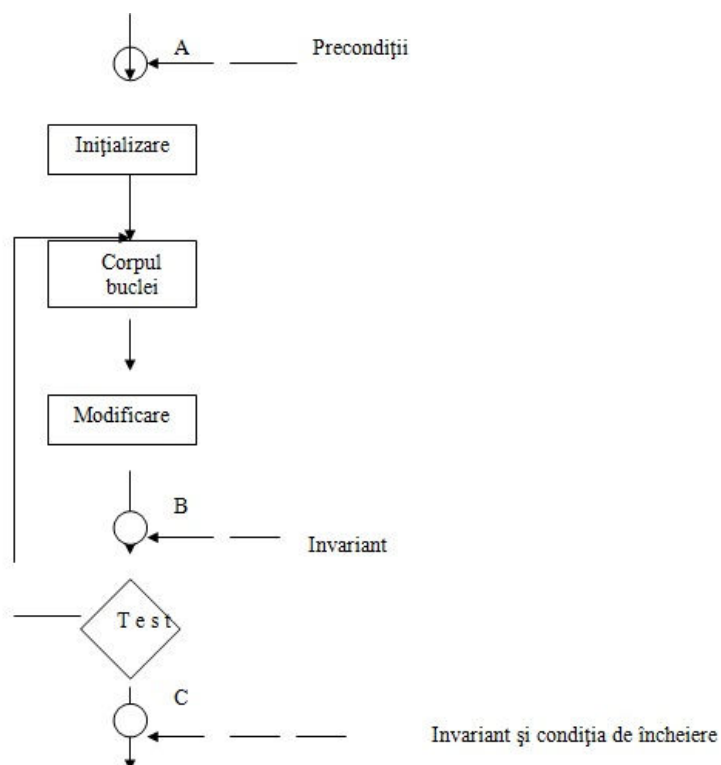


Figura 4.9 – Aserțiuni asociate cu o structura *repeat*

Să presupunem că în urma precondițiilor din punctul A putem stabili că o anumită aserțiune este adevărată de fiecare dată când în timpul procesului repetitiv este atins punctul B (o astfel de aserțiune care apare în cadrul unui ciclu este cunoscută sub numele de *invariant al buclei*). La încheierea procesului repetitiv, execuția ajunge în punctul C, unde putem spune că atât invariantul, cât și condiția de ieșire sunt adevărate.

Invariantul este în continuare adevărat, deoarece testul de încheiere a ciclului nu modifică nici o valoare de program, iar condiția de încheiere este adevărată, pentru că astfel ciclul nu s-ar terminat). Dacă aceste aserțiuni combinate implică obținerea rezultatului de ieșire dorit, atunci pentru a demonstra că astfel ciclul nu s-ar fi terminat.

Dacă aserțiunile de mai sus implică obținerea rezultatului de ieșire dorit, pentru a demonstra că programul este corect trebuie să arătăm că cele două componente ale controlului, inițializarea și modificarea, conduc la îndeplinirea condiției de încheiere.

Tehnicile formate de verificare a programelor nu au atins un stadiu care să faciliteze aplicarea lor asupra sistemelor generale de prelucrare a datelor. În cele mai multe cazuri, programele sunt „verificate” prin aplicarea lor pe seturi de date de test, procedură care nu este foarte sigură. Alegerea datelor de test folosite pentru verificarea programelor ar trebui tratată cu aceeași atenție ca și obținerea unui eșantion reprezentativ pentru o cercetare statistică.

TEST AUTOEVALUARE 4 (Algoritmii)

1. Definiți noțiunea de algoritm și prezentați principalele trăsături ce caracterizează un algoritm.
2. Programul este reprezentarea unui:
 - a. Algoritm
 - b. Pseudocod
 - c. Liste de instrucțiuni
3. Care dintre următoarele exemple pot constitui probleme ce se pot rezolva prin intermediul unui algoritm:
 - a. Construirea unei liste cu toate numerele întregi și pozitive
 - b. Sortarea unei liste în ordine descrescătoare
 - c. Extragerea primului număr întreg din lista rezultată
4. Care dintre următoarele exemple nu pot constitui probleme ce se pot rezolva prin intermediul unui algoritm:
 - a. Construirea unei liste cu toate numerele întregi și pozitive
 - b. Sortarea unei liste în ordine descrescătoare
5. Un algoritm este alcătuit din:
 - a. Procese
 - b. Instrucțiuni
 - c. Liste
6. Definiți noțiunea de primitive și ilustrați-le printr-un exemplu.
7. O primitivă constă din două elemente. Ele sunt:
 - a. Sintaxa și Semantică
 - b. Procese și Semafoare
 - c. Secvențiator și Executor
8. Sintaxa se referă la:
 - a. Reprezentarea simbolică a unei primitive
 - b. Reprezentarea în pseudocod a unui algoritm
 - c. Reprezentarea gradului de complexitate a unui algoritm

9. Semantica se referă la:
- a. Conceptul reprezentat (semnificația primitivei respective)
 - b. Procese și semafoare
 - c. Reprezentarea simbolică a unei primitive
10. Definirea precisă a primitivelor elimină:
- a. Ambiguitatea
 - b. Claritatea
 - c. Certitudinea
11. Definirea precisă a unei primitive permite:
- a. Un grad uniform de detaliere
 - b. Un grad uniform de amplificare
 - c. Stabilirea de blocuri elementare care stau la baza reprezentării algoritmilor
12. Un limbaj de programare este constituit din:
- a. Semafoare și procese
 - b. Primitive ce pot fi combinate pentru reprezentarea ideilor mai complexe
 - c. Instrucțiuni de ciclare
13. Un pseudocod este:
- a. Sistem de notare intuitiv
 - b. Diagrame UML
 - c. Șablon de proiectare
14. Definiți noțiunea de pseudocod și ilustrați folosirea acestuia printr-un exemplu, maxim două.
15. Funcția *If* poate fi reprezentată prin pseudocod?
- a. Adevărat
 - b. Fals
16. Funcția *If* poate exista fără instrucțiunea *else*?
- a. Adevărat
 - b. Fals
17. Funcția *If* poate fi definită ca:
- a. *if*(condiție) *then* (activitate)
else(activitate)

- b. *if*(condiție) *then*(activitate)
 - c. *if*(condiție) *do*(activitate)
18. Selectarea între două activități în funcție de îndeplinirea unei condiții, poate fi reprezentată prin:
- a. Funcția *If*
 - b. Funcția *While*
 - c. Funcția *Assign*
19. Funcția *While* constă în:
- a. Executarea repetată a unei instrucțiuni sau a unei secvențe de instrucțiuni atâta timp cât o anumită condiție este adevărată.
 - b. Executarea unei instrucțiuni folosind un număr exact de pași
 - c. Executarea unei instrucțiuni la infinit
20. Modelul pseudocod al instrucțiunii *while* este:
- a. *while*(condiție) *do*(activitate)
 - b. *if*(condiție) *while*(activitate)
 - c. *do*(activitate) *if*(condiție)
21. Definiți funcția *while* și ilustrați folosirea acesteia printr-un exemplu în pseudocod.
22. Definiți pe scurt funcția *assign* și folosirea acesteia printr-un exemplu în pseudocod.
23. Modelul pseudocod pentru funcția *assign* este:
- a. *assign*(nume) *the value*(expresie)
 - b. *assign*(instrucțiune) *do*(activitate)
 - c. *do*(activitate) *while*(*assign*(instrucțiune))
24. Definiți noțiunea de *procedură* și ilustrați folosirea acesteia printr-un exemplu
25. Identificați numele procedurii din următorul cod:

```
procedure Salutări
assign Count the value 3
while Count < 0 do
  (tipărește mesajul „Hello” și
  assign Count the value Count + 1)
```

26. Procedurile trebuiesc să fie:

- a. Cât mai generale
- b. Cât mai lizibile
- c. Greu de înțeles pentru alți programatori cu excepția celui care a scris-o

27. Dezvoltarea unui program constă din:

- a. Dezvoltarea algoritmului
- b. Reprezentarea algoritmului ca program
- c. Gradul de complexitate să fie cât mai mic

28. Definiția teoriei rezolvării problemelor și a programelor precizând principalele faze și principii.

UNITATEA DE ÎNVĂȚARE 5

5. Limbaje de programare

5.1. Perspective istorice

Primele generații

La început, programarea presupune o metodă foarte laborioasă, anume transpunerea algoritmilor respectivi în limbaj-mașină, suplimentar față de proiectarea algoritmului. După localizarea și corectarea erorilor, se poate spune că programul este utilizabil.

Prima simplificare a procesului de programare a fost să se renunțe la reprezentarea operațiilor și operatorilor sub formă de cifre hexazecimale, atribuindu-se memoria pentru instrucțiunile limbajului-mașină. În cazul operanzilor s-au stabilit reguli pentru atribuirea unor nume descriptive (identificatori) pentru locațiile de mnemonice și utilizarea acestor nume în instrucțiuni.

Inițial programatorii foloseau aceste notații pentru dezvoltarea programului pe hârtie, ulterior translatându-l într-o formă utilizabilă de calculator. Procesul de translatare este un tip de activitate care poate fi realizată direct de calculator. Utilizarea mnemonicelor s-a oficializat ca limbaj de asamblare, s-a creat **asamblorul**, care translatează programele scrise în limbaj de asamblare într-o formă compatibilă cu calculatorul.

Limbajele de asamblare au apărut ca un pas uriaș pe drumul către medii de programare mai eficiente. Limbajele de asamblare, ca **limbaje de generația a doua**, prezentau multe avantaje față de limbajele-mașină (**prima generație**), dar nu constituiau un mediu de programare foarte propice.

Orice *limbaj de asamblare este dependent de mașină*, deoarece instrucțiunile folosite se referă la caracteristicile unui anumit calculator.

Alt dezavantaj al limbajelor de asamblare este acela că programul trebuie gândit la un nivel foarte coborât (*instrucțiunile urmăresc pas cu pas limbajul-mașină*), în loc să se concentreze asupra unei soluții globale a problemei de rezolvat.

Pe baza acestor concepte s-au dezvoltat limbaje de programare, din **a treia generație**, la care primitivele erau de nivel înalt și independente față de mașină (calculator). Principala preocupare în dezvoltarea limbajelor de programare din generația a treia a fost identificarea unui ansamblu de primitive de nivel înalt, similare pseudocodului despre care am discutat, care să permită dezvoltarea produselor software. După identificarea setului de primitive la nivel înalt, a fost scris un program, numit **translator**, care translatează programele exprimate în primitive de nivel înalt în programe în limbaj-mașină. Aceste programe de translatare au ajuns să fie cunoscute sub denumirea de **compilatoare**.

Independența față de mașină

Un program scris într-un limbaj din generația a treia poate fi utilizat la orice calculator, utilizând pur și simplu asupra lui compilatorul corespunzător.

O componentă a problemei portabilității este faptul că în unele cazuri nu s-a ajuns la un acord cu privire la definiția corectă (exactă) a unui anumit limbaj. Pentru a elimina probleme de acest tip, American National Standards Institute (ANSI) și International Standards Organization (ISO) au adoptat și publicat standarde pentru multe din limbajele cele mai răspândite.

Faptul că mașinile pot acum să răspundă la instrucțiuni de nivel înalt a permis specialiștilor să vizeze medii de programare în care oamenii să poată să comunice cu mașina direct prin intermediul unor concepte abstracte specifice gândirii umane, fără să mai fie necesară transpunerea într-o formă compatibilă mașinii.

De asemenea, există preocupări pentru realizarea unor mașini (calculatoare) care să participe nu numai la execuția algoritmilor, ci și la dezvoltarea algoritmilor. Rezultatul acestor preocupări a condus la apariția unei game de limbaje de programare în continuă diversificare, care cu greutate pot fi clasificate în generații.

Ca regulă generală, termenul de **limbaje de programare din generația a patra** se referă la pachetele software care permit utilizatorilor fără o pregătire de specialitate să poată adapta produsele propriilor lor necesități. În această categorie se încadrează sistemele de calcul tabelar, sistemele de baze de date, pachete pentru grafică, procesoare de text, aceste pachete sunt adesea grupate într-un singur sistem integrat.

Termenul de **limbaje din generația a cincea** este folosit adesea cu referire la programarea declarativă și în special pentru ceea ce se numește programarea logică.

Paradigme de programare

Dezvoltarea istorică a limbajelor de programare se poate reprezenta printr-o diagramă cu mai multe piste, în care căile de dezvoltare (conform diferitelor paradigme) sunt reprezentate separat (fig. 5.1). Apar astfel patru piste care ilustrează patru paradigme diferite: **funcțională, orientată spre obiecte, imperativă, declarativă**, limbajele asociate fiecăruia fiind înfățișate într-un mod care să arate evoluția lor în timp, ceea ce nu înseamnă că aceste limbaje au evoluat neapărat unele din altele.

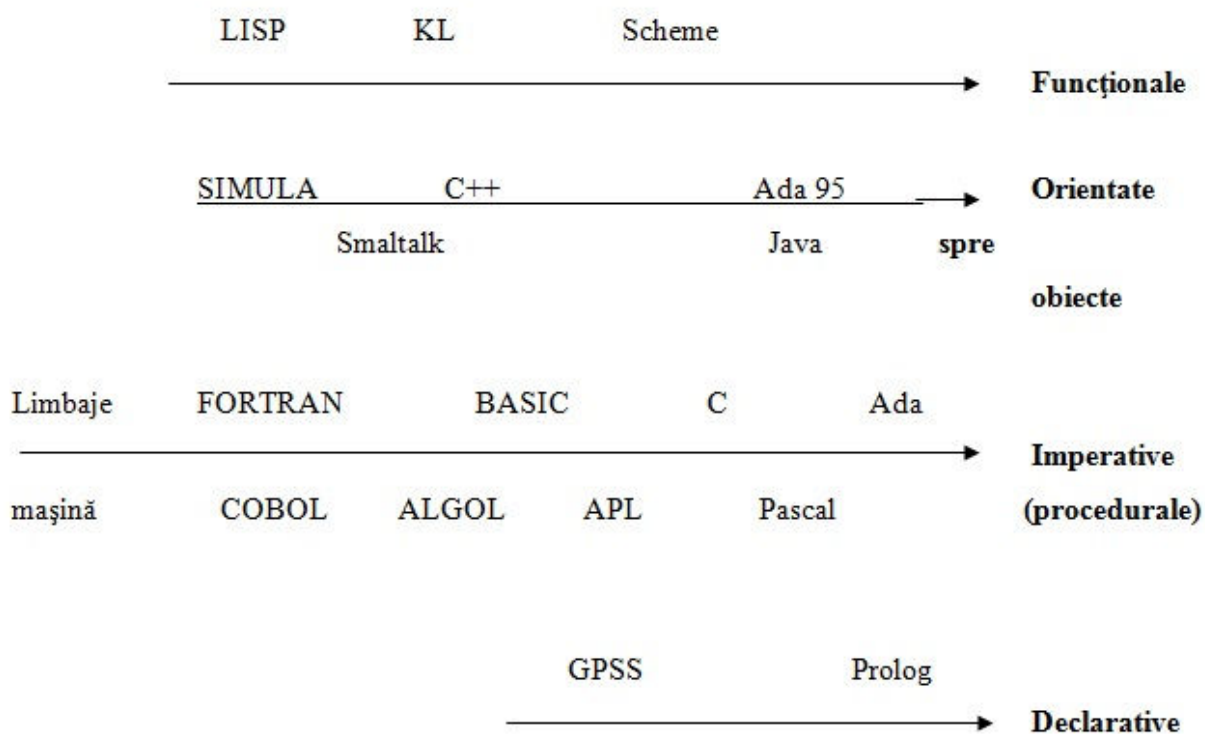


Figura 5.1 - Evoluția paradigmelor de programare

Paradigma imperativă (procedurală) reprezintă modul clasic de abordare a procesului de programare. Pe această paradigmă se bazează ciclul parcurs de unitatea centrală: citirea instrucțiunii – decodificare – execuție. Prin această paradigmă se definește procesul de programare ca o secvență de comenzi care, urmate, conduc la rezultatul dorit.

Paradigma declarativă propune dezvoltarea și implementarea unui algoritm general de rezolvare a problemelor. Pentru rezolvarea unei probleme nu va trebui decât ca ea să fie formulată într-o manieră compatibilă cu algoritmul și să i se aplice algoritmul respectiv.

Principalul obstacol în calea dezvoltării unui limbaj de programare bazat pe paradigma declarativă este descoperirea algoritmului (general) de rezolvare a problemelor.

Paradigma funcțională prezintă procesul de dezvoltare a programelor ca o construcție de „cutii negre”, fiecare acceptând intrări și furnizând ieșiri.

Matematicienii desemnează aceste „cutii” drept funcții, de unde denumirea de paradigmă funcțională. Primitivile unui limbaj de programare funcțional constau din funcții elementare de bază cu care programatorul construiește funcțiile complexe necesare rezolvării problemei respective. Paradigma de programare funcțională încurajează abordarea modulară a programelor, deci programele sunt bine organizate în mod natural. Cu această metodă se dezvoltă de obicei pachete de software complexe.

Paradigma orientată spre obiecte, această metodă conduce la un proces de programare cunoscut sub numele de **programare orientată spre obiecte** (OOP – *object oriented Programming*). În această abordare *datele* sunt considerate a fi **obiecte „active”**, spre deosebire de *rolul „pasiv”* care le este atribuit în paradigma imperativă.

În abordarea orientată pe obiecte *lista* e considerată a fi **un obiect** care se compune din *lista propriu-zisă*, la care se adaugă *un ansamblu de rutine* pentru gestionarea ei. Programul care lucrează cu lista *nu are nevoie de algoritmi pentru efectuarea acestor operații*, ci *folosește rutinele oferite de obiect*. Limbajele de programare orientate spre obiecte sunt Visual Basic (Microsoft Corporation), Delphi (Borland International) și Java.

5.2. Conceptele programării clasice

În principiu, instrucțiunile într-un limbaj de programare se împart în: instrucțiuni declarative, instrucțiuni imperative și comentarii.

Instrucțiunile declarative definesc terminologia specifică utilizată în cadrul programului (ex. numele folosite pentru diferite date).

Instrucțiunile imperative sunt cele care descriu pașii algoritmului respectiv.

Comentariile îmbunătățesc înțelegerea algoritmului.

Orice modul de program începe cu o parte declarativă (descrierea terminologiei), urmată de o parte procedurală (instrucțiuni imperative pentru indicarea acțiunilor). Comentarii sunt plasate în program ori de câte ori se consideră necesar acest lucru.

Variabile, constante și literali

Identificatori de tip nume descriptive a unor locații de memorie sunt cunoscuți sub numele de **variabile**. Prin modificarea valorii stocate în locația respectivă în timpul execuției programului se schimbă valoarea asociată identificatorului.

Uneori programul folosește valori predefinite care nu se modifică. Această valoare se poate include ca un literal. Unele limbaje de programare permit atribuirea de nume descriptive și unor valori particulare care nu pot fi modificate. Aceste nume se numesc **constante**.

Ex.:

Pentru scrierea unui program referitor la un aeroport este necesară altitudinea acestuia. Într-o linie de program ce utilizează literali aceasta se scrie astfel:

assign Effective Alt the value (Altimeter – 392)

, unde: *Effective Alt*, *Altimeter* sunt variabile, 392 este o valoare literală.

Într-o linie de program ce utilizează constante același lucru se scrie astfel:

assign Effective Alt the value (Altimeter – Airpor Alt)

const Aieport Alt = 392

, (instrucțiunea declarativă **const** asociază constanta AirporAlt cu valoarea 392, iar numele Airpot Alt poate fi folosit oriunde în program în locul valorii 392).

Categorii de date

Instrucțiunile declarative identifică și tipul datelor. **Tipul datelor** determină modul în care este interpretat șirul respectiv de biți, cât și operațiile care pot fi efectuate cu data respectivă. Cele mai utilizate tipuri de date sunt: **întreg** (*integer*), **real** (*real*), **caracter** (*character*) și **boolean**.

Tipul integer se referă la date numerice întregi, reprezentate în complement față de doi. Se pot efectua cu aceste numere operații aritmetice clasice și comparațiile.

Tipul real se referă la numere reale, stocate în general în virgulă mobilă. Activitățile care trebuie executate pentru adunarea a două numere reale sunt diferite de cele pentru adunarea doi întregi.

Tipul character se referă la date care constau din simboluri, codificate de obicei în ASCII. Asupra acestui tip de date se pot efectua: comparații asupra poziționării alfabetice a unui

simbol față de altul; testarea apariției unui șir de simboluri în cadrul altui șir, concatenarea unui șir de simboluri la sfârșitul altuia pentru a forma un șir de mai lung.

Tipul boolean se referă la acele date cu valori doar de adevărat sau fals, ce apar ca urmare a efectuării unor comparații. Operațiile compatibile cu acest tip de date sunt verificările dacă o valoare este adevărată sau falsă.

În figura 5.2 sunt prezentate exemple de astfel de instrucțiuni declarative în Pascal, C, C++, Java și FORTRAN. În aceste exemple variabilele Length și Width sunt declarate de tip real, iar Price, Tax, Total de tip întreg.

Declarații de variabile în Pascal:

Var

Length, Width: real;

Price, Tax, Total : integer.

Declarații de variabile în C, C++ și Java:

float Length, Width;

int Price, Tax, Total.

Declarații de variabile în FORTRAN:

REAL Length, Width;

INTEGER Price, Tax, Total.

Figura 5.2 - Declarații de variabile în Pascal, C, C++, Java, FORTRAN

Structuri de date

Un concept asociat instrucțiunilor declarative este acela de structură de date care se referă la forma conceptuală a datelor. Datele pe lângă tip, deja explicat în capitolul anterior, au și o lungime (mărime).

Un șir de caractere este un caz particular al unei structuri generice de date care poartă numele de vector omogen (*homogeneous array*). Pentru a declara un astfel de vector, majoritatea

limbajelor de programare utilizează o instrucțiune declarativă în care se precizează mărimea pentru fiecare dimensiune a vectorului.

În fig. 5.3 se prezintă instrucțiuni declarative în C și Pascal care declară **Nume** ca vector unidimensional de tip caracter și lungime opt, iar **Scores** ca vector bidimensional de tip întreg, cu două linii și nouă coloane.

Vectori declarați în Pascal

Var

Name: packet array [1...8] of char;

Scores: array [1...2, 1...9] of integer.

Vectori declarați a C

char Name [8];

int Scores [2] . [9].

Structura conceptuală a vectorilor

Name:

--	--	--	--	--	--	--	--

Scores:

Figura 5.3 - Declararea unor vectori omogeni în Pascal și C

Există și vectori neomogeni, din punct de vedere al tipului de date incluse, numiți vectori heterogeni (*heterogenous array*). În fig. 5.4 se prezintă modul de declarare a unui asemenea vector.

După declararea vectorului se poate face referire la el în întregime, prin nume, iar la fiecare componentă a sa prin poziția acesteia. În cazul vectorilor omogeni componentele sunt identificabile prin indici care specifică linia, coloana etc.

Vectori declarați în Pascal

Var

Employe: record

Name: packed array [1...8] of char;

Age : integer;

Skill Rating: real

End

Vectori declarați în C

Struct

(char Name [8];

int Age ;

float Skill Rating;

Employe.

Organizarea conceptuală a vectorului

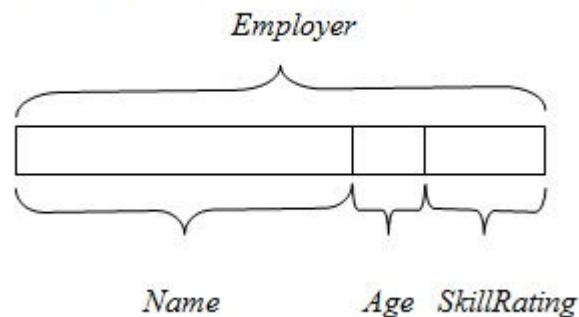


Figura 5.4 - Declararea unor vectori heterogeni în Pascal și C

Instrucțiuni de atribuire

Instrucțiunea de atribuire solicită ca unei variabile să i se atribue o valoare. O astfel de instrucțiune are următoarea sintaxă: variabila urmată de un simbol care reprezintă operația de atribuire și de o expresie care indică valoarea ce se atribue.

În limbajele C, C++ instrucțiunea **Total = Price + Tax** cere ca variabilei **Total** să-i fie atribuită valoarea sumei dintre **Price** și **Tax**.

În paradigmele imperativă și cea orientată pe obiecte instrucțiunea de atribuire este foarte utilizată, deoarece datele sunt manipulate în principal prin intermediul său.

Instrucțiuni de control

Instrucțiunile de control sunt instrucțiuni imperative care modifică ordinea de execuție a programului. Aceste instrucțiuni au stârnit cele mai multe controverse, iar „principalul vinovat” este cea mai simplă dintre ele, „go to”. Această instrucțiune furnizează o metodă de a continua programul din alt punct, care a fost identificat printr-o etichetă (nume, număr). Includerea unei astfel de instrucțiuni într-un limbaj de nivel înalt conduce la scrierea unor secvențe foarte încâlcite. Pentru evitarea unor asemenea situații, limbajele moderne de programare dispun de instrucțiuni de control mai elaborate (ex. *if – then – else*) care permit ramificarea programului prin intermediul unei singure structuri sintactice.

În figura 5.5 sunt prezentate câteva din cele mai utilizate structuri de ramificare și instrucțiuni de control, precum și modul lor de reprezentare în diverse limbaje.

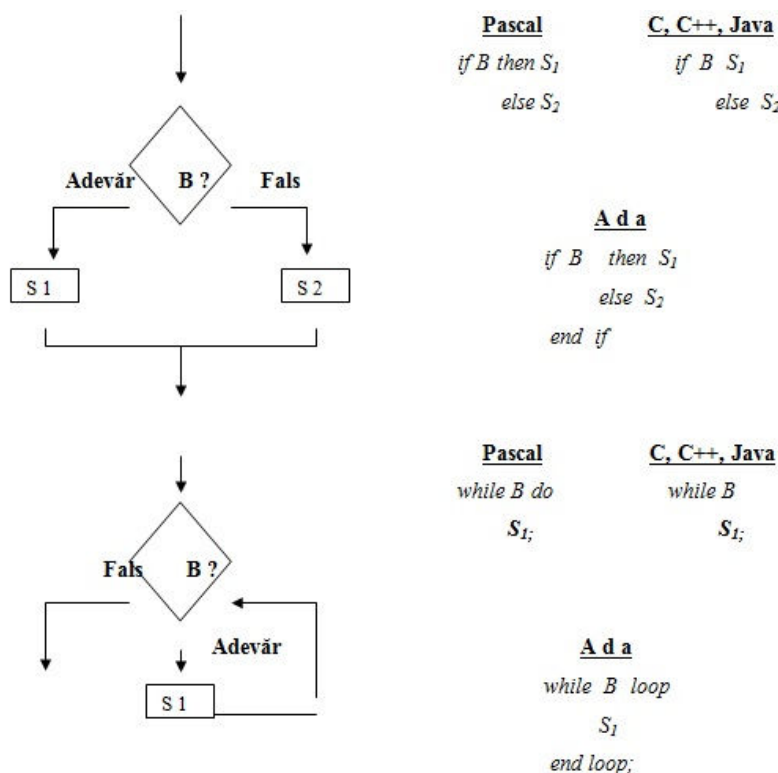


Figura 5.5 a - Structuri de control și reprezentarea lor în Pascal, C, C++, Java și Ada

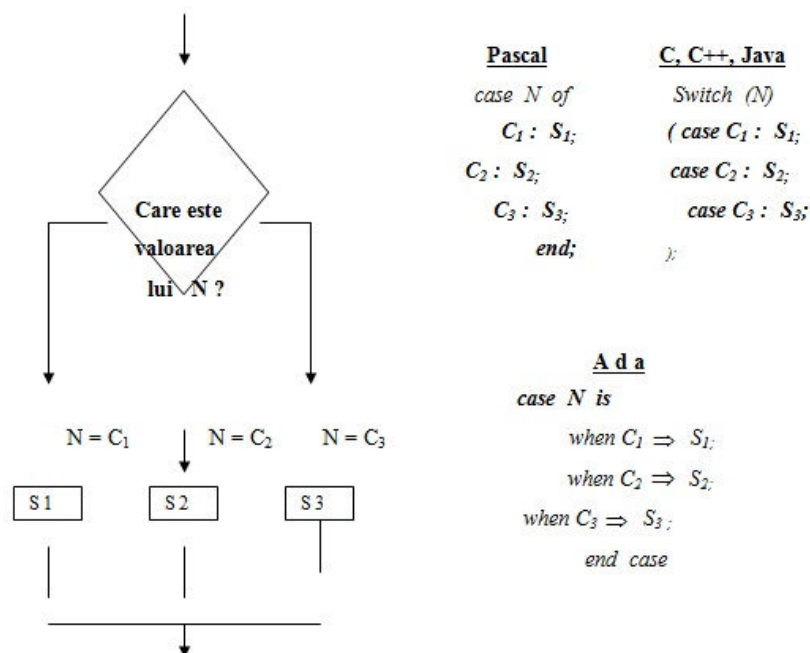


Fig 5.5. b. Structuri de control și reprezentarea lor în Pascal, C, C++, Java și Ada

O altă structură larg folosită este cea numită *for*. Ea este similară cu structura *while* utilizată în pseudocod, dar inițializarea, modificarea și încheierea ciclului sunt încorporate într-o singură instrucțiune.

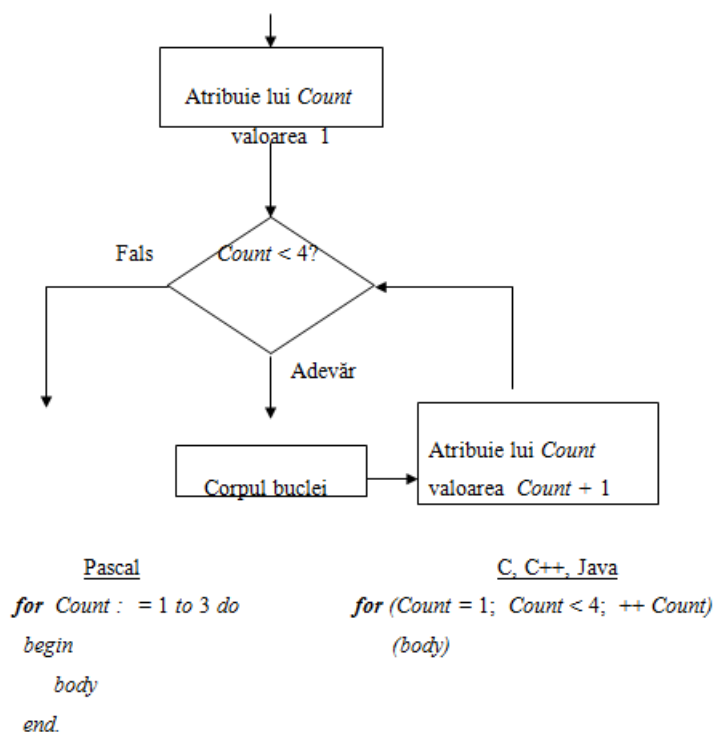


Figura 5.6 - Structura *for* și reprezentarea ei în Pascal, C, C++ și Java

Comentarii

Experiența arată că oricât de bine ar fi proiectat un limbaj de programare și oricât de bine ar fi folosit, informațiile suplimentare sunt utile întotdeauna pentru orice doritor care încearcă să înțeleagă un program de oarecare complexitate. În acest scop limbajele de programare furnizează o sintaxă pentru inserarea în program a unor instrucțiuni explicative numite **comentarii**. Documentarea constituită din aceste comentarii se numește **documentație internă**

Documentația internă este ignorată de către translator, pentru a include comentariile în program, limbajele de programare furnizează două metode de delimitare a acestora de restul programului. Fie se include comentariul între niște marcaje speciale, fie se marchează doar începutul comentariului, acesta urmând să ocupe tot rândul sau chiar mai multe rânduri (în acest caz se adaugă și la începutul rândurilor suplimentare semnul specific comentariului. Este recomandabil ca toate comentariile care se referă la un anumit modul de program să fie grupate într-un singur loc.

5.3. Module de program

Limbajele bazate pe paradigma funcțională împart în mod natural programele în funcții, iar limbajele bazate pe paradigma orientată spre obiecte au ca rezultat module de program care reprezintă obiectele. Ne vom concentra asupra tehnicilor prin care se poate obține o reprezentare modulară a unui algoritm.

Proceduri

Procedura este un modul de program scris independent de programul principal, dar legat de acesta printr-un proces de transfer/revenire conform fig.5.7.

Când este nevoie de serviciile procedurii, controlul se transmite acesteia (instrucțiunea JUMP, limbajul-mașină), iar după execuție controlul revine modulului principal.

Ca regulă generală, variabilele declarate în interiorul procedurii sunt variabile locale (se utilizează numai în interiorul procedurii). Sunt totuși cazuri în care unele date trebuie utilizate în comun de mai multe module. Variabilele utilizate într-o asemenea situație se numesc **variabile globale**.

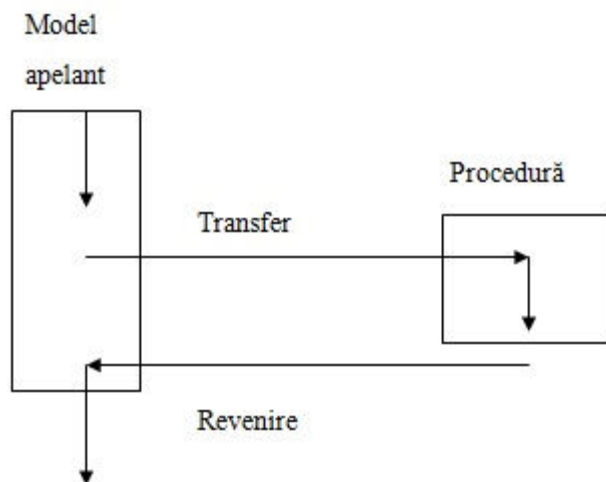


Figura 5.7 - Transferul controlului la apelarea unui subprogram

Parametrii

Folosirea în comun a unor informații prin utilizarea unor variabile globale nu este recomandată, deoarece acest mod de lucru maschează activitățile modulelor de program care partajează datele respective.

Se recomandă identificarea datelor partajate de mai multe module de program, ceea ce se poate face prin includerea explicită a listei acestora în instrucțiunea prin care se solicită execuția procedurii. Astfel și antetul procedurii trebuie să conțină o listă a variabilelor cărora li se atribuie valori la apelarea procedurii. Elementele care compun aceste două liste se numesc **parametri**.

La apelarea procedurii, lista de parametri din modulul apelant este asociată, element cu element, listei de parametri din antetul procedurii. Valorile parametrilor din modulul apelant sunt transferate efectiv parametrilor corespondenți din procedură.

După aceea, procedura este executată, iar valorile (eventual modificate) sunt transferate înapoi către modulul apelant. În alte situații transferul poate avea loc într-un singur sens, fie către procedură, fie către modulul apelant.

Un avantaj major al acestui sistem de substituie este acela că aceeași procedură poate fi apelată de mai multe ori, cu diferite seturi de date.

Funcții

Termenul de **funcție** se referă la un modul de program asemănător unei proceduri, cu deosebirea că o valoare se transferă înapoi către programul principal nu prin intermediul listei de parametri, ci ca „valoare a funcției”. Altfel spus, valoarea respectivă este asociată numelui funcției, așa cum se asociază o valoare unei variabile. În cazul funcției, valoarea se obține prin executarea instrucțiunilor din cadrul funcție.

Instrucțiuni de intrare/ieșire

Procedurile și funcțiile sunt metode foarte bune de a extinde caracteristicile unui limbaj de programare.

Dacă limbajul nu furnizează o anumită operație ca primitivă, se poate scrie o procedură sau o funcție care să efectueze acea operație, iar apoi se va apela la modulul respectiv ori de câte ori este nevoie. În majoritatea limbajelor, operațiile de intrare/ieșire sunt implementate în acest mod, cu diferența că procedurile și funcțiile apelate sunt de fapt rutine ale sistemului de operare.

Pentru a citi o valoare de la tastatură și a o atribui variabilei **Value**, în Pascal se scrie:

readln (Value),

, iar pentru a afișa valoarea respectivă pe ecran:

writeln (Value).

Limbajul C++, fiind un limbaj orientat pe obiecte, furnizează două obiecte prefabricate, *cin* și *cout*, care reprezintă dispozitivul standard de intrare și, respectiv, dispozitivul standard de ieșire. Aceeași acțiune de mai sus în limbajul C++ se va scrie astfel:

cin >> Value; [citirea valorii de la tastatură]

cout << Value; [afișarea variabilei *Value* pe ecran].

5.4. Implementarea limbajelor

În continuare vom analiza procesul de convertire a unui program scris într-un limbaj de nivel înalt într-o formă executabilă de mașină.

Procesul de translatare

Procesul de convertire a unui program dintr-un limbaj în altul se numește **translatare**. Programul în forma inițială este **programul sursă**, iar versiunea translatată este programul obiect.

Procesul de translatare constă din trei activități: analiza lexicală, analiza sintactică (*parsing*) și generarea codului.

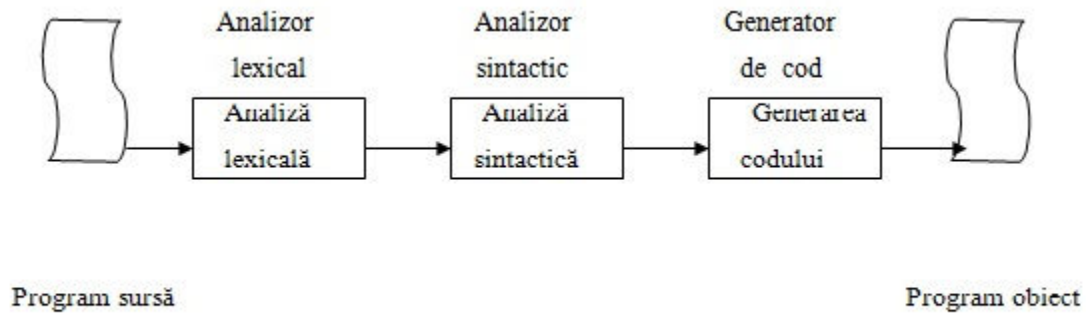


Figura 5.8 - Procesul de translatare

Analiza lexicală este procesul prin care se identifică șirurile de simboluri din programul sursă care reprezintă entitățile distincte. Pe măsură ce analizorul lexical identifică grupuri de simboluri care constituie entități distincte, le clasifică în funcție de ceea ce reprezintă – o valoare numerică, un cuvânt, un operator aritmetic, ... — și generează un model de biți cunoscut sub numele de simbol (*token*), care arată clasa entității respective.

Analiza sintactică este un proces de identificare a structurii gramaticale a programului și de recunoaștere a rolului fiecărei componente. Majoritatea limbajelor de programare sunt limbaje cu format liber, ceea ce înseamnă că locul instrucțiunilor în program nu are importanță. Pentru ca un calculator să poată analiza sintactic un program scris într-un limbaj cu format liber trebuie ca analizorul să-l identifice indiferent de spațiile utilizate în programul sursă.

În acest scop majoritatea limbajelor folosesc semnele de punctuație, cum ar fi punct și virgulă, pentru a marca sfârșitul unei instrucțiuni, precum și de **cuvinte-cheie** (cuvinte rezervate), cum ar fi *if*, *then* sau *else* pentru a marca începutul unor propoziții distincte.

Procesul de analiză sintactică se bazează pe un set de reguli care definesc sintaxa limbajului de programare, reprezentare prin diagramele de sintaxă. În fig. 5.9 se prezintă diagrama de sintaxă a instrucțiunii *if – then - else*.

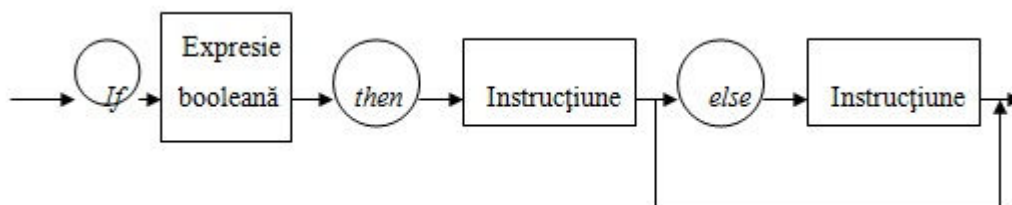


Figura 5.9 - Diagrama de sintaxă a instrucțiunii *if – then – else* din pseudocod

Procesul de analiză sintactică constă, în esență, din construirea arborelui de analiză sintactică a programului sursă. Din acest motiv, regulile de sintaxă care descriu structura gramaticală a unui program nu trebuie să permită obținerea a doi arbori diferiți pentru același șir analizat. Pentru exemplificare, regula din fig. 5.9 conține o eroare care permite ca, pentru instrucțiunea :

if B₁ if B₂ then S₁ else S₂

să se obțină doi arbori de analiză diferiți, prezentați în fig. 5.10.

Definițiile de sintaxă ale limbajelor formale de programare sunt proiectate astfel încât să evite asemenea ambiguități, folosind parantezele. Pentru a distinge cele două interpretări posibile vom scrie:

if B₁
 then (if B₂ then S₁)
 else S₂

if B₁
 then (if B₂ then S₁
 else S₂)

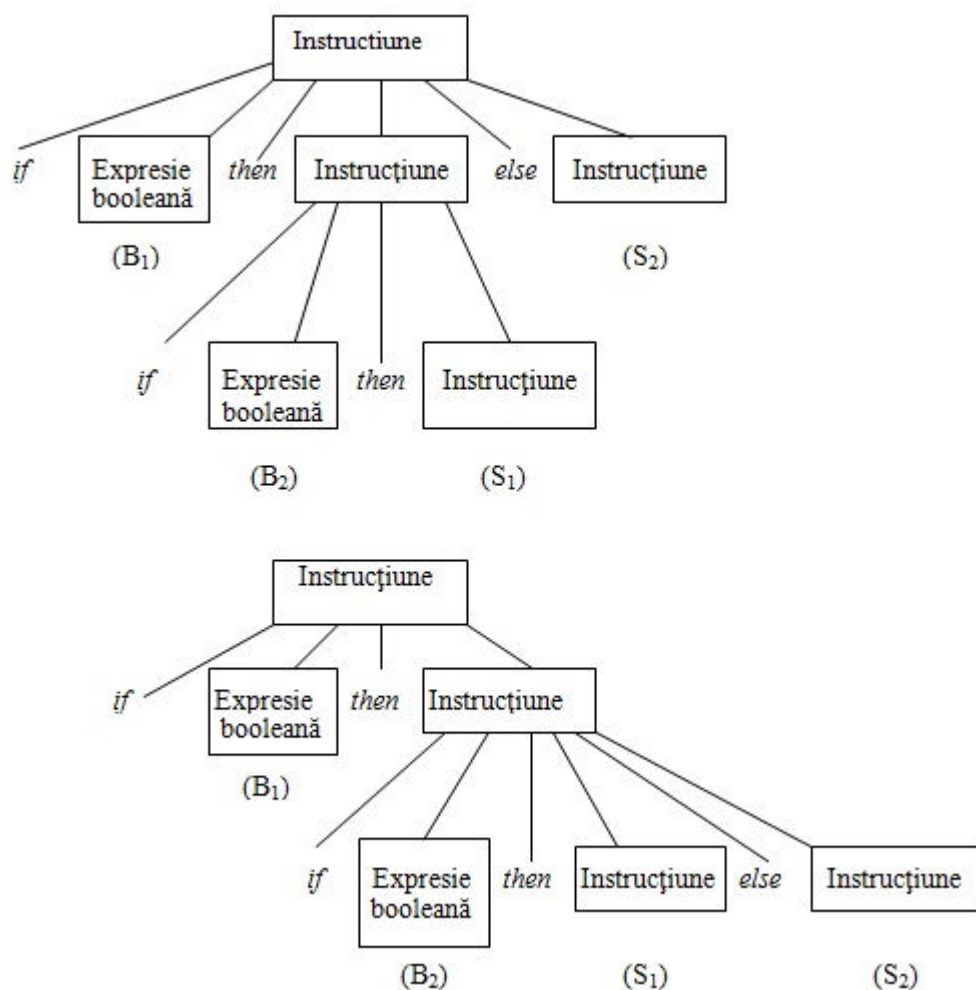


Figura 5.10 - Arbori de analiză sintactică diferiți pentru instrucțiunea
(if B₁ then if B₂ then S₁ else S₂)

Informațiile extrase din instrucțiunile declarative sunt înregistrate într-o tabelă numită **tabela de simboluri**, care conține date referitoare la variabilele care au fost declarate și la tipurile și structurile de date asociate.

O altă activitate din cadrul translatării unui program este **generarea codului**, reprezentând procesul de construire a instrucțiunilor în limbajul-mașină care simulează instrucțiunile recunoscute de analizorul sintactic. Una din sarcinile importante ale generatorului de cod este optimizarea codului.

Analiza lexicală, sintactică și generarea codului nu se realizează strict în această ordine, ci se împletesc: analizorul lexical identifică primul simbol și îl transferă analizorului sintactic care ar trebui să urmeze și îi cere următorul simbol.

În momentul în care sunt recunoscute instrucțiuni sau propoziții complete, analizorul sintactic apelează generatorul de cod, care produce instrucțiunile mașină necesare.

Editarea de legături și încărcarea

Programul obiect care rezultă în urma procesului de traducere, deși exprimat în limbaj mașină, este arareori într-o formă executabilă. Majoritatea mediilor de programare permit dezvoltarea și traducerea individuală a modulelor de program, la momente diferite, ceea ce permite construcția modulară a produselor software. Chiar dacă programul este dezvoltat și tradus sub forma unui singur modul, în cele mai multe cazuri programul obiect tot nu este pregătit pentru execuție, deoarece conține cereri de servicii software care sunt oferite direct de sistemul de operare sau pot fi obținute prin intermediul acestuia.

Un program obiect este un program mașină căruia îi lipsesc unele legături care trebuie „înnodate” pentru a-l transforma într-un program executabil.

Înnodarea acestor legături este efectuată de un program numit editor de legături (*linker*). Sarcina acestui *linker* este de a lega între ele programele-obiect, rutinele sistemului de operare și alte utilitare software într-un program executabil (modul încărcabil), memorat ca fișier în sistemul de stocare de masă al calculatorului.

Pentru executarea unui program astfel tradus, el trebuie plasat în memorie de către programul de încărcare (*loader*), parte a secvențiatorului.

Pe scurt, pregătirea unui program scris într-un limbaj de nivel înalt constă dintr-o secvență de trei pași: traducere, editare de legături și încărcare, așa cum se vede în fig.7.11. După traducere și editare de legături, programul poate fi încărcat și executat în mod repetat, fără a mai reveni la versiunea-sursă. Dacă sunt necesare modificări în program, acestea se fac în programe-sursă, după care sursa modificată este tradusă și se face editarea de legături, obținându-se un nou model încărcabil.

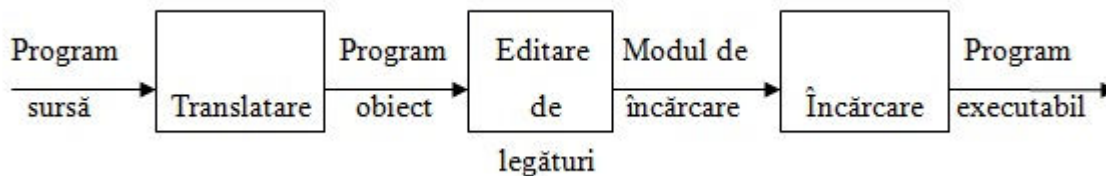


Figura 5.11 - Procesul complet de pregătire a programului

Pachete pentru dezvoltarea produselor software

Tendința curentă este aceea de a grupa translatorul și celelalte elemente software utilizate în procesul de dezvoltare a unui produs software într-un pachet care funcționează ca un sistem integral. Avantajele utilizării unui astfel de sistem integrat sunt reprezentate de posibilitatea ca programatorul să poată trece cu ușurință de la editor la instrumente de depanare modificând și testând imediat programul.

5.5. Programarea declarativă

Logica formală furnizează un algoritm general de rezolvare a problemelor, în jurul căruia se poate construi un sistem de programare declarativă.

Deductia logică

Să presupunem că următoarea afirmație este adevărată: Kermit fie este bolnav, fie se află pe scenă. În aceste condiții, dacă ni se spune despre Kermit că nu este pe scenă, tragem concluzia că este bolnav.

Acesta este un exemplu de aplicare a principiului deductiv numit **rezoluție**. Ca să înțelegem mai bine acest principiu, să începem prin a reprezenta propozițiile logice prin litere, iar negarea negațiilor lor prin simbolul „—” însoțit de o literă.

În formă generală principiul rezoluției precizează faptul că dacă propozițiile:

$P \text{ OR } Q$ și $R \text{ OR } \text{—}Q$

sunt adevărate, atunci și propoziția:

$P \text{ OR } R$ este adevărată.

Altfel spus, cele două propoziții inițiale se rezolvă prin a treia, numită **rezolvent**.

Se observă că rezoluția poate fi aplicată doar perechilor de propoziții cu **forma clauzală** (propoziții legate prin operatorul boolean OR).

În fig. 5.12 este reprezentată grafic rezoluția a două propoziții.

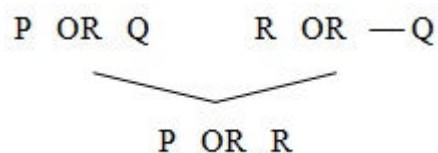


Figura 5.12 - Rezolvarea propozițiilor $(P \text{ OR } Q)$ și $(R \text{ OR } -Q)$ prin $(P \text{ OR } Q)$

Se spune că o serie de propoziții este inconsistentă dacă este imposibil ca toate propozițiile să fie adevărate în același timp. În fig. 5.13 se demonstrează faptul că setul de propoziții $P \text{ OR } Q$; $R \text{ OR } -Q$; $-R$; $-P$ este inconsistent.

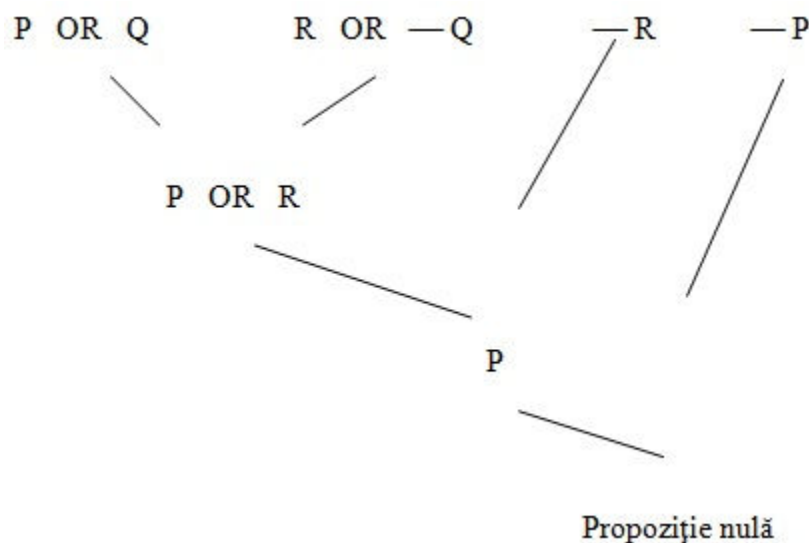


Figura 5.13 - Rezolvarea propozițiilor $(P \text{ OR } Q)$ și $(R \text{ OR } -Q)$, $-R$ și $-P$

Procesul prin care, pentru a face posibilă aplicarea principiului rezoluției variabilelor, li se atribuie valori se numește **unificare**. Acest proces, de unificare, permite ca într-un sistem deductiv propozițiile generale să fie aplicate în cazuri particulare.

Limbajul PROLOG

Limbajul Prolog (*PROgramming in LOGic*) este un limbaj de programare declarativă al cărui algoritm de rezolvare a programelor are la bază rezoluția repetată.

Un program Prolog constă dintr-o serie de propoziții inițiale, pe care algoritmul își bazează toate deducțiile.

Componentele din care sunt construite aceste propoziții se numesc predicate. Un astfel de predicat se compune dintr-un identificator, urmat de o paranteză cu lista argumentelor predicatului.

Argumentele predicatului încep întotdeauna cu literă mică. Limbajul Prolog face distincția între constante (literă mică) și variabile (majuscule). Instrucțiunile într-un limbaj Prolog sunt fie afirmații, fie reguli; ele se termină cu punct.

O afirmație constă dintr-un singur predicat. Nu trebuie să vă închipuiți că sistemul Prolog înțelege semnificația predicatelor din program; el le manipulează într-o manieră absolut simbolică, aplicând regulile de inferență ale rezoluției.

Programatorului îi revine responsabilitatea de a descrie toate caracteristicile și predicatele sub forma unor afirmații și reguli. Majoritatea implementărilor de Prolog sunt proiectate pentru utilizare interactivă.

Atunci când sistemul Prolog primește o întrebare, el îi aplică principiul rezoluției încercând să confirme faptul că o afirmație corespunzătoare întrebării este o consecință a afirmațiilor inițiale.

Astfel, pe baza seriei de instrucțiuni care descriu viteza relativă, pot fi confirmate toate întrebările următoare:

faster (broasca, melc)

faster (iepure, broască)

faster (iepure, melc).

Primele două sunt afirmații care apar în setul inițial de instrucțiuni, iar cea de a treia poate fi dedusă de către sistem. Dacă furnizăm sistemului întrebări care au argumente variabile, nu constante, vom obține exemple mult mai interesante.

De exemplu, pornind de la întrebarea:

faster (w, melc)

vom obține ca răspuns:

faster (broasca, melc).

Putem cere sistemului Prolog să găsească animale care sunt mai lente decât iepurele punându-i întrebarea:

faster (iepure, w).

Astfel, utilizând un singur program Prolog, vom putea să confirmăm faptul că un animal este mai rapid decât altul, să găsim toate animalele care sunt mai rapide decât un animal dat sau să găsim toate relațiile dintre animale în ceea ce privește viteza. Această flexibilitate este unul dintre aspectele care a trezit interesul specialiștilor pentru limbajul Prolog.

TEST AUTOEVALUARE 5 (Limbaje de programare)

1. Un program este utilizabil atunci când:
 - a. Gradul de complexitate este mic
 - b. Gradul de complexitate este mare
 - c. Când localizarea și corectarea erorilor a fost înlăturată
2. Asamblorul are rolul de:
 - a. Scrierea algoritmilor sub formă de pseudocod
 - b. Traducere a programelor scrise în limbaj de asamblare
 - c. Organizare a priorităților proceselor folosind semafoare
3. Definiți pe scurt limbajul de asamblare.
4. Descrieți independența programelor față de mașină.
5. ANSI se referă la:
 - a. American National Standards Institute
 - b. International Standards Organization
 - c. International Electrotechnical Commission
6. Prezentați pe scurt clasificarea limbajelor de programare.
7. Translatorul are rolul de:
 - a. Traducere a programelor exprimate în primitive de nivel înalt în limbaj-mașină
 - b. Traducerea programelor în pseudocod
 - c. Realizare a instrucțiunilor din cadrul unui program în semafoare și procese
8. Programele de traducere mai sunt cunoscute și sub numele de:
 - a. Asamblatoare
 - b. Compilatoare
 - c. Translatoare
9. Descrieți pe scurt paradigmele de programare.
10. Limbajul C++ este un limbaj bazat pe o paradigmă de tip:
 - a. Funcțională
 - b. Orientată pe obiect
 - c. Procedurale

- d. Declarativă
11. Limbajul Pascal este un limbaj bazat pe o paradigmă de tip:
- a. Funcțională
 - b. Procedurale
 - c. Declarativă
12. Limbajul Prolog este un limbaj bazat pe o paradigmă de tip:
- a. Declarativă
 - b. Orientată pe obiecte
 - c. Funcțională
13. Definiți **paradigma imperativă (procedurală)** și dați exemple de limbaje de programare care se bazează pe această paradigmă.
14. Definiți **paradigma declarativă** și dați exemple de limbaje de programare care se bazează pe această paradigmă.
15. Definiți **paradigma funcțională** și dați exemple de limbaje de programare care se bazează pe această paradigmă.
16. Definiți **paradigma orientată spre obiecte** și dați exemple de limbaje de programare care se bazează pe această paradigmă.
17. Definiți conceptele programării clasice.
18. Care sunt conceptele programării clasice:
- a. Instrucțiuni declarative
 - b. Procese
 - c. Instrucțiuni imperative
 - d. Translatoare
 - e. Comentariile
19. Caracterizați pe scurt variabilele, constantele și literalii.
20. O variabilă poate fi definită ca:
- a. locație de memorie
 - b. o valoare oarecare
 - c. valoare binară

21. Caracterizați pe scurt principalele categorii de date, furnizând câte un exemplu pentru fiecare.
22. Tipul datelor determină:
- Modul în care este interpretat șirul
 - Valorile pe care le poate accepta o variabilă
 - Modul în care compilatorul alocă memorie pentru tipul dat
23. Tipul *integer* se referă la:
- Date de tip numere întregi
 - Date tip numere reale
 - Date de tip caracter
 - Date de tip adevărat sau fals
24. Cu ajutorul tipului *integer* se pot efectua operații:
- Aritmetice
 - Booleene
 - De substituție
25. Tipul *real* se referă la numerele:
- Reale, stocate în virgulă mobilă
 - Reale, stocate ca numere întregi
 - Întregi
26. Tipul *character* se referă la date care constau:
- Din simboluri, codificate ca ASCII
 - În valori binare
 - În valori hexazecimale
27. Tipul *boolean* se referă la date care constau:
- În valori adevărate (true) sau false (false)
 - În valori octale
 - În valori hexazecimale
28. Descrieți pe scurt tipul de date *integer*.

29. Descrieți pe scurt tipul de date *real*.
30. Descrieți pe scurt tipul de date *character*.
31. Descrieți pe scurt tipul de date *boolean*.
32. Definiți și dați exemple de diferite variabile (integer, real, character, boolean) dând precizând limbajul în care au fost definite.
33. Conceptul de structură de dată este asociat:
- a. Instrucțiunilor ciclice
 - b. Instrucțiunilor declarative
 - c. Instrucțiunilor de control
34. Conceptul de structură de date se referă la:
- a. Forma abstractă a datelor
 - b. Forma generală a datelor
 - c. Forma conceptuală a datelor
35. Definiți și caracterizați prin exemplu structura conceptuală a vectorilor omogeni.
36. Numele unui șir de caractere este un caz particular al unei:
- a. Structuri generice de algoritmi
 - b. Structuri generice de date
 - c. Structuri ciclice și de control
37. Pentru a putea face referire la un vector în întregime, el trebuie să fie mai întâi:
- a. Declarat
 - b. Inițializat
 - c. Scris în pseudoco
38. Ne putem referi la un vector în întregime:
- a. Prin nume
 - b. Prin poziția componentei
 - c. Prin tipul vectorului
39. Prin ce sunt identificabile componentele unui vector omogen:
- a. Indici
 - b. Coloane
 - c. Rânduri

40. Definiți conceptul de *instrucțiune de atribuire* și dați exemple.

41. Ce se solicită unei variabile prin instrucțiunea de atribuire:

- a. Valoarea să fie incrementat cu 1
- b. Valoarea să fie decrementată cu 2
- c. Să i se atribuie o valoare

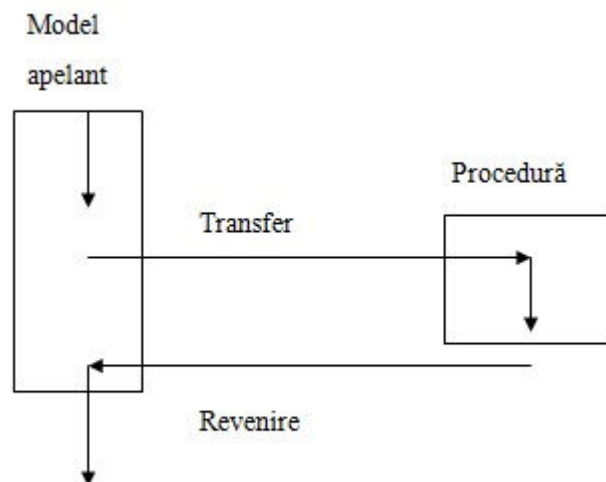
42. Definiți conceptul de instrucțiune de control și dați exemple.

43. Instrucțiunile de control sunt instrucțiuni:

- a. Imperative
- b. Prin Alegeri opționale
- c. Prin argumente justificabile

44. Comentariile într-un program sunt utile pentru:

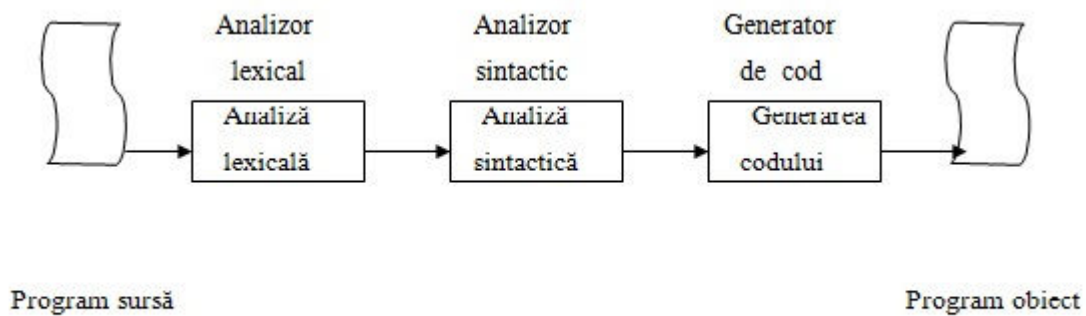
- a. Comentarea anumitor instrucțiuni
- b. Furnizarea de informații suplimentare pentru înțelegerea programului
- c. Marcarea variabilelor aflate în memorie în timpul execuției



45. Variabilele globale se definesc:

- a. La începutul programului
- b. În afara funcțiilor
- c. În interiorul funcțiilor

46. Definiți conceptul de *parametru* și dați exemple.
47. Definiți conceptul de *funcție* și dați exemple.
48. Definiți conceptul de *instrucțiuni de intrare/ieșire*.
49. Fie următorul proces de traducere. Definiți și caracterizați fiecare componentă ce face parte din acest proces (analizor lexical, analizor sintactic, generator de cod).



50. Analiza lexicală este procesul prin care:
- Se identifică șirurile de simboluri din programul sursă
 - Se identifică structura gramaticală a programului
 - Se ajunge de la program sursa la program obiect
51. Definiția analiza lexicală.
52. Definiți analiza sintactică și dați un exemplu.
53. Definiți generatorul de coduri.
54. Informațiile extrase din instrucțiunile declarative sunt înregistrate într-o tabelă numită:
- Tabelă de rutare
 - Tabelă statică
 - Tabelă de simboluri
55. Activitatea de generare a codului reprezintă:
- Procesul de construire a instrucțiunilor în limbaj mașină
 - Trecerea de la cod sursă la cod mașină
 - Trecerea de la hexazecimal la binar

56. Definiți editarea de legături și procesul complet de pregătire a programului.
57. Înainte ca un program să fie executat, el trebuie plasat în:
- a. Memorie
 - b. Pe un dispozitiv optic
 - c. În memoria virtuală
58. Logica formală stă la baza:
- a. Unui sistem de programare declarativ
 - b. Unui sistem de programare obiectul
 - c. Unui sistem de programare procedural
59. Definiți și argumentați deducția logică.
60. Procesul prin care este posibilă aplicarea principiului rezoluției variabilelor se numește:
- a. Unificare
 - b. Potențare
 - c. Divizare
61. Procesul de unificare permite într-un sistem deductiv ca propozițiile generale să fie aplicate în cazuri:
- a. Particulare
 - b. Generale
 - c. Specifice
62. Limbajul PROLOG (Programming in Logic) este un limbaj de:
- a. Programare declarativă
 - b. Programare orientată pe obiect
 - c. Programare procedurală
63. Într-un program PROLOG propozițiile din care este construit, se numesc:
- a. Verbe
 - b. Adjective
 - c. Predicate
64. Caracterizați limbajul PROLOG (Programming in Logic) dând și exemple.

UNITATEA DE ÎNVĂȚARE 6

6. Structuri de date

6.1. Vectori

Vectorii permit exprimarea unui algoritm ca și cum datele manipulate ar fi stocate într-un aranjament rectangular. În acest fel, ne putem referi la cel de al cincelea element al unui vector rectangular sau la elementul situat pe a treia linie și a șasea coloană al unui vector bidimensional.

6.1.1. Vectori unidimensionali

Să presupunem necesitatea stocării și prelucrării unui șir de temperaturi măsurate din oră în oră. Cea mai comodă formă de organizare a acestor date este un vector unidimensional. Acest vector unidimensional poate fi considerat o listă numită. *Citiri* ale cărei articole sunt specificate prin poziția lor în listă.

Prima valoare înregistrată va fi specificată cu *Citiri(1)*, a doua cu *Citiri(2)* și așa mai departe. Organizarea fizică, în exemplul nostru, este evidentă datele vor fi stocate într-o secvență de 24 de celule de memorie care au adrese consecutive. Cunoșcând adresa primei celule de memorie translatorul va putea să convertească referirile de tipul *Citiri(4)* în adrese de memorie corespunzătoare.

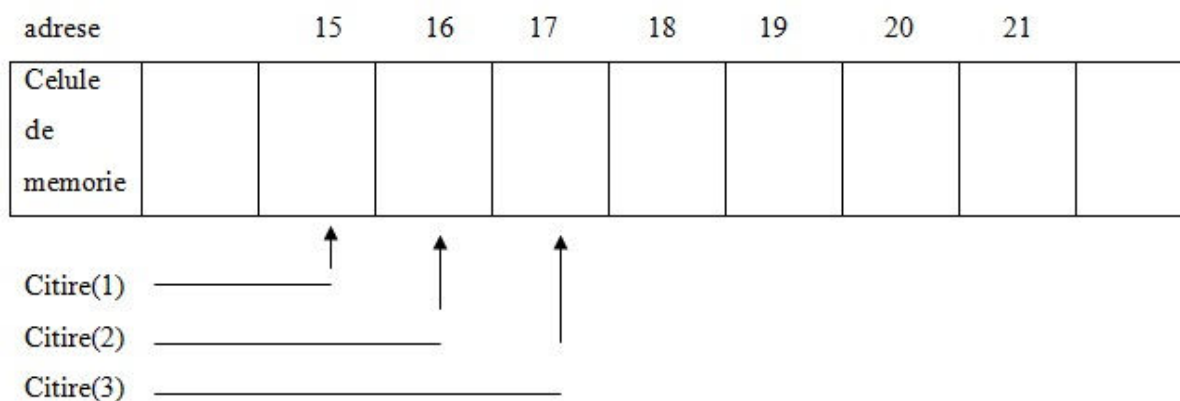


Figura 6.1 - Vectorul *Citiri* stocat în memorie începând de la adresa 15

6.1.2. Vectori multidimensionali

În cazul acestor vectori destinați organizării datelor se poate sugera o formă tabelară. Să imaginăm exemplul vânzărilor făcute de un număr de agenți comerciali pe o perioadă de o săptămână. Forma de organizare tabelară este următoarea, în stânga apar pe verticală agenții, iar pe prima linie zilele săptămânii. Deci datele vor fi dispuse pe linii și coloane. Valorile de pe o linie reprezintă vânzările efectuate de un agent, iar pe o coloană vânzările efectuate într-o anumită zi. Extragerea informației din tabel presupune găsirea valorii care se află la intersecția unei linii cu o coloană. Memoria calculatorului nu are o structură matricială, motiv pentru care structura tabelului va trebui simulată. Calculăm pentru început spațiul de memorie necesar și rezervăm un bloc continuu de celule de memorie de dimensiunea respectivă. Apoi se vor stoca datele în memorie linie cu linie în ordine astfel, valorile din prima linie, apoi valorile din linia doua, etc.

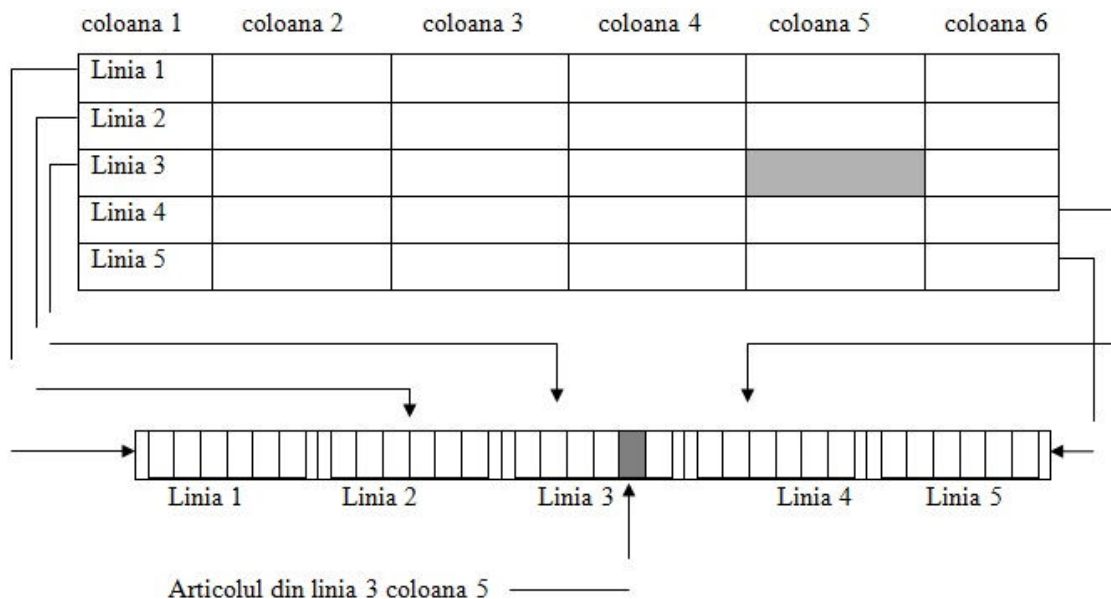


Figura 6.2 - Vector bidimensional stocat în ordinea liniilor

Acest sistem de stocare se numește în ordinea liniilor (row major order); existând și posibilitatea stocării în ordinea coloanelor (column major order).

Cum putem localiza elementele vectorului atunci când este nevoie de ele? Notăm cu C numărul de coloane ale vectorului (numărul elementelor de pe o linie). Pentru a găsi elementul

din linia I, coloana J va trebui să ne deplasăm de la începutul blocului cu $C * (I-1) + J$ poziții. Această exprimare se numește polinom de adresa (address polynomial).

Altfel spus, trebuie să depășim I-1 linii, fiecare având C articole, pentru a ajunge la linia I, apoi încă J elemente pentru a ajunge la elementul J, din linia curentă. În fig. 2 acest calcul este urmatorul $C=6, I=3, J=5$; formula devine $6*(3-1)+5 = 17$

6.2. Liste

Există și structuri vectoriale dinamice, cu forma și dimensiuni variabile. În anumite cazuri, pe lângă localizarea elementelor în cadrul structurii ne vom confrunta și cu variațiile lungimii acesteia.

Pointeri

Locațiile din memoria calculatorului sunt identificate prin adrese numerice. Prin adresa unei anumite date vom găsi elementul respectiv fără nici o dificultate. Așa cum elementul propriu-zis poate fi înregistrat într-o celulă de memorie și adresa lui poate fi stocată în alta celulă. Ulterior dacă accesăm adresa găsim elementul respectiv prin intermediul respectivei adrese. Putem considera că celula de memorie care conține adresa “indică” (point) către elementul respective, de unde apare și numele de pointeri dat celulelor de memorie care conțin adrese.

Multe limbaje de programare permit declararea, alocarea și manipularea pointerilor. Să exemplificăm prin situația unei biblioteci care ține evidența cărților în ordine alfabetică, după titlu. În respectiva organizare este dificil de găsit toate cărțile scrise de un anumit autor. Pentru a rezolva problema se rezervă în blocul care conține informații referitoare la carte o celulă suplimentară de tip pointer. În fiecare celulă pointer se stochează adresa altui bloc care corespunde unei cărți scrise de același autor. În acest fel, cărțile care au același autor vor fi legate într-un fel de buclă (fig.6.3).

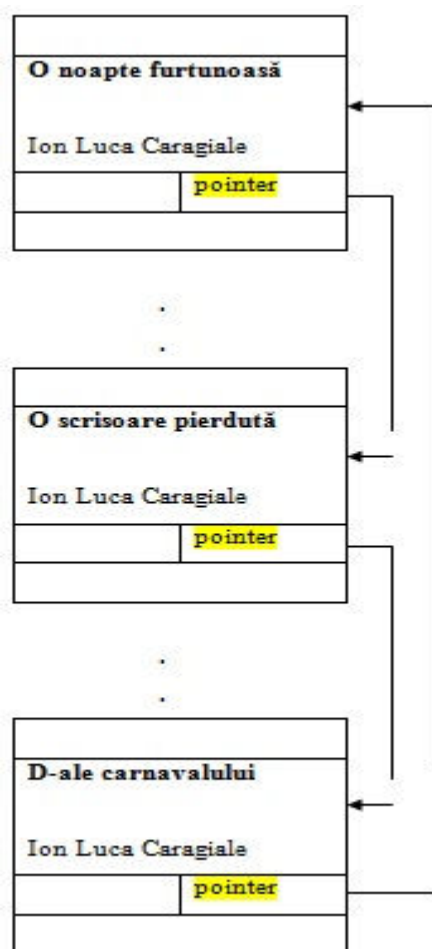


Figura 6.3 Lista de cărți dintr-o bibliotecă stocată după titlu și înlanțuită după autor

În continuare vom discuta despre structurile dinamice analizând modalitățile de stocare a unei liste de nume în memoria calculatorului.

Liste contigue

Una din tehnicile de stocare a unei liste de nume în memoria calculatorului este plasarea listei într-un bloc de celule cu adrese consecutive. Presupunem că numele respective nu depășesc opt caractere. Vom împărți blocul în sub-blocuri de câte opt celule. În fiecare sub-bloc se va putea stoca un nume (înregistrând în fiecare celulă codul ASCII pentru o literă - fig.6.4). Dacă

numele nu ocupă toate celulele din blocul alocat, vom complete celulele libere cu codul ASCII pentru spațiu.

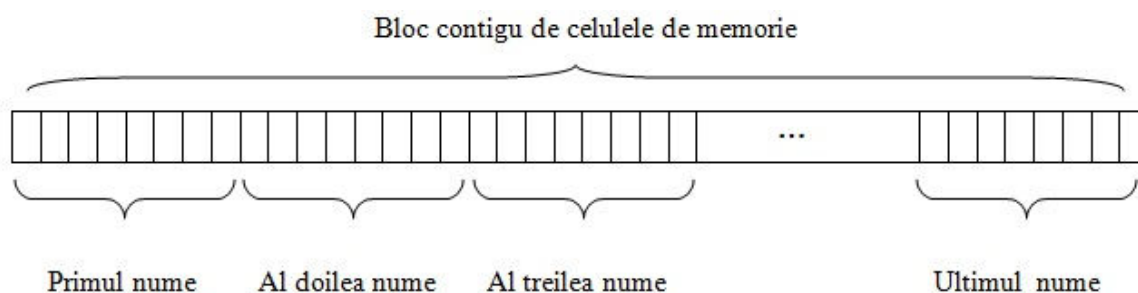


Figura 6.4 - Nume stocate în memorie ca listă contiguă

Acest mod de organizare se numește listă contiguă, în cadrul sistemului pentru stocarea listei. Exemplu. Instrucțiune Pascal

Var

List packed array (1 .. 8, 1..10) of char

, declară un vector bidimensional de 8 linii și 10 coloane. Problemele acestei structuri de stocare (structura contigua) apar la :

- Ștergerea unui nume din listă - Dacă numele șters nu este ultimul pentru a păstra ordinea de stocare (eventual alfabetică) trebuie să fie mutate toate numele care se află în lista după numele șters.
- Adaugarea unui nume nou în listă - O problemă posibilă, să nu mai fie loc în memorie pentru acest nou nume, deoarece folosim un bloc contiguu. O altă problemă este poziționarea noului nume într-o listă alfabetică, acolo unde-i este locul.

Liste înlanțuite

Fiecare nume se înregistrează într-un șir de nouă celule, primele opt vor fi utilizate pentru stocarea numelui propriu-zis, iar ultima va fi folosită ca pointer către următorul nume din listă. Astfel, lista poate fi fragmentată în mici blocuri, de câte nouă celule, blocuri legate între ele prin pointeri. Acest mod de organizare se numește *listă înlanțuită* (linked list).

Pentru a ști de unde începe lista trebuie rezervată o celulă de memorie în care se salvează adresa primului articol din listă – pointer cap de listă (head pointer). Citirea listei va începe de la locația indicată de pointer-ul cap de listă, unde se găsește primul nume din listă și un pointer

către articolul următor. În același fel se poate parcurge întreaga listă. Problema detectării sfârșitului de listă se rezolvă prin utilizarea pointer-ului NIL (o combinație specială de biți înscrisă în celula de tip pointer a ultimului element din listă). În figura 6.5 este prezentată structura unei liste înlănțuite.

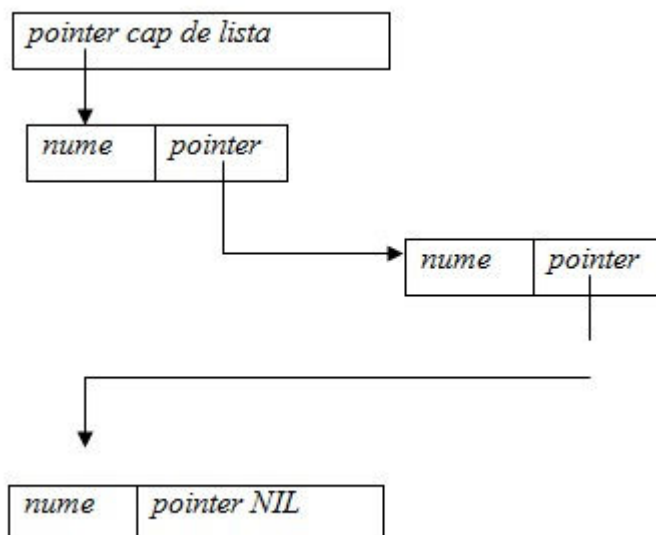


Figura 6.5 - Structura unei liste înlănțuite

Revenind la problemele care apar la structura contiguă, ștergerea unui nume și adaugarea unui nume în listă, ele au o soluționare mult simplificată în cadrul listei înlănțuite.

- a. Ștergerea unui nume implică doar schimbarea unui pointer. Pointer-ul care arată inițial către numele pe care-l ștergem va arăta către următorul nume din listă fig.6.6.

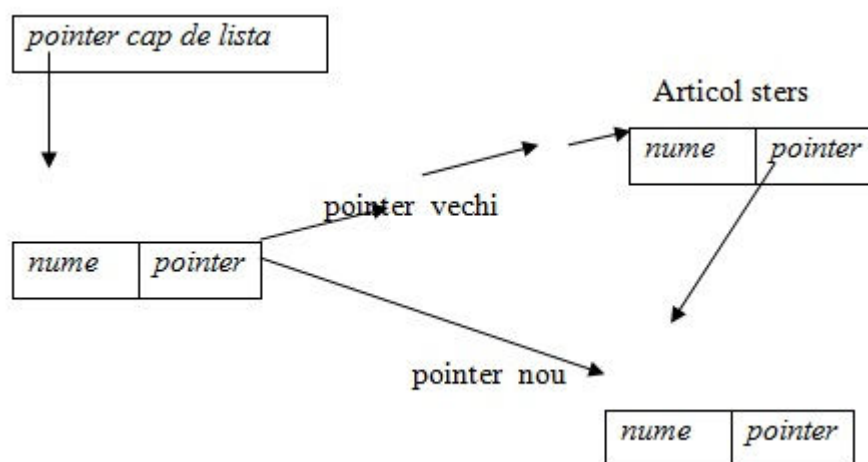


Figura 6.6 - Ștergerea unui articol dintr-o listă înlănțuită

- b. Pentru inserarea unui nou nume trebuie găsit mai întâi un loc liber de nouă celule în memorie, apoi se stochează noul nume în primele opt celule și se completează cea de a noua celulă cu adresa numelui din lista care trebuie precedată în lista numelui inserat - fig.6.7.

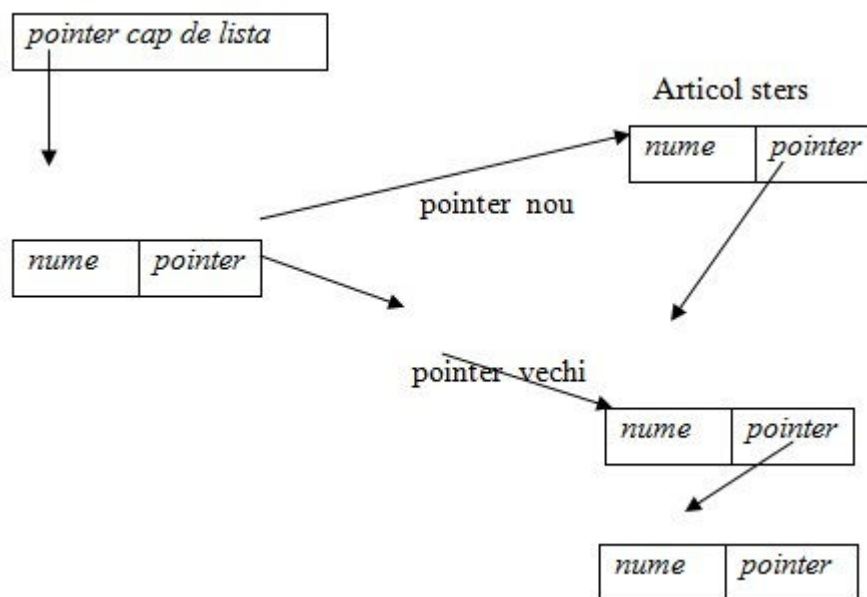


Figura 6.7 - Inserarea unui articol într-o listă înlănțuită

La ștergerea și inserarea unui nume într-o listă trebuie să se țină seama și de evidența blocurilor de celule eliminate din listă (aceste blocuri pot fi refolosite pentru stocarea noilor articole inserate). Evidența blocurilor șterse se poate ține prin construirea unei noi liste înlănțuite.

Stive

Stiva reprezintă o listă în care inserările și ștergerile se fac la un singur capăt al structurii. Capătul listei la care se efectuează aceste operații se numește vârful stivei (top), iar celălalt capăt se numește baza stivei. Pentru inserarea unui element în stivă se va folosi denumirea de operație **push**, iar pentru ștergerea unui element din stivă se folosește denumirea de operație **pop**.

Una dintre cele mai comune aplicații prin care se poate explica modul de lucru într-o stivă este execuția unor module de program din categoria procedurilor din pseudocod.

La solicitarea execuției unei proceduri, calculatorul accesează proceduri, iar încheierea procedurii trebuie să se termine în punctul anterior și să continue execuția programului apelant, ca în figura 6.8.

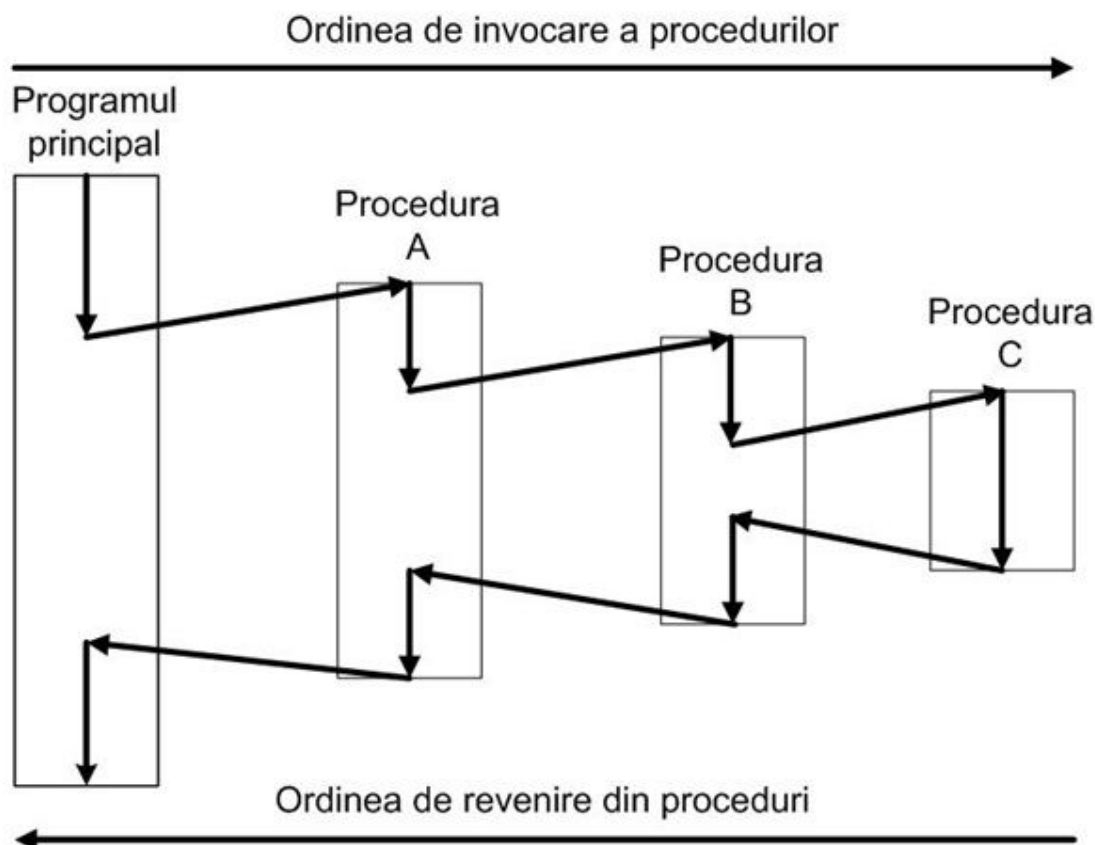


Figura 6.8 - Procedurile își încheie execuția în ordinea inversă celei în care au fost apelate

După realizarea transferului inițial trebuie să existe și să funcționeze un mecanism de memorare a punctului în care trebuie să revină. Mecanismul de memorare trebuie să salveze punctele de revenire și să le regăsească în ordine corectă. Conceptul de stivă este inerent proceselor care presupun ieșirea dintr-un sistem urmându-se drumul invers celui de intrare.

Implementarea stivei

În mod obișnuit se rezervă pentru stivă un bloc de memorie suficient de mare pentru a permite creșterea dimensiunii ei. După rezervarea blocului de memorie, trebuie să stabilim capătul care va servi ca bază stivei. Acesta este punctul în care vom include în stivă primul

element, articolele următoare fiind stocate în continuare, spre cealaltă extremitate a blocului rezervat.

Este nevoie de un instrument suplimentar: posibilitatea identificării în orice moment a vârfului stivei. Această celulă suplimentară poartă numele de pointer al stivei (stack pointer). Sistemul astfel completat este prezentat în figura 6.9.

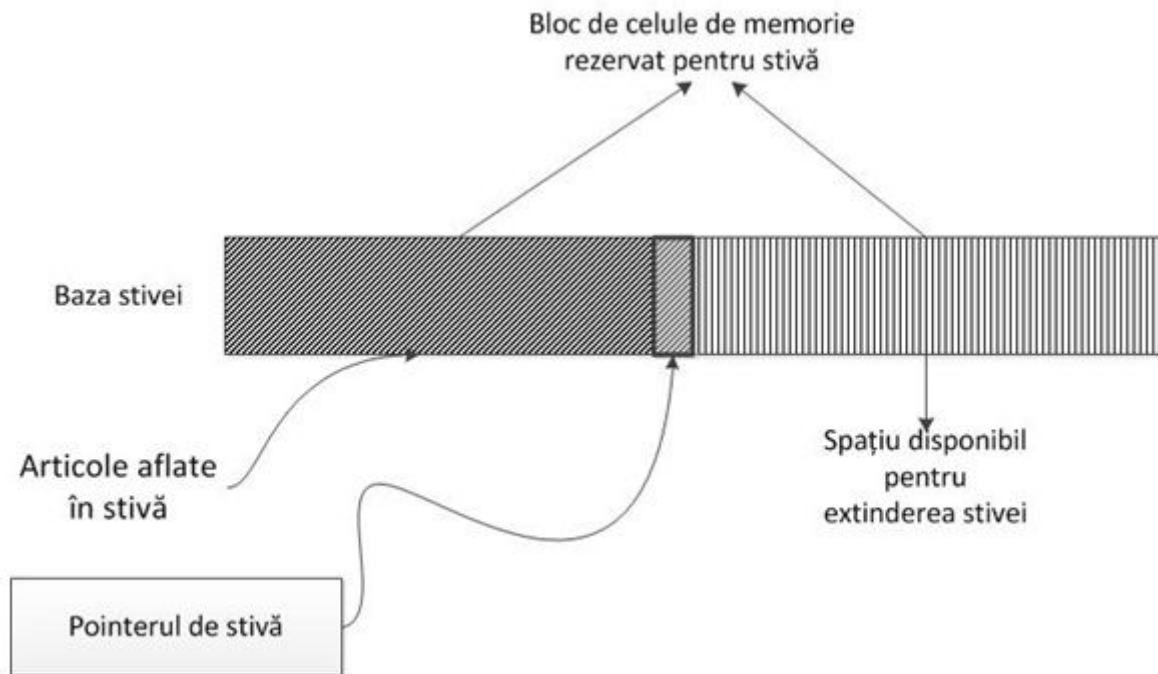


Figura 6.9 - Stocarea stivei în memorie

Modul de funcționare al sistemului:

- pentru adăugarea unui element în stivă, se modifică (mai întâi) pointerul de stivă astfel încât să indice către primul loc gol de deasupra vârfului, apoi plasăm respectivul element în locația respectivă;
- pentru a șterge (scoate) un element din stivă citim datele din locația indicată de pointerul de stivă, apoi ajustăm pointerul astfel încât să arate către următorul articol situat spre baza stivei.

Dacă nu putem rezerva un bloc fix de memorie suficient de mare pentru a se putea șterge stiva, trebuie implementată stiva ca structură înlanțuită.

Cozi

Organizarea de tip coadă este o formă de listă în care adăugarea (inserarea) se face pe la un capăt al listei, iar ștergerea la celălalt capăt. Astfel spus coada este un obiect de stocare de tip FIFO (primul venit, primul plecat).

Capetele de la care sunt eliminate articolele se numesc cap (head), iar capătul la care se adaugă articole se numește coadă (tail); pentru evitarea confuziilor vom folosi în acest scop termenii de început și sfârșit.

Implementarea cozilor

O metodă este folosirea unui bloc continuu de celule de memorie. În cazul cozilor se efectuează operații la ambele extremități ale structurii, deci se vor folosi doi pointeri: un pointer de început și un pointer de sfârșit.

Presupunem la început, coada goală deci ambii pointeri indică aceeași locație de memorie, figura 6.10.

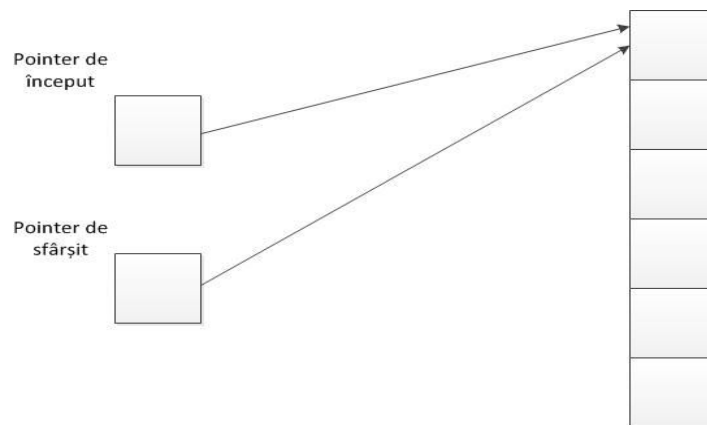


Figura 6.10 - La început coada este goală

La inserarea unui articol, acesta va fi stocat la locația indicată de pointerul de sfârșit după care se va modifica pointerul de sfârșit (va arăta următoarea locație neutilizată) - figura 6.11.

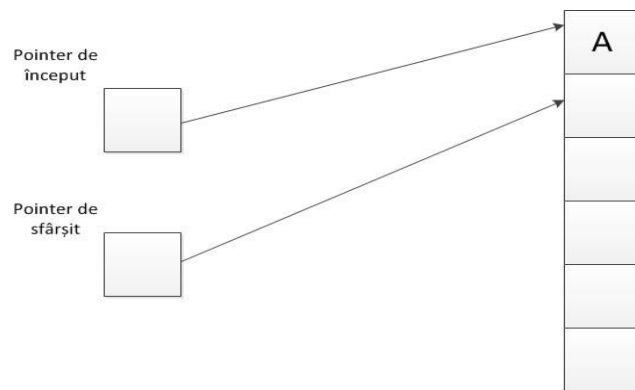


Figura 6.11 - După inserarea unui articol (A)

În același fel se va realiza inserarea a încă un articol - figura 6.12.

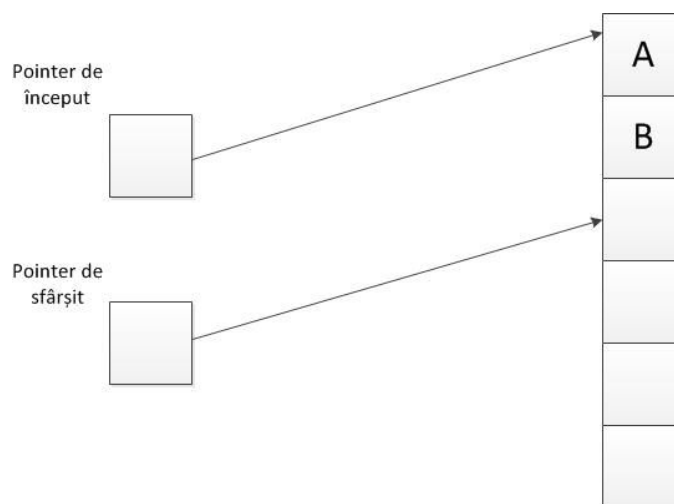


Figura 6.12 - După inserarea încă unui articol

La eliminarea unui articol din coadă, se va extrage obiectul care indică locația indicată de pointerul de început, după care se va ajusta acest pointer așa încât să arate către articolul care urmează după cel eliminat - figura 6.13.

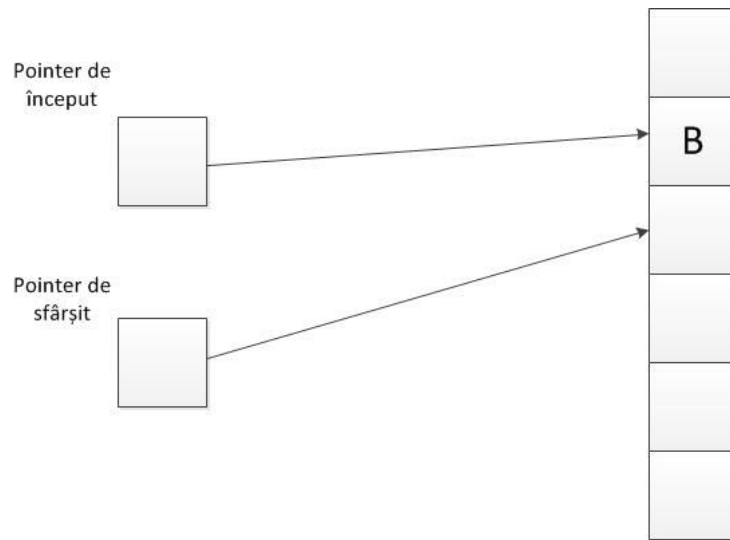


Figura 6.13 - După eliminarea articolului A

În cadrul acestui sistem de stocare, coada tinde să se deplaseze lent prin memorie, distrugând în drumul ei alte date. Această „sete” de memorie este un efect colateral al procedurii de acces.

O soluție la această problemă este ca articolele rămase în coadă vor înainta pe măsură ce articolele de la începutul cozii sunt eliminate. În această situație, ar trebui găsită o metodă de a încadra coada într-o anumită zonă de memorie, fără să fie nevoie de rearanjarea datelor.

Faptic rezervăm un bloc de memorie și plasăm coada, pentru început, la una dintre marginile blocului și îi permitem să migreze către cealaltă margine.

Când sfârșitul cozii atinge marginea blocului, vom începe să inserăm elemente la celălalt capăt al blocului, unde au apărut noi articole, care așteaptă. Astfel coada se va învârti în cadrul blocului rezervat (coadă circulară).

6.3. Arbori

Spre exemplificarea acestei structuri ne vom inspira din diagrama de organizare a unei companii - figura 6.14.

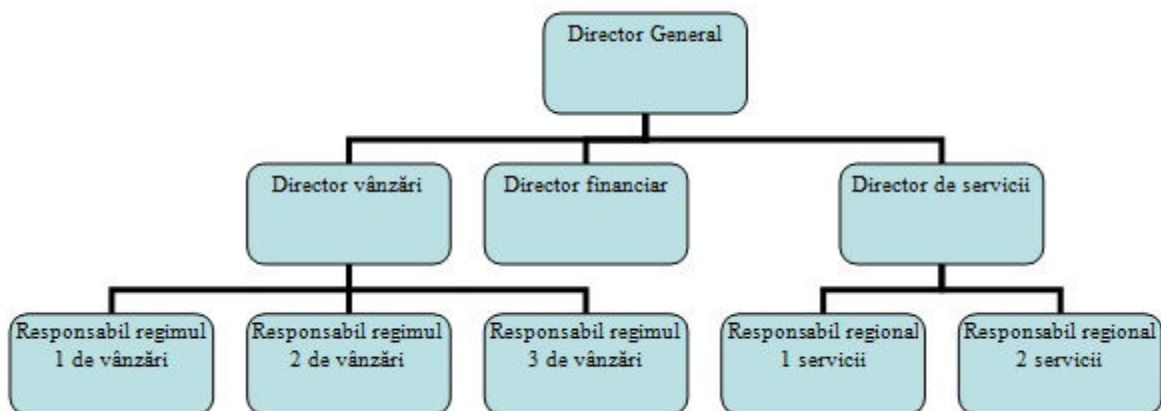


Figura 6.14 - Un exemplu de diagramă de organizare

Fiecare poziție din cadrul arborelui poartă denumirea de nod. Nodul din vârful ierarhiei se numește nodul rădăcină, nodurile de la cealaltă extremitate a arborelui sunt denumite noduri terminale (sau frunze). Linia care unește două noduri se numește „are”. Oricare componentă a unui arbore poartă numele de sub-arbore. Se vorbește de în cadrul acestei structuri de *strămoși* și de *descendenți* unui nod.

Descendenții de la nivelul imediat inferior se numesc *fii*. *Nodul superior* este denumit *nod părinte*; nodurile generate de același nod părinte se numesc frați.

Adâncimea arborelui, reprezintă numărul de noduri care formează cea mai lungă cale de la nodul rădăcină la un nod terminal; *adâncimea arborelui* este egală cu *numărul de straturi orizontale* din componenta acestuia.

Implementarea arborilor

Ne vom rezuma la cazul particular al arborilor binari – arbori în care fiecare nod are cel mult doi descendenți.

În cazul arborilor binari, fiecare artiol (nod) al arborelui conține trei componente:

- datele;
- un pointer către primul descendent (pointer către descendentul stâng)
- un pointer către al doilea descendent (pointer către descendentul drept)

Fiecare nod al arborelui va fi reprezentat într-un bloc continuu de celule de memorie conform figurii 6.15.

Celule de date	Pointer către descendentul stâng	Pointer către descendentul drept
----------------	----------------------------------	----------------------------------

Figura 6.15 - Structura unui nod dintr-un arbore binar

Stocarea arborelui în memorie presupune atât identificarea unor blocuri disponibile în care să se înregistreze nodurile, cât și legarea acestor noduri pentru a se forma structura dorită. Fiecare pointer trebuie definit astfel încât să indice către descendentul stâng/drept al nodului sau să i se atribuie valoarea NIL (dacă nodul nu are descendenți în direcția respectivă).

Evident pentru un nod terminal ambii pointeri sunt NIL. De asemenea trebuie să rezervăm o locație specială pentru adresele nodului rădăcină (pointer către rădăcină).

În figura 6.16 este prezentată structura conceptuală a arborelui, iar în figura 6.17, modul de stocare a structurii în memorie.

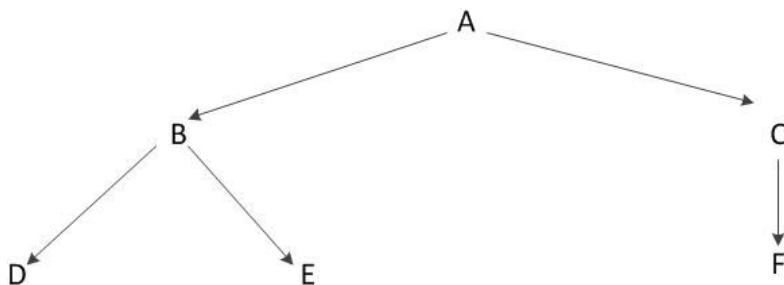


Figura 6.16 - Organizarea conceptuală a arborelui binar

Cu ajutorul sistemului de stocare deschis se va putea identifica nodul rădăcină și se va putea urma orice turneu în cadrul arborelui mergând din nod în nod, cu ajutorul pointerilor.

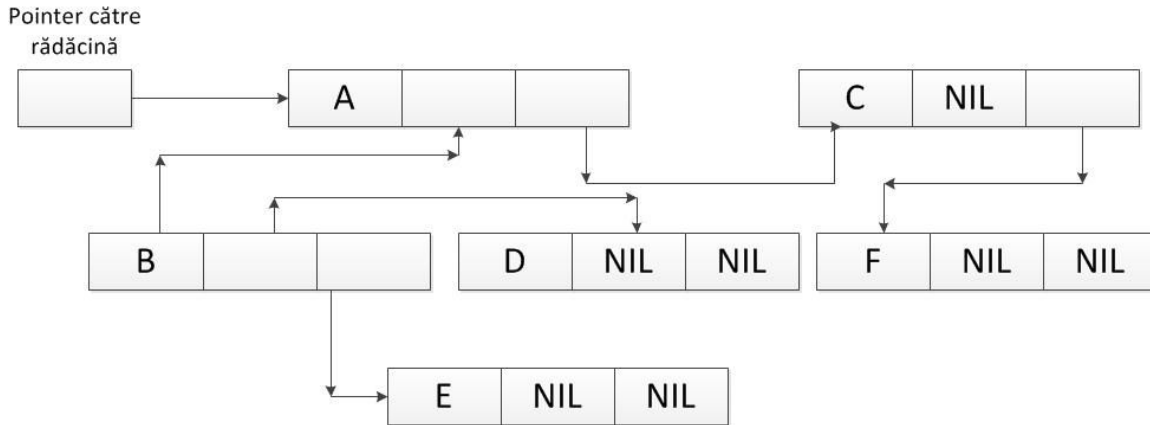


Figura 6.17 - Organizarea reală a unui arbore binar

O metodă alternativă de stocare a arborilor binari își propune rezervarea unui bloc de memorie continuu, stocarea nodului rădăcină în prima celulă (fiecare nod ocupă o singură celulă), stocarea descendentului stâng al nodului rădăcină în a doua celulă, stocarea descendentului drept al nodului rădăcină în a treia rădăcină (în general stocarea descendentului „stâng” respectiv „drept” al nodului din celula n în celulele $2n$, respectiv $2n+1$). Celule neutilizate pentru stocarea structurii arborescente vor avea o valoare particulară, indicând absența datelor – figura 6.18.



Figura 6.18 - Stocarea arborelui din figura 6.16 fără pointeri

Această metodă de stocare oferă un mod convenabil de regăsire a părinților sau fraților oricărui nod. Pentru a regăsi părinții și frații în cadrul structurii înlanțuite, trebuie folosiți pointeri suplimentari.

TEST AUTOEVALUARE 6 (Structuri de date)

1. In contextul structurilor de date, ce reprezinta un vector ?
2. Descrieti implementarea vectorului unidimensional avand in vedere modul de rezervare a blocului de memorie.
3. Descrieti implementarea vectorului multidimensional avand in vedere modul de rezervare a blocului de memorie.
4. In contextul structurilor de date, prin ce parametrii se poate caracteriza o lista ?
5. Explicati succint notiunea de pointer, care apare in cadrul unei liste.
6. Precizati si explicati succint doua dintre problemele care pot apare la actualizarea unei liste contigue.
7. Descrieti implementarea listei inlantuite avand in vedere modul de rezervare a blocului de memorie.
8. Descrieti modul de stergere a unui articol folosind structura de tip lista inlantuita.
9. Descrieti modul de inserare a unui articol folosind structura de tip lista inlantuita.
10. In contextul structurilor de date , ce reprezintă o stivă?
11. Operația de inserare într-o listă poartă denumirea de?
 - a. Push
 - b. Pop
 - c. Insert
12. Descrieți implementarea stivei având în vedere modul de funcționare al sistemului și rezervarea blocului de memorie.
13. Celula suplimentară într-o stivă poartă denumirea de?
 - a. Referință
 - b. Pointer
 - c. Definiție
14. In contextul structurilor de date ,ce reprezintă o coadă?

15. Ce tip de obiect de stocare este o coadă?
- a. LIFO
 - b. FIFO
 - c. Depth-first search
16. Descrieți implementarea cozilor.
17. In contextul structurilor de date , ce reprezintă un arbore?
18. Poziția din cadrul unui arbore poartă denumirea de?
- a. Frunză
 - b. Nod
 - c. Copil
19. Nodurile aflate la extremitățile arborelui sunt denumite noduri?
- a. Frunză
 - b. Părinte
 - c. Copil
20. Descrieți implementarea arborilor.
21. Un nod are?
- a. Descendenți
 - b. Ascendenți
 - c. Copii
22. Ce reprezintă adâncimea unui arbore?
23. Stiva poate fi descrisă astfel:
- a. primul element introdus în stivă este ultimul care poate fi extras
 - b. primul element introdus în stivă este primul care poate fi extras
 - c. ultimul element introdus în stivă este primul care poate fi extras.
24. Eliminarea unui element din stivă se poate face astfel:
- a. se elimină elementul din vârf
 - b. se elimină elementul de la bază
 - c. se poate elimina orice element din stivă

25. Notăm cu VARF poziția pe care se află elementul din vârful stivei , și cu BAZA poziția celui de la capătul opus; stiva este vidă dacă:
- a. $\text{VARF} = \text{BAZA}$
 - b. $\text{VARF} = 0$
 - c. $\text{VARF} < \text{BAZA}$
26. La o coadă adăugarea unui nou element se poate face:
- a. după ultimul element
 - b. în fața primului element
 - c. într-o poziție intermediară
27. In contextul structurilor de date ,când o listă este vidă?
28. In contextul structurilor de date ,ce este o listă circulară?
29. Coadă poate fi descrisă astfel:
- a. primul element introdus în coadă este ultimul care poate fi extras
 - b. primul element introdus în coadă este primul care poate fi extras
 - c. ultimul element introdus în coadă este primul care poate fi extras

UNITATEA DE ÎNVĂȚARE 7

7. Structuri de fișiere

7.1. Fișierele secvențiale

Noțiuni de bază

Organizarea secvențială este, desigur, una conceptuală. Funcție de caracteristicile fizice ale dispozitivului periferic utilizat se poate stoca fișierul respectiv în alt format și să fie prezentat utilizatorului ca un fișier secvențial. Noțiunea de organizare secvențială – aici a înregistrărilor într-un fișier – este legată de dispozitivul periferic și de suportul fizic utilizat (ex. banda magnetică este, prin natura sa, secvențială). Desigur, fișiere secvențiale se pot stoca și pe suportul disc magnetic dispersând înregistrările în diferite locații disponibile (care nu sunt una după alta ca adresare). Sistemele de operare păstrează evidența locațiilor (sectoarelor) ocupate de înregistrări și fișiere și a ordinii în care trebuie citite.

Sfârșitul fiecărui fișier secvențial este indicat de un marcaj de sfârșit de fișier – EDF (*end of file*). În figura 7.1 este prezentată organizarea secvențială a fișierului.

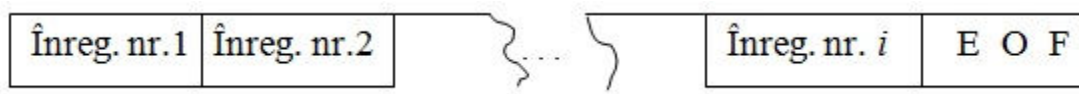


Fig. 7.1. Fișier secvențial

Utilizatorul unui fișier secvențial poate să vadă înregistrările numai în ordine secvențială. Singurul mod de a găsi o înregistrare este acela de a începe căutarea de la începutul fișierului și de a le găsi și extrage în ordinea în care se află în fișier.

Structura secvențială a unui fișier este convenabilă pentru generarea unor rapoarte (ex. statele de salarii), dar aceeași structură secvențială este totală nefavorabilă la actualizarea înregistrărilor. (De exemplu, pentru actualizarea orelor lucrate de un angajat se citește cartela sa de pontaj și apoi se caută în tot fișierul înregistrarea angajatului respectiv pentru actualizare, iar pentru următorul angajat trebuie reluat procesul de la început. Procesul poate fi simplificat dacă ordinea cartelelor de pontaj corespunde cu ordinea înregistrărilor din fișierul angajaților).

Din aceste motive fișierele secvențiale sunt de regulă stocate în ordine alfabetică sau numerică, în funcție de conținutul unui câmp, numit **câmp-cheie** (*key field*). Având în vedere avantajele oferite de o ordine bine stabilită în prelucrarea fișierelor secvențiale, sortarea ocupă un loc important.

Pentru actualizarea unui fișier secvențial, noile informații (în cazul nostru colecția de cartele de pontaj) sunt înregistrate sub forma unui fișier secvențial independent, cunoscut sub numele de **fișier tranzacțional**. Acest fișier tranzacțional este sortat în ordinea în care este sortat fișierul care trebuie actualizat și înregistrările fișierului inițial sunt actualizate în ordinea lor de apariție.

Aspecte legate de programare

În ceea ce privește manipularea fișierelor secvențiale din punct de vedere al programatorului, tendința în cadrul limbajelor de nivel înalt este ca operațiile cu fișiere să se realizeze prin proceduri care sunt fie definite ca parte a limbajului formal, fie furnizate de extensii ale acestuia, sub formă de biblioteci. În ambele situații, parametrii procedurilor indică fișierul-țintă și zona de memorie principală în care se scrie sau din care se citesc înregistrările manipulate.

De exemplu, în Pascal se pot folosi instrucțiuni de forma:

read (Lista poștală, Înregistrare poștală)

și

write (Listă poștală, Înregistrare poștală)

pentru a citi, respectiv a scrie (salva) informații dintr-un/într-un fișier secvențial identificat ca **Listă poștală**. Alături de identificatorul de fișier, în lista de parametri apare și numele **Înregistrare poștală**, utilizat în cadrul programului pentru identificarea blocului de date care se transferă.

Observați că aceste instrucțiuni nu conțin nici o informație explicită cu privire la poziția pe care o au în fișier înregistrările manipulate. Fișierul fiind secvențial, înregistrarea citită este cea care urmează poziției curente în cadrul fișierului, iar înregistrarea care se scrie în fișier este plasată imediat după poziția curentă.

Pe lângă subrutine care permit manipularea înregistrărilor dintr-un fișier secvențial, majoritatea limbajelor de nivel înalt furnizează și instrumente pentru detectarea marcajului de sfârșit de fișier.

7.2. Fișiere de text

Dacă înregistrarea logică dintr-un fișier secvențial se rastrange la, un singur octet, se consideră că avem un **fișier de text** (*text file*). Denumirea fișierului reflectă faptul că aceste fișiere sunt utilizate de obicei pentru stocarea unor documente care conțin text.

Altfel spus, un fișier de text poate fi considerat compus din secvențe de rânduri separate prin marcaje de sfârșit de rând.

Manipularea fișierelor de text

Din motive legate de stocare, fișierele de text sunt împărțite în secvențe de octeți care formează înregistrări fizice a căror dimensiune e compatibilă cu sistemul de stocare utilizat. Manipularea acestor înregistrări fizice este realizată în mod transparent de program.

În realitate, la solicitarea preluării din fișier a primului octet/primul rând de text, programul citește una sau mai multe înregistrări fizice și le memorează într-un *buffer* (zonă de tampon) din memoria principală. Pe măsură ce programul transferă conținutul buffer-ului către utilizator, în *buffer* sunt aduse noi înregistrări și așa mai departe, până la atingerea sfârșitului de fișier.

La scriere în fișier, programul colectează octeții într-un buffer, până când se acumulează acolo o înregistrare fizică întreagă – sau s-a ajuns la capătul fișierului și apoi se trece la transferarea efectivă a înregistrării fizice pe dispozitivul de stocare.

Aspecte legate de programare

Manevrarea fișierelor de text se face cu ajutorul procedurilor predefinite, incluse în definiția formală a limbajului sau în biblioteci oferite ca extensii ale acestuia.

Considerăm **Symbol** o variabilă de tip caracter implicată în instrucțiunea:

read (Vechiul manuscris, Symbol).

Instrucțiunea de mai sus citește un octet din fișierul de text identificat ca **Vechiul manuscris** și atribuie acea valoare variabilei **Symbol**.

Similar, instrucțiunea:

write (Noul manuscris, **Symbol**)

, plasează octetul atribuit variabilei **Symbol** la poziția curentă în fișierul **Noul manuscris**.

7.3. Fișiere indexate

Ideea de indexare este preluată din modul de lucru cu o carte, unde se utilizează un index care permite localizarea unui subiect mult mai repede decât prin parcurgerea secvențială a cărții.

Noțiuni de bază

Indexul fișierului constă dintr-o listă a valorilor care apar în câmpurile-cheie ale fișierului, la care se adaugă poziția înregistrării respective în cadrul dispozitivului de stocare masivă a informațiilor. Acest mod de organizare presupune să cunoaștem valoarea câmpului-cheie corespunzător înregistrării căutate.

Organizarea indexului

Pentru a căuta ceva în înregistrarea identificată prin index este necesar ca indexul să fie transferat în memoria principală și în acest sens dimensiunea lui trebuie să se încadreze în limite rezonabile. O metodă de rezolvare a problemei indecșilor de mari dimensiuni este utilizarea unui index al indexului inițial. Astfel indexul general va lua o formă stratificată sau arborescentă.

Aspecte legate de programare

Foarte puține dintre limbajele actuale de programare de nivel înalt conțin comenzi specifice pentru manipularea fișierelor prin intermediul unui index. Sistemele de gestiunea bazelor de date furnizează instrumente care scutesc programatorul de sarcina de a-și întreține propriile sisteme de fișiere indexate.

Totuși există și la nivelul limbajelor de programare „cărămizile elementare” pentru construirea unui fișier indexat.

De exemplu: Într-un program scris în limbajul **C** se poate utiliza funcția **fsetpos** pentru a stabili poziția curentă în cadrul unui fișier.

Astfel, instrucțiunea:

fsetpos (Personal, & Poziție)

solicită ca poziția curentă în cadrul fișierului **Personal** să fie stabilită la locația identificată prin valoarea variabilei **Poziție**.

Dacă imediat după această instrucțiune se scrie:

fscanf (Personal, „%s”, Nume)

, vom obține numele situat la respectiva poziție din fișier.

7.4. Fișiere dispersate (hashed files)

Sistemul de stocare reprezentat prin fișiere dispersate permite doar accesul direct, fără a utiliza un mod de indexare. Ideea de bază este calcularea poziției fiecărei înregistrări în memorie prin aplicarea unui algoritm (algoritmul de dispersie) asupra valorii câmpului-cheie.

Altfel spus, fiind dat câmpul-cheie căutat, se poate determina rapid poziția înregistrării respective, fără a utiliza o tabelă specială (cazul fișierelor indexate).

Exemplu de tehnică de dispersie

Se împarte zona de memorie de masă alocată pentru fișierul respectiv în mai multe secțiuni, pe care le vom numi **containere**.

Să presupunem că zona de stocare se împarte în 20 de containere. De asemenea, presupunem că înregistrările din fișier vor fi căutate după numărul de identificare și vom stabili acest câmp drept câmp-cheie.

Obiectivul nostru următor este să convertim orice valoare a câmpului-cheie într-o valoare numerică. Acest lucru este oarecum simplu, deoarece vom putea scrie forma binară a fiecărei valori din câmpul-cheie. Apoi utilizând această interpretare numerică, se va împărți această valoare binară a câmpului-cheie la numărul de containere, obținându-se câtul și restul. Aceste resturi vor fi întotdeauna un număr între 0 și 19.

Utilizând acest sistem se poate converti orice valoare din câmpul cheie într-un întreg care identifică unul dintre containerele din memoria de masă, unde vom stoca înregistrarea respectivă.

Rezolvarea situațiilor de depășire superioară

Un fișier dispersat trebuie implementat în ideea că se va ajunge la depășirea capacității containerelor. În acest sens trebuie găsită o metodă de soluționare a situației.

Metoda tipică de rezolvare a problemei este rezervarea unei zone suplimentare de memorie de masă, unde să fie înmagazinate înregistrările care depășesc capacitatea containerelor. Această zonă suplimentară a memoriei de masă se numește zona de depășire.

În concluzie, dacă numărul depășirilor este foarte mare, eficiența acestui tip de fișier scade considerabil. Din acest motiv, proiectarea unui fișier dispersat presupune o atentă alegere a algoritmului de dispersie, a numărului și a dimensiunilor containerelor, precum și a mărimii și structurii zonei de depășire.

Aspecte legate de programare

Limbajele procedurale de nivel înalt folosite în prezent nu oferă o implementare directă a fișierelor dispersate. Acest fapt se datorează atât apariției unor sisteme de gestiunea bazelor de date performante, cât și dependenței de aplicație a detaliilor de proiectare (algoritmul de dispersie, numărul și dimensiunea containerelor).

În limbajul C va fi nevoie doar să se memoreze pozițiile de stocare ale containerelor și să se utilizeze funcția **fsetpos** pentru localizarea containerului indicat de funcția de dispersie.

Tehnicile de dispersie sunt folosite cu mare succes și în distribuția datelor în zone ale memoriei interne principale.

7.5. Rolul sistemului de operare

Mediul de lucru, în cazul limbajelor de nivel înalt, oferă rutine predefinite pentru manipularea fișierelor componente, care nu intră în sarcina programatorului.

Pentru utilizarea operațiilor de regăsire și inserare a înregistrărilor, rutinele comunică cu sistemul de operare, deci acestuia îi revine sarcina manipulării fișierelor. Sistemul de operare trebuie să cunoască structura fișierului, câmpul-cheie și dacă fișierul trebuie salvat.

Funcție de tipul fișierului se pot prelua: poziția curentă în cadrul fișierului, înregistrarea fizică care se află în buffer-ul din memoria principală.

Pentru gestionarea acestor informații, sistemul de operare folosește câte o tabelă – numită **descriptor de fișier** sau **bloc de control** – pentru fiecare fișier utilizat.

Dacă un program lucrează cu trei fișiere, sistemul de operare va trebui să construiască trei descriptori de fișier care să permită gestionarea acestora. În limbajele de nivel înalt, construirea descriptorului de fișier se inițializează printr-o rutină numită **Open**.

În FORTRAN, instrucțiunea tipică are următoarea formă:

```
OPEN (UNIT = 10, FILE = ,Test File', STATUS = OLD,  
      ACCESS = SEQUENTIAL)
```

și solicită sistemului de operare să construiască un descriptor de fișier pentru fișierul cu numele **Test File**.

Parametrii precizați specifică că:

- fișierul va fi indicat ulterior în program ca unitatea logică numărul 10 (UNIT = 10);
- numele fișierului este **Test File** (*File = ,Test File'*);
- sistemul de operare ar trebui să găsească deja fișierul în memoria de masă (STATUS = OLD);
- structura fișierului este secvențială (ACCESS = SEQUENTIAL).

Din exemplul furnizat se observă că un fișier poate fi desemnat ulterior în program prin alt nume decât numele lui propriu-zis. Această distincție care se face între numele extern al fișierului și identificatorul utilizat în program reflectă diferența între regulile sintactice ale sistemului de operare și cele ale limbajului de programare. Această distincție între identificatorii externi și interni de fișier permite, de altfel, un grad mai mare de flexibilitate în sensul că o procedură proiectată să manipuleze fișierul prin intermediul identificatorului intern poate fi utilizată ca rutină generică pentru prelucrarea a diferite fișiere.

După prelucrarea fișierului, multe limbaje de programare cer să fie apelată o rutină numită **Close**. Această rutină informează sistemul de operare că spațiul de memorie rezervat pentru descriptorul de fișier poate fi folosit în alte scopuri. Pentru anumite fișiere (ex. fișierele txt), rutina **Close** are și rolul de a comunica sistemului de operare necesitatea transferării ultimei înregistrări fizice pe dispozitivul de stocare.

TEST AUTOEVALUARE 7 (Structuri de fişiere)

1. De câte tipuri pot fi structurile de fişiere? Enumeraţi-le.
2. Daţi exemple de suporturi fizice specifice organizării secvenţiale.
3. Cum se marchează sfârşitul fiecărui fişier secvenţial?
4. Fişierele secvenţiale sunt stocate în ordine sau, în funcţie de conţinutul unui câmp, numit
5. Singurul mod de a găsi o înregistrare este acela de a:
 - a) începe căutarea de la începutul fişierului;
 - b) folosi o cheie de căutare;
 - c) căuta în arhivă, folosind un algoritm.
6. Ce este un *fişier tranzacţional*?
7. Pentru a citi, respectiv a scrie (salva) informaţii dintr-un/într-un fişier secvenţial identificat ca **Listă poştală**, în Pascal se pot folosi instrucţiuni de forma :
 - a) *read*
 - b) *compare*
 - c) *summ up*
 - d) *write*
8. Cum se numeşte parametrul utilizat în cadrul programului pentru identificarea blocului de date care se transferă:
 - a) Înregistrare poştală;
 - b) Înregistrare fiscală;
 - c) Separare numerică.
9. Ce este un *fişier de text*?
10. Fişierele de text sunt împărţite în :
 - a) secvenţe de octeţi ;
 - b) secvenţe de rânduri separate ;
 - c) secvenţe de unităţi lingvistice.

11. Unde sunt memorate mai întâi înregistrările înregistrările fizice, înainte de transferarea efectivă a acestora pe dispozitivul de stocare:
- a) Într-un buffer;
 - b) Pe un hard-disk extern;
 - c) Pe o partiție separată.
12. Descrieți aspectele legate de programarea fișierelor de text.
13. Ce este *indexul fișierului*?
14. Care este condiția necesară pentru ca un index să devină viabil:
- a) Dimensiunea lui trebuie să se încadreze în limite rezonabile;
 - b) Trebuie să rămână în memoria secundară a calculatorului;
 - c) Trebuie să se regăsească pe ambele partiții.
15. Descrieți etapele de aplicare a funcției **fsetpos** și rolul acesteia (limbajul C).
16. Calcularea poziției fișierelor dispersate în memorie se realizează prin aplicarea unui algoritm numit asupra valorii
17. Descrieți o tehnică de dispersie cunoscută.
18. Ce este o *zonă de depășire*?
19. Care este rolul sistemului de operare?
20. Cum se numește tableta folosită de sistemul de operare pentru gestionarea informațiilor?

UNITATEA DE INVATARE 8

8. Structuri de Baze de date

8.1. Considerații generale

În general, înțelegem prin **bază de date** orice ansamblu de date, dar termenul semnifică de obicei un ansamblu de date stocate pe un dispozitiv magnetic de stocare de masă, având diferite forme de prezentare (organizare), în funcție de necesități și disponibil ca sursă de date pentru o gamă largă de aplicații.

Din alt punct de vedere, o **bază de date** este rezultatul combinării mai multor serii de date (proiectate și culese inițial pentru aplicații diferite) într-un singur ansamblu unificat. Această abordare integrată a conceptului de bază de date își are originea în evoluția istorică a sistemelor automate de stocare și întreținere a datelor.

De exemplu, nevoia calculării salariilor a condus la apariția unor fișiere secvențiale, iar necesitatea regăsirii interactive a datelor a avut ca rezultat realizarea unor fișiere cu acces direct.

Deși fiecare din aceste sisteme de fișiere a reprezentat o îmbunătățire față de procedurile manuale, datorită lipsei de comunicare între ele, totuși permiteau numai o utilizare limitată și neeficientă a resurselor. În aceste condiții, bazele de date au reprezentat soluția adecvată a organizării informațiilor stocate și gestionate de instituții și organizații.

Dacă se implementează o bază centrală de date referitoare la datele solicitate de mai multe compartimente, administrarea acestora se poate executa dintr-o unică poziție cunoscută sub numele de **administrator al bazei de date** (DBA – *database administrator*).

Administratorul știe atât ce date sunt disponibile în cadrul organizației, cât și care sunt datele necesare diferitelor compartimente, putând lua hotărâri referitoare la organizarea și accesul la date. Există și dezavantaje care apar la organizarea datelor sub forma unor baze de date. Una din problemele delicate este **controlul accesului** la datele importante. Controlul asupra accesului la informațiile din baza de date este la fel de important ca și **capacitatea lor de partajare**.

Acordarea de drepturi distincte de acces la sistemele de baze de date se face ținând seama de schema de descriere a întregii structuri a bazei de date. În ultima vreme, dimensiunile și aria de cuprindere a bazelor de date a cunoscut o creștere rapidă, ceea ce a condus la apariția unor grupări foarte mari de date. Aceste grupări de date pot fi fără efort organizate, asamblate și interogate de la distanță. Acest mod de manipulare generează propagarea cu ușurință a informațiilor greșite și apariția incidentelor legate de utilizarea neautorizată sau vicierea intenționată a informațiilor cum ar fi falsificarea creditelor, cazierelor judiciare sau accesul neautorizat la informații personale.

8.2. Implementarea stratificată a bazelor de date

Cel care utilizează o bază de date este numit **utilizator** (*user*), iar uneori **utilizator final** (*end-user*). Acest utilizator final poate fi în egală măsură un funcționar al unei companii aviatice care efectuează rezervări de locuri sau un director executiv al unei mari societăți constructoare de automobile.

În ambele cazuri, probabil că utilizatorul nu este specialist în calculatoare și nici nu trebuie să cunoască în detaliu tehnologia și tehnicile utilizate în domeniu. Utilizatorul trebuie să se poată concentra asupra problemei pe care o are de rezolvat, iar sistemul de gestionare a bazei de date trebuie să îi prezinte informațiile necesare într-o formă specifică aplicației respective, și nu într-un limbaj tehnic greu de înțeles.

Pentru a răspunde la aceste cerințe bazele de date sunt construite pe mai multe niveluri de abstractizare (fig. 8.1).

Imaginea datelor oferită utilizatorului final este produsă de un *software* de aplicație, a cărui comunicare cu utilizatorul se desfășoară în mod interactiv și în termeni specifici programului.

Proiectarea acestui *software* de aplicație imprimă personalitate sistemului de uz general. De exemplu, comunicarea cu utilizatorul se poate face printr-un sistem de întrebări și răspunsuri sau prin formulare care trebuie completate.

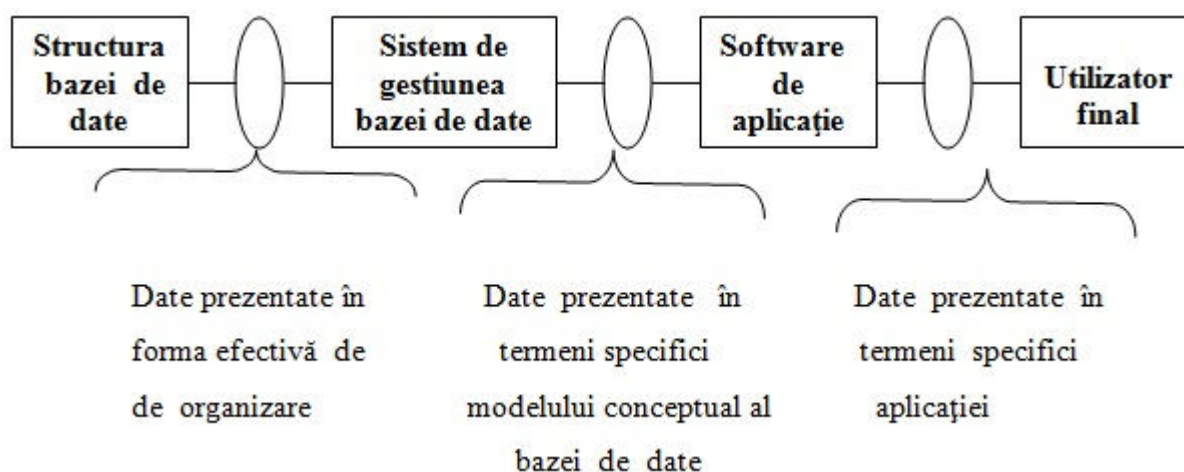


Figura 8.1 - Stratificarea conceptuală a bazelor de date

Indiferent de interfața adoptată, aplicația trebuie să comunice cu utilizatorul pentru a obține de la acesta informațiile necesare (inclusiv cerințele/solicitările), pe care ulterior să le prezinte într-un format cât mai „prietenos” (*friendly user interface*).

De manipularea datelor, în cadrul bazei de date, se ocupă un pachet de programe numit **sistem de gestiunea bazei de date – SGBD** (*data base management system – DBMS*).

Primul avantaj al utilizării acestor sisteme de gestiunea bazei de date – SGBD este **simplificarea activității programatorului aplicației** în ceea ce privește manipularea efectivă a datelor solicitate de aplicație. Această manipulare s-ar complica și mai mult din punct de vedere al rezolvării problemei în contextul unei baze de date distribuite (o bază de date împărțită pe mai multe calculatoare legate în rețea).

Alt avantaj al separării aplicației de sistemul de gestiune a bazei de date este că o astfel de organizare permite **controlul accesului la baza de date**.

Un alt avantaj important al utilizării a două pachete de programe, unul pentru interfața cu utilizatorul, celălalt pentru manipularea efectivă a datelor este obținerea **independenței datelor**. Altfel spus, este posibilă schimbarea organizării bazei de date fără a fi necesară modificarea aplicațiilor.

Și un ultim avantaj al separării între aplicații și SGBD este acela că permite ca **aplicația** să fie **scrisă utilizându-se un model simplificat conceptual**, al bazei de date, fără a lua în considerare structura efectivă a acesteia.

Aplicațiile sunt scrise în limbaje de uz general, dispunând de instrumente necesare exprimării algoritmilor, dar nu și pentru operații care să permită manipularea facilă a bazei de date. Rutinele furnizate de SGBD extind facilitățile limbajului utilizat, permițând folosirea imaginii conceptuale a modelului bazei de date.

Un astfel de limbaj de uz general care se adaugă capacităților SGBD-ului se numește **limbaj de gazdă**.

Căutarea unor model mai performante de baze de date este în plină desfășurare, scopul fiind acela de a se găsi un model care să permită conceptualizarea unor sisteme complexe, să conducă la moduri cât mai concise de solicitare a informațiilor și să poată fi implementat eficient prin furnizarea de către SGBD a unor instrumente abstracte utilizabile în aplicații.

8.3. Modelul relațional

Popularitatea de care se bucură modelul relațional este motivată de simplitatea sa structurală, modelul prezintă datele ca fiind stocate în niște **tabele numite relații**. Modelul relațional permite reprezentarea informațiilor referitoare la salariații unei firme printr-o relație ca cea din tabelul următor:

Marca Salariatului	N u m e l e	A d r e s a	Cod numeric
12 X 95	Ion C. Ion	Șos. Ștefan cel Mare, 56	144051440022
34 A 70	Gh. Vasilescu	Bd. Carol I, 50	152092030124
30 Y 24	Marcela Patache	Str. Dionisie Lupu, 22	260102211128

Figura 8.2 - Relația care conține informații despre angajații unei firme

O linie din cadrul relației (tabelului) poartă numele de **tuplu**. Coloanele relației se numesc **attribute**.

Proiectarea relațională

Proiectarea unei baze de date utilizând modelul relațional se concentrează pe proiectarea relațiilor din componența sa. Să presupunem că pe lângă informațiile conținute în relația din fig. 8.2, dorim să includem și alte informații legate de un istoric al pozițiilor deținute de un salariat în cadrul firmei cu următoarele atribute: denumirea postului (secretar, șef birou, șef compartiment), codul de identificare a postului (unic pentru fiecare post), codul referitor la pregătirea profesională necesară fiecărui post, compartimentul în cadrul căruia a fost deținut postul și perioada de ocupare a postului (data/început și data/sfârșit activitate; în cazul unui post deținut în prezent se va nota acest atribut cu un asterisc).

Un mod de rezolvare a noi probleme este extinderea relației din fig. 8.2, prin adăugarea de noi coloane care să cuprindă noile atribute conform fig. 8.3. La o examinare mai atentă această relație ridică o serie de probleme. Una dintre ele este ineficiența relației: nu mai conține câte un tuplu pentru fiecare salariat, fiind posibil ca un salariat să fi fost avansat în decursul timpului.

Descriind legăturile dintre atribute, ca mai sus, informațiile inițiale se vor repeta, fie: marca, numele, adresa, codul numeric, fie: departamentul, codul pregătire profesională. O altă problemă apare la ștergerea unor informații înregistrate o singură dată în baza de date, cum ar fi codul funcției pentru poziția a treia din tabel (S8 – director compartiment).

Această situație a apărut deoarece am combinat în aceeași relație informații care se referă la lucruri diferite. Modul de rezolvare a problemei este reproiectarea bazei de date folosind atâtea relații câte subiecte am ales. În cazul nostru este vorba de trei subiecte – identificare salariat, locul său de muncă și perioada angajării salariatului pentru fiecare loc de muncă.

Relația identificarea salariatului:

Marca Salariatului	N u m e l e	A d r e s a	Cod numeric
12 X 95	Ion C. Ion	Șos. Ștefan cel Mare, 56	144051440022
34 A 70	Gh. Vasilescu	Bd. Carol I, 50	152092030124
30 Y 24	Marcela Patache	Str. Dionisie Lupu, 22	260102211128

Relația identificarea locului de muncă:

Cod funcție	Denumire funcție	Cod pregătire profesională pentru funcție	Denumire compartiment
J 25 X	Secretară	S C 5	Personal
J 25 Y	Secretară	S C 5	Contabilitate
J 25 Z	Secretară	S C 6	Financiar
S 8	Director departament	D 3	Aprovizionare
.	.	.	.
.	.	.	.
.	.	.	.

Relația perioada angajării salariatului pe fiecare loc de muncă :

Marca salariatului	Codul funcției	Data debut	Data sfârșit
12 X 15	A 6	15.1.93	30.9.95
34 A 70	S 8	1.3.91	*
12 X 15	B 7	1.10.95	*
30 Y 24	J 25 X	1.1.91	30.10.93
30 Y 24	J 25 Y	1.11.93	15.9.96
.	.	.	.
.	.	.	.
.	.	.	.

Figura 8.3 - Bază de date care conține trei relații

Utilizând această bază de date informațiile suplimentare sunt disponibile implicit prin combinarea informațiilor conținute în aceste relații. Câte odată împărțirea atributelor în relații foarte simple duc la pierderea de informații. Pornind de la proprietățile relațiilor s-au putut construi ierarhii de clase de relații numite **prima formă normală**, **a doua formă normală**, **a**

Operații relaționale

Analizăm în continuare operațiile efectuate asupra relațiilor.

Pentru a regăsi informațiile referitoare la un angajat se va selecta din relația **identificarea salariatului** tuplul cu atributul de identificare dorit, iar pentru a obține lista posturilor dintr-un compartiment se vor selecta tuplurile din relația **identificarea locului de muncă** care pentru atributul compartiment au valoarea codului departamentului precizat.

Realizarea selecției tuplului și plasarea sa într-o nouă relație se poate exprima sintactic astfel:

Astfel se creează o nouă relație numită NEW care conține tuplurile din relația **salar**iat, al căror atribut Marca este egal cu 34A70 (fig. 8.4.).

Marca Salariatului	N u m e l e	A d r e s a	Cod numeric
12 X 95	Ion C. Ion	Șos. Ștefan cel Mare, 56	144051440022
34 A 70	Gh. Vasilescu	Bd. Carol I, 50	152092030124
30 Y 24	Marcela Patache	Str. Dionisie Lupu, 22	260102211128

Marca Salariatului	N u m e l e	A d r e s a	Cod numeric
34 A 70	Gh. Vasilescu	Bd. Carol I, 50	152092030124

172

Operația PROJECT extrage anumite coloane. Dacă dorim să aflăm titulaturile posturilor dintr-un compartiment, după ce am efectuat o operație SELECT pentru a extrage din relația LOCURI de MUNCĂ tuplurile care conțin departamentul-țintă plasându-le într-o relație NEW, lista pe care o căutăm este cea a valorilor din coloana *Denumire compartiment* a acestei noi relații.

Pentru a extrage această nouă coloană și a plasa rezultatul într-o nouă relație, utilizăm operația PROJECT, care se va exprima sintactic astfel:

MAIL ← PROJECT Nume, Adresă from SALARIAT

și va realiza obținerea unei liste cu numele și adresele tuturor salariaților firmei. Lista se va găsi în noua relație MAIL care are două coloane (vezi fig. 8.5).

Relația **identificare salariat**

Marca Salariatului	N u m e l e	A d r e s a	Cod număr
12 X 95	Ion C. Ion	Șos. Ștefan cel Mare, 56	144051440022
34 A 70	Gh. Vasilescu	Bd. Carol I, 50	152092030124
30 Y 24	Marcela Patache	Str. Dionisie Lupu, 22	260102211128

NEW ← SELECT from SALARIAT where Marca = „34A70”

Relația **NEW**

N u m e l e	A d r e s a
Ion C. Ion	Șos. Ștefan cel Mare, 56
Gh. Vasilescu	Bd. Carol I, 50
Marcela Patache	Str. Dionisie Lupu, 22

Figura 8.5 - Operația PROJECT

Relația JOIN aplicată asupra a două relații are ca rezultat obținerea unei noi relații ale cărei atribute sunt atributele relațiilor inițiale (fig. 8.6).

Modul în care sunt concatenate tuplurile este determinat de condiția specificată pentru operația JOIN.

$C \longleftarrow \text{JOIN } A \text{ and } B \text{ where } A.W = B.X$

În acest exemplu un tuplu din relația A va fi concatenat cu unul din relația B dacă și numai dacă atributele W și X ale celor două tupluri sunt egale.

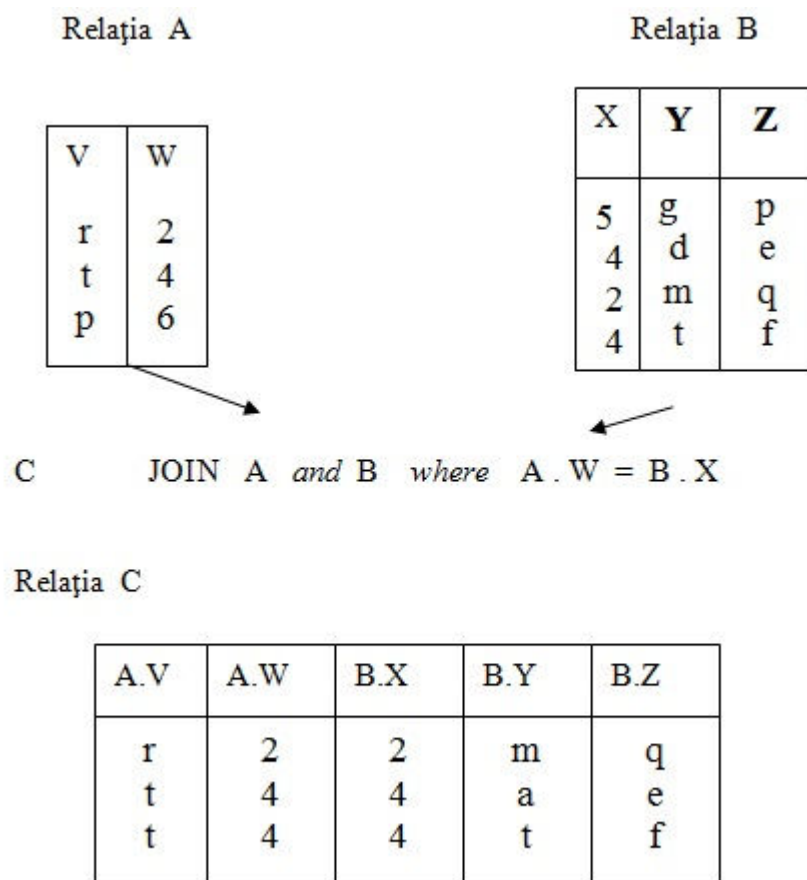


Figura 8.6 - Operația JOIN

Se va exemplifica mai jos operația JOIN asupra bazei de date din fig 8.3 pentru a obține o listă a codurilor de identificare ale tuturor angajaților, împreună cu departamentul în care lucrează fiecare.

Instrucțiunea necesară este :

PLACE 1 ← JOIN ANGAJARE SALARIAT AND LOC DE MUNCĂ unde
ANGAJARE SALARIAT . Cod funcție = LOC DE MUNCĂ .

Cod funcție, rezultatul este prezentat în fig. 8.7. Pentru a rezolva problema **se selectează** mai întâi acele tupluri din relația ANGAJARE SALARIAT Data sfârșit, unde Data sfârșit = „* ”, iar după aceea **se proiectează** attributele:

ANGAJARE SALARIAT Marca salariatului

și

LOC DE MUNCĂ Denumire departament.

În concluzie, obținerea informațiilor se realizează prin executarea instrucțiunilor :

PLACE 1 ← JOIN ANGAJARE SALARIAT AND LOC DE MUNCĂ unde
ANGAJARE SALARIAT . Cod funcție = LOC DE MUNCĂ .Cod funcție

PLACE 2 ← SELECT *from* PLACE 1 unde
ANGAJARE SALARIAT Data sfârșit = „* ”

LIST ← PROJECT ANGAJARE SALARIAT, Marca salariat LOC DE MUNCĂ
Denumire departament *from* PLACE 2.

Relația ANGAJARE SALARIAT

Marca salaria t	Codul funcție i	Data debut	Data sfârșit
12 X 15	A6	15.1.93	30.9.95
34 A 70	S 8	1.3.91	*
12 X 15	B 7	1.10.95	*
30 Y 24	J 25 X	1.1.91	30.10.93
⋮	⋮	⋮	⋮

Relația LOC DE MUNCĂ

Codul funcției	Denumire funcție	Cod pregătire profesiona lă funcție	Denumire compartime nt
J 25 X	Secretară	S C 5	Personal
J 25 Y	Secretară	S C 5	Contabilitate
J 25 Z	Secretară	S C 6	Financiar
S 8	Director departame nt	D 3	Aprovizionar e
⋮	⋮	⋮	⋮

PLACE 1 JOIN ANGAJARE SALARIAT AND LOC DE MUNCĂ unde
ANGAJARE SALARIAT . Cod funcție = LOC DE MUNCĂ . Cod funcție

Relația PLACE 1

Marca salaria - tului	Codul funcției	Data debut	Data sfârșit	Codul funcției	Denumire funcție	Cod pre- gătire pro- fesională pt.funcție	Denumire comparti- ment
12 X 15	A6	15.1.93	30.9.95	A 6	Șef birou	S B 5	Aprovizionare
34 A 70	S 8	1.3.91	*	S 8	Director departame nt	D 3	Aprovizionare
12 X 15	B 7	1.10.95	*	B 7	Șef serviciu	S S 4	Aprovizionare
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figura 8.7 - Exemplu de aplicare a operației JOIN

Sistemul de gestiune a bazei de date are rolul de a accepta comenzi exprimate în termenii modelului relațional și de a le transforma în acțiuni ce țin cont de structura efectivă de stocare. Un sistem de gestiune a bazei de date care utilizează un model relațional va include rutine pentru efectuarea operațiilor SELECT, PROJECT și JOIN, rutine care vor fi apelate din aplicație printr-o structură sintactică compatibilă cu limbajul-gazdă.

În realitate sistemele de gestiune a bazelor de date conțin operații care pot fi combinații ale unor pași elementari într-o formă prietenoasă pentru utilizatori. Un astfel de limbaj este limbajul SQL (*Structured Query Language*).

SQL

Limbajul SQL este destinat interogării bazelor de date. Acest limbaj permite ca printr-o singură instrucțiune SQL se poate exprima o interogare care presupune o secvență de operații SELECT, PROJECT și JOIN.

De exemplu interogarea din ultimul exemplu (fig. 7.7) poate fi exprimată în SQL printr-o instrucțiune:

```
select Marca salariat, Denumire departament  
from ANGAJARE SALARIAT, LOC DE MUNCĂ  
unde ANGAJARE SALARIAT . Cod funcție = COD LOC DE MUNCĂ .  
Cod funcție și ANGAJARE SALARIAT . Dată sfârșit = „* ”.
```

Iată câteva exemple în SQL.

Instrucțiunea:

```
select Nume, Adresă  
from IDENTIFICAREA SALARIATULUI
```

generează o listă care conține numele și adresele tuturor angajaților din relația IDENTIFICAREA SALARIATULUI. (Această operație este identică cu operația PROJECT).

Instrucțiunea :

```
select IDENTIFICAREA SALARIATULUI Nume, ANGAJAREA  
SALARIATULUI . Dată debut  
from IDENTIFICAREA SALARIATULUI, ANGAJAREA SALARIATULUI  
where IDENTIFICAREA SALARIATULUI . Cod funcție = ANGAJAREA  
SALARIATULUI .
```

Cod funcție furnizează o listă a tuturor angajaților cu informații privind data angajării.

Pe lângă interogări, limbajul SQL permite și definirea structurii unei relații, crearea de noi relații, ca și modificarea conținutului unor relații existente.

Baze de date orientate spre [pe] obiecte

Una dintre cele mai noi direcții de cercetare în domeniul bazelor de date o constituie aplicarea paradigmei orientate spre obiecte. Direcția este încurajată de următoarele motive:

- independența datelor se poate obține prin încapsulare;
- clasele și moștenirea par concepute dedicat pentru descrierea schemei generale a bazei de date și a schemelor parțiale;
- baze de date constituite din „obiecte inteligente” care pot răspunde direct la întrebările ce li se pun, fără să utilizeze un program supervisor de interogare;
- abordarea orientată pe obiecte elimină anumite restricții inerente altor metode de organizare a bazelor de date.

Mentținerea integrității bazelor de date

Sistemele de gestiune a bazelor de date (SGBD) de „uz personal” sunt în general ieftine și relativ simplu de manipulat. Instalarea lor nu-l implică de utilizator în detalii tehnice legate de implementare. Volumul acestor baze de date este de mărime mică sau medie, iar pierderea sau alterarea informațiilor conținute constituie o neplăcere și nu un dezastru. În general problemele apărute nu afectează de obicei decât câțiva oameni, iar pierderile de ordin financiar sunt desul de mici.

Desigur altfel stau lucrurile cu SGBD-urile de mari dimensiuni (ca volum de informații înmagazinat și manipulat) utilizate în scop comercial. În astfel de sisteme unul din principalele roluri ale sistemului de gestiune este de a veghea la menținerea integrității bazei de date, evitând operațiile efectuate parțial sau acre, acționând într-un mod neprevăzut, care conduc la apariția de informații eronate.

Protocolul *Commit/Rollback*

Transferarea unei sume de bani dintr-un cont în altul presupune retragerea sumei din contul-sursă și adăugarea ei la contul-destinație. În faza intermediară între două etape de actualizare a bazei de date, informațiile din baza de date pot fi contradictorii. Astfel în scurta

perioadă dintre retragerea (scoaterea) banilor dintr-un cont și depunerea lor în celălalt cont, suma respectivă dispare din evidențe.

În cazul bazelor de date de mari dimensiuni în care sunt în curs de execuție foarte multe tranzacții, probabilitatea ca la un moment ales aleator să fie în curs o tranzacție este foarte mare.

În cazul unei defecțiuni, SGBD nu lasă baza de date în stare de contradicție internă. Acest deziderat este dus la îndeplinire prin menținerea unui **jurnal** în care se înregistrează toate activitățile efectuate asupra unei tranzacții.

Punctul în care s-au înregistrat toate etapele tranzacției se numește **punct de angajare** (*commit point*).

În acest fel SGBD-ul dispune de toate informațiile necesare pentru reconstruirea pe cont propriu a tranzacției dacă acest lucru este necesar. În cazul defectării unui dispozitiv, SGBD-ul utilizează informațiile din jurnal pentru a reconstitui tranzacțiile încheiate de la efectuarea ultimei salvări. Dacă apare o problemă înainte ca o tranzacție să atingă punctul de angajare, SGBD-ul se va găsi în situația de a fi executat parțial o tranzacție pe care nu o poate finaliza.

În această situație jurnalul poate fi utilizat pentru **anularea** (derulare înapoi – *roll back*) activităților deja efectuate de tranzacția respectivă. Uneori derularea înapoi a unei tranzacții poate afecta elementele bazei de date care au fost utilizate între timp la alte tranzacții.

Este posibil ca tranzacția anulată să fi actualizat un cont bancar, iar altă tranzacție efectuată între timp să fi utilizat noua valoare. Această situație necesită anularea și a altor tranzacții, ceea ce va avea influență și asupra altora ș.a.m.d. Fenomenul se numește **anulare în cascadă** (*cascading rollback*).

Blocarea

Dacă studiem problema unei tranzacții care se execută în timp ce baze de date se modifică ca urmare a altei tranzacții uneori observăm apariția unor interacțiuni dorite între tranzacții care pot conduce la rezultate eronate. Atunci când o tranzacție efectuează un transfer de fonduri dintr-un cont în altul, în timp ce alte tranzacții calculează valoarea totală a depozitelor, poate apare problema cunoscută sub numele de **însurare incorectă** (*incorrect summary problem*).

Pentru înlăturarea unei astfel de situații, SGBD-ul poate obliga tranzacțiile să se execute integral, una după alta (serial), noile tranzacții plasându-se într-o coadă de așteptare până la finalizarea celor precedente.

Majoritatea sistemelor mari de gestiune a bazelor de date utilizează un modul de coordonare a partajării timpului de calcul între tranzacții.

Pentru evitarea unor anomalii din categoria precizată, modulele de coordonare recurg la un **protocol de blocare** (*locking protocol*), în care elementele bazei de date care sunt utilizate în mod curent într-o tranzacție sunt marcate ca fiind blocate.

Se utilizează două tipuri de blocare – **blocarea partajabilă** și **blocarea exclusivă**, corespunzând celor două tipuri de acces – partajabil și exclusiv. Dacă o tranzacție nu modifică datele, atunci are nevoie de acces partajabil (permite altor tranzacții să citească datele respective).

Dacă tranzacția are ca obiectiv modificarea datelor, atunci are nevoie de acces exclusiv (nu permite accesul altor tranzacții la articolul respectiv).

Pentru tratarea cazurilor în care accesul solicitat de o tranzacție la un articol de date este refuzat, se pot utiliza mai mulți algoritmi. Unul dintre aceștia forțează tranzacția respectivă să aștepte până când cererea poate fi aprobată, dar acest lucru poate conduce la apariția interblocării.

Pentru evitarea interblocării SGBD-urile acordă prioritate primei tranzacții.

Acest protocol este cunoscut sub numele de **protocol de așteptare** (*wound wait protocol*) și asigură efectuarea tuturor tranzacțiilor.

TEST AUTOEVALUARE 8 (Structura bazelor de date)

1. Definiți conceptul de bază de date.
2. O bază de date este:
 - a. Ansamblu de date
 - b. O colecție de mai multe date
 - c. Un fisier
3. Cine utilizează o bază de date?
 - a. Utilizatorii
 - b. Hackerii
 - c. Crackerii
 - d. Functionarii publici
4. Definiți comunicarea cu utilizatorul și stratificarea conceptuală a bazelor de date.
5. SGBD înseamnă:
 - a. Sistem de gestiune al fișierelor
 - b. Sistem de gestiune a bazelor de date
 - c. Data base management system
 - d. Sistem de gestiune al bazelor direcționale
6. O linie din cadrul relației (tabelului) poartă numele de:
 - a. Cvintuplu
 - b. Quadruplu
 - c. Tuplu
7. Coloanele unei relații se numesc:
 - a. Proprietăți
 - b. attribute
 - c. Parametrii
8. Selectarea unui tuplu cu anumite caracteristici dintr-o relație se face cu:
 - a. NEW SELECT FROM NumeTabel WHERE camp = "valoare"
 - b. SELECT from NumeTabel WHERE camp = "valoare"
 - c. FROM NumeTabel (SELECT NumeColoana from NumeTabel2 WHERE camp="valoare")

9. Relația JOIN este folosită pentru a:
 - a. Extrage date dintr-un tabel
 - b. Extrage date din doua tabele in functie de un criteriu de selectare
 - c. Extrage date sub forma de produs cartezian din doua tabele
10. SQL înseamnă:
 - a. Structured Query Language
 - b. Integrated Query Language
 - c. Metadata Quert Language
11. SQL este un:
 - a. Limbaj de interogare
 - b. Limba de programare
 - c. Comanda de batch
12. Definiți conceptul de bază de date orientată pe obiect.
13. Definiți protocolul Commit/Rollback.
14. Definiți conceptul Commit și dați un exemplu.
15. Definiți fenomenul de anulare în cascadă (cascading rollback)
16. Definiți conceptul de roll back.
17. Definiți blocarea și tranzacțiilor.

UNITATEA DE ÎNVĂȚARE 9

9. Sistem informatic și sistem informațional

9.1 Conceptul de informație

Activitatea umană, în cele mai diverse forme ale sale, trebuie să respecte un anumit număr de legi și reguli și este caracterizată prin *entități factice* exprimate fie sub formă de valori numerice, fie ca percepții sau observații nenumerate. Aceste entități factice independente și neevaluate, există în general în număr nelimitat și se numesc **informații**. Obținerea materialului informațional presupune operații de căutare, iar valorificarea lui, în scopul obținerii unor cunoștințe necesită un proces de prelucrare (evaluare, selectare, ordonare, transformare, stocare, transmitere).

Este necesar a se face distincție între noțiunea de *informație* (reprezentând cunoștințe despre o situație, un individ sau un obiect) și noțiunea de *dată*. Deosebirea dintre informație și dată este echivalentă cu deosebirea dintre obiect și modelul său. Informația și data se pot utiliza ca sinonime numai în măsura în care convenim să identificăm un obiect prin modelul său. În general se identifică următoarele niveluri la care poate fi considerată informația:

- **Nivelul sintactic** se referă la un sistem de simboluri și reguli de grupare ale acestora pentru reprezentarea informației în procesul culegerii, transmiterii și prelucrării acesteia. În sistemele informatice modul de reprezentare sintactică a informației este *data* căreia trebuie să-i fie asociate noțiunea de *valoare* împreună cu un sistem de reguli pentru transformarea acesteia în scopul obținerii unor noi date.
- **Nivelul semantic** presupune *semnificația* informației. Sensul informației la nivel semantic este corespondența dintre o dată și un obiect real sau situația pe care o reprezintă această dată.
- **Nivelul pragmatic** este concretizarea informației la necesitățile receptorului, traduse în *importanța și utilitatea ei*. Abordarea pragmatică include probleme legate de conducere, de necesarul de informație și de eficiența sistemelor informaționale. Acest nivel reflectă cel mai fidel procesul de cunoaștere.

Criterii de clasificare a informațiilor

1. *după forma de exprimare a fenomenelor* pe care le reflectă:

- informație *analogică* care caracterizează parametrii cu variație continuă din cadrul proceselor tehnologice;
- informație *cantitativă* sau *numerică* exprimând aspectul cantitativ al fenomenelor;
- informație *calitativă* sau *nenumerică* prezentată într-o varietate de forme, concepte, etc.

2. *după situarea în timp* față de procesul sau fenomenul reprezentat:

- informații *active* cu privire la procese sau fenomene în curs de desfășurare;
- informații *pasive* se referă la procese sau fenomene în curs de desfășurare;
- informații *previzionale*, sunt cele cuprinse în scenarii și fenomene care vor avea loc în viitor oferind modele cantitative și calitative ale activităților care se vor desfășura.

3. *după conținut*:

- informații *elementare* care definesc operații și fenomene indivizibile;
- informații *complexe* constituite din rezultatele agregării rezultatelor elementare pentru a caracteriza un proces sau un fenomen;
- informații *sintetice* rezultând din adăugarea informațiilor elementare de același tip.

Gradul de utilizare al informațiilor

Gradul de utilizare al informațiilor și eficacitatea utilizării lor în diverse activități sunt determinate de indici de calitate specifici:

- **Exactitatea** (*precizia*) reprezintă cantitatea de informație corectă în raport cu întregul volum de informații produs într-o perioadă de timp. O informație eronată poate influența decisiv desfășurarea proceselor, iar rectificarea erorii înseamnă consum de timp și costuri suplimentare.
- **Actualitatea** (*oportunitatea*) exprimă faptul că o informație este utilă într-un anumit moment, legat de desfășurarea în timp a unor fenomene. Informația trebuie frecvent pusă la zi.
- **Utilitatea** unei informații poate face obiectul unui studiu de oportunitate. O informație nu este prin ea însăși utilă sau inutilă, ea este raportată la necesitățile deciziei. În tehnica

de calcul multe informații pot fi înregistrate în fișiere provizorii care vor putea deveni utile prin perfecționarea posibilităților de prelucrare existente.

- **Fiabilitatea** presupune un control sistematic la nivelul informației de bază și a informației rezultate:
 1. *chei de control* (literă, cifră sau grup de două cifre asociate de exemplu la marca de identificare a unui individ, a unei întreprinderi, la un cont bancar sau poștal, la numărul de identificare al unui articol vândut);
 2. *control de verosimibilitate* (verificarea unui principiu, de exemplu cel de egalitate între totalul debitului și totalul creditului);
 3. *control de feed – back* (de exemplu un mesaj primit este reemis spre expeditor, acesta putând verifica corectitudinea recepționării).
- **Completitudinea** – necesitatea de a dispune de cât mai multe sau chiar de totalitatea informațiilor referitoare la un domeniu al activității.
- **Costurile** informațiilor nu trebuie să fie superioare valorii informațiilor obținute (*costuri de noncalitate*).

9.2 Noțiunea de sistem

Definiție

Prin sistem se înțelege orice secțiune a realității în care se identifică un ansamblu de fenomene, obiecte, procese, concepte, ființe sau grupuri, interconectate printr-o mulțime de relații reciproce, precum și cu mediul înconjurător și care acționează în comun în vederea realizării unor obiective bine definite. La un sistem se disting:

- o mulțime de elemente ($e_1, e_2, \dots, e_n$);
- relații interne (endogene) între elemente;
- relații exogene (intrări și ieșiri din sistem);
- variabilitatea în timp (caracterul procesual, dinamic) a sistemelor și relațiilor;
- scopul sau finalitatea sistemului.

Mulțimea relațiilor dintre componentele sistemului, precum și a relațiilor dintre componente și ansamblu, formează *structura sistemului*. Mulțimea caracteristicilor unui sistem, la un moment dat, determină *starea sistemului*. Sistemele se pot clasifica după mai multe criterii:

- a) după *natură*: sisteme naturale și sisteme elaborate;
- b) după *comportament*: sisteme deterministe și sisteme nedeterministe;
- c) după *modul de funcționare*: sisteme deschise și sisteme închise.

Sistem deschis

Un sistem deschis este caracterizat de ieșiri care răspund intrărilor din sistem, dar ieșirile sunt izolate de intrări și nu au nici o influență asupra acestora. Rezultatele acțiunilor trecute nu influențează acțiunile viitoare.

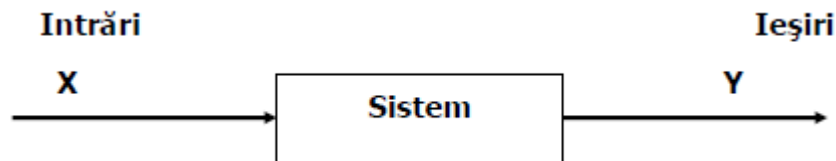


Figura 9.1 – Sistem deschis

Sistem închis

Un sistem închis denumit și *sistem cu conexiune inversă*, (*cu reacție* sau *cu feed-back*) este influențat de propriul comportament:

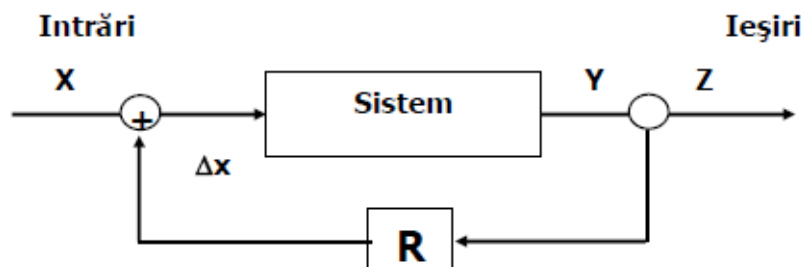


Figura 9.2 – Sistem închis

,unde: $\mathbf{X}$ este vectorul intrărilor;

$\mathbf{Y}$ este vectorul ieșirilor, $\mathbf{Y} = \mathbf{f}(\mathbf{X})$

Dacă se notează cu $\mathbf{Z}$ vectorul obiectivelor atunci $\mathbf{Z} = \mathbf{f}(\mathbf{X} + \Delta \mathbf{x})$.

Valoarea $\Delta \mathbf{x}$ este vectorul de reglare.

Se deosebesc două sisteme cu conexiune inversă:

- Sistemele cu **conexiune inversă negativă** au un obiectiv, iar evoluția lor este o consecință a neatingerii acestui obiectiv. Rolul conexiunii este de a limita anumite mărimi, limitând cauzalitatea intrare + ieșire.
- Sistemele cu **conexiune inversă pozitivă** pentru care ieșirea influențează intrarea în sensul accentuării cauzalității intrare – ieșire. Aceasta generează procese de creștere, în care rezultatul unei acțiuni produce o amplificare continuă a acțiunii (de exemplu beneficiile obținute sunt investite în dezvoltare, rezultă un spor de producție, de beneficii, etc.)

Dacă un sistem poate fi descompus în minimum două părți, în care se pot identifica intrări și ieșiri, astfel încât ieșirile unei părți să constituie intrări pentru cealaltă parte, aceste părți se numesc **subsisteme**. De exemplu, un sistem de producție, într-o reprezentare simplificată poate fi descompus în două subsisteme :

producție și comercial.

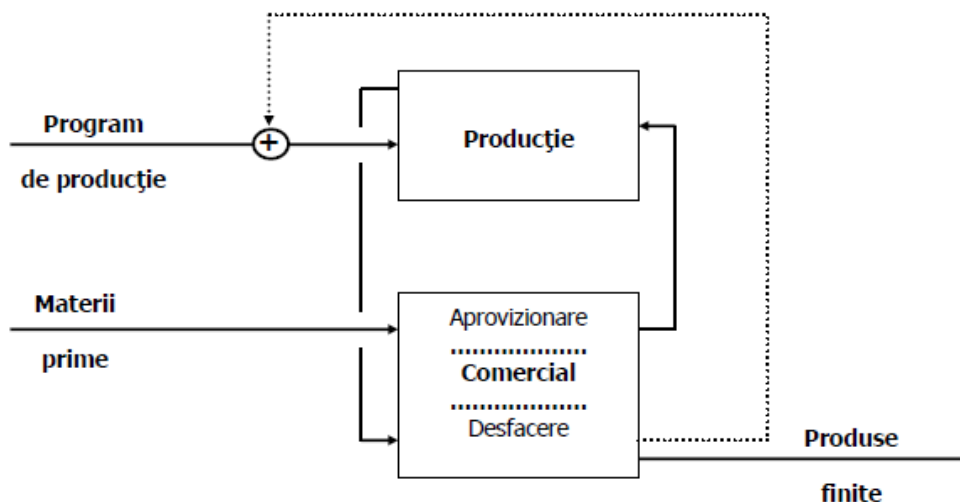


Figura 9.3 - Sistem informațional

Definiție

Sistemul informațional al unui organism economic reprezintă ansamblul informațiilor, surselor și nivelurilor consumatoare, canalelor de circulație, procedurilor și mijloacelor de tratare a informațiilor din cadrul respectivului organism.

Sistemul informațional are rolul de sesizare și colectare a informației de intrare și de difuzare a informației rezultante. Sistemele informaționale, în afară de *informațiile de origine externă* sunt alimentate și de *informații interne* (directivele sistemului de decizie, rezultatele acțiunilor sistemului operant). Observațiile (de *origine internă* sau *origine externă*) sunt captate printr-un anumit număr de observatori:

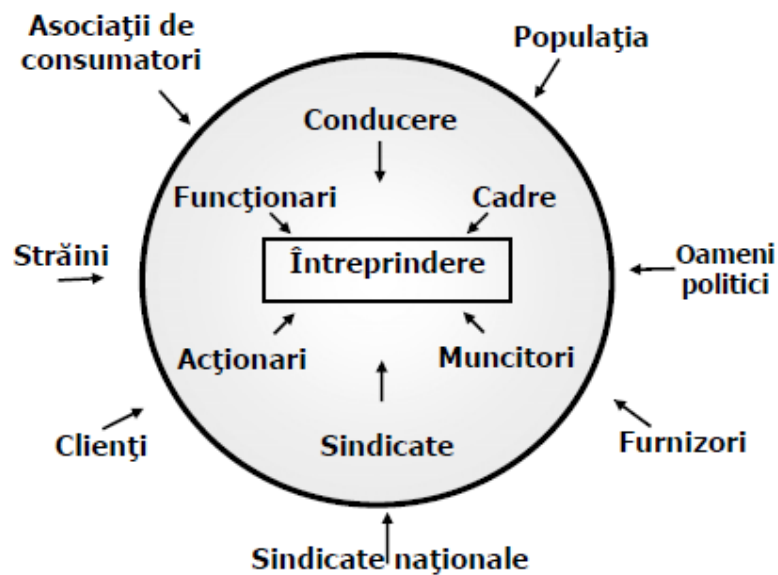


Figura 9.4 – Sistemul informațional al unei unități economice

Sistemul informațional al unei unități economice asigură culegerea, păstrarea, transmiterea și prelucrarea informațiilor necesare luării deciziilor de către sistemul de conducere, cu scopul realizării funcțiilor conducerii asupra nivelului condus. În structura oricărei unități economice se pot identifica trei sisteme corelate între ele:

- *sistemul de conducere (decizional)* constând din mulțimea centrelor sau organismelor unde se analizează informațiile și se elaborează deciziile;
- *sistemul condus (de execuție, operațional)* în care deciziile sunt transpuse în acțiuni;

- *sistemul informațional* care asigură legătura în ambele sensuri între sistemul de conducere și cel condus, într-un sens transmițându-se decizii privind activitatea operațională, iar în celălalt sens informații referitoare la desfășurarea proceselor în sistemul condus.

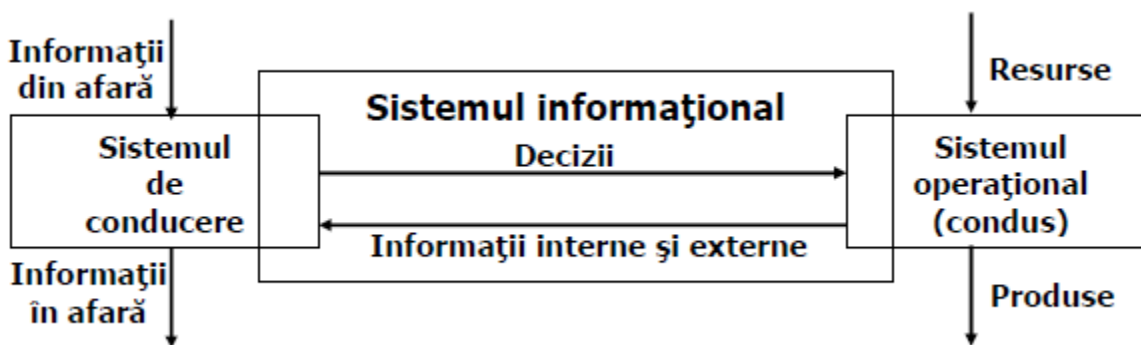


Figura 9.4 – Circulația informațiilor în sistemul informațional

Legăturile și circulația informațiilor în sistemul informațional, care definesc *fluxul informațional*, sunt strâns legate de structura celorlalte două sisteme (sistemul de conducere și sistemul condus). Sistemul informațional al unei unități economice poate fi perfecționat și raționalizat, având în vedere următoarele aspecte:

- **sporirea calității** informației, astfel încât să răspundă cerințelor enunțate;
- **circulația rațională** a informației, prin continuitatea fluxurilor și asigurarea legăturilor inverse în reglarea traiectoriei sistemului;
- **circulația economică** a informației prin eliminarea paralelismelor de informare, a prelucrării repetate;
- **circulația rațională a suporturilor** de date primare prin tipizare și standardizare;
- **finalizarea informației** prin decizie sau acțiune;
- **reducerea costului informației**;
- **adaptarea unor modele matematice** pentru utilizarea optimă a resurselor;
- **asigurarea unității sistemului informațional** prin abordarea integrată a metodelor, tehnicilor și mijloacelor de tratare a datelor.

Principala modalitate de raționalizare a unui sistem informațional este realizarea unui *sistem informatic*.

9.3 Sistem informatic

Definiție

Sistemul informatic este un ansamblu coerent structurat, format din echipamente electronice de calcul și comunicație, procese, proceduri automate și manuale, inclusiv structurile organizatorice și salariații, care folosesc calculatorul ca instrument de prelucrare automată a datelor în domeniul concret de activitate al agentului economic, cu scopul maximizării profitului realizat din activitatea economică.

Un sistem informatic este componenta sistemului informațional în care operațiile de culegere, stocare, prelucrare și transmitere a datelor se realizează cu calculatorul electronic.

Sistemul informatic este conceput să funcționeze la nivelul unui singur agent economic sau grup de societăți comerciale, în vederea asigurării informațiilor complexe, necesare acțiunii manageriale și desfășurării eficiente a întregii activități, cu respectarea cadrului legislativ normativ în vigoare.

Un sistem informatic cuprinde:

- *Baza tehnică (hardware)* – constituită din ansamblul de echipamente pentru culegerea, transmiterea, prelucrarea și stocarea informațiilor.
-

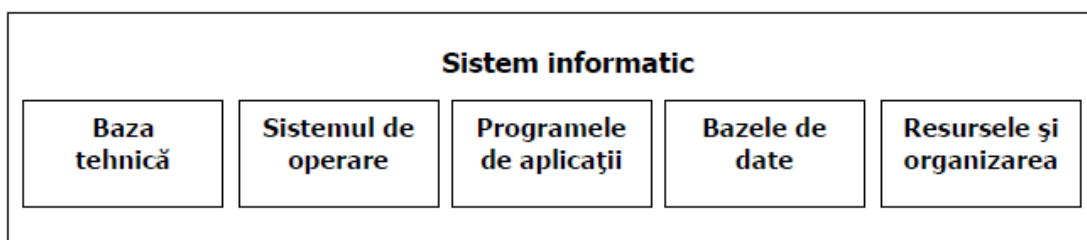


Figura 9.5 – Sistem informatic pentru o unitate economică

- *Sistemul de operare (software)* – cuprinde totalitatea programelor care asigură utilizarea optimă a resurselor fizice.

- *Programele de aplicații* – reprezintă totalitatea programelor care realizează prelucrarea datelor pentru obținerea diferitelor rapoarte (situații).
- *Baza de date* – constituie un ansamblu de date organizat în fișiere interconectate.
- *Resursele umane și cadrul organizatoric* – cuprinde personalul de specialitate și cadrul necesar funcționării sistemului informatic.

Obiectivele utilizării sistemelor informatice

Obiectivul principal al oricărui sistem informatic îl constituie asigurarea selectivă și în timp a tuturor nivelelor de conducere cu informațiile necesare și reale, pentru fundamentarea și elaborarea operativă a deciziilor cu privire la desfășurarea mai eficientă a întregii activități din unitatea economică.

Utilizarea și proiectarea sistemelor informatice trebuie să țină cont de următoarele:

- *Ciclul de viață* – intervalul de timp de la începutul lucrărilor de realizare a unui sistem informatic până la introducerea unui nou sistem mai mare.
- *Cantitatea de date* prin care se încearcă realizarea unor sisteme integrate, care să poată răspunde la cele mai variate cerințe ale conducerii.
- *Numărul de utilizatori*.
- *Aparatul matematic* folosit pentru cercetarea operațională, analiza factorială, teoria stocurilor, teoria așteptării, teoria reînnoirii, etc.
- *Procedurile automate sau manuale* utilizate.
- *Posibilitățile de modificare* impuse de schimbările frecvente legate de cerințele utilizatorilor.
- *Costurile* resurselor materiale, atât în faza de proiectare cât și în exploatare.

Structurarea sistemelor informatice

Structurarea sistemelor informatice pentru unitățile economice se poate face pe subsisteme și în cadrul acestora pe aplicații sau module:

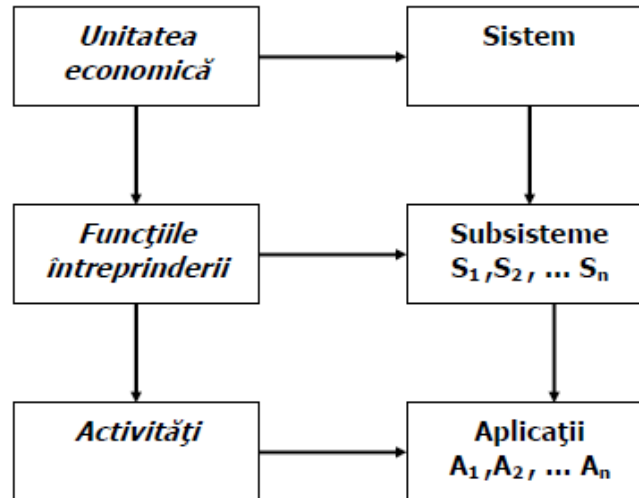


Figura 9.6 – Structurarea unui sistem informatic

La nivelul unei întreprinderi, subsistemele corespunzătoare pot apare astfel:

- A. Subsistemul *cercetare –dezvoltare*.
- B. Subsistemul *producție* cu următoarele componente:
 - a. planificarea tehnico – materială;
 - b. pregătirea tehnică a fabricației;
 - c. programarea, lansarea și urmărirea producției;
 - d. conducerea activității de reparații și întreținere a utilajelor.
- C. Subsistemul *comercial* care cuprinde:
 - a. aprovizionarea tehnico – materială;
 - b. desfacerea produselor finite;
 - c. gestiunea stocurilor;
 - d. organizarea activității de transport.
- D. Subsistemul *financiar contabil*.
- E. Subsistemul *personal*.

O altă posibilitate de structurare a sistemului informatic derivă din corelația stabilită între sistemul informațional și sistemul de conducere, situație în care se poate face și împărțirea în raport cu nivelele de decizie existente în unitatea economică:

- **Subsistemul strategic** pentru rezolvarea problemelor de perspectivă și generale (afectează ansamblul unității economice).
- **Subsistemul tactic** având ca scop rezolvarea problemelor pe o perioadă mai scurtă (sub un an) pe anumite domenii de activitate sau subactivități.
- **Subsistemul operativ** care deservește conducerea curentă la nivel de decadă, săptămână, zi, formație de lucru, schimb, etc.

Clasificarea sistemelor informatice

Clasificarea sistemelor informatice poate fi făcută în raport cu gradul de cuprindere al domeniului sistemului informațional:

- ***Sisteme informatice parțiale*** pentru prelucrarea automată a datelor dintr-un sector de activitate, de regulă cel mai important.
- ***Sisteme informatice totale*** care cuprind toate activitățile informaționale pentru prelucrarea datelor cu ajutorul calculatorului.

Aceste sisteme abordează sistemul ca fiind suma unor subsisteme considerate ca entități distincte care deservește anumite activități, *fără a evidenția legăturile dintre ele* (acestea nu sunt recomandabile, deoarece nu asigură cunoașterea relațiilor de cauzalitate dintre subsisteme și nu permit utilizarea mai eficientă a capacității de prelucrare a calculatorului).

- ***Sisteme informatice integrate*** care abordează procesul de prelucrare a datelor din cadrul sistemului informațional al unității economice, *reliefând legăturile de cauzalitate dintre subsistemele acestuia*. Ele se bazează pe principiul prelucrării, în toate modurile utile, a datelor primare introduse o singură dată în sistem.

Avantajele implementării sistemelor informatice

Datorită extinderii, în ultimii ani, într-o măsură din ce în ce mai mare a tehnologiei informației, a progreselor rapide ale microelectronicii, infrastructura tehnologică a societății a suferit o schimbare importantă prin includerea unui domeniu numit uzual *tehnologia informației* în care se evidențiază, în mod decisiv *informatica*. Acest lucru marchează trecerea de la orientarea industrială, în care accentul se pune pe mașină și energie, la o nouă orientare informațională, în care accentul este pus pe *informație*.

Proiectarea sistemelor informatice la nivel micro și macroeconomic, pe baza unor modele matematice și pe o bună cunoaștere a legilor economice, utilizând tehnica bazelor de date, face posibilă ca activitatea de analiză a fenomenelor economice să devină dintr-un instrument pasiv de constatare într-un instrument activ de previziune și control al acestora. Baza de date este sursa de la care vin și de la care pleacă toate informațiile, de la proces spre punctele de decizie și invers, eliminând redundanța existentă în cele mai multe sisteme informaționale.

În practica dezvoltării sistemelor informatice având ca scop informatizarea activităților economico-sociale s-au produs importante transformări datorită unor schimbări și tendințe cum ar fi:

Informatizarea informației. Inovațiile introduse de informatică, de exemplu în domeniul comunicațiilor, induc o concepere nouă a informației și a conținutului său. Pentru definirea elementelor noi apar, în mod firesc, neologisme cum ar fi: „*bază de date*”, „*bancă de date*”, „*bază de cunoștințe*”, „*inteligență artificială*”, „*hypertext*”, „*hypermedia*”, „*multimedia*”, legate de specificul informaticii și a suportului informatic.

Scăderea costurilor produselor informatice datorită, pe de o parte, reducerii costurilor hardware-lui, iar pe de altă parte, reducerii costurilor software-lui.

Creșterea gradului de generalitate și de adaptabilitate al aplicațiilor ceea ce oferă posibilitatea generalizării implementării sistemelor informatice în mai multe unități economice, pe baza unei platforme comune de aplicații, cu efecte imediate de reducere a costurilor pe unitatea de implementare. În acest sens aplicațiile implementate furnizează funcții software de bază și funcții specifice activității companiei. Prin funcțiile software de bază se definesc și se rezolvă problemele comune aplicației în proporție de circa 80-90%, iar prin soft-ul specific aplicației, se definesc proprietățile comportamentale suplimentare companiei. De multe ori însă, gradul de generalitate al produselor informatice este prea mare, astfel încât adaptabilitatea lor la necesitățile utilizatorului devine aproape imposibilă, acesta recurgând, în mod firesc, la realizarea propriului produs informatic.

Dezvoltarea telecomunicărilor prin apariția de noi produse cu un raport preț/performanță din ce în ce mai avantajos care au făcut rentabilă și necesară conectarea între ele a calculatoarelor în cadrul unor **rețele de calculatoare**. Principalul obiectiv al constituirii rețelelor de calculatoare este de a permite transmiterea datelor la distanță dar și acela al partajării (punerii în comun) unor resurse hardware și software. Sistemele telecomunicărilor au dimensiuni din ce în

ce mai mari, rețelele de calculatoare se pot interconecta putând conține și componente eterogene - calculatoare din familii diferite, platforme diferite și producători diferiți.

Introducerea standardelor internaționale, prin elaborarea de către *ISO* (*International Standard Organization*) a unor modele de referință pe baza conceptului de "*sistem deschis*", care să permită asigurarea unei baze comune pentru coordonarea elaborării de noi standarde (cum ar fi interconectarea sistemelor eterogene). Termenul de sistem deschis se referă doar la recunoașterea reciprocă și aplicabilitatea acelorași standarde. Standardizarea asigură o creștere a gradului de portabilitate, de pe o platformă de sistem la alta, atât a datelor cât și a produselor software, astfel încât producătorii se simt încurajați să le implementeze, având în vedere larga circulație a acestor standarde. Standardizarea în domeniul sistemelor de gestiune a bazelor de date sau în domeniul limbajelor de programare conform unor standarde precum *CODASYL* și *ANSI* au impus folosirea unor limbaje cum ar fi *SQL* și *C* datorită performanțelor și facilităților pe care le oferă.

Extinderea bazelor de date clasice bazate pe text și valori numerice spre *baze de date orientate obiect*. Bazele de date clasice sau relaționale oferă prea puțin suport teoretic și practic pentru tipurile neconvenționale de date. Bazele de date orientate obiect permit crearea de obiecte complexe din componente mai simple, fiecare având propriile attribute și propriul comportament, reușind să ofere noi soluții pentru rezolvarea problemelor și crearea unor aplicații moderne.

Orientarea spre multimedia. Transpunerea unei părți a informației pe un suport multimedia conjugă interactivitatea cu atracția vizuală, oferă un nou potențial în materie de informare și este de un interes evident în toate domeniile. În raport cu utilitățile tradiționale de utilizare a informației, multimedia oferă următoarele avantaje: *facilitatea de utilizare* (interfața utilizator, grafica, audio, video), *independența de utilizator* (parcurs personalizat, independența în raport cu un grup), *interactivitatea* (atractivitate, convivialitate).

TEST AUTOEVALUARE 9 (Sistem Informațional și Sistem Informatic de fișiere)

1. Definiți conceptul de informație.
2. Ce presupune obținerea materialului informațional?
 - a. Operații de adunare și scădere a informației
 - b. Operații de căutare
 - c. Operații de introducere a unei noi cantități de informație
3. Comentați nivelurile la care poate fi considerată informația.
4. Definiți nivelul sintactic.
5. Definiți nivelul semnatic.
6. Definiți nivelul pragmatic.
7. Nivelul sintactic se referă la un sistem?
 - a. Eterogen
 - b. Binar
 - c. De simboluri
8. Modul de reprezentare sintactică a informației în sistemele informatice sunt?
 - a. Date
 - b. Numere
 - c. Structuri de date
9. Nivelul semantic presupune?
 - a. Semnificația informației
 - b. Semnificația structurilor de informații
 - c. Structuri de arbori
10. Nivelul pragmatic reprezintă?
 - a. Concretizarea informației
 - b. Diminuarea informației
 - c. Blocarea informației
11. Definiți noțiunea de sistem.
12. Ce este un sistem deschis?
13. Ce este un sistem închis?

14. Definiți sistemul informațional.
15. Definiți noțiunea de sistem informatic.
16. Argumentați clasificarea sistemelor informatice.
17. Care sunt avantajele implementării sistemelor informatice.

BIBLIOGRAFIE

1. Andrew Tanenbaum, Organizarea structurată a calculatoarelor, ISBN: 978-9-738669-91-8 (ISBN10: 9-738669-91-X).
2. Andrew Tanenbaum, Rețele de calculatoare, ISBN: 978-9-730030-00-6 (ISBN10: 9-730030-00-6).
3. Andrew Tanenbaum, Sisteme de operare moderne, ISBN: 978-9-738669-92-5 (ISBN10: 9-738669-92-8) 2004.
4. Thomas Cormen, Introducere în algorimi, ISBN: 978-9-739753-47-0 (ISBN10: 9-739753-47-7) 2004.
5. Knuth D.E., Arta programării calculatoarelor, Editura TEORA, ISBN: 1-59496-098-4