

**Alexandru Vancea**

**Darius Bufnea**

**Adrian Dărăbant**

**Florian Boian**

**Anca Andreica**

**Andreea Navroschi**

**Arhitectura calculatoarelor.**

**Limbajul de asamblare 80x86.**

**Editura RISOPRINT  
Cluj-Napoca • 2014**



## C U P R I N S

<b>1. REPREZENTAREA DATELOR.....</b>	<b>1</b>
1.1. Tipuri de date elementare.....	1
1.2. Numere întregi.....	2
1.2.1. Baze de numerație.....	2
1.2.2. Conversii între baze de numerație.....	3
1.2.3. Conversii rapide între bazele 2, 8, 16.....	5
1.3. Reprezentări binare și ordini de plasare.....	7
1.3.1. Dimensiune a reprezentării.....	7
1.3.2. Organizarea și memorarea datelor.....	9
1.3.2.1. Bit, octet, locație, adresă.....	9
1.3.2.2. Tipuri elementare de date: dimensiuni ale standardelor de reprezentare.....	12
1.3.2.3. Ordinea octețiilor într-o locație; mașini little-endian și mașini big-endian.....	13
1.3.2.4. Unități de capacitate a memoriei.....	17
1.4. Codificarea caracterelor.....	18
1.5. Codificarea numerelor întregi.....	20
1.5.1. Convenție cu semn și convenție fără semn.....	20
1.5.2. Bitul de semn; codul complementar.....	21
1.5.2.1. Reguli alternative de complementare.....	22
1.5.3. Operații aritmetice; conceptul de depășire.....	25
1.5.3.1. De ce codul complementar?.....	25
1.5.3.2. Conceptul de depășire.....	26
1.5.3.3. Adunări și scăderi.....	26
1.5.3.4. Înmulțiri și împărțiri.....	30
1.5.4. Conversia la o locație de alte dimensiuni.....	32
<b>2. ARHITECTURA SISTEMELOR DE CALCUL.....</b>	<b>35</b>
2.1. Definiții. Organizarea unui sistem de calcul.....	35

2.2. Unitatea centrală (Central Processing Unit - CPU).....	38
3.2.1.4. Operanzi cu adresare indirectă.....	78
3.2.2. Utilizarea operatorilor.....	79
3.2.2.1. Operatori aritmétici.....	80
3.2.2.2. Operatori de indexare.....	81
3.2.2.3. Operatori de desplasare de biți.....	81
3.2.2.4. Operatori logici pe biți.....	81
3.2.2.5. Operatori relativnați.....	82
3.2.2.6. Operatori de specificare a segmentului.....	82
3.3.1. Directive standard pentru definirea segmentelor.....	85
3.3.1.1. Directive SEGMENT.....	85
3.3.1.2. Directive ASSUME și gestiunea segmentelor.....	85
3.3.3. Directive LABEL, EQU, =.....	97
3.3.4. Directive PROC.....	99
3.3.5. Blocuri repetitive.....	99
3.3.6. Directive INCLUDE.....	101
3.3.7. Macroouri.....	102
4. INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE.....	107
4.1. Manipularea datelor.....	108
4.1.1. Instrucțiuni de transfer al informației.....	108
4.1.1.1. Instrucțiuni de transfer de uz general.....	108
4.1.1.2. Instrucțiuni de transfer de interesi.....	112
4.1.1.3. Instrucțiuni de transfer al adreselor.....	113
4.1.1.4. Instrucțiuni asupra flagurilor.....	116
4.1.2. Instrucțiuni de conversie.....	117
4.1.3. Instrucțiuni asupra accesării datelor.....	119
4.1.4. Instrucțiuni de reprezentare în little-endian sau pe același plan.....	122
4.2. Operatiile.....	77
3.2.1.1. Utilizarea operatorilor imediați.....	73
3.2.1.2. Utilizarea operatorilor regiștri.....	76
3.2.1.3. Utilizarea operatorilor imediați în memorie.....	77
3. ELEMENTELE LIMBAJULUI DE ASAMBLARE.....	69
3.1. Formatarea limbii sursei.....	71
3.2. Eșpresii.....	73
3.2.1. Moduri de adresare.....	74
3.2.1.1. Utilizarea operatorilor imediați.....	74
3.2.1.2. Utilizarea operatorilor regiștri.....	76
3.2.1.3. Utilizarea operatorilor imediați în memorie.....	77
3.2.2. Calculul offsetului unui operand. Moduri de adresare.....	67
3.2.2.1. Adresă FAR și NEAR.....	66
3.2.2.2. Reprezentarea instrucțiunilor masive.....	66
3.2.2.3. Reprezentarea adresei și calculul de adresa.....	64
3.2.2.4. Reprezentarea în generală EU.....	62
3.2.2.5. Flagurile.....	63
3.2.2.6. Structura microprocesorului.....	69
3.2.2.7. Arhitectura microprocesorului 8086.....	69
3.2.2.8. Arhitectura și performanțele unității de calcul.....	61
3.2.2.9. Componentele unității de calcul.....	61
3.2.2.10. Performanțele unității de calcul.....	61
3.2.2.11. Directiva SEGMENT.....	65
3.2.2.12. Directiva ASSUME și gestiunea segmentelor.....	67
3.2.2.13. Directive pentru definirea segmentelor.....	67
3.2.2.14. Directive LABEL, EQU, =.....	94
3.2.2.15. Directive PROC.....	99
3.2.2.16. Directive repetitive.....	99
3.2.2.17. Directive INCLUDE.....	101
3.2.2.18. Macroouri.....	102
2.6. ARHİTECTURA LIMBAJULUI DE ASAMBLARE.....	107
2.6.1. Structura microprocesorului.....	19
2.6.2. Regelești generali EU.....	22
2.6.3. Flagurile.....	23
2.6.4. Reprezentarea adresei și calculul de adresa.....	24
2.6.5. Reprezentarea instrucțiunilor masive.....	25
2.6.6. Adresă FAR și NEAR.....	26
2.6.7. Calculul offsetului unui operand. Moduri de adresare.....	27
2.7. FORMATAREA LIMBAJULUI DE ASAMBLARE.....	69
2.7.1. Formatarea limbii sursei.....	71
2.7.2. Eșpresii.....	73
2.7.3. Moduri de adresare.....	74
2.7.4. Utilizarea operatorilor imediați.....	74
2.7.5. Utilizarea operatorilor regiștri.....	76
2.7.6. Utilizarea operatorilor imediați în memorie.....	77

<b>4.2.1. Operații aritmetice.....</b>	<b>122</b>	<b>5.2.3. Întreruperi software.....</b>	<b>187</b>
4.2.1.1. Adunarea și scăderea.....	122	5.3. Câteva observații asupra întreruperilor 8086.....	192
4.2.1.2. Înmulțirea și împărțirea.....	126	5.4. Instrucțiuni specifice lucrului cu întreruperi.....	193
4.2.1.3. Exemple și exerciții propuse.....	129	5.5. Formatele COM și EXE.....	195
<b>4.2.2. Operații logice pe biți.....</b>	<b>131</b>	5.5.1. Prefixul unui program executabil (PSP).....	195
4.2.3. Deplasări și rotiri de biți.....	133	5.5.2. Structura unui program EXE.....	197
<b>4.3. Ramificări, salturi, cicluri.....</b>	<b>138</b>	5.5.3. Structura unui program COM.....	202
4.3.1. Saltul necondiționat.....	138	5.5.4. Depanarea programelor .EXE și .COM.....	205
4.3.1.1. Instrucțiunea JMP.....	139		
4.3.2. Instrucțiuni de salt condiționat.....	144	<b>6. REDIRECTAREA ÎNTRERUPERILOR.....</b>	<b>207</b>
4.3.2.1. Comparații între operanzi.....	144	6.1. Redirectarea întreruperilor.....	207
4.3.2.2. Salturi condiționate de flaguri.....	145	6.2. Programe TSR.....	210
4.3.2.3. Exemple comentate.....	148	6.3. Harta de memorie DOS și programele TSR.....	210
4.3.3. Instrucțiuni de ciclare.....	162	6.4. TSR-uri active și TSR-uri pasive.....	214
4.3.4. Instrucțiunile CALL și RET.....	164	6.4.1. Problema funcțiilor DOS non reentrant.....	219
<b>4.4. Instrucțiuni pe șiruri.....</b>	<b>166</b>	6.4.2. Problema întreruperilor BIOS non reentrant.....	223
4.4.1. Generalități privind șirurile și instrucțiunile pe șiruri.....	166	6.5. Întreruperea multiplex (INT 2Fh).....	224
4.4.2. Instrucțiuni pe șiruri pentru transferul de date.....	168	6.6. Instalarea unui TSR.....	225
4.4.3. Instrucțiuni pe șiruri pentru consultarea și compararea datelor.....	170	6.7. Dezinstalarea unui TSR.....	227
4.4.4. Execuția repetată a unei instrucțiuni pe șiruri.....	173	6.8. TSR monitor tastatură.....	229
4.4.5. Utilizarea de operanzi pentru instrucțiuni pe șiruri.....	174	6.9. Depanarea programelor TSR.....	244
4.5. Un exemplu complet de program.....	175	6.10. TSR-uri și redirectare întreruperi în cadrul SO Windows.....	249
<b>5. ÎNTRERUPERI.....</b>	<b>181</b>		
5.1. Probleme generale privind întreruperile.....	181	<b>7. IMPLEMENTAREA APELULUI DE SUBPROGRAME.....</b>	<b>251</b>
5.2. Clasificarea întreruperilor.....	182	7.1. Cod de apel, cod de intrare, cod de ieșire.....	251
5.2.1. Întreruperi hardware.....	183	7.1.1. Cod de apel.....	251
5.2.2. Excepții.....	185	7.1.2. Cod de intrare.....	252
		7.1.3. Cod de ieșire.....	255
		7.2. Implementarea subprogramelor în Turbo Pascal.....	256
		7.2.1. Modele de memorie.....	256

7.2.2. Hărta memoriei Turbo Pascal.....	257
7.2.3. Cod de apel al subprogramelor Pascal.....	259
7.2.3.1. Tipuri de apel al subprogramelor.....	260
7.2.3.2. Cod de apel al subprogramelor.....	261
7.2.3.3. Exemplu.....	262
7.2.4. Cod de intrare în subprogramele Pascal.....	262
7.2.4.1. Împlementarea rezultatului de către funcții.....	262
7.2.4.2. Exemplu.....	262
7.2.5. Cod de ieșire din subprogramele Pascal.....	264
7.2.6. Proceduri și funcții imbricate în Turbo Pascal.....	265
8.3.2.6. Utilizarea direcțiilor de segment simplificate.....	269
8.3.2.7. Exemplu 1.....	271
7.3.1. Puncte fixe și near.....	267
7.3.2. Cod de apel subprogramelor C.....	268
7.3.3. Cod de imprimare în subprogramele C.....	269
7.3.4. Cod de ieșire din subprogramele C.....	270
8. PROGRAMAREA MULTIMODULI.....	271
8.1. Directiva MODEL. Directiva de segment simplificate.....	271
8.2. Cerințele unei module asamblabile la legătură cu un alt modul.....	274
8.2.2. Directiva EXTRN.....	275
8.2.3. Directiva GLOBAL. Legătura de module asamblabile.....	276
8.3. Legea care se aplică unei module asamblabile scrise în limbaj de nivel înalt.....	278
8.3.1. Etapele legării unei module asamblare cu un modul scris în limbaj de nivel înalt.....	278
8.3.1.1. Cerințe ale editorului de legătură.....	278
8.3.1.2. Intrarea în procedura.....	279
8.3.1.3. Negrile care valoarea unor registri.....	279
8.3.1.4. Transmiterea și accesarea parametrilor.....	279
8.3.1.5. Allocarea de spațiu de memorie pentru datele locale.....	281
9.2.1.1. Înstrucțiunile assembly.....	312
9.2.1.2. Înstrucțiunile assembly.....	312
9.2.1.3. Expressii.....	315
9.2.1.4. Proceduri și funcții assemble.....	321
9.2.1.5. Exemplu.....	322

7.2.2. Hărta memoriei Turbo Pascal.....	282
7.2.3. Cod de apel al subprogramelor Pascal.....	282
7.2.3.1. Transmisarea parametrilor.....	283
7.2.3.2. Tipuri de apel al subprogramelor.....	285
8.3.2. Interfața dintre Turbo Assembler și Turbo Pascal.....	285
8.3.2.1. Directiva de compilare \$L și subprogramele external.....	285
8.3.2.2. Reguli de utilizare a regiștilor.....	288
8.3.2.3. Transmiterea și accesarea parametrilor.....	288
8.3.2.4. Împortarea rezultatului de către funcții.....	289
8.3.2.5. Allocarea de spațiu pentru datele locale.....	289
8.3.2.6. Utilizarea direcțiilor de segment simplificate.....	289
8.3.2.7. Exemplu 1.....	291
8.3.2.8. Exemplu 2.....	293
8.3.3. Interfața dintre Turbo Assembler și Borland C++.....	301
8.3.3.1. Cerințe ale editorului de legătură de definiție modulului asamblare.....	301
8.3.3.2. Transmiterea parametrilor.....	304
8.3.3.3. Împortarea de valori.....	306
8.3.3.4. Convorbiri privind utilizarea regiștilor.....	306
8.3.3.5. Exemplu.....	306
9.1. Încarcarea de cod masivă în textul surșă Pascal.....	309
9.1.1. Înstrucțiunea inline.....	309
9.1.2. Directiva inline.....	311
9.2. Asamblarea în linie.....	312
9.2.1. Asamblarea în linie al lui Borland Pascal 6.0.....	312
9.2.1.1. Înstrucțiunea ASM.....	312
9.2.1.2. Înstrucțiunile assembly.....	312
9.2.1.3. Expressii.....	315
9.2.1.4. Proceduri și funcții assemble.....	321
9.2.1.5. Exemplu.....	322

<b>9.2.2. Asamblorul inline al lui Borland C++.....</b>	<b>327</b>
9.2.2.1. Instrucțiunea asm.....	327
9.2.2.2. Exemplu.....	330
<b>9.3. Proceduri și funcții imbricate în Borland Pascal.....</b>	<b>331</b>
<b>9.4. Accesarea regiștrilor și apelarea de întreruperi.....</b>	<b>335</b>
9.4.1. Borland Pascal 6.0.....	335
9.4.2. Borland C++.....	336
<b>9.5. Scrierea de rutine de tratare a întreruperilor în limbajele Pascal și C.....</b>	<b>337</b>
9.5.1. Proceduri interrupt în Pascal.....	337
9.5.2. Funcții interrupt în C.....	340
<b>10. EXTENSII x86.....</b>	<b>341</b>
10.1. Memoria înaltă ( <i>high memory</i> ) și memoria extinsă.....	342
10.2. Procesorul 80386 și modul de lucru protejat.....	355
10.3. Noi instrucțiuni aduse de urmașii procesorului 8086.....	361
10.4. Exemplu de program care lucrează cu procesorul în mod protejat.....	365
<b>11. PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS.....</b>	<b>371</b>
11.1. Modul de adresare protejat și microprocesoare pe 32 de biți.....	371
11.2. Programare în limbaj de asamblare sub Windows.....	372
11.3. Microsoft Macro Assembler.....	373
11.4. Modalități de folosire a limbajului de asamblare sub Windows.....	374
11.4.1. Programe scrise în întregime în limbaj de asamblare.....	374
11.4.2. Inserare de cod sursă asamblare în cadrul limbajelor de nivel înalt (studiul de caz – Visual C++).....	382
11.4.3. Programare multimodul.....	384
11.5 Limbaj de asamblare vs. limbi de nivel înalt.....	390

## Cap.1. Reprezentarea datelor.

### C A P I T O L U L 1

#### REPREZENTAREA DATELOR

##### 1.1. TIPURI DE DATE ELEMENTARE

Un element esențial în utilizarea unui sistem de calcul este cunoașterea tipurilor de date primitive cu care lucrează sistemul și modul în care acesta și le reprezintă. Datele care se prelucrează sunt fie numerice, fie nenumerice. La rândul lor, prelucrările numerice le putem separa în calcule numai asupra numerelor întregi și calcule în care numerele inițiale, rezultatele intermediare și finale nu sunt neapărat întregi. Deși a doua categorie de calcule o include pe prima, în sistemele de calcul ele sunt tratate ca și două categorii distincte, din rațiuni care vor fi lămurite mai târziu. Să exemplificăm aceste trei categorii de date.

Crearea și întreținerea listei studenților dintr-un an de studii nu reclamă, cel puțin în primă instanță, nici un fel de prelucrări aritmétice. Prelucrarea unei astfel de liste presupune numai introducere, stergere și modificare de caractere din cadrul listei. În acest caz putem considera că avem de-a face cu prelucrări nenumerice. Vom spune că datele primare din această categorie sunt *date de tip caracter*, sau în argoul informaticienilor *caractere*.

Să considerăm operația 7 : 3, o operație simplă de împărțire. După caz, noi oamenii interpretăm rezultatul în unul din următoarele două moduri:

7 : 3 = 2.3333 . . . și vom numi aceasta *împărțire reală*

7 : 3 dă câtul 2 și restul 1 și vom numi aceasta *împărțire întreagă*

Dacă oamenii dau o interpretare adecvată a rezultatelor împărțirii, nu același lucru se poate spune despre un calculator. Acesta trebuie să știe, apriori, dacă va efectua o împărțire reală sau una întreagă. Astfel de situații au stat la baza separării calculelor numerice în calcule întregi și calcule care nu sunt neapărat întregi.

Pentru a se verifica dacă un număr este prim sau nu, indiferent de metoda folosită, este necesară efectuarea de împărțiri întregi cu obținerea de câturi și de resturi întregi. Datele primare cu care se fac astfel de operații vom spune că sunt *date de tip întreg* sau în argoul informaticienilor *numere întregi*.

În multe aplicații ingineresci apar tot felul de calcule în care se operează cu diversi coeficienți care nu sunt neapărat numere întregi. De exemplu aflarea unei soluții a unui sistem de ecuații presupune întrinsec efectuarea de operații, în particular împărțiri în care se vor lua în calcul cât mai multe zecimale. Datele primare din această categorie vom spune că sunt *date de tip real*, sau adesea în argoul informaticienilor *numere reprezentate în virgulă flotantă* sau *numere reale*.



$$d = i * c + r$$

unde  $d$  este deîmpărțitul,  $i$  este împărtitorul (baza în care se dorește conversia),  $c$  este câtul, iar  $r$  este restul împărțirii. Din această scriere se deduce faptul că valoarea cifrei unităților în baza  $i$  (coeficientul lui  $i^0$  în scrierea polinomială) este restul  $r$  al împărțirii lui  $d$  la  $i$ . Aplicând lui  $c$  aceeași rețetă se obține valoarea cifrei unităților lui  $c$  care este coeficientul lui  $i^1$  în scrierea lui  $d$  s.a.m.d.

De exemplu, să convertim numărul 39549 în baza 8:

39549 : 8 =	4943	rest	5	deci cifra de rang 0 în baza 8 este 5
4943 : 8 =	617	rest	7	deci cifra de rang 1 în baza 8 este 7
617 : 8 =	77	rest	1	deci cifra de rang 2 în baza 8 este 1
77 : 8 =	9	rest	5	deci cifra de rang 3 în baza 8 este 5
9 : 8 =	1	rest	1	deci cifra de rang 4 în baza 8 este 1
1 : 8 =	0	rest	1	deci cifra de rang 5 în baza 8 este 1
0				Conversia se oprește la ultimul deîmpărțit 0

Scriind resturile în ordine inversă obținem reprezentarea numărului în baza 8:

$$(39549)_{10} = (115175)_8$$

Reluăm aceleași calcule pentru trecerea în baza 2:

39549 : 2 =	4943	rest	1
19774 : 2 =	9887	rest	0
9887 : 2 =	4943	rest	1
4943 : 2 =	2471	rest	1
2471 : 2 =	1235	rest	1
1235 : 2 =	617	rest	1
617 : 2 =	308	rest	1
308 : 2 =	154	rest	0
154 : 2 =	77	rest	0
77 : 2 =	38	rest	1
38 : 2 =	19	rest	0
19 : 2 =	9	rest	1
9 : 2 =	4	rest	1
4 : 2 =	2	rest	0
2 : 2 =	1	rest	0
1 : 2 =	0	rest	1
0 :			

Deci avem:

$$(39549)_{10} = (1001101001111101)_2$$

În exemplul care urmează convertim numărul în baza 16. În urma împărțirilor, vor apărea resturi între 0 și 15. Drept urmare, pentru resturile cu valorile 10, ..., 15 vom folosi în scrierea în nouă

## Cap.1. Reprezentarea datelor.

bază simbolurile A, ..., F corespunzătoare. Si acum exemplul de reprezentare în baza 16 a numărului 39549.

39549 : 16 =	2471	rest	13	deci cifra de rang 0 în baza 16 este D
2471 : 16 =	154	rest	7	
154 : 16 =	9	rest	10	deci cifra de rang 2 în baza 16 este A
9 : 8 =	0	rest	9	
0				

Deci avem:

$$(39549)_{10} = (9A7D)_{16}$$

Să mai considerăm numărul  $(985437)_{10}$  pe care vrem să-l reprezentăm, pe rând, în bazele 6 și 16. Calculele sunt următoarele:

985437 : 6 =	164239	rest	3
164239 : 6 =	27373	rest	1
27373 : 6 =	4562	rest	1
4562 : 6 =	760	rest	2
760 : 6 =	126	rest	4
126 : 6 =	21	rest	0
21 : 6 =	3	rest	3
3 : 6 =	0	rest	3

Deci avem:

$$(985437)_{10} = (33042113)_6$$

985437 : 16 =	61589	rest	13	D
61589 : 16 =	3849	rest	5	
3849 : 16 =	240	rest	9	
240 : 16 =	15	rest	0	
15 : 16 =	0	rest	15	F

Deci avem:

$$(985437)_{10} = (F095D)_{16}$$

### 1.2.3. Conversii rapide între bazele 2, 8, 16

Rugăm cititorul să compare conversia în baza 2 a numărului 39549 cu conversia același număr în baza 8. Primele trei linii de la conversia în baza 2 au același efect ca și prima linie de la conversia în baza 8. Acest lucru este normal, deoarece o împărțire la 8 este echivalentă cu trei împărțiri la 2. Deci, luând în ordine inversă și concatenând cele trei resturi din baza 2 se obține  $(101)_2$ , adică restul  $(5)_{10}$  de la conversia în baza 8. Continuând comparația, cea de-a doua linie de la conversia în

## 6. Arhitectura Calculatoarelor. Limba și deasemblare 80x86.

baza 8 cu multoarele trei liniile de la conversia în baza 2, restul în baza 8 se obține prin concatenarea celor trei resturi de la conversia în baza 2 și a.m.d.

Un efect similar se poate observa dacă se compară prima linie de la conversia în baza 16 cu primele patru liniile de la conversia în baza 2 etc. Aceste similitudini nu sunt întotdeauna, ele decurg din față și împărtășătoare, deoarece într-un grup de 4 cifre binare, se pot defini înșile reguli simple de conversie între bazele 2 și 8, respectiv 2 și 16.

Orică grup de 3 cifre binare determină, în mod unic, o cifră hexazecimală. Reciproc, o cifră octală se reprezintă în binar printă-un grup de 3 cifre binare, compunând zerourile necesară la stanga. Un astfel de grup de 4 cifre binare îl vom numi tetradă.

Analog, orice grup de 4 cifre binare determină, în mod unic, o cifră hexazecimală. Reciproc, o cifră hexazecimală se reprezintă în binar printă-un grup de 4 cifre binare, compunând zerourile necesară la stanga. Un astfel de grup de 3 cifre binare este regula de la conversie între bazele 2 și 8:

1. Pentru treccerea din baza 2 în baza 8, se grupăază cifrele reprezentării binare în tridec.  
2. Pentru treccerea din baza 8 în baza 16, se grupăază cifrele reprezentării binare în triade.  
Apoi se folosesc trei cifre. Dacă cel mai din stanga grup al patru cifre este exact trei cifre binare, se completează cu zerouri la stanga pentru a întregi numărul de cifre.  
Pentru treccerea din baza 8 în baza 16 se folosesc cifrele octale ale lui 8, care au loc în triadele de la mijlociu și la mijlociu.

$(11515)_8 = (001\ 001\ 001\ 001\ 111\ 101)_2$   
 $(100\ 101\ 110)_2 = (456)_8$   
 $(110\ 101\ 110)_2 = (347015)_8$   
 $(111\ 001\ 111\ 001\ 101\ 011)_2 = (001\ 001\ 001\ 001\ 110\ 111)_2$

În același spirit, avem următoarele două reguli practice de treccere între bazele 2 și 16:

1. Pentru treccerea din baza 2 în baza 16, se grupăază cifrele reprezentării binare în tetrade.  
Pentru treccerea din baza 16 în baza 2, se completează cu zerouri la stanga pentru a întregi numărul de cifre. Dacă cel mai din stanga grup al patru cifre este exact patru cifre binare, se folosesc cifrele octale.  
Apoi se folosesc cifrele hexazecimale cu tetradă binară corespunzătoare ei (fiecare cifră octală se folosește fiecare tetradă din baza 16 în baza 2, pozitionat de la cifra octală de rang 0 spre stanga).  
Pentru treccerea fiecare tetradă din baza 16 în baza 2, se folosesc cifrele hexazecimale cu tetradă binară corespunzătoare (fiecare cifră binară este valoarea exactă cu care este reprezentată în baza 16).

2. Pentru treccerea din baza 16 în baza 2, pozitionat de la cifra hexazecimală cu tetradă binară corespunzătoare ei (fiecare cifră binară este valoarea exactă cu care este reprezentată în baza 16).

(fiecare cifră hexazecimală este valoarea exactă cu care este reprezentată în baza 16).

## Cap.1. Reprezentarea datelor.

Să considerăm căteva exemple:

$$\begin{aligned} (9AD)_{16} &= (1001\ 1010\ 0111\ 1101)_2 \\ (1\ 0010\ 1110)_2 &= (12E)_{16} \\ (1\ 1100\ 1110\ 0000\ 1101)_2 &= (1CE0D)_{16} \\ (26B)_{16} &= (0010\ 0110\ 1011)_2 \end{aligned}$$

O rezultă practică de treccere între bazele 8 și 16 este aceea să folosim baza 2 ca intermedier. Pentru un opera cu similar neșarristic de cifre binare, facem următoarele recomandări:

1. Pentru treccerea din baza 8 în baza 16, se grupăază la stanga, câte 4 cifre octale. Acestea vor fi transformate în 12 cifre binare, care apoi vor fi transformate în 3 cifre hexazecimale.

2. Pentru treccerea din baza 16 în baza 8, se procedează analog: se grupăază la stanga câte 3 cifre hexazecimale. Acestea vor fi transformate mai întâi în 12 cifre binare, care apoi vor fi transformate în 4 cifre octale.

De exemplu,  $(27354357)_8 = (5DID8F)_{16}$ . Grupările intermediere în baza 2 se pot organiza astfel:  
cazul bazelor care sunt puteri ale lui 2, mecanismele de conversie mai sus prezente. De asemenea, amatori de aritmética pot generaiza mai departe, la orice altă bază de numărăfi. Printre particularități aritmetică în baza 10 sunt folosul alternativ termenului de bit ca sinonim pentru cifră binară.

## 1.3. REPREZENTĂRI BINARE SI ORDINI DE PLASARE

Find vorba de calculul efectuate cu o mașină, se impun trese o serie de restricții legate de reprezentarea întregei număruri întregi (și nu numai). Cea mai importantă dintre ele este dimensiunea a calculatorului, fixată la proprietatea sistemului de calcul respectiv.

### 1.3.1. Dimensiunea a reprezentării

## 8 Arhitectura calculatoarelor. Limbajul de asamblare 80x86.

Valorile lui  $n$  la calculatoarele actuale pot fi: 8, 16, 32 și 64. Sistemele care folosesc alte dimensiuni de reprezentare sunt atât de rare, încât nu merită să ne ocupăm de ele.

Tot în categoria restricțiilor intră și faptul că dimensiunile a doi operanzi care participă la o operație, precum și dimensiunile rezultatului sunt de asemenea constante ale calculatorului, indiferent de tipul de codificare a numerelor. Pentru a preciza regulile de dimensionare în operațiile binare asupra numerelor întregi, vom folosi sintagma "operație pe  $n$  biți". Regulile de dimensionare în urma operațiilor sunt:

Operațiile de adunare pe  $n$  biți și scădere pe  $n$  biți presupun că ambii termeni sunt reprezentați pe câte  $n$  biți, iar rezultatul, suma sau diferența, se va reprezenta tot pe  $n$  biți, vezi figura 1.1.



Fig. 1.1. Dimensiuni de reprezentare la adunare și scădere

Înmulțirea pe  $n$  biți presupune că ambii factori sunt reprezentați pe câte  $n$  biți, iar produsul lor va fi reprezentat pe  $2 * n$  biți, vezi figura 1.2.

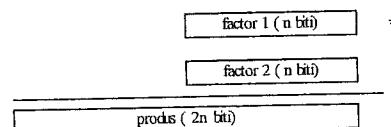


Fig. 1.2. Dimensiuni de reprezentare la înmulțire

Împărțirea pe  $n$  biți (oarecum invers față de înmulțire), impune condiția ca deîmpărțitul să fie reprezentat pe  $2 * n$  biți, iar împărtitorul pe  $n$  biți. Operația furnizează două rezultate: câștul reprezentat pe  $n$  biți și restul reprezentat tot pe  $n$  biți, vezi figura 1.3.

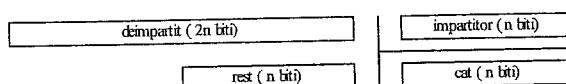


Fig. 1.3. Dimensiuni de reprezentare la împărțire

Dacă rezultatul unei operații nu începe în dimensiunea de reprezentare, atunci se vor pierde biți cei mai semnificativi, rămânând biți mai puțin semnificativi: 0, 1, 2 ș.a.m.d, calculatorul semnalând fenomenul de depășire.

## Cap.1. Reprezentarea datelor.

### 1.3.2. Organizarea și memorarea datelor

#### 1.3.2.1. Bit, octet, locație, adresă

Atât pentru reprezentarea întregilor, cât și pentru reprezentarea altor tipuri de date, orice sistem de calcul folosește o componentă specială, numită unitate de memorie. Fără să intrăm în detaliu constructive, prezentăm principalele elemente de structurare a acesteia.

Unitatea elementară de informație este **bitul**. Într-un bit se poate reprezenta o informație care poate să aibă doar două valori posibile. Tradițional, aceste valori vor fi 0 și 1. De exemplu, în funcție de context, un bit poate să însemne 0 sau 1, true sau false, bărbat sau femeie, bine sau rău, alb sau negru etc. Totul depinde de interpretarea care se dă bitului respectiv. Din punct de vedere tehnologic, un bit este materializat foarte simplu. De exemplu, o tensiune de 0 voltă poate însemna 0, în timp ce o tensiune de +5V poate însemna 1.

**Definiție:** Un octet (byte) este o succesiune de 8 biți, numerotați de la 0 la 7, ca în figura 1.4.

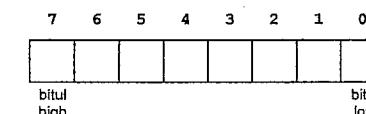


Fig. 1.4. Numerotarea bițiilor în cadrul unui octet

**Octetul** este unitatea elementară de adresare a memoriei. Fiecare octet are atașat un număr întreg și nenegativ numit adresa octetului respectiv. Primul octet din memorie are adresa 0, al doilea octet are adresa 1, al treilea are adresa 2 ș.a.m.d. Sistemul poate referi / identifica fiecare octet din memorie folosind adresa acestuia. Intuitiv, ne putem imagina memoria ca o mulțime de căsuțe (aşa cum sunt cele de la post-restant). Căsuțele sunt numerotate începând de la 0 și în fiecare căsuță poate fi un singur număr. În momentul în care depunem în căsuță un alt număr, vechiul număr se pierde (ca și la înregistrările audio / video pe bandă, rămâne doar ultima înregistrare). Numărul de pe ușa căsuței reprezintă adresa / referința, iar numărul din căsuță reprezintă conținutul.

Referirea la un octet se face, deci, prin adresa lui. De multe ori se practică referirea la un octet nu prin adresa lui, ci prin poziția lui față de un alt octet. În primul caz vorbim de adresa absolută a unui octet, iar în al doilea caz vorbim de adresa relativă față de un alt octet. În al doilea caz adresa absolută se obține adunând la adresa octetului de referință adresa relativă. De exemplu, dacă un octet A are adresa absolută 5643 și un octet B are adresa relativă 5 față de A, atunci adresa absolută a octetului B este 5648. Ca și terminologie, spunem că octetul B este cu 5 octeți *mai la dreapta* decât A. Vezi în figura 1.5. ilustrarea acestei situații.

Prin deținere, o succesiune de acrile corespondă ad dimensiunea lăzii, privind că o entitate de sine

**Dimensiunea** unui locuri este egală cu numărul de acenzi care o compun.

- patru octeți consecutivi formează un dublu cuvant.

La astrel de sistemă vorbind de locații oclete, locuind într-o zonă cu subdistribuție (subenfitejile) în prezentă sunt într-o fază de dezvoltare în care nu au încă dezvoltat toate caracteristicile lor de adăpostire și de reproducere.

**Fig. 1.7.** Un cuvant IBM-PC

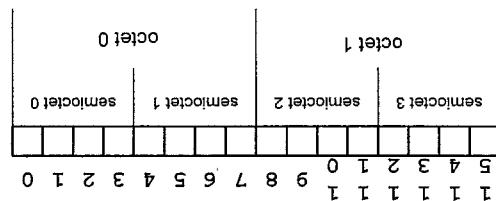
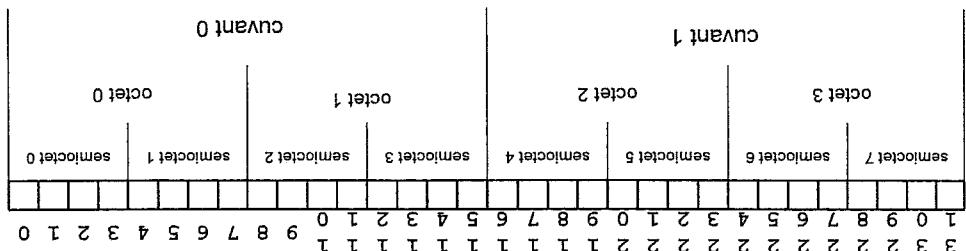


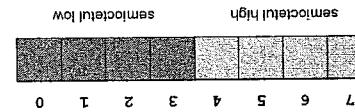
Fig. 1.8. Un dublu cuvânt IBM-PC



**Fig. 1.8.** Un dublu cuvânt IBM-PC

Preleucările fundamentele sunt efectuate pe ocazii și pe grupuri de ocazii consecutive. În particular, operațiile cu numere întregi se pot efectua fie pe cete, fie pe grupuri de ocazii consecutive.

Fig. 1.6. Semioctetii componenti ai unui octet



Accessul la băti și unită ocetă se poate face prin înțemnădilă unor instuigări speciaлизate. În particular, sunt situații în care se folosesc tehnici de execuție semioctet (nibble). Un semioctet este format din patru biți, alăturiți. Vorbim de semioctet low, sau semioctet mai puțin semisfătă, sau semioctet drapți, sau cîțiva hexazecimale dreapta, sau între drapți etc. Analog, vorbim de semioctet high, sau semioctet semisfătă, sau semioctet slings, sau cîțiva hexazecimale stângă, sau nibble etc. Schematic, aceasta împărtează paralelă în figura 1.6.

Emititarea acestor este folosita practic de catre toate institutiile de prelucrare si de schimb cu exteriorul si este unul sistem de calcul. In contextul comunicariilor trebuie reformat postulatul ca: obtinerea unei informatii se poate face doar prin intermediul unei altor informatii.

Ajici este momentul să clarificăm două noțiuni fundamentele: **adresa - adreseare și confiștut**. În cadrul unei obișnuințe de la dreptă sprie slanga în ordinea crescătoare a puterilor bazelor, numerotările consecutive sunt același (vezi Figura 1.4); în cele ce urmărează vom mai vedea astfel de numerotări de la slanga același curier. În ceea ce privește **confiștut** unei locații de memorie:

Fig. 1.5. Succesiunea octefiilor în memorie

Evident, pentru dublucuvântul din figura 1.8., semioctetul 0, octetul 0 și cuvântul 0 sunt respectiv *semoctetul low*, *octetul low* și *cuvântul low* din cadrul locației. Similar, semioctetul 7, octetul 3 și cuvântul 1 sunt respectiv *semoctetul high*, *octetul high* și *cuvântul high* din cadrul locației.

La alte tipuri de sisteme de calcul, printre care și la supercalculatoare, se vorbește de locații octet, locații semicuvânt, locații cuvânt și locații dublucuvânt:

- doi octeți consecutivi, care formează un *semicuvânt*;
- patru octeți consecutivi, care formează un *cuvânt*;
- opt octeți consecutivi, care formează un *dublucuvânt*

### 1.3.2.2. Tipuri elementare de date: dimensiuni ale standardelor de reprezentare

Așa cum am arătat în secțiunea 1.1, cele mai utilizate tipuri elementare de date sunt caracterele, numerele întregi și numerele reale. Pentru fiecare dintre aceste tipuri de date sunt stabilite o serie de *standarde de reprezentare* a datei într-o locație. În particular, standardul de reprezentare fixează dimensiunea locației pentru fiecare tip de dată.

Lăsând detaliiile de reprezentare pentru mai târziu, enumerăm câteva dimensiuni ale locațiilor de reprezentare:

- Tip de date *caracter*:
  - 1 octet – standardul ASCII;
  - 2 octeți – standardul UNICODE.
- Tip de date *întreg*: 1, 2, 4, 8 octeți, depinde de sistemul de calcul.
- Tip de date *real*:
  - 4 octeți – standardul IEEE simplă precizie;
  - 8 octeți – standardul IEEE dublă precizie;
  - 6 octeți – standardul Turbo Pascal.

Asupra standardului ASCII și asupra reprezentărilor numerelor întregi vom reveni cu detalii în secțiunile următoare. În ceea ce privește standardul UNICODE și standardele de reprezentare ale numerelor reale, cititorul interesat poate consulta [Boian96].

Pentru ca cititorul să își facă o idee despre ce înseamnă interpretarea conținutului unei locații, vom da câteva exemple de reprezentări ale unor date elementare. Continuturile locațiilor le vom scrie în hexazecimal:

- Literele 'M' și 'm' interpretate ca și *caracter* se reprezintă:
  - 4D respectiv 6D - standardul ASCII;
  - 004D respectiv 006D – standardul UNICODE.
- Numerele 1 și -1, interpretate ca *întregi* pe 2 octeți se reprezintă 0001 respectiv FFFF. Aceleași numere interpretate ca *întregi* pe 4 octeți se reprezintă 00000001 respectiv FFFFFFFF.
- Numerele 2 și -2, interpretate ca *întregi* pe 2 octeți se reprezintă 0002 respectiv FFFE. Aceleași numere interpretate ca *întregi* pe 4 octeți se reprezintă 00000002 respectiv FFFFFFFE.

- Numere 1 și -1, dar de această dată interpretate ca și numere *reale* se reprezintă:
  - 3F800000 respectiv BF800000 – standardul IEEE simplă precizie,
  - 3FF000000000000 respectiv BFF000000000000 – standardul IEEE dublă precizie,
  - 000000000081 respectiv 800000000081 – standardul Turbo Pascal.

Conform cu cele prezentate la numerotarea octețiilor într-o locație, pentru exemplele de mai sus, avem:

- Octetul 0 al reprezentării caracterului 'm' în standardul UNICODE are valoarea 6D.
- Octetul 1 al reprezentării caracterului 'm' în standardul UNICODE are valoarea 00.
- Octetul 0 al reprezentării numărului -1 în standardul Turbo Pascal are valoarea 81.
- Octetul 5 al reprezentării numărului -1 în standardul Turbo Pascal are valoarea 80.
- Octetul 0 al reprezentării numărului -1 în standardul IEEE dublă precizie are valoarea 00.
- Octetul 6 al reprezentării numărului -1 în standardul IEEE dublă precizie are valoarea F0.
- Octetul 7 al reprezentării numărului -1 în standardul IEEE dublă precizie are valoarea BF.
- §.a.m.d.

### 1.3.2.3. Ordinea octețiilor într-o locație; mașini little-endian și mașini big-endian

La nivelul unei locații, privită ca o entitate de sine stătătoare cu conținut interpretat în funcție de standardul de reprezentare, se disting două abstracțiuni:

- Numerotarea octețiilor în cadrul unei locații fixată de standardul de reprezentare;
- Adresele octețiilor care compun locația.

Așa cum am precizat deja în secțiunea 1.3.2.1, numerotările de conținuturi se fac de la dreapta spre stânga, iar adresele cresc de la stânga spre dreapta. În mod normal, se pune problema unei corespondențe între aceste două elemente. Pe de o parte vorbim de *memorarea structurală* a unui tip de date, impusă de standardul de reprezentare. Pe de altă parte vorbim de *memorarea concretă* a datei în locație: în care octet al locației memorăm octetul 0 al reprezentării, în care octet al locației memorăm octetul 1 §.a.m.d. Cu alte cuvinte trebuie stabilită o *corespondență* între ordinea octețiilor impusă de reprezentarea structurală și adresele din locație în care se memorează valorile acestor octeți.

Această corespondență constituie o caracteristică a sistemului de calcul și ea poate fi una dintre următoarele două:

- Plasarea *little-endian*, în care octetul cu cea mai mică adresă din locație va conține octetul cu numărul 0 al reprezentării, octetul cu adresa următoare va conține octetul 1 al reprezentării §.a.m.d. (octetul „end” al reprezentării are adresa cea mai „little”).
- Plasarea *big-endian*, în care octetul cu cea mai mare adresă din locație va conține octetul 0 al reprezentării, octetul cu adresa precedentă va conține octetul 1 al reprezentării §.a.m.d. (octetul „end” al reprezentării are adresa cea mai „big”).

Să alegem, spre exemplu, numărul (1025)<sub>10</sub> pe care să îl reprezentăm într-o locație de patru octeți. Pentru această dimensiune a locației, reprezentările lui în bazele 16 și 2 sunt (0000401)<sub>16</sub>,

Plasarea într-o matrice cu un tip de nichidină și o caracteristică a sistemului de calcul. De multe ori se folosesc elemente de arhitectură big-endian, respectiv arhitectură little-endian. Un procesor are instrucțiuni speciale care să opereze tip de locație, instații care "stiu" standardele de reprezentare și ordinea de plasare. Utilizatorul trebuie dor să comande procesorul operației de adresa locală de unde aceasta să își ia reprezentarea datelor.

Cap. 1. Repräsentative area dateior.

- De exemplu, se arată că două moduri de plasare ale unei rezervații sunt posibile și corecte în ceea ce privește legile românești:
  - rezervația este înregistrată pe numele proprietarului și este acordată de ambelor parți.
  - rezervația este înregistrată pe numele unui alt persoană și este acordată de proprietarul și de către un alt persoană.
- În primul caz, proprietarul nu poate să obțină înapoiarea platilor făcute pentru rezervație, deoarece rezervația nu este înregistrată pe numele său.
- În al doilea caz, proprietarul poate să obțină înapoiarea platilor făcute pentru rezervație, deoarece rezervația este înregistrată pe numele său.

respective (00000000 00000000 00000100 00000001). Se presupunem că B este adresa locației în care numărul este plasat în ordinea big-endian, iar L este adresa locației în care se află numărul este plasat în ordinea little-endian. Cele două plasari sunt ilustrate în figura 1.9., cu configurațile octetelor scrise astăzi în hexazecimal, cătări în binar:

14 Arhitectura culturală calculatorică. Limba jocului de asamblare B0UX66.

Fig. 1.9. Plasari big-endian si little-endian

In secțiunea prezentată am prezentat căteva exemple de reprezentări, cu diverse standarde, a caracterelor I, I<sub>1</sub>, I<sub>2</sub> și I<sub>3</sub>. Tabelele care urmărează prezintă, în hexazecimal, plasarea bi-ig-endian și little-endian a acestor reprezentări.

număr negativ. Indiferent de standardul de reprezentare, bitul de semn este bitul high al octetului high din reprezentare. Comparativ, acest criteriu reprezintă:

- avantaj big-endian, dezavantaj little-endian, argumentarea în paragraful următor.

De ce? Să luăm ca și exemplu de comparare reprezentările lui 2 pe 4 octeți, atât în plasarea big-endian, cât și în plasarea little-endian, ilustrate în figura 1.10.

Pentru conversie, în cazul unei mașini little endian adresa locației rămâne **L**, indiferent că aceasta are dimensiunea de 4 octeți, de 2 octeți, sau chiar de 1 octet, modificându-se doar dimensiunea locației (deci locația ce conține numărul 2 va avea aici aceeași adresă **L**, indiferent dacă el este reprezentat pe 1, 2 sau 4 octeți!). La mașina big-endian, adresele locațiilor trebuie modificate în funcție de dimensiunea acestora: **B+2** pentru locația de 2 octeți (conținând octeții de adrese **B+2** și **B+3**) și **B+3** pentru locația de 1 octet. Deci în cazul big-endian procesorul trebuie să facă în plus calculul de adresă.

Big-endian:	00 00000000	00 00000000	00 00000000	02 00000010
	Adresa <b>B</b>	Adresa <b>B+1</b>	Adresa <b>B+2</b>	Adresa <b>B+3</b>
Little-endian:	02 00000010	00 00000000	00 00000000	00 00000000
	Adresa <b>L</b>	Adresa <b>L+1</b>	Adresa <b>L+2</b>	Adresa <b>L+3</b>

Fig. 1.10. Reprezentarea lui 2 în cele două tipuri de plasări

Pentru bitul de semn, în cazul unei mașini big-endian adresa octetului cu bitul de semn coincide cu adresa locației. Cu notațiile din figura 1.10., bitul de semn, la mașina big-endian, se află la adresa **B**, indiferent de faptul că se va prelucra o locație de 4 octeți, de 2 octeți sau de 1 octet. În cazul unei mașini little-endian, bitul de semn se află în ultimul octet al locației (cel cu cea mai mare adresă), aşa că pentru a-l obține procesorul trebuie să calculeze adresa acești octet: ea este **L** pentru locația de 1 octet, **L+1** pentru locația de 2 octeți și **L+3** pentru cea de 4 octeți.

Utilizatorul poate să interpreteze în mod diferit conținutul aceleiași locații! De exemplu, poate să memoreze într-un sir de 4 octeți un număr întreg, după care să comande operarea asupra aceleiași arii de memorie prin patru instrucțiuni care să utilizeze 4 locații consecutive de tip caracter reprezentat pe octet. În astfel de situații, când la momente diferite se interpretează diferit aceeași arie de memorie, trebuie să se țină cont de ordinea de plasare și de standardele de reprezentare ale datelor elementare.

Sistemele de calcul de dimensiuni mari, cum ar fi procesoarele **SPARC** sau **MOTOROLA**, mașinile **RISC**, ca și supercalculatoarele **CDC-Cyber** sau **CRAY**, folosesc arhitectura big-endian. Calculatoarele uzuale actuale, în particular cele de tip **IBM-PC**, procesoarele **INTEL** și **DEC**,

**Alpha** folosesc arhitectură little-endian. De o factură aparte sunt calculatoarele din familia **PowerPC**, care sunt mașini *bi-endian*, ele "înțelegând" ambele arhitecturi.

Și acum câteva cuvinte "exotice" legate de arhitecturile big-endian și little-endian. Conversia datelor între cele două arhitecturi este cunoscută în literatură sub numele de problema '**NUXI**'. De ce? '**UNIX**' este numele unui sistem de operare celebru. Să presupunem că într-un sir de 4 octeți se dorește încărcarea cuvântului '**UNIX**'. Un programator "isteț", în loc să comande încărcarea pe rând în cele patru locații succesive a căte unui caracter pentru a obține succesiunea ('U','N','T','X'), face numai două încărcări a căte unui întreg pe doi octeți: mai întâi încarcă sirul '**UN**' privit ca un întreg, apoi în locația întreagă următoare încarcă sirul '**IX**' privit tot ca un întreg! În acest mod a făcut "economie" de două instrucțiuni.

Toate bune și frumoase, el speră că a obținut ('UN','IX'), ceea ce coincide cu ceea ce dorește. Într-adevăr este așa dacă mașina este una big-endian.

În schimb dacă este una little-endian, atunci primii doi octeți vor conține 'NU' și următorii doi 'XI'. Deci, el va obține ('NU','XI') privind ca un sir de patru caractere, deci de fapt obține '**NUXI**' !

Continuând prelucrarea "isteță" a programatorului nostru, el va vrea să folosească o încărcare directă a unui întreg pe 4 octeți, cu valoarea '**UNIX**'. Ok, dacă mașina este big-endian. În schimb, dacă mașina este little-endian, el va obține în cei patru octeți succesiunea '**XINU**' și cu asta am spus tot!

Adjectivele *big-endian* și *little-endian* au și ele o toponimie "exotică". După unii autori, numele au fost stabilite de către un proiectant cu fantezie care a citit cu plăcere povestea lui Jonathan Swift "Călătoriile lui Gulliver". Acolo se vorbește de faționea politică "Big Endians" care susținea că după fierbere oul trebuie spart pe la capătul mai larg (big-endian), în opozиie cu faționea rebelă "Little Endian" potrivnică regelui liliputanilor, care susține că oul trebuie spart la capătul mai ascuțit (little-endian).

#### 1.3.2.4. Unități de capacitate a memoriei

Prin capacitatea de memorare a unui sistem de calcul înțelegem numărul total de octeți ai unității de memorie.

În practică se folosesc o serie de multipli ai numărului de octeți. Spre deosebire de multiplii folosiți în activitatea cotidiană, unitățile multipli ale capacitatii de memorare sunt exprimate sub formă de puteri ale lui 2, astfel:

1 Ko	Kilo-octet	= $2^{10}$ octeți	= 1_024 octeți.
1 Mo	Mega-octet	= $2^{20}$ octeți	= 1_048_576 octeți.
1 Go	Giga-octet	= $2^{30}$ octeți	= $2^{10}$ Mo = $2^{20}$ Ko = 1_073_741_824 octeți.
1 To	Tera-octet	= $2^{40}$ octeți	= $2^{10}$ Go = $2^{20}$ Mo = $2^{30}$ Ko = 1_099_511_627_776 octeți.

19

5. Un set de caractere suntionale care nu apar la tipar / sășare, ci doar într-oaza tiparirea / afisarea.

- Condițiile de codificare pe care le respectă standardul ASCII sunt:
- Toate caracterele funcționale au coduri mai mici decât codul caracterului spațiu;
- Codul caracterului spațiu este mai mic decât codurile celelalte tipăribile.
- Literalele mici sunt codificate prin 26 numere consecutive, în ordine alfabetica.
- Literalele mari sunt codificate prin 26 numere consecutive, în ordine alfabetica.
- Cifrele zecimale sunt codificate prin 10 numere consecutive, în ordinea valorilor.

Dec Hex Oct Char Dec Hex Oct Char Dec Hex Oct Char Dec Hex Oct Char

Fig. 1.11. Codul ASCII standard

4. O sete de caractere *speciale*: spatiu, virgula, punctu, +, -, ., \$, & s.a.m.d

In case of an emergency, it is important to stay calm and follow standard safety procedures.

In contextul cerimiei de înmormântare impuse de extrema însemnătate, în urmă cu 10 ani se impune din ce în ce mai mult standardul de codificare UNICODE. Acesta codifică un caracter pe doi octeți, pentru a permite codificarea simbolurilor și a caracterelor folosite în scrierile majorității limbilor

În prezent, cel mai folosit sistem de codificare este ASCII (American Standard Code for Information Interchange). ASCII ca și standard este un cod pe 7 biți, folosind numerele întregi din intervalul [0,127]. În ASCII este codificat un caracter pe un octet, iar bițul 7 (cel de-al 8-lea) este automat 0. Practic, totate calculatorele actuale folosesc ASCII pentru codificarea caracterelor. Fără IBM a propus (și la propunere au aderat practic toate măritile case de software și hardware), extinderile ASCII folosite fiind de-a lungul timpului.

Un prim sistem de codificare stabilită a fost EBCDIC (Extended Binary Decimale Code), care codifică un caracter pe un octet folosind numere întregi din intervalul [0,255]. Calculatoarele medii-marii de tip IBM-360, IBM-370, precum și FELIX-C folosesc acest cod.

#### 1.4. CODIFICARIA CARACTERE LOR

Spre a avea o idee asupra capacitatilor de memorare actuale, un calculator personal obisnuit are memorii interne cu 256 Mo și 4 Go. Capacitatea unită hard - disc actual este între 20 Go și 512 Go. Pentru comparație, o pagină de format A4 - echivalentă cu o pagină dintr-o carte, conține cu puțin peste 3000 de caractere: litere mari / mici, cifre, semne speciale. Așa cum vom vedea mai târziu, memorarea unui text se face un caracter pe ocol. Rezultă că pentru o pagină sunt necesari 3 Ko de memorie. Deși în mare, într-un Mo de memorie se pot memoră cam 350 pagini de carte.

O altă comparație: în CD obișnuit (nu DVD) are capacitatea de 700 Mo. Într-un astfel de CD se poate memoră lista tuturor abonanților telefonic Romtelecom din România, cu toate informațiile necesare: nume, prenume, adresa completă, telefon.

În figura 1.11., preluată din [www.LookupTables.com] se prezintă complet standardul ASCII. Numerele de cod sunt prezentate în zecimal, în hexazecimal și în octal. Pe prima coloană apar caracterele funcționale, incluzând atât grupurile de litere prin care se simbolizează și semnificația pe scurt a fiecăruiu. Pe celelalte coloane apare în plus modul în care caracterele se pot specifica în limbajul HTML, limbaj de bază în descrierea paginilor Web. Din aceeași sursă, în figura 1.12. se prezintă extinderea IBM prin folosirea celui de-al 8-lea bit.

128	ç	124	é	161	í	177	»	193	-	209	™	225	ß	241	±
129	ä	145	ø	162	ö	178	»	194	Γ	210	π	226	Γ	242	≥
130	è	146	æ	163	ú	179		195	†	211	≤	227	π	243	≤
131	ã	147	ó	164	ñ	180	+	196	-	212	≤	228	Σ	244	Γ
132	ë	148	ö	165	ÿ	181	+	197	†	213	π	229	ø	245	↓
133	à	149	â	166	˜	182		198	†	214	π	230	μ	246	+
134	å	150	å	167	°	183	π	199	†	215	†	231	π	247	≈
135	ş	151	ş	168	þ	184	π	200	†	216	†	232	Φ	248	°
136	đ	152	đ	169	—	185		201	π	217	↓	233	⊕	249	·
137	ë	153	õ	170	—	186		202	↓	218	Γ	234	□	250	·
138	ë	154	Ü	171	˜	187	π	203	π	219	■	235	δ	251	↓
139	ř	156	ě	172	ÿ	188	π	204	†	220	■	236	⊗	252	—
140	ř	157	ř	173	—	189		205	=	221	↓	237	⊕	253	◦
141	ř	158	ř	174	«	190	↓	206	†	222	■	238	π	254	■
142	À	159	à	175	»	191	↑	207	↓	223	■	239	π	255	—
143	Ã	160	ã	176	»	192	L	208	±	224	π	240	≡		

Source: [www.LookupTables.com](http://www.LookupTables.com)

Fig. 1.12. Codul ASCII extins

## 1.5. CODIFICAREA NUMERELOR ÎNTREGI

### 1.5.1. Convenție cu semn și convenție fără semn

Așa cum am arătat în 1.3.2.3, pentru reprezentarea semnului unui număr se utilizează bitul high al octetului high din reprezentare. Fiecare programator poate să interpreteze conținuturile locațiilor de numere întregi cu care operează în una din următoarele două convenții:

- convenția de reprezentare fără semn, în care se operează numai cu numere naturale;
- convenția de reprezentare cu semn, în care se operează atât cu numere pozitive, cât și cu numere negative.

Astfel, într-o locație de  $n$  biți, programatorul poate considera fie că se află un număr între 0 și  $2^n - 1$  dacă adoptă convenția fără semn, fie un număr între  $-2^{n-1}$  și  $2^{n-1} - 1$  dacă adoptă convenția cu semn. (Se observă că într-o locație pot fi memorate tot atâtea numere pozitive câte negative). De exemplu, într-o locație de un octet pot fi reprezentate, în convenția fără semn, numerele de la 0 la 255. În convenția cu semn, din cauza faptului că un bit este ocupat de semn, pot fi reprezentate numerele pozitive de la 0 la 127 și numerele negative de la -128 la -1.

## Cap.1. Reprezentarea datelor.

21

Tabelul următor prezintă intervalele de numere reprezentabile într-o locație, atât în convenția fără semn, cât și în convenția cu semn, în funcție de dimensiunea acesteia.

Nr. octeți	Convenția fără semn	Convenția cu semn
1	$[0, 2^8 - 1] = [0, 255]$	$[-2^7, 2^7 - 1] = [-128, 127]$
2	$[0, 2^{16} - 1] = [0, 65535]$	$[-2^{15}, 2^{15} - 1] = [-32768, 32767]$
4	$[0, 2^{32} - 1] = [0, 4294967295]$	$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$
8	$[0, 2^{64} - 1] = [0, 18446824753]$	$[-2^{63}, 2^{63} - 1] = [-9223412376694775808, 9223412376694775807]$

Acestea sunt convențiile impuse constructorilor de procesoare. Implementările operațiilor peste întregi trebuie, pe de o parte să fie eficiente, iar pe de altă parte să se folosească, pe cât posibil, algoritmi comuni de evaluare a operațiilor fundamentale peste numere întregi, indiferent de convenția de reprezentare. Modul în care se realizează aceste implementări și măsura în care se vor folosi algoritmi comuni ambelor convenții de reprezentare se va vedea în secțiunea următoare.

### 1.5.2. Bitul de semn; codul complementar

Dacă interpretăm o anumită configurație de biți drept întreg cu semn, atunci prin convenție, pentru reprezentarea semnului unui număr se folosește un singur bit. Este vorba de *bitul high* (bitul 7) din *octetul high* al locației în care se reprezintă numărul. Dacă valoarea acestui bit este *zero* atunci numărul este *pozitiv*. Dacă bitul este *unu*, atunci numărul este *negativ*.

Astfel, pentru o locație pe doi octeți (16 biți), dăm câteva exemple de configurații (scrise în hexazecimal și în binar), pe care dacă le interpretăm ca și numere cu semn avem:

- $(8000)_{16} = (1000\ 0000\ 0000\ 0000)_2$  este număr negativ, deoarece bitul cel mai semnificativ este 1;
- $(1000)_{16} = (0001\ 0000\ 0000\ 0000)_2$  este număr pozitiv, deoarece bitul cel mai semnificativ este 0;
- $(7FFF)_{16} = (0111\ 1111\ 1111\ 1111)_2$  este număr pozitiv;
- $(FFFF)_{16} = (1111\ 1111\ 1111\ 1111)_2$  este număr negativ;
- $(0FFF)_{16} = (0000\ 1111\ 1111\ 1111)_2$  este număr pozitiv;
- etc.

Și acum întrebarea (pe care ne-o punem de câteva secțiuni încoace): cum se reprezintă numerele întregi în convenția cu semn?

Răspunsul a stârnit dispute aprinse de-a lungul istoriei calculatoarelor. Au existat trei direcții din care s-a impus una.

O primă direcție este aceea a reprezentării valorii absolute a numărului pe  $n-1$  biți din cei  $n$  ai locației, iar în bitul cel mai semnificativ să se pună semnul. Această reprezentare poartă numele de

O două direcție este accea o reprezentare valori absolute a numerelor pe n-1 biți din cei n ai locației, iar în cauză în care numărul este negativ, să se inverseze toți cei n biți ai reprezentării. În acest mod bițul de semn va devine automat l. Accesătă reprezentare portă numele de cod invers, sau complementărlă de 1. A fost, de asemenea, o mică cruce de cod invers, care de la direcție, care de la un impuls pentru reprezentarea numerelor întregi cu semn portată numele de cod complementar, sau complementărlă de 2. Pe tot rezultul prezentat încărcăt vom discuta de semn.

cod direct. Soluția, foarte apropiată de cea naturală — pe care o folosim în calculul manuală, s-a dovedită și mai puțin eficientă decât altă. Azi se folosete codul direct numai la reprezentarea numerelor pozitive.

O două direcție este accea o reprezentare valori absolute a numerelor pe n-1 biți din cei n ai locației, care de la direcție, care de la un impuls pentru reprezentarea numerelor întregi cu semn portată numele de cod invers, care de la un impuls pentru reprezentarea numerelor întregi cu semn portată numele de cod complementar, sau complementărlă de 2. Pe tot rezultul prezentat încărcăt vom discuta de semn.

Definiție: Pentru complementarea unui număr întreg reprezentat pe n biți, mai întâi se inversează după care se adaugă 1 la valoarea obținută.

Inaltele variante, convenabile mai multe și se complementăza manuală.

Se scadă cifra binară configurată (viden binar) al locației zeroi de complementă din 100...00, unde după cifra hexazecimală 1 apar altăea zeroi cu către cifre hexazemiale ară locația de complemenat.

Locația pe care să fie adăugat 1 este înlocuită cu cifra binară care conține numărul (18)10.

De exemplu, dacă vom să complememăm o locație de un octet și care conține numărul (18)10:

Locația inițială: 00010010+  
Complemenat: 11101110  
Se împără acces transport: 11101110  
Se împără acces transport: 11101110  
00010010+

De asemenea, se poate verifica că pe 1 octet numărul **TF** și **81** sunt complementare și pe 2 octetii **TFFF** și **8001** sunt complementare.

Natural, regălile de complementare sunt valabile și la locații de 2 octetii, și la locații de 4 octetii etc.

Locația inițială: 00000000+  
Complemenat: 11111111  
Se adaugă 1  
Locația inițială: 00010010  
Complemenat: 11101110  
Se adaugă 1  
Locația inițială: 00000000  
Complemenat: 11111111

Se cădă neștiință! bîți începând din dreptă reprezentării binare până la biți n-1 inclusiv!

restul biților se inversează pînă la primul biț I inclusiv.

Se lastă neștiință! bîți începând din dreptă reprezentării binare până la biți n-1 inclusiv!

de după cifra binară configurată (viden binar) al locației zeroi de complementă din 100...00, unde după cifra hexazecimală 1 apar altăea zeroi cu către cifre hexazemiale ară locația de complemenat.

Locația inițială: 00010010+  
Complemenat: 11101110  
Se împără acces transport: 11101110  
Se împără acces transport: 11101110  
00010010+

De exemplu transporul de către semnificația începeand cu cifra binara de rang n. Aveni: și acum să adunăm pe 8 bîți un număr cu complementul său. Print-o adunare pe n bîți înseamnă că se ignorează transporul de către semnificația începeand cu cifra binara de rang n.

Deci toamă numărul de la care am plecat într-adăvar, complementarea complementului este numărul îninal spus complementar.

Locația inițială: 00010010+  
Complemenat: 11101110  
Se împără acces transport: 11101110  
00010010+

De exemplu, dacă vom să complememăm o locație de un octet și care conține numărul (18)10:

Locația inițială: 00010010+  
Complemenat: 11101110  
Se împără acces transport: 11101110  
00010010+

Altfel spus, ignorând transportul în afara locației, suma dintre un număr și complementul său este 0. Este interesant și de reținut faptul că numărul 0, reprezentat prin  $n$  zerouri într-o locație de  $n$  biți este propriul lui complement, iar numărul de  $n$  biți 100...00 este, de asemenea, propriul lui complement. Aceasta datorită faptului că adunarea se efectuează pe  $n$  biți, ultimul transport fiind ignorat, aşa cum am arătat mai sus. Să exemplificăm pentru locații de 8 biți.

Locația inițială:	00000000
După inversarea bițiilor	11111111
Se adăugă 1	11111111+
(se ignoră ultimul transport)	<u>00000001</u>
Complementul:	00000000
Locația inițială:	10000000
După inversarea bițiilor	01111111
Se adăugă 1	01111111+
	<u>00000001</u>
Complementul:	10000000

În sfârșit, prezentăm **regula de reprezentare a numerelor întregi cu semn**: Un număr întreg între  $-2^{n-1}$  și  $2^{n-1}$  se reprezintă într-o locație de  $n$  biți astfel:

- dacă numărul este pozitiv, atunci în locație se reprezintă numărul respectiv scris în baza 2;
- dacă numărul este negativ, atunci în locație se înscrie complementul reprezentării în baza 2 a numărului.

Înainte de a trece la exemple, trebuie să clarificăm situația reprezentării numărului  $-2^{n-1}$ . Valoarea lui absolută nu poate fi reprezentată pe  $n-1$  biți ca să rămână loc și pentru bitul de semn, ci el se reprezintă pe  $n$  biți și este 100...0. Pe de o parte această reprezentare indică un număr negativ! Pe de altă parte, am arătat deja că acest număr este propriul lui complement. Din aceste motive, prin convenție s-a stabilit că numărul  $-2^{n-1}$  se reprezintă în cod complementar pe  $n$  biți prin 100...0. Aceeași configurație interpretată fără semn reprezintă numărul  $2^{n-1}$ .

Tabelul următor prezintă reprezentările mai multor numere, în locații de 8 biți – 1 octet, 16 biți – 2 octeți și 32 de biți – 4 octeți.

Dim. locație (octeți)	Număr în baza 10	Reprezentare în cod complementar (hexazecimal)	Reprezentare în cod complementar (binar)
1	0	00	00000000
2	0	0000	0000000000000000
1	1	01	00000001
2	1	0001	0000000000000001
1	-1	FF	11111111
2	-1	FFFF	1111111111111111
1	127	7F	01111111

2	127	007F	0000000011111111
1	-128	80	10000000
2	-128	FF80	1111111100000000
2	128	0080	0000000010000000
2	32767	7FFF	0111111111111111
2	-32767	8001	1000000000000001
2	-32768	8000	1000000000000000
4	-32768	FFFF8000	11111111111111110000000000000000
4	32768	00008000	00000000000000010000000000000000
1	18	12	00010010
2	18	0012	0000000000010010
1	-18	EE	11101110
2	-18	FFEE	111111111101110
4	39549	00009A7D	000000000000000100110100111101
4	-39549	FFFF6583	111111111111110110010110000011
4	985437	000F095D	0000000000001110000100101011101
4	-985437	FFF0F6A3	1111111111000011101010100011

### 1.5.3. Operări aritmetice; conceptul de depășire

#### 1.5.3.1. De ce codul complementar?

Spuneam într-o secțiune precedentă că implementările operațiilor peste întregi trebuie, pe de o parte să fie eficiente, iar pe de altă parte să se folosească, pe cât posibil, algoritmi comuni de evaluare a operațiilor fundamentale peste numere întregi, indiferent de convenția de reprezentare.

Până în prezent reprezentarea în cod complementar răspunde cel mai bine cerințelor de mai sus. Principalele motive sunt următoarele două:

- Operația de adunare se execută la fel, indiferent de faptul că avem de-a face cu convenția de reprezentare fără semn sau cea de reprezentare cu semn. Operația executată este o adunare simplă, pe  $n$  biți ( $n$  – dimensiunea locației), cu ignorarea ultimului transport.
- Operația de scădere se reduce la operația de adunare a descăzutului cu complementul scăzătorului.

După cum se apreciază, procentul operațiilor additive – adunări și scăderi este mult mai mare în aplicații decât cel al operațiilor multiplicative. De aici și preferința proiectanților pentru adoptarea codului complementar pentru reprezentarea întregilor cu semn. În schimb, operațiile de înmulțire și împărțire sunt efectuate cu algoritmi separați pentru reprezentările fără semn și reprezentările cu semn.

În consecință, programatorul își alege convenția cu semn sau fără semn în funcție de specificul problemei. El utilizează aceleași operații pentru adunări și scăderi și operații specifice cu semn sau fără semn pentru operațiile de înmulțire și împărțire.

Baza 10	Baza 16	Baza 2	Observații
18+	18	12+ 00010010+	Asă după cum am arătat mai sus, reprezentarea numerelor în baza 2 este rezervată pentru apăratori. În mod natural, trebuie să ne punem problema: ce se întâmplă când rezultatul nu încapă în spațiu deoarece să rezolvăm?
243-	243- F3-	11110011- 00010010	Specific vorbind, condiție de apărare a unei depășiri apar în mod direct în funcție de context.
243+	243+ F3+	11110011+ 00010010	Contextul definește imacătă semnala, în mod specific, fiecare situație de depășire.
18-	18- 00010010-	00010011 11100111	Procesarele sunt astfel constituite încât să semnalaize, în mod specific, fiecare situație de depășire.
575+	575+ 023F+	000000100011111+ 000000100010101+	Operațiile de adunare și de scădere sunt efectuate de calculator exact după algoritmi cunoscuți din clasa I primă, cu singura deosebire că operațiile sunt efectuate în baza 2. În schimb, calculatorul rezultatului de dimensiuniare și-a raportat că operația este efectuată de masina și nu de om, există posibilitatea de apărare a depășirilor.
650-	650- 028A-	0000001010001010- 0000000010001111	R1) Dacă rezultatul adunării nu încape pe n biți, atunci apără depășire la adunare și rămasă în rezultat numai biți de la ordinul 0 la ordinul n-1, iar biți de ordin n se pierde.
6535+	6535+ FFF+	1111111111111111+ 0005A	Pentru cauză operativă frază semn, există două căre provoacă depășire:
65625	65625 90	00059 00059	R2) Dacă într-o operație de scădere rezultatul este mai mic decât scăzătorul, atunci are loc depășire la scădere, dar operația se execută, jucându-se "împurunt jicăvă" de la un rang înexistenț.
575-	575- 023F-	0000000100011111- 00000001010001010	Dam, în continuare, către exemplu de operări de adunare și scădere analizate în convenția frază baza 2. Dimensiunea locaților este fie de un octet, fie de doi octetii.
75-	75- 650	1111111110110101 028A FFBB3	Aceste reguli sunt intuitiv clare (justificabile) și se pot aplica și tehnic ca astăzi.

### 1.5.3.2. Conceptul de depășire

### 1.5.3.3. Adunări și scăderi

In cele ce urmează vom aborda, pe rand tipurile de operații și în cadrul acestora vom semnala situațile de depășire.

Rămâne în sarcina utilizatorului dacă ia în calcul semnalele procesorului și pe care anume.

Procesarele sunt astfel constituite încât să semnalaize, în mod specific, fiecare situație de depășire.

de altă parte, contextul depășide de tipul operației; aditiva sau multiplicativa.

Contextul definește imacătă semnala, în mod direct în funcție de context.

Specifice vorbind, condiție de apărare a unei depășiri apar în mod direct în funcție de context.

Conținutul de date de la adunare și scădere sunt rezultatul unitării de calcul din imacătă semnala.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

În cele ce urmează vom aborda, pe rand tipurile de operații și în cadrul acestora vom semnala situațile de depășire.

Ramane în sarcina utilizatorului dacă ia în calcul semnalele procesorului și pe care anume.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

de altă parte, contextul depășide de tipul operației; aditiva sau multiplicativa.

Contextul definește imacătă semnala, în mod direct în funcție de context.

Specifice vorbind, condiție de apărare a unei depășiri apar în mod direct în funcție de context.

Conținutul de date de la adunare și scădere sunt rezultatul unitării de calcul din imacătă semnala.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

În cele ce urmează vom aborda, pe rand tipurile de operații și în cadrul acestora vom semnala situațile de depășire.

Ramane în sarcina utilizatorului dacă ia în calcul semnalele procesorului și pe care anume.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

de altă parte, contextul depășide de tipul operației; aditiva sau multiplicativa.

Contextul definește imacătă semnala, în mod direct în funcție de context.

Specifice vorbind, condiție de apărare a unei depășiri apar în mod direct în funcție de context.

Conținutul de date de la adunare și scădere sunt rezultatul unitării de calcul din imacătă semnala.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

În cele ce urmează vom aborda, pe rand tipurile de operații și în cadrul acestora vom semnala situațile de depășire.

Ramane în sarcina utilizatorului dacă ia în calcul semnalele procesorului și pe care anume.

Procesarele sunt astfel constătoare imacătă semnala, în mod specific, fiecare situație de depășire.

Să considerăm două numere  $x, y$  din intervalul  $[-2^{n-1}, 2^{n-1}-1]$  și să considerăm suma lor algebrică  $x + y$ . Este suficient să studiem următoarele trei cazuri:

- 1)  $x < 0$  și  $y \geq 0$ . Din apartenența la interval avem că  $-2^{n-1} \leq x < 0$  și  $0 \leq y \leq 2^{n-1}-1$ .
- 2)  $x \geq 0$  și  $y \geq 0$ . Din apartenența la interval avem că  $0 \leq x \leq 2^{n-1}-1$  și  $0 \leq y \leq 2^{n-1}-1$ .
- 3)  $x < 0$  și  $y < 0$ . Din apartenența la interval avem că  $-2^{n-1} \leq x < 0$  și  $-2^{n-1} \leq y < 0$ .

Inegalitățile din cazul 1) ne asigură că în primul caz suma algebrică aparține același interval. Într-adevăr, din inegalitățile:  $-2^{n-1} \leq x < 0$  și  $0 \leq y \leq 2^{n-1}-1$  avem, pe de o parte, că  $x \leq x + y$ , deci  $-2^{n-1} \leq x + y$ . Pe de altă parte, din aceeași inegalitate rezultă că  $x + y \leq y$ , deci  $x + y \leq 2^{n-1}-1$ . În concluzie,  $x + y$  aparține intervalului  $[-2^{n-1}, 2^{n-1}-1]$ . De aici se deduce că dacă două numere sunt de semne contrare, atunci suma lor nu poate produce depășire. Așa cum se va vedea din exemplele care urmează, atunci când se face suma codurilor complementare este posibil să apară transport de cifră semnificativă, dar acest fapt este ignorat, deoarece nu produce depășire.

În cazurile 2) și 3) este posibilă apariția depășirii. În cazul 2)  $x + y$  poate avea valoarea maximă  $2^n - 2$ , iar depășire apare dacă  $x + y > 2^{n-1}-1$ . Deoarece codurile complementare se adună ca și numere fără semn, rezultă că depășire apare atunci când pe poziția bitului de semn apare cifra 1, ceea ce în cod complementar înseamnă număr negativ!

În cazul 3)  $x + y$  poate avea valoarea minimă  $-2^n$ , iar depășire apare dacă  $x + y < -2^{n-1}$ . Analog ca mai sus, rezultă depășire dacă pe poziția bitului de semn apare cifra 0, ceea ce în cod complementar înseamnă număr pozitiv!

Din studiul celor trei cazuri rezultă o regulă simplă a depășirii la adunare în cod complementar:

*R3) Suma a două numere reprezentate în cod complementar provoacă depășire dacă și numai dacă sunt de același semn și rezultatul sumei lor este de semn contrar.*

Din această regulă se poate deduce și regula depășirii la scăderea cu semn. Având în vedere faptul că o scădere  $a - b = c$  este echivalentă cu adunarea  $a = b + c$ , din regula R3 rezultă că:

*R4) Diferența a două numere reprezentate în cod complementar provoacă depășire dacă și numai dacă scăzătorul și diferența sunt de același semn și descăzutul este de semn contrar.*

Practic, putem identifica două tipuri de situații ce vor semnală depășire la scăderea cu semn, în situația b) fiind necesar un "împrumut fictiv".

$$\begin{array}{c} \text{a)} \\ \begin{array}{r} 1 \cdots \cdots - \\ 0 \cdots \cdots - \\ \hline 0 \cdots \cdots \end{array} \end{array} \quad \begin{array}{c} \text{b)} \\ \begin{array}{r} 0 \cdots \cdots - \\ 1 \cdots \cdots - \\ \hline 1 \cdots \cdots \end{array} \end{array}$$

Intuitiv, în cazul a) depășirea se justifică prin imposibilitatea obținerii unui număr pozitiv ca rezultat al scăderii unui număr pozitiv dintr-unul negativ. În cazul b) depășirea se justifică intuitiv

dacă facem referire la adunarea echivalentă ( $a - b = c \Leftrightarrow a = b + c$ ); aici diferența și scăzătorul negative nu pot furniza descăzut pozitiv.

Așa cum se va vedea din exemplele care urmează, atunci când se face diferența codurilor complementare este posibil să apară împrumut fictiv de cifră semnificativă, dar acest fapt este ignorat, deoarece nu produce depășire.

În continuare prezentăm câteva exemple de adunări și scăderi ale unor numere reprezentate în cod complementar. Pentru simplificarea expunerii, vom utiliza operanții reprezentări în cod complementar pe 4 biți. Conform celor de mai sus, pentru patru biți intervalul de reprezentare este  $[-2^3, 2^3-1]$ , adică  $[-8, 7]$ . Operanții îi vom transcrie din baza 10 direct în cod complementar. Vom semnală cazurile de transport, împrumut fictiv și depășire.

Suma în baza 10	Suma în cod complementar	Transport sau depășire	Diferența în baza 10	Diferența în cod complementar	Împrumut sau depășire
$(-7) + 5 = -2$	$1001+$ <u>0101</u> 1110		$5 - 7 = -2$	<u>0101-</u> <u>0111</u> 1110	
$(-4) + 4 = 0$	$1100+$ <u>0100</u> 0000	Transport	$4 - 4 = 0$	<u>0100-</u> <u>0100</u> 0000	
$2 + (-7) = -5$	$0010+$ <u>1001</u> 1011		$2 - 7 = -5$	<u>0010-</u> <u>0111</u> 1011	Împrumut
$5 + (-2) = 3$	$0101+$ <u>1110</u> 0011	Transport	$5 - 2 = 3$	<u>0101-</u> <u>0010</u> 0011	
$3 + 4 = 7$	$0011+$ <u>0100</u> 0111				
$(-4) + (-1) = -5$	$1100+$ <u>1111</u> 1011		$(-4) - 1 = -5$	<u>1100-</u> <u>0001</u> 1011	
$5 + 2 = 7$	$0101+$ <u>0010</u> 0111				

Reprezentare în cod complememtar. Cătăl mparfintiu va respecta regula semnelor. Imparfintiu va fi, în lăzioră absoluită mai mult decât valoarea absolută a imparfitorului și va avea același semn ca și demiparfintiu! De exemplu,  $-7 : 3$  da cátul  $-2$  și restul  $-1$ , adică  $-7 = -2 * 3 + (-1)$ . Rungam ctitorial să compare acest rezultat cu cel impus de teorema împărțirii cu rest din aritmetică, care impune că restul să fie un număr pozitiv, deci în aritmetică avem  $(-7) = -(-3) * 3 + 2$ , deci cátul

Operația de împărțire semnalarea eroare la semnalarea eroare este de tipul  $\text{divide by zero}$  ( $\text{Divide by zero!}$ ). În dimensiuni semnalarea eroare la semnalarea eroare este de tipul  $\text{divide by zero!}$ . În cauză neîncadrarii efectua, deoarece deimpărțitul se poate reprezenta pe  $16 \text{ biti}$  și împărțitorul se poate reprezenta pe  $8 \text{ biti}$ . La aceasta operatie va apărea deosebită, deoarece ceea ce este  $33$  și nu se poate reprezenta pe  $8 \text{ biti}$ . Restul împărțitului este  $1$  și se poate reprezenta pe  $16 \text{ biti}$ , adică să reprezinte deimparțitul pe  $32$  de biți. Programatorul poate decide să fie operația pe  $16 \text{ biti}$ , adică să obțină ceea ce este  $16 \text{ biti}$  (33 împărțit de  $16 \text{ biti}$ ) și împărțitorul pe  $16 \text{ biti}$ , urmând să obțină ceea ce este  $16 \text{ biti}$  (restul pe care îl obține după ce împărțește  $32$  de biți astfel nu vom mai avea dețapări).

Plieánnd de la acest exemplu, se poate observa că:  
Imt-adrevăr,  $11 \times 13 = 143 = (10001111)_2$

1) Iată un exemplu de calea de rezolvare a unei ecuații de gradul II:

$$x^2 - 5x + 6 = 0$$

Soluția este:

$$x_1 = \frac{5 + \sqrt{25 - 24}}{2} = \frac{5 + 1}{2} = 3$$

$$x_2 = \frac{5 - \sqrt{25 - 24}}{2} = \frac{5 - 1}{2} = 2$$

2) Rezolvarea ecuației de gradul II:

$$ax^2 + bx + c = 0$$

Se aplică formulele:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$(-3) + 2 = -3$	01011+	00010	0010-	2 - 3 = -3	Impurmuit	Nu se poate scrie $T-8$ de 4 biți, deoarece 8 nu aparține intervalului $[-8, 7]$ .	0100+	1000+	1111	1111	0111	0101+	5 + 4 = 9
$(-8) + 7 = -1$	1000+	0000+	0010-				1111	1111	0111	0111	1001+	0101+	$(-7) + (-6) = -13$
$(-7) + (-6) = -13$	1001+	0100	0010-				1001-	1001-	0110	0110	0011	0101+	$(-6) + (-4) = -10$
$(-6) + (-4) = -10$	1010+	0110	0100-				1010-	1010-	0110	0110	0011	0101+	$(-6) + (-4) = -10$
$(-7) + 14 = 7$	0111+	0111	0110-				0111	0111	0110	0110	0011	0101+	$(-8) + (-8) = -16$
$(-8) + (-8) = -16$	1000+	1000	1000-				1110	1110	0110	0110	0011	0101+	$(-8) + (-8) = -16$

#### 1.5.3.4. Immulti si impartri

10010011	1011
1011	
=1110	1101
1011	
=0111	
0000	
=1111	
1011	
=100	

Într-adevăr,  $147 \div 11 = 13$  și restul 4.

Ca și la înmulțire, putem face o serie de observații. Acestea evidențiază faptul că, în ultimă instanță, împărțirea în baza 2 se reduce la o succesiune de scăderi succesive combinate cu operări de deplasare.

Să reamintim câteva observații privind înmulțirile și împărțirile cu semn.

- 1) Toți operanții implicați în operații sunt reprezentați în cod complementar, în conformitate cu cerințele de dimensionare expuse mai sus.
- 2) Atât la înmulțirea cu semn cât și la împărțirea cu semn, se respectă regula semnelor.
- 3) Pentru operația de împărțire, restul este în modul mai mic decât modulul împărtitorului, iar semnul restului este același cu semnul deîmpărtitului.

Algoritmi de înmulțire și împărțire în cod complementar nu pot fi preluati natural de la cei fără semn, așa cum stau lucrurile la adunare și scădere. Dacă ar fi aşa, atunci în exemplul de mai sus, interpretând **1011** ca **-5** și **1101** ca **-3** ar rezulta produsul **-113** în loc de **-15!** Normal, deoarece configurația de biți **10001111** care este rezultatul înmulțirii binare, este reprezentarea în cod complementar a numărului **-113!** La fel, dacă interpretăm cu semn exemplul dat la împărțirea fără semn, avem "egalitatea" în cod complementar:

$$10010011 = 1011 * 1101 + 0100 \text{ adică } -109 = (-5) * (-3) + 4 !$$

Nu vom detalia algoritmi specifici înmulțirii și împărțirii cu semn. Cititorii pot obține detalii din [Boian96].

#### 1.5.4. Conversia la o locație de alte dimensiuni

Până acum am presupus că operanții au lungimi fixe, așa cum pretind regulile de derulare a operațiilor. Dar ce-i de făcut atunci când, spre exemplu, trebuie să se convertească un cod complementar pe 8 biți la unul pe 16 biți? Sau dacă trebuie să reducem un număr reprezentat fără semn pe 16 biți la unul similar pe 8 biți?

În fapt este vorba de patru operații:

- *Extensie cu semn* a unui cod complementar într-o locație mai mare.
- *Extensie cu zero* a unui număr fără semn într-o locație mai mare.
- *Contraction cu semn* a unui cod complementar într-o locație mai mică.
- *Contraction de zero* a unui număr fără semn într-o locație mai mică.

Regulile de conversie sunt foarte simple. Extensia cu semn înseamnă că în spațiul suplimentar toți biții vor avea ca valoare valoarea bitului de semn al reprezentării care se convertește. Extensia cu zero înseamnă că în spațiul suplimentar toți biții vor avea valoarea zero. Tabelul următor prezintă câteva exemple cu ambele extensii. În fiecare celulă a tabelului pe primul rând este scrisă configurația în hexazecimal, iar pe următoarele configurația binară:

8 biți:	16 biți: extensie cu semn	32 biți: extensie cu semn	16 biți: extensie cu zero	32 biți: extensie cu zero
80 10000000	FF80 1111111100000000	FFFFFF80 1111111111111111 1111111110000000	0080 0000000100000000	000000080 0000000000000000 0000000010000000
28 00101000	0028 0000000000101000	00000028 0000000000000000 0000000000101000	0028 000000000101000	00000028 0000000000000000 0000000000000000
9A 10011010	FF9A 1111111110011010	FFFFFF9A 1111111111111111 1111111110011010	009A 000000010011010	0000009A 0000000000000000 0000000010011010
7F 01111111	007F 0000000011111111	0000007F 0000000000000000 0000000011111111	007F 0000000011111111	0000007F 0000000000000000 0000000011111111
--	1020 0001000000100000	0001020 0000000000000000 0001000000100000	----	0001020 0000000000000000 0001000000100000
--	8088 1000000010001000	8088 1000000010001000	FFFF8088 1111111111111111 1000000010001000	00008088 0000000000000000 1000000010001000

Operațiile de contracție nu se pot executa întotdeauna. Spre exemplu, într-o locație pe 16 biți există numărul **-448** în baza 10, care în cod complementar se reprezintă **FE40**. Dorim să efectuăm o contracție la 8 biți. Eliminând pur și simplu primul octet se obține **40**, adică numărul **64** în baza 10! Avem, evident, o situație de depășire. Cu alte cuvinte, contracțiile (conversii prin îngustare) se pot executa numai dacă NU se provoacă pierderea de informație.

Pentru contracția cu semn, contracția se poate face numai dacă toți biții care se elimină trebuie să coincidă cu bitul de semn, adică cu primul bit care rămâne. Pentru contracția fără semn, trebuie ca toți biții care se elimină să fie zero. Tabelul următor prezintă câteva exemple.

## 2.1. DEFINITII. ORGANIZAREA UNUI SISTEM DE CALCUL.

### ARHTECTURA SISTEMELOR DE CALCUL

#### CAPITOLUL 2

Cap.2. Arhitectura sistemelor de calcul.

35

Prințe nivellele ieerarhice ale hardware-ului se pot identifica structuri din siliciu sau alte materiale, componente electronice (transistori și.a.), componente logice, circuite logice, unități de calculanților hard și soft asupra complexității sistemelor de calcul.

de abordare este reprezentați abstractă, el constând din mediul în care componenta este realizată sau în detaliile de implementare față de componentă superioară imediată. O astfel de nivela are scopul să formă unor nivale (componente) ieerarhice, fiecare componentă

- hardware - parte a microporogramelor;
- software - parte a programelor;
- software - parte a echipamentei;
- dispozitivele periferice;
- memoria;
- unitatea centrală de procesare (Central Processing Unit - CPU);
- hardware - parte a echipamentei;
- hardware - parte a niveli structurii.

Arhitectura unui sistem de calcul este:

frizează și căile de comunicare între acestea) sau la nivel logic (funcție fizică a unei componente în cardul structural).

- controlul tuturor componentelor SC;
- transmisiile de informații;
- memorarea de date;
- procesarea de date;
- funcțile de bază ale unui SC sunt:

O definiție similară a unui sistem de calcul (The American Heritage Dictionary of the English Language, 2000) este: un dispozitiv care efectuează calcule, în special o mașină electronică programabilă care poate exectua operațiuni aritmice, logice sau care asamblează, coreleză sau efectuează un alt tip de procesare a informației, cu viteză ridicată.

Numează, prelucrând date în vederea producării unor rezultate ca efect al programelor, un sistem de calcul (SC) un dispozitiv care lucraza automat, sub controlul unui program

16 biti	8 biti: contractie cu semn	8 biti: contractie cu zero		
FF80	80	Depășire!	Se schimbă bitul de semn	0000000010001000
0028	28	Depășire!	Se prelde 8 biti 1	0000000100000000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010000000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010001000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	0000000010000000
0028	28	Depășire!	Se prelde 8 biti 1	000101000
FF80	80	Depășire!	Se schimbă bitul de semn	0000000100000000
1111111100000000	100000000	Depășire!	Se prelde 8 biti 1	1111110000000000
0000000001010000	28	Depășire!	Se schimbă bitul de semn	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
0028	28	Depășire!	Se schimbă bitul de semn	0000000100000000
FF80	80	Depășire!	Se schimbă bitul de semn	0000000100000000
1111111100000000	100000000	Depășire!	Se prelde 8 biti 1	1111110000000000
0000000001010000	28	Depășire!	Se schimbă bitul de semn	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!	Se prelde 8 biti 1	000101000
F9A	9A	Depășire!	Se schimbă bitul de semn	0100000100010000
FE40	10011010	Depășire!	Se prelde 8 biti 1	1111110010001000
111111110011010	111111110011010	Depășire!	Se prelde 8 biti 1	1111110010000000
FF9A	00101000	000101000	000101000	0000000010100000
0000000001010000	28	Depășire!</		

funcționale (ALU, CU și.a.), componente ale calculatorului (CPU, memorie, sistem de I/O), iar nivelele ierarhice ale software-ului sunt reprezentate de limbajul mașină, limbajul de asamblare și limbajele de nivel înalt.

*Arhitectura (organizarea) unui sistem de calcul* se referă la acele atribute ale sistemului care sunt vizibile programatorului și care au un impact direct asupra execuției unui program: setul de instrucțiuni mașină, caracteristicile de reprezentare a datelor, modurile de adresare și sistemul de intrare / ieșire (I/O). Din punct de vedere organizatoric, componentele unui SC sunt (vezi figura 2.1.):

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

unde controlul și calea de date sunt componente ale procesorului (vezi paragraful 2.2.).

Această organizare este independentă de tehnologia hard adoptată pentru construcția sistemului de calcul. Orice componentă a unui SC poate fi încadrată în una din aceste 5 categorii.

Văzută dintr-un alt unghi, arhitectura unui SC este compusă din *mulțimea instrucțiunilor mașină și organizarea mașinii*.

**Mulțimea instrucțiunilor mașină** (*Instruction Set Architecture – ISA*) este o interfață cheie între nivelele de abstractizare, fiind interfața dintre hard și soft-ul de nivel scăzut (*low-level software*). O astfel de interfață permite unor implementări diferite ale SC să ruleze soft identic, caz în care vorbim despre calculatoare compatibile (de exemplu – "calculatoare compatibile IBM-PC" – nu au același hardware, dar răspund aceleiași ISA).

ISA definește:

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

**Organizarea unei mașini** se referă la:

- implementarea, capacitatea și performanța unităților funcționale;
- interconexiunile dintre aceste unități;
- fluxul de informație dintre unități;
- controlul fluxului de informație.

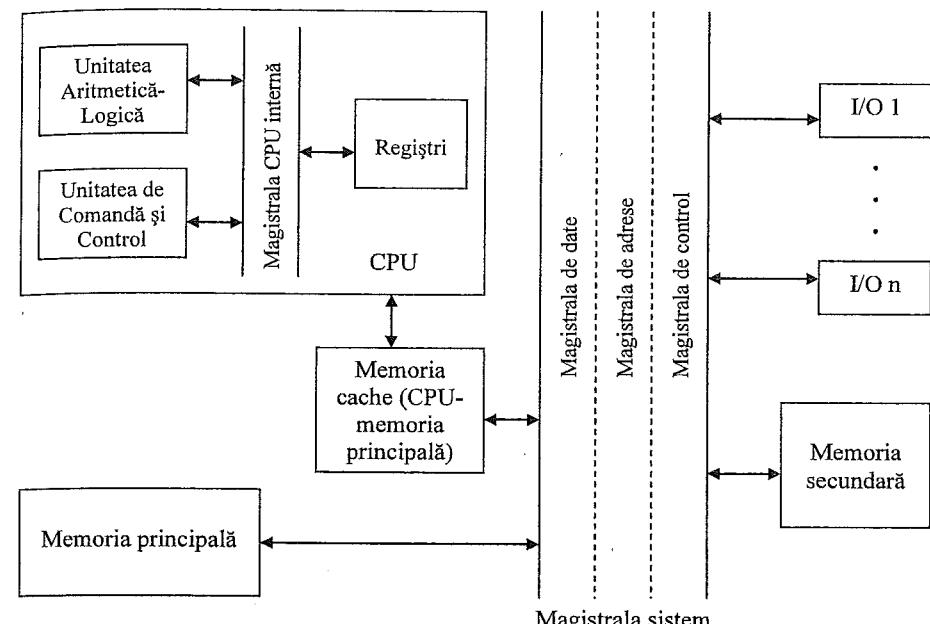


Fig. 2.1. Arhitectura unui sistem de calcul

Cea mai mare parte din calculatoarele momentului sunt construite pe baza *architecturii von Neumann*. Ideea de plecare este utilizarea memoriei interne pentru a stoca secvențe de control pentru îndeplinirea unei anumite sarcini – secvențe de programe; astfel putem vorbi despre mașini programabile. Acesta reprezintă așa-numitul concept al **programului memorat** (*stored-program concept*). În contrast, primele mașini erau construite pentru îndeplinirea unei anumite operații și era necesară modificarea acestora pentru a putea efectua un alt tip de operație.

Caracteristicile arhitecturii von Neumann sunt:

- atât datele, cât și instrucțiunile sunt reprezentate ca siruri de biți și sunt stocate într-o memorie read-write; e important de subliniat faptul că nu se poate face diferență între date și instrucțiuni prin simpla citire a unei locații de memorie - trebuie cunoscut ce reprezintă pentru a-i se stabili semnificația (de exemplu: valoarea 98h - poate reprezenta o valoare a unei variabile de dimensiune un cuvânt sau codul instrucțiunii mașină corespunzător instrucțiunii de asamblare 8086 CBW);
- conținutul memoriei se poate accesa în funcție de locație (adresă), indiferent de tipul informației conținute;
- execuția unui set de instrucțiuni se efectuează secvențial, prin citirea de instrucțiuni consecutive din memorie.

#### 2.2. UNITATEA CENTRALĂ (CENTRAL PROCESSING UNIT - CPU)

Procesorul controlereză modul de operare a calculatorului și executa în modul de procesare a datelor.

- obținerea instanțialilor care trebuie executate;
- obținerea datelor necesare instanțialor;
- procesarea datelor (execuția instanțialor);

- Utilizarea de Comunicații și Control (Control Unit – CU);
- Structura fizică a unui microprocesor (memorie, logica, control, interfațe);
- Rezistența – secesia sună disponibilă de tecnoare emporata a deteleri și informațiilor de control (instrucțiunile), de capacitatea mășii și utiliză a acces marie;
- magistrala interne CPU – dispozitive pentru comunicare intre componentele CPU și comunicare cu exteriorul, pentru traficul de informații.

Unitatea de Calculuri și Control este compozită din componente CFU care alcătuiesc un sistem binar de calculuri direct asupra acestora. Pentru accesarea lor este disponibile comunități de memorie secundară sau de la un dispozitiv de memorie în memoria primăplă. CU coordonează ciklul de execuție a instrucțiunii, aceasta și datele necesare trebuie aduse din memoria secundară pentru execuția unei instrucțiuni, după care funcționarea CU sunt codificate în combinaționale AND / OR) sau într-o memorie de tip ROM (Read-Only Memory).

Consecrada estructura unit CP si pasó din círculu de ejecutife a unet institucifón, plemento concurra estructura funcionala a procesosuniti: modulu de control si caldea de date (datopath). Modulu de

Loate operațiile efectuate de către microprocessor sunt sincronizate cu ceasul sistem, aceasta nesemnă că procesorul nu poate efectua de o manieră securvențială operarării mai rapid decât asupra unei adrese de la care rulează procesorul respectiv. Așteptării sunt, de asemenea, efectuate de către un ceas intern care este capabil să trage de la un ceas de 200 MHz care este capabil să trage de la un ceas de 1 / 200.000.000 secunde.

Teoremele de sincronizare și stabilitatea semnalelor electrice pe magistrala de control sunt obținute în cadrul unei sisteme de control a unui sistem de sincronizare și operării celor două sisteme. Semnalul de referință din starea 0 în I și apoi în II este numelese perioada de cca. 100 ms. În cadrul acestui interval de timp, se aplică un semnal de sincronizare la cele două sisteme. Semnalul de sincronizare este generat de un generator de semnal de sincronizare care este conectat la un controlor de sincronizare. Controlorul de sincronizare este conectat la un sistem de control al sistemului de sincronizare și este responsabil să genereze semnalul de sincronizare și să își sincronizeze următorul ciclu de cca. 100 ms.

## 2.1. Casual system (The System Clock)

controlor este raspunzător de comunicarea cu exteriorul și interpreterea instițuijilor și de transmiterea rezultatelor), precum și de controlul execujielui instițuijilor și de transmisie rezultatelor), precum și de exteriorul și interpreterea instițuijilor și de transmisie rezultatelor), precum și de exteriorul și interpritera emisierei semnalelor de control către călea de date și receptiunea semnalelor de stocare (registri de date), unități funcționale (memoria, ALU) și căi de comunicare. Practic, călea de date componentă de stocare (registri de date), unități funcționale (memoria, ALU), și căi de date la care se adaugă căile de date și de control ale instițuijilor și de transmisie rezultatelor.

al instrucțiunilor mașină a fost unul secvențial. Procesorul preia o instrucțiune, *petrece un număr de cicluri de ceas la execuția completă a acesteia*, după care trece la următoarea instrucțiune și.a.m.d. În general CPI se referă la numărul mediu de cicluri procesor per instrucțiune executată de procesor – în raport cu toate instrucțiunile ce compun un program sau în raport cu întreg setul de instrucțiuni al procesorului. De remarcat aici, însă, că deși o instrucțiune **MOV** se execută în același număr  $N$  de cicluri procesor, durata de execuție în timp (secunde) este diferită de la un procesor la altul. Astfel un procesor ce rulează la viteza de 200 MHz va executa o instrucțiune **MOV** în 20 nanosec, în timp ce un procesor ce rulează la 1GHz va executa exact aceeași instrucțiune **MOV** de aproximativ 5 ori mai rapid (4 nanosec).

Întrucât în zilele noastre s-a ajuns la o limită tehnologică din punct de vedere al frecvenței de ceas a procesoarelor, se urmărește creșterea vitezei acestora prin alte metode. Cea mai des întâlnită este paraleлизarea execuției instrucțiunilor mașină. Această paraleлизare la nivel de instrucțiune mașină se referă la paraleлизarea etapelor execuției unei instrucțiuni (vezi paragraful anterior). În loc să execute toate cele 5 faze ale unei instrucțiuni și să treacă abia apoi la următoarea, ne putem închipui o arhitectură în care, după execuția fazei **FETCH** pentru o instrucțiune **I1**, aceasta trece în faza **DECODE**, iar procesorul preia următoarea instrucțiune a programului în faza **FETCH**. Putem avea astfel în arhitectură prezentată mai sus maximum cinci instrucțiuni care se execută în paralel. **I1** în faza **STORE (S)**, **I2** în faza **EXECUTE (E)**, **I3** în faza **READ MEMORY (R)**, **I4** în faza **DECODE (D)** și **I5** în faza **FETCH (F)** (vezi figura 2.2.). Această tehnică de paraleлизare se numește *pipelining* (de la cuvântul *pipeline* – conductă).

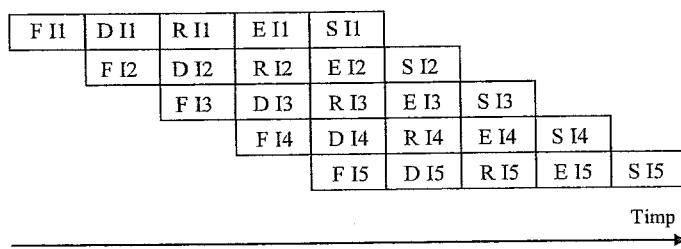


Fig. 2.2. Mecanismul de paraleлизare în execuția instrucțiunilor (pipeline)

De remarcat faptul că prin această tehnică nu se crește viteza de execuție a unei instrucțiuni (ea se execută în același număr de cicluri de ceas), ci se mărește numărul de instrucțiuni executate de către procesor per unitate de timp.

Folosind tehnica de pipeline ajungem la un caz cu totul surprinzător: putem să ajungem, pentru unele secvențe de instrucțiuni să măsurăm numărul de instrucțiuni / ciclu de ceas – *Instructions per Cycle (IPC)* - termen ce poate induce o stare de confuzie dacă ne gândim că un CPU nu poate executa mai mult de o singură acțiune / instrucțiune în fiecare ciclu de ceas. Tehnica de pipeline permite însă execuția *virtual paralelă* a mai multor instrucțiuni în același timp, fapt care

## Cap.2. Arhitectura sistemelor de calcul.

poate conduce la situații cu valori CPI medii subunitare. Cu cât pipeline-ul este mai lung și numărul de cicluri de ceas pentru instrucțiuni mai mic, cu atât crește factorul IPC pentru procesorul respectiv.

O problemă spinoasă de-a lungul timpului a fost măsurarea vitezei calculatoarelor. Pentru aceasta s-a pornit inițial cu frecvența de ceas a CPU. Aceasta s-a dovedit rapid a fi o măsură neadecvată, întrucât două procesoare (de exemplu – unul de 1GHz și unul de 500 MHz) pot rezolva o problemă în același timp, timp care depinde și de puterea de calcul a acestora (în exemplul nostru, puterea de calcul a procesorului de 500 MHz poate fi mai mare decât a celui de 1 GHz).

S-a introdus apoi noțiunea de *MIPS (Millions of Instructions per Second)* – care nu mai depinde de ciclul de ceas. Aceasta măsoară numărul (exprimat în milioane) de instrucțiuni (operând doar pe numere întregi) pe care le poate executa un procesor într-o secundă. *MFLOPS (Millions of Floating-Point Instructions per second)* reprezintă numărul (exprimat în milioane) de instrucțiuni în virgulă flotantă pe care un procesor le poate executa în unitatea de timp.

În general, un microprocesor este caracterizat de viteza de lucru, capacitatea maximă de memorie pe care o poate adresa (de exemplu – 1 MB la PC-uri), respectiv de setul de instrucțiuni pe care le poate executa (vezi ISA). Ca și criteriu de performanță se consideră deseori viteza de lucru a CPU, care depinde atât de frecvența ceasului intern, cât și de capacitatea de paraleлизare (organizarea execuției instrucțiunilor), dimensiunea regiștrilor interni și a magistralei de date, tipul microprocesorului sau dimensiunea memoriei cache a CPU (ca factor de influență a vitezei de comunicare cu exteriorul).

Viteza de lucru a CPU, ca și în cazul altor componente ale SC, a cunoscut în timp o continuă evoluție. De exemplu: microprocesorul IBM PC (1981) - 4.77 MHz (4 770 000 cicluri/secundă), microprocesorul Intel Pentium (1995) - 100 MHz (100 milioane cicluri/secundă), microprocesorul Intel Pentium 4 (2005) - 3.6 GHz. (3.6 miliarde cicluri/secundă).

### 2.2.2. "Dimensiunea" unui microprocesor sau răspunsul la întrebarea "ce înseamnă calculator pe n biți?"

Există două perspective sub care se interpretează răspunsul la această întrebare în literatură:

- perspectiva hard (punctul de vedere hardware): dimensiunea magistralei de date (de exemplu: Pentium are o magistrală de date pe 64 biți = 64 linii de date, astfel că la fiecare "memory cycle" procesorul poate accesa 8 octeți din memorie);
- perspectiva soft (punctul de vedere software): dimensiunea unui cuvânt de memorie (dimensiunea regiștrilor CPU);

În multe cazuri cele două perspective au coincis ca dimensiune. Diferențe de interpretare apar spre exemplu la:



exemplu de ordin  $\mu$ s (de aici denumirea *dinamică*). Datele nu sunt disponibile în timpul operațiilor de reactualizare. Deși timpul consumat de aceste operații constituie aproximativ 1% din timpul de funcționare, acestea contribuie la viteza de acces mai redusă față de alte tipuri de memorie (vezi SRAM). Conținutul unei asemenea memorii este organizat ca tablou bidimensional de biți. La citirea unui element al tabloului, se citește întreg rândul, care apoi este rescris (*refresh*). Pentru operația de scriere a unui element, se citește întreg rândul, se modifică elementul, apoi se rescrie întreg rândul înapoi. Elementele unei memorii DRAM sunt mai mici și mai ieftine decât elementele SRAM. Tipuri particulare de memorie DRAM: *Fast Page Mode DRAM* (FPM DRAM), *Extended Data Out DRAM* (EDO DRAM), *Burst EDO DRAM* (BEDO DRAM), *Synchronous Dynamic RAM* (SDRAM) – o versiune îmbunătățită a DRAM, *Double Data Rate SDRAM* (DDR SDRAM) – o îmbunătățire ulterioară a SDRAM, *Direct Rambus DRAM* (DRDRAM sau RDRAM), *Synchronous Graphics RAM* (SGRAM) – o formă a SDRAM specializată pentru adaptoare grafice.

**Observație:** O alternativă a memoriei DRAM ca organizare este *memoria flash*, întâlnită în prezent în dispozitive precum carduri de memorie, dispozitive flash USB, camere digitale, telefoane mobile. Acest tip de memorie are un cost per bit mai mic decât al memoriei DRAM, este non-volatile, dar de viteză mai mică la citire / scriere.

*Memoria SRAM (Static RAM)* este un tip de memorie semiconductor, volatile. După cum indică denumirea, conținutul unei memorii SRAM se păstrează atât timp cât sistemul este conectat la o sursă, spre deosebire de DRAM care necesită reactualizări periodice ale conținutului. Structura SRAM permite un acces mai rapid la locațiile acesteia, în comparație cu DRAM, motiv pentru care este utilizată ca memorie cache a CPU. Memoriile SRAM de viteză și capacitate mai mici sunt folosite atunci când se cere un consum de energie și cost scăzut, de exemplu pentru backup RAM cu sursă de tip baterie. Deoarece este mai puțin densă față de DRAM (conține mai puțini biți pe unitate de suprafață), în general capacitatea unei memorii SRAM este mai mică față de a unei memorii DRAM.

**Observație:** De obicei, raportul de capacitate DRAM/SRAM = 4-8; raportul de cost și timp de acces SRAM/DRAM = 8-16.

Deoarece nu poate fi (ușor) scrisă, memoria ROM este utilizată în general ca spațiu de stocare al *firmware-ului*, care nu necesită actualizări frecvente. Memoria ROM a multor sisteme de calcul din generațiile trecute (anii '80) conținea încă de la furnizare sistemul de operare, iar o parte din acestea includeau și un interpretor al limbajului de programare BASIC. Era cea mai practică alternativă, dischetele nefiind utilizate încă pe scară largă. În prezent, tendința este de a stoca cât mai puține informații în memorile ROM și o cantitate tot mai mare de date pe dispozitivele de memorare externe. Deși memoria ROM este de capacitate mică, avantajul principal este viteza mare de accesare a datelor. În general este întâlnită ca și componentă CPU, caz în care conține programul de control al acestuia, sau ca suport pentru BIOS. BIOS-ul (Basic Input/Output System) este un set de rutine de nivel scăzut care sunt responsabile de inițializarea sistemului, verificarea echipamentelor periferice din sistem și accesul primar la acestea. BIOS-ul poate fi considerat și *firmware-ul* placii de bază (vezi 2.4.4.), rutinele sale fiind printre primele care se

execută la pornirea unui calculator. De asemenea, poate să conțină rutinele cu funcțiile de bază pentru dispozitive precum telefoane mobile, controlere de rețea, controlere video.

Memoria ROM este în general cunoscută ca memorie scrisă în faza de producție, al cărui conținut nu poate fi modificat ulterior. Mai există, însă, câteva alte tipuri de memorie ROM al căror conținut poate fi rescris, precum:

- PROM (*Programmable Read-Only Memory*) – poate fi scrisă (programată) o singură dată cu ajutorul unui echipament specializat;
- EPROM (*Erasable Programmable Read-Only Memory*) – permite ștergerea conținutului (prin expunere la ultraviolete) și rescrierea acestuia cu ajutorul unui ‘programator EPROM’; numărul de rescrieri este însă limitat din cauza degradării progresive din faza de ștergere;
- EAROM (*Electrically Alterable Read-Only Memory*) – este folosită în general pentru memorarea permanentă a unor parametri ai sistemului, motiv pentru care este rar modificată; la un moment dat, poate fi alterată o parte a conținutului, bit cu bit;
- EEPROM (*Electrically Erasable Read-Only Memory*) – permite ștergerea electrică a întregului conținut sau doar a unui bloc din conținutul memoriei și rescrierea acestuia; exemplu – memoria flash a camerelor digitale, MP3 player-elor și.a.

### 2.3.2. Memoria externă / secundară

Memoria secundară reprezintă un dispozitiv de stocare pe termen lung a datelor, care nu sunt curent folosite de către CPU. În general este de capacitate mai mare și are o viteză mai mică de accesare a datelor față de memoria internă și face parte din categoria memorilor non-volatile.

Câteva dispozitive incluse în această categorie de memorie sunt: hard disk (HDD), floppy disk (FDD), compact disc (CD), DVD, banda magnetică, memoria flash.

#### Structura unui volum disc

Din punct de vedere fizic, un dispozitiv (volum) disc poate fi alcătuit din unul sau mai multe *discuri*, plasate concentric pe un *ax*, în jurul căruia se rotesc cu o viteză constantă. Informația poate fi înregistrată magnetic pe una sau ambele *fete* ale unui disc. Pentru accesarea informației, fiecare suprafață de memorare are asociat un *cap de citire / scriere*. Brațele capetelor de accesare corespunzătoare discurilor sunt situate pe un suport unic al dispozitivului. Mișcarea acestora permite deplasarea capetelor de acces radial pe suprafața discului, permitând astfel accesarea informației indiferent de localizarea acesteia față de axul central.

Din punct de vedere logic, o suprafață de memorare a discului este divizată în benzi concentrice numite *piste*. Pentru un dispozitiv ce deține mai multe suprafete de memorare, numărul de piste de pe aceste suprafete este același, iar pistele de aceeași rază formează un *cilindru*. O pistă este divizată în porțiuni numite *sectoare*. Numărul de sectoare este același pentru fiecare pistă a discului și fiecare sector are aceeași dimensiune. Un sector reprezintă în general unitatea de

- Hard disk - dispozitive de stocare a datelor pe suport magnetic, alcătuite din mai multe discurți; capacitatea de memorare a acestora este mare, astăzi fiind în prezent plană la unitatea de memorare.
- Floppy disk) - permit acces direct la date, au preț de produs și achiziționare foarte scăzut și sunt portabile; cele mai utilizate sunt cele de 1.44 megabytes (1 MB = 2<sup>20</sup> bytes);
- Unitate magnetică - permite acces secvențial la date, asigură o capacitate mare de stocare, sunt ieftine și sunt folosite în general pentru arhivare și memorare copiilor de siguranță (support de backup);
- CD (Compact Disc) - sunt dispozitive de capacitate de memorare relativă mare (650-700 MB), de tip Read-Only sau Read / Write, pentru cărți produse și dupăcare nu sunt costisitoare; de exemplu: WORM CD (Write Once, Read Many) - permit înregistrarea permanentă a unui volum mare de date (CD-ROM).
- DVD (Digital Versatile Disc) - sunt dispozitive disc similare CD-urilor, însă prezintă un mod de codificare a datelor different și o densitate a acestora mai mare; capacitatea de a accesa este în prezent de 4,7 GB plană la 17,1 GB; pot fi
- de tip Read Only (DVD-ROM) sau Read / Write (DVD-RW);

La TimpAccess mai plătește ca interfață de comunicare între CPU și disc. Dispozitivul, care prezintă multă similaritate cu modelul de controlerul de evoluție. De exemplu, capacitatea de stocare poate să crească cu aproape 100% în 1-1.5 ani, rata de transfer cu 40% / an, impuls de acces cu doar 8% / an, în timp ce raportul cost / capacitate scade de aproape 2 ori / an.

- TimPCalitate depinde de numarul de piste peste care trebuie sa de depласzeze capul de acasă și viteza de căutare a discului;
- TimPTransfer depinde de rata de transfer a datelor (bandwidth) caracteristica disponibilă.

CAP.2. Arquitectura sistémica de calculo. 47

Din cele menționate mai sus rezulta următori parametri (constante) ai unui dispozitiv disc:  
numărul de discuri, numărul capeteelor de citire / scrisere pentru un disc (numărul de fele active ale unui disc), numărul capeteelor de citire de pe o feală, numărul de sectoare de pe o pistă și numărul de sectoare mai scrisă în general utilizată este CHS (Cylinder-head-sector). După cum sugerază numele, identificarea seccorului se face astfel ca să realizezeaza o secție din discul respectiv (sector). De exemplu, o discete de 3.5 inch este conformată în general la dimensiunile de cap și cilindru. De altfel, numărul sectorilor este de obicei de 11, 16 sau 22, în funcție de capacitatea discului.

transfere date între disc și memorie intermă. Structura unui volum disc este reprezentată în figura 3.

<sup>46</sup> Arquitectura calculadora relacional. Limbal de asamblea Box86.

**Fig. 2.3.** Structura unui volum din ce poate fi calculat după formula:

$$\text{TimPAccess} = \text{TimPCautare} + \text{TimProiecte} + \text{TimTransfer}$$

Un criteriu de evaluare a performanței unui dispozitiv disc este

Fig. 2.3. Struktura unit volum disc

- Alte tipuri de discuri optice: *Blu-Ray Disc*, *High Density Digital Versatile Disc* (HD DVD), *Enhanced Versatile Disc*, *Holographic Versatile Disc* (deocamdată în faza de dezvoltare).

### 2.3.3. Ierarhia memoriei

#### Motivație

Pe măsură ce sistemele de calcul se dezvoltă, diferența de performanță dintre diferitele componente poate să crească tot mai mult. Cel mai grăitor exemplu este diferența dintre performanța CPU și cea a memoriei interne de tip DRAM.

Din cauza diferenței timpului de acces al CPU și al memoriei principale, CPU este nevoit să aștepte destul de mult pentru a primi datele din memorie. O asemenea diferență este defavorabilă și în cazul interacțiunii dintre memoria principală și memoria secundară.

Pentru a gestiona cât mai eficient accesul la date, un sistem de calcul deține un sistem complex al memoriei, în care combină memorie de capacitate mică, dar rapidă, și memorie de capacitate mare, însă de viteză redusă. Drept rezultat, un asemenea sistem se comportă în general ca o memorie rapidă, de capacitate mare. Nivelele ierarhice ale unui asemenea sistem pot fi reprezentate astfel:

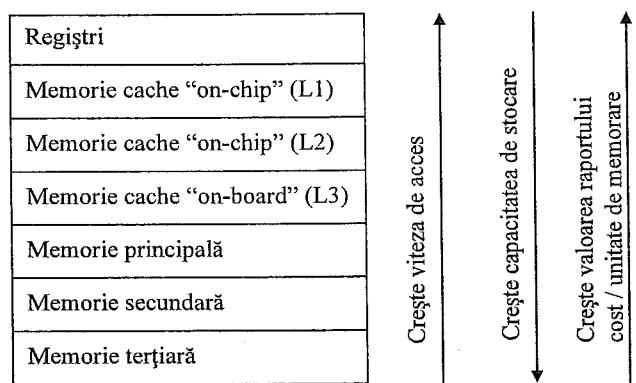


Fig. 2.4. Ierarhia memoriei

Se observă că ierarhia memoriei unui SC este organizată astfel încât nivelele de memorie de capacitate mai mică, însă mai rapide se găsesc mai aproape de procesor decât memoriile de capacitate mare, dar de viteză de acces mai mică.

În general, un nivel al ierarhiei reprezintă o submulțime de informații a unui nivel inferior: datele care se găsesc în primul nivel sunt aduse din următorul nivel de memorie, mai îndepărtat de CPU. Pentru a gestiona un asemenea trafic al datelor între diferite nivele este nevoie de funcții de transformare a adreselor de pe nivelul inferior către cel imediat superior. Întotdeauna datele sunt copiate numai între două nivele adiacente.

Eficiența unui asemenea sistem este asigurată de *principiul localizării*:

- *localizare temporală*: după accesarea unei date sunt mari şanse ca ea să fie accesată din nou în scurt timp => ar trebui să se mai rețină datea respectivă pentru o perioadă de timp (de exemplu: instrucțiunile dintr-o structură repetitivă sau ale unei subrute)
- *localizare spațială*: dacă se accesează o locație, sunt mari şanse să urmeze accesarea unor locații din vecinătatea primei => ar trebui ca la accesarea datei curente să se aducă un întreg bloc de informație care să conțină atât informația necesară în momentul curent, cât și informația conținută la adrese învecinate (de exemplu: variabile locale unei subrute sau elementele unui sir)

În urma acestor observații statistice s-au dezvoltat așa-numitele memorii de tip cache.

#### Memoria cache

O *memorie de tip cache* este o colecție de date ce reprezintă duplicarea valorilor originale stocate într-un alt tip de dispozitiv de memorare, a căror accesare pentru citire / procesare este mai costisitoare (ca timp) decât accesarea lor din cache. Memoria cache are capacitate mai mică, însă oferă un timp de acces la date cu mult mai rapid față de timpul asigurat de componenta asociată. Odată ce datele sunt aduse în memoria cache, ele vor fi accesate de aici, fără a fi nevoie de repetarea operației de copiere, scăzând astfel semnificativ timpul mediu de accesare.

Memoria cache exploatează localitatea spațială și temporală. O putem întâlni ca interfață între diferite nivele ale ierarhiei memoriei sau în asociere cu diferite dispozitive periferice sau chiar componente soft. Astfel, în prezent noțiunea de memorie cache reprezintă o tehnică de optimizare a accesului la date, indiferent de tipul clientului cache pe care îl deservește (memorie, dispozitiv periferic sau componentă software – sistem de operare sau aplicație utilizator). În general, însă, se face referință la memoria cache ca fiind interfața dintre CPU și memoria principală.

Ca exemplu de funcționare a unei memorii cache, considerăm interfața dintre CPU și memoria internă de tip DRAM. Aceasta poate fi alcătuită din unul, două sau trei nivele de memorie cache. O mare parte a sistemelor de calcul din prezent folosesc memorie cache pe două nivele (L1 și L2). Primul nivel de cache este integrat pe chip-ul CPU (cache “on-chip”) și asigură o viteză de acces similară CPU. Capacitatea acestei memorii poate varia de la 16, 32 până la 64, chiar 128 KB. Al doilea nivel asigură interfața dintre primul nivel și memoria internă și în general este memorie de tip SRAM. Este plasată de obicei tot pe chip-ul CPU, însă viteză de acces este mai redusă decât cea asigurată de primul nivel de cache, iar capacitatea de stocare este mai mare (512-1024 KB, sau chiar 2MB în cazul procesoarelor proiectate pentru server-e). În cazul în care

erformanța unită disponibilă de intrare / ieșire (*Input / Output - I/O*) depinde în general start de la nivelul de transfer a datelor (*I/O bandwidth* = cantitatea de date transmită și receptată într-un interval de timp) și împărțește alături de periferice *latency*.

#### 2.4.1. Magistrale – strucuri de interconectare

- dispõe de dispositivos periféricos para ligar a rede: impressora, mouse, scanner, monitor
  - dispõe de interface: interface serial, paralela, USB, FireWire, etc.
  - dispõe de dispositivo de armazenamento: disco rígido, unidade óptica, unidade magnética

magistratura și în studiile primări se efectuează într-o atmosferă de transporțare a informației (date, instituții, seminături de control) sau energetică inter-componențială (punct la punct), o singură magistrată poate realiza o conexiune între două sau mai multe componente.

În sistemele de calcul moderne o asemenea magistrală poate fi de tip paralel sau serial. Prin magistrală serială se transmite informația ca și în de biți (bit după bit). Magistrală paralelă transmite simultan informație prin mai multe firuri, măritându-se astfel rata de transfer. De exemplu, pe o magistrală paralelă de 16 biți se pot transmite simultan doi octetii. Această lucru nu înseamnă că pe o magistrală de 16 biți se poate să devină mai rapidă decât pe una de 8 biți, deoarece la transferul de date este implicată mai multă lărgime de bandă. Într-un sistem de transfer de date, magistrala serială este mai rapidă decât magistrala paralelă, deoarece este mai scurtă și mai ușoară de realizat.

Un sistem de calcul include magistrala interne (locale) care face legătura între componente interne și sistemele (de exemplu: Intel CPU și memoria internă) și magistrala extensie, penetrată în cadrul unei unități de memorie sau către alte masini.

magistrală de adresa (address bus) — informația comunicată este adresa locației de memorie pe care componenta solicitanta dorește să o acceseze (în cîndre sau scrisoare); dimensiunea magistralei determină capacitatea maximă de memorie adresabilă din sistem (de exemplu: sistemele de calcul cu regrătiri pe 8 biți definesc un magistrală pe 16 biți, de 64 biți).

magistrate de date (data bus) - transports à l'information de la magistrature intra-un SC;

Sub-sistemul de magistrală al unui SC poate numele de magistrală sistem (system bus). Raportat

Este esquema se aplica tanto para la evaluación de las partículas sólidas en el aire como para las líquidas.

In sistemele de calcul incluse magistratice interne (locale) care face legătura între componente interne și externe, pe măsură ce este realizată interacția cu utilizatorul, se poate observa că este deosebit de complexă.

Transfer mai mare decat magistrala paralela IDE/ATA; similar pentru interfața serială seriala USB în suport cu orice interface parallele la I2C.

mp se remarcă renunțarea la magistratul paralel și concentrarea pe magistratul serial care să recruteze la frecvențe de transfer mai mari (de exemplu: magistratul serial S-ATA are o frecvență de

Pe o magistrala paralela vineza de transfer a informasiile va fi neaparat mai deosebit pe una paralela. Dimpotrivă, datorita costurilor mari implicate de transmisia paralela datele, în ultimul

Exemplu, pe o magistratura de 16 ani se pot transmite simultan doi octetii. Acest lucru nu înseamnă să se transmită simultan informații din mai multe fizice, mărimi și se astfel ratează transferul de date.

1. sistemele de calcul moderne o asemenea magistratilor care să trimită informații ca să fie de bine (bit după bit). Magistratul paralel este serial. Prin magistratul serial se transmite datele în paralel sau serial. Prin

unique component.

ce conexiunile punct la punct, o singura magistrală potrivită realiză o conexiune între două sau mai multe unități sau întregi sisteme diferențiate compozante ale unui SC sau multe diferențe SC. Spre deosebire

magistrala este un subsistem prin care se transportă informație (date, instrucțiuni, semnale de

2.4.1. Magistrale – structuri de interconectare

despolarizante de sódio; aise (hard disk, floppy disk), banda magnética

- dispositiva de mitare sau ieșire: modern, placă de referință
- dispositiva de stocare: dischete (hard disk floppy disk) hărțile de date

- dispositivo de interar: tastatură, mouse, scanner
- dispositivo de ieșire: imprimația, monitor

Tipurile de dispozitive preferite des întâlnite sunt:

performanța unită dispozitivă de intrare / ieșire (*Input / Output - I/O*) depinde în general atât de timpul de transfer a datelor (*I/O bandwidth* = cantitatea de date transmită și receptată într-un interval de timp) și de timpul de rezoluție al dispozitivului preferit (*latency*).

#### 2.4. DISPOSITIVE PREFERÈCE

15

unde rezultă  $2^{16} = 64K$  locații de memorie adresabile, iar sistemele PC din prezent au magistrale pe 32 biți –  $2^{32} = 4G$  locații)

- **magistrale de control (control bus)** – transmit informație de control și semnalizare (de exemplu: semnale de citire / scriere a memoriei, cerere de utilizare a magistralei de date, acceptarea cererii de utilizare a magistralei, semnale de ceas, reset)

În funcție de tipul componentelor interconectate, magistralele interne pot fi:

- magistrale CPU-memorie: au arhitectură specifică producătorului, asigură o comunicare rapidă directă între procesor și memorie și sunt de lungime redusă;
- magistrale de I/O: au în general arhitectură standardizată și asigură o viteză ridicată în comunicarea informației dintre CPU, memorie și un controler de I/O

Cele mai cunoscute tipuri de magistrale de I/O sunt sau au fost: ISA (*Industry Standard Architecture*), PCI (*Peripheral Component Interconnect*) și AGP (*Accelerated Graphics Port*). Aceste magistrale de I/O permit comunicarea cu dispozitivul periferic prin intermediul unui controler.

Un controler în forma sa fizică se materializează printr-o placă de extensie atașabilă sistemului de calcul. De obicei, fiecare controler prezintă două interfețe:

- o interfață de comunicare cu procesorul prin intermediul magistralei de I/O. În funcție de tipul de magistrală folosit, această interfață poate fi de exemplu ISA, PCI sau AGP;
- o interfață de comunicare cu echipamentul periferic propriu zis. Această interfață diferă de la controler la controler în funcție de echipamentul periferic care îl este atașat.

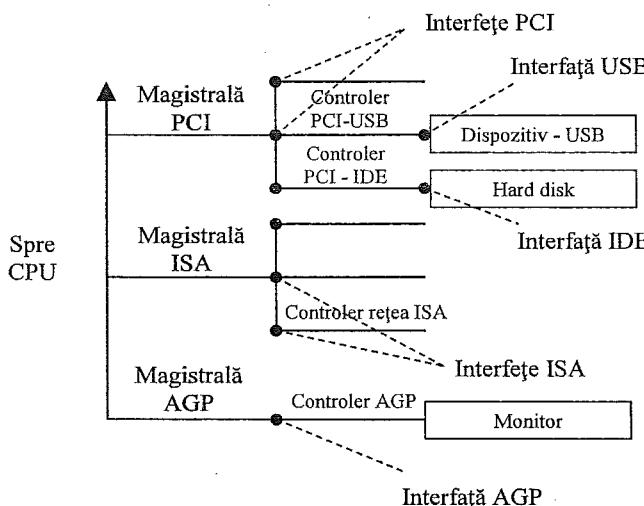


Fig. 2.5. Magistrale de I/O

Foarte des, vom folosi pentru controler și denumirea de adaptor sau chiar de placă (spre exemplu placă video, adaptor de rețea).

Practic prin interfață (fie că este vorba de interfața dintre controler și magistrala I/O internă, fie că este vorba de interfața dintre controler și echipamentul periferic) vom înțelege atât expresia fizică a acesteia (specificațiile de interconectare fizică, portul, slotul, mufa), cât și setul de caracteristici funcționale, protocoale, specificații logice necesare comunicării pe magistrala asociată.

#### 2.4.2. Magistrale I/O interne și interfețele asociate

##### Magistrala ISA (Industry Standard Architecture)

Este una dintre cele mai vechi tipuri de magistrale de comunicare cu echipamentele periferice. A fost introdusă de IBM la începutul anilor '80, rezistând cu succes până la sfârșitul anilor '90. ISA a fost dezvoltată mai întâi pe 8 biți, iar ulterior pe 16 biți, operând la 8 MHz. Aceste valori au fost potrivite pentru dimensiunea magistralei sistem și frecvența procesorului 286, permitând viteze de până la 16 Megaocetii/secundă. Odată cu creșterea vitezei procesoarelor și a foamei de lățime de bandă de către unele controlere și periferice, ca de exemplu adaptoarele video, hard disk-urile, controlerle de rețea, viteza oferită de o magistrală ISA a devenit insuficientă. Deși, teoretic, pe o magistrală ISA se pot obține viteze de până la 16 Megaocetii/secundă, vitezele reale sunt mult mai mici. A fost folosită cu succes pentru toate tipurile de controlere, însă s-a bucurat de succes până la sfârșitul anilor '90 pentru conectarea la calculator mai ales a controlerelor de rețea de până la 10 Mbps (Mbps = megabiți pe secundă), a placilor de sunet și a modemurilor. Calculatoarele personale de astăzi, păstrează încă o relicvă a acestui tip de magistrală. Portul pentru tastatură și mouse, iar în unele cazuri porturile seriale și paralele, precum și controlerul unității de dischetă, sunt conectate la o magistrală ISA care este cascadată la magistrala de I/O a sistemului prin intermediul magistralei PCI.

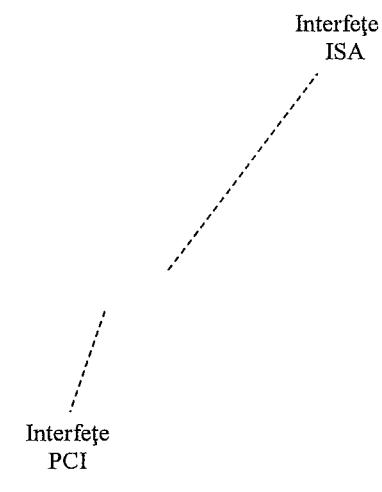


Fig. 2.6. Interfețele ISA și PCI din punct de vedere fizic în cadrul unui calculator personal

#### 2.4.3. Magistrale I/O extreme și interfețe associate

metrefata IDE sau ATA cum mai este cunoscută, a fost folosită de la milioane de ani în prezent.

meteza sa DDE sau ALA cum mai este cunoscută, a fost folosita de la mijlocul anilor '80 în cadrul unei comunități preferințelor de socotea cum ar fi hard disk-unite și unitate optică. Prin cele două extremități ale socotei se poate să se conecteze la un computer care să suporte acest tip de unitate.

permisă în moduri și moduri de a crește capacitatea de a rezista la presiunea exercitată de oțelul de la 48 de bini, lucru care permite folosirea teorica a dispozitivelor de erăzare pe 32 de bini. Au existat situații când proiectanții de IDE au sugerat să se extindă capacitatea de a rezista la presiunea exercitată de la 32 de bini, lucru care permite folosirea teorica a dispozitivelor de erăzare la capacitatea de a rezista la presiunea exercitată de la 40 de bini, ceea ce ar fi putut să aducă la o creștere a capacitatii de a rezista la presiunea exercitată de la 40 de bini.

CSI (Small Computer System Interface)

Cea ce este tip de magistrală S-a născut la mijlocul anilor '80, fiind cel mai des folosită pentru neconectarea dispozitivelor de stocare cum ar fi hard disk-uri, unități optice, unități de bandă magnetica. Poate că în să folosita și pentru conectarea slotură tipuri de periferice cum ar fi scannere și imprimante. Nu s-a bucurat niciodată de popularitate în calculatoarele personale, fiind numai utilizată mai des în calculatoarele Apple și în statifice și serverele Sun. Acest tip de magistrală, fiind mai scumpă și mai lentă decât celelalte, nu a reușit să își impună în mod semnificativ pe restul pieței.

la frecvențe încreștind cu 33 MHz. Aceste valori permit vîrfurile teoretice de pînă la 132 MHz acelerate și de secundă (33 • 10<sup>6</sup> • 4 octetă). Majoritatea controlerelor existente în momentul de față se conectează la aceasta magistrală; controlerele IDE/ATA pentru conectorile hard disk-uri, etc. Bise specifică și altor tipuri de calculatoare, nu numai calculatorelor personale, cum ar fi Power Macintosh.

S-a nascut din nevoiea de a înființa o societate care să producă și să vândă produsele de informație și tehnologie.

MHz, obtendo o vireza de aproximativi 266 MHz aceleri pe secunda — dublu faža de viteza oferita de magistrala PCI. Aceasta lăsime de banda este dedicata în întregime controlerului video, în

interfața AGP și interfața PCI, utilizândă paralela PCI, mată consumătoare de energie și controler video AGP de bandă, cum ar fi controlere IDE/ATA. Magistrala AGP 8x actuală, permite atingerea unor viteze de 8 ori mai mari decât specificația AGP inițială (2133 Megabitefi pe secundă).

**Fig. 2.7.** Interpretation of AGP as a controller

AGP vor fi îmlocuită de PCI cat și cea magistrala PCI Express, care promite, cel puțin teoretic, viteze de până la 4 GigaOctet/s astăzi disponibile controler înspri- procesor, cat și în sens invers.

Asamblatul format din controller, midifiret și tipul de tipul de magistrală. **LO** interfața la care este conectat același tip de interface cu interfața dintr-o unitate de caseta. Împreună cu interfața dintr-o unitate de caseta și echipamentul preferit este referit și sub numele de **multisala extrema**. Printre cele mai cunoscute multisale extreme sunt: **IDE/ATA**, **SCSI**, **LSI**.

Fiecare dispozitiv periferic conectat la o magistrală SCSI i se asociază un identificator – *SCSI id*. Numărul de biți pe care se reprezintă acest identificator implică și numărul de dispozitive care se pot conecta pe aceeași magistrală. Astfel, spre deosebire de IDE care permite doar două dispozitive periferice pe controler (magistrală), o magistrală SCSI poate suporta până la 8 sau 16 echipamente periferice. Un alt avantaj al controlerelor SCSI este că suportă *hot-swapping* (*hot-plugging*) – schimbarea în timpul mersului calculatorului a echipamentelor periferice conectate.

### Interfața serială

Comunica serial cu un dispozitiv periferic, transferul de date făcându-se pe principiul serial, bit cu bit. Cele mai des întâlnite periferice seriale sunt *mouse-urile*, modemurile (vezi paragraful 2.4.4) și terminalele virtuale. Controlerul serial la calculatoarele personale de azi este pe cale de dispariție, majoritatea perifericelor care se conectau la calculator pe această interfață conectându-se în prezent prin intermediul unei interfețe USB. Acolo unde este încă prezent, controlerul serial este legat de magistrala ISA sau PCI. Viteza maximă atinsă pe un port serial este foarte mică, 128000 biți /secundă ≈ 16 kiloocteți / secundă.

### Interfața paralelă

Este și ea o relicvă în calculatoarele moderne, încet renunțându-se la ea și datorită nevoii de a reduce costurile unui sistem de calcul și mai ales datorită numărului mic de periferice care mai folosesc acest tip de interfață. Principiul de comunicare pe o asemenea interfață este bineînțeles cel paralel – mai mulți biți sunt transferați în același timp pe mai multe fire fizice. În prezent, singurele echipamente întâlnite care folosesc această interfață sunt imprimantele. În trecut, interfața paralelă a fost folosită și pentru conectarea altor periferice cum ar fi camerele video sau scannerele. Producătorii de asemenea echipamente periferice au renunțat la interfața paralelă în favoarea celei USB. Ca și în cazul controlerului serial, unde mai este prezent, controlerul paralel este legat de magistrala ISA sau PCI.

Înainte de reducerea costurilor controlerelor de rețea și a răspândirii rețelelor locale, interfețele seriale și paralele au fost des folosite la legarea în rețea a două calculatoare. Chiar și azi, multe legături în rețeaua Internet sunt legături punct la punct, realizate între interfețele seriale a două calculatoare.

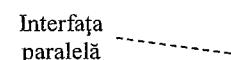
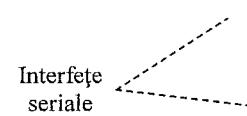


Fig. 2.8. Interfețe seriale și paralele

### USB (Universal Serial Bus)

Interfața USB s-a născut din nevoie unui mecanism universal de conectare a echipamentelor periferice externe la calculator, indiferent de tipul acestora: imprimante, tastaturi, *mouse-uri*, hard disk-uri, unități optice, camere video, scannere, telefoane mobile, etc. Viteza prevăzută de specificațiile actuale, USB 2.0, de 57 Megaoceteți pe secundă este practic suficientă pentru conectarea oricărui dispozitiv extern calculatorului, mai puțin a unui monitor. Principiul de comunicare cu acestea este serial, interfața USB folosind pentru date doar două fire fizice. Pe un singur controler USB se pot conecta până la 127 de echipamente USB, fiecare echipament primind un USB id reprezentat pe 7 biți (de fapt un id este consumat de controlerul USB însuși). Lucru deosebit de important, specificațiile USB permit adăugarea în timpul funcționării calculatorului a echipamentelor periferice USB sau înlocuirea acestora.

#### 2.4.4. Componentele unui calculator personal

Poate cea mai importantă componentă a unui calculator personal nu este procesorul, ci placa de bază. Placa de bază este suportul fizic pe care se monteză procesorul, memoria, pe ea sunt cablate fizic magistralele interne ale calculatorului, interfețele PCI, ISA și AGP.

La calculatoarele personale unele controlere (ISA, PCI și chiar AGP) sunt cablate pe placa de bază și fac parte integrantă din aceasta (termenul consacrat este de „*on-board*”). Spre exemplu, toate placile de bază au incorporat un controler IDE/ATA PCI. Chiar dacă acest controler nu este o componentă fizică distinctă care se conectează la calculator prin intermediul unei interfețe PCI, atât logic, cât și fizic, acest controler este conectat la magistrala PCI a calculatorului. De asemenea, controlerele USB, de rețea, sunet, cel serial și paralel sunt și ele prezente pe placa de bază a calculatoarelor personale de azi, fiind conectate cel mai adesea la magistrala PCI.

Cu toate aceste controlere integrate pe placa de bază, aceasta trebuie să prezinte interfețele fizice necesare conectării dispozitivelor periferice suportate de controlerele respective. Astfel, pe aproape fiecare placă de bază putem distinge următoarele interfețe fizice:

- conector pentru alimentarea placii de bază și, prin intermediul acesteia, a tuturor controlerelor, fie *on-board*, fie atașate în interfețele ISA, PCI sau AGP (nu și a componentelor periferice interne care sunt alimentate separat);
- slot (sau *socket*) pentru procesor;
- între două și patru sloturi pentru memorie;
- două interfețe IDE/ATA, care permit fiecare conectarea a două dispozitive de stocare internă (hard disk sau unitate optică);
- interfață oferită de controlerul unității de dischetă (FDD – *Floppy Disk Controller*). Pe această interfață se pot conecta până la două unități de dischetă;
- mai multe interfețe PCI, de obicei între trei și cinci;
- o interfață AGP pentru atașarea controlerului video;

Aceste interfețe sunt ilustrate în figura 2.9. Există situații în care echipamentul preferat nu se atașează direct interfeții oferite de controler sau de placă de bază, fiind necesar un cablu (sau "paniglică") prelungitor care să pară integrată a interfeței fizice de conectare oferita de controlerul respectiv. Spă exemplu, un hard disk nu se conectează direct interfeței IDE/ATA de pe placă de bază, fiind necesar un cablu (numit eronat "neuter" controler IDE) prelungitor de la interfața IDE a placii de bază la hard disk. De asemenea, un mouse serial se poate conecta la interfață serial serială prin intermediul unei cabluri prelungitor seriale.

Prezentăm în continuare lista echipamentelor preferate ce se pot conecta pe fiecare din trei interfețe fizice:

- interfețe PCI (vezi figura 2.6); permit conectarea unui controler video. Un controler video din interfața AGP (vezi figura 2.7), permite conectarea unui controler optică (CD-RW, DVD-ROM, etc). Pentru conectarea acestor echipamente la aceeași interfață este necesară o "paniglică" (ROM, etc).
- interfața IDE/ATA permite conectarea hard disk-urilor și unităților optice (CD-RW, DVD-ROM, etc), unde masterul sau unitatea principală este hard disk sau unitate optică IDE.
- interfața SCSI (vezi figura 2.7), permite conectarea unui controler de calculator.

USB suplimentară, placă de sunet, modemuri, controlere wireless (fără fir), etc., sunt procesorul printului magistrală PCI. Majoritatea controlerelor standard sunt pe placă comunica cu baza, însă la nevoie se pot adăuga atele noi: adaptori SCSI, TV-Tuner, controler IDE sau interfață serială și paralelă.

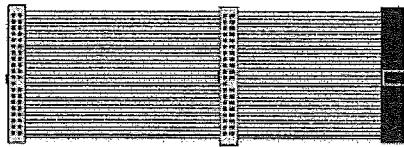


Fig. 2.10. Hard disk și cablu de conectare IDE/ATA

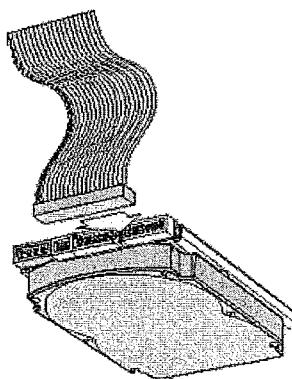


Fig. 2.9. Placă de bază a unui calculator personal

- interfața FDD: permite tot prin intermediul unei "panglici" atașarea a două unități de dischetă.

- interfața serială (vezi figura 2.8.): este încă folosită pentru conectarea mouse-urilor seriale, precum și a modemurilor externe. Modemurile sunt echipamente periferice care transformă semnalul audio analogic în semnal digital și invers. Sunt folosite pentru conectarea la Internet a calculatoarelor prin intermediul unei linii telefoniice obișnuite. Interfața serială este des folosită și pentru a controla și configura prin intermediul unei legături punct la punct echipamente active de rețea (switch-uri, routere) care nu sunt dotate cu un controler video, ci doar cu un controler serial.

- interfața de rețea: permite conectarea calculatorului la o rețea locală și, prin intermediul acesteia, la Internet;

- interfețele audio: permit conectarea la calculator a unor boxe, căști sau a unui microfon;

- interfețele PS/2 permit conectarea la calculator a unei tastaturi și a unui mouse.

- interfețele USB permit conectarea de echipamente periferice externe. Cele mai des întâlnite echipamente periferice USB sunt: tasturi și mouse-uri USB, echipamente de stocare (*memory-stickuri*, hard disk-uri, unități optice externe, unități de dischetă externe), telefoane mobile, imprimante, adaptoare pentru comunicații fără fir (*wireless*), *joystick*-uri și alte echipamente destinate jocurilor pe calculator, camere video și aparate de fotografat digitale, scannere, etc. Practic interfața USB va duce la dispariția totală din calculator a unor interfețe ca cele seriale, paralele, PS/2 și a *game port*-ului.

Conector RJ45

Conector AUI

Conector BNC

Fig. 2.11. Interfețe de rețea

Fig.2.12. Interfețe audio

Interfață PS/2



Interfețe USB

Fig. 2.13.  
Interfețe USB și PS/2

## 2.5. PERFORMANȚELE UNUI SISTEM DE CALCUL

În general sunt utilizate două tipuri de criterii de evaluare a performanțelor:

- *timpul de execuție (Execution Time)* = timpul în care este executată o sarcină, timpul de răspuns
- *rata de execuție (Throughput, Bandwidth)* = numărul de sarcini rezolvate într-un anumit interval de timp (zi, oră, secundă, milisecundă...),

de unde rezultă că îmbunătățirea performanței poate fi privită sub două aspecte:

- reducerea timpului de rezolvare a unei sarcini
- creșterea ratei de execuție.

Acordarea unei „note” întregului sistem de calcul este dificil de realizat, motiv pentru care se face referire la diferite aspecte ale performanțelor componentelor sale. De exemplu:

- CPU:
  - o timpul de execuție (perioadă de ceas redusă, execuție paralelă a instrucțiunilor)
  - o rata de execuție (MIPS, MFLOPS, planificarea eficientă a instrucțiunilor)
  - o capacitatea memoriei cache a CPU
- memoria cache: rata de transfer, hit rate vs. miss rate
- memoria internă: capacitatea, rata de transfer
- dispozitivele periferice: viteza de căutare, capacitatea de stocare, viteza de transfer, numărul de pixeli sau poligoane afișate pe secundă
- s.a.m.d.

Totuși, alături de performanțele individuale ale componentelor, trebuie avute în vedere și alte aspecte precum: compatibilitatea componentelor, tipurile de date gestionate sau de aplicații executate, sistemul de operare, disponibilitatea software-ului, costurile de proiectare sau costurile de achiziționare sau întreținere.

## 2.6. ARHITECTURA MICROPROCESORULUI 8086

### 2.6.1. Structura microprocesorului

Această structură este ilustrată în figura 2.14. Microprocesorul dispune de mai mulți registri generali pe 16 biți. El este format din două componente mari:

- **EU (Executive Unit)** care execută instrucțiunile mașină prin intermediul componentei ALU (*Arithmetic and Logic Unit*).
- **BIU (Bus Interface Unit)** este componenta care pregătește execuția fiecărei instrucțiuni mașină. În esență, această componentă citește o instrucțiune din memorie, o decodifică și calculează adresa din memorie a unui eventual operand. Configurația rezultată este depusă într-o zonă tampon cu dimensiunea de 6 octeți, de unde va fi preluată de EU.

89

Cap. 2. Arhitectura Sistemelor de Calcul.

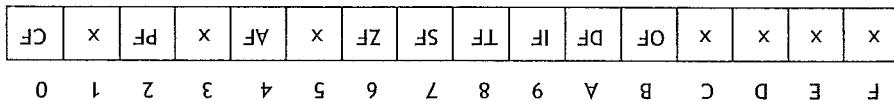
Hificare dimite regisimii AX, BX, CX, DX cu capacitate de 16 biti. Fiecare dintre ei poate fi privat in acelasi timp ca si mod format prin concatenarea (alipptare) a doi (sub)regisimi. Subregisimii superioi conțin cele mai semnificative 8 biti (partea HIGH) și regisimii de 16 biti din care face parte. Regisimii inferioi conțin cele mai puțin semnificative 8 biti astfel regisimii AH, BH, CH, DH. Subregisimii inferioi conțin cele mai puțin semnificative 8 biti (partea LOW) și regisimii de 16 biti din care face parte. Există astfel regisimii AL, BL, CL, DL.

Regisitru SP și BP sunt regiștri de estimări lucrului cu stivă. O stivă se definește ca fiind o zonă de memorie în care se pot depune succesiiv valori, extagereă lor ulterior facându-se în ordinea inversă depunerei.

Regeștiți **DL** și **SI** sunt reprezintă de index utilizat de obiect penitentiar să acceseze elementelor din structura de date cuvinte. Denumirea lor (*Destination Index și Source Index*) precum și rolurile lor vor fi clarificate în capitolul 4, secțiunea 4.4.2.

### 2.6.3. Flageolets

**Fig. 2.15.** Strucutra regiștrului de flaguri 8086



$$\begin{array}{r}
 100000110 \\
 01110011 \\
 \hline
 10010011 +
 \end{array}$$

Rezultatele depășesc dimensiunea unei cuvinte.

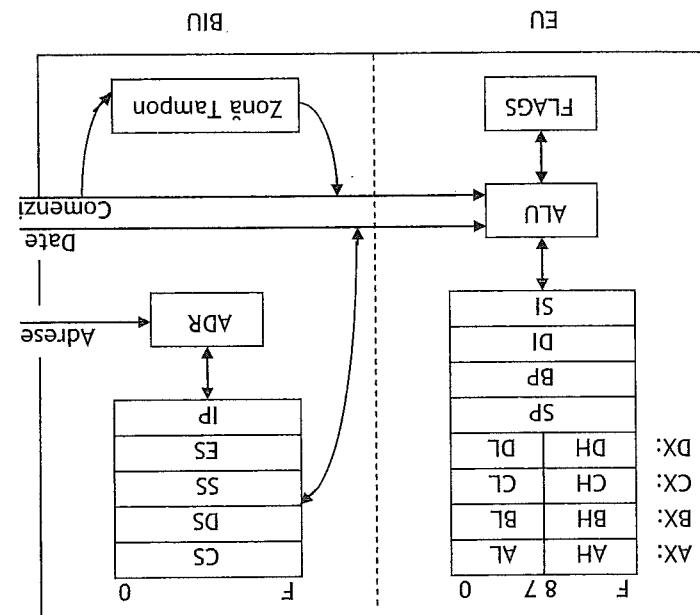
Regăsiștii CX este folosit în principal ca numărare (register counter) pentru instrucțiunile care au nevoie de indicații numerice.

Registruul BX este folosit în primul rând ca reģistratuur de baza. Põhiosene läbi selle seestatud aktsipliürite va tulla seefüüne 2.6.7.

Registru AX este registrul acumulator. El este folosit de către majoritatea instituțiilor ca unul dintre operații.

## 2.6.2. Registri generali EU

**Fig. 2.14.** Arhitectura microprocesorului 8086



**BII**, pregează unele instanțe în paralel, în sensul că într-un răspuns se sincronizează datele din cele două compuși și se obțin rezultatele. Cele două instanțe sunt separate de o cale dedicată.

• 9

rezultă un transport de cifră semnificativă. Valoarea 1 este depusă automat în CF. În absența transportului, în CF se va depune valoarea 0.

**PF (Parity Flag)** este flagul de paritate. Valoarea lui se stabilește în așa fel încât împreună cu numărul de biți 1 din reprezentarea rezultatului instrucțiunii să rezulte un număr impar de cifre 1.

**AF (Auxiliary Flag)** indică valoarea transportului de la bitul 3 la bitul 4 al rezultatului execuției instrucțiunii. De exemplu, în adunarea de mai sus transportul este 0.

**ZF (Zero Flag)** primește valoarea 1 dacă rezultatul instrucțiunii este egal cu zero și valoarea 0 la rezultat diferit de zero.

**SF (Sign Flag)** primește valoarea 1 dacă rezultatul execuției instrucțiunii este un număr strict negativ și valoarea 0 în caz contrar.

**TF (Trap Flag)** este un flag de depanare. Dacă are valoarea 1, atunci mașina se oprește după fiecare instrucțiune.

**IF (Interrupt Flag)** este flag de întrerupere. Asupra acestui flag vom reveni în capitolul 7.

**DF (Direction Flag)** este folosit când se operează asupra sirurilor de octeți sau de cuvinte. Dacă are valoarea 0, atunci deplasarea în sir se face de la început spre sfârșit, iar dacă are valoarea 1 este vorba de deplasări de la sfârșit spre început.

**OF (Overflow Flag)** este flag pentru depășire. Dacă rezultatul ultimei instrucțiuni nu a încăput în spațiul rezervat operanzilor, atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

Semnificațiile de mai sus sunt generale. De fapt, fiecare instrucțiune își specifică modul propriu de setare și interpretare a flagurilor.

#### 2.6.4. Regiștri de adresă și calculul de adresă

Prin definiție, *adresa unei locații de memorie* este numărul de octeți consecutivi aflați între începutul memoriei RAM și începutul locației respective.

Dată fiind capacitatea de 1 Mo a memoriei microcalculatoarelor care folosesc 8086, o adresă trebuie să se reprezinte pe 20 de biți. Capacitatea regiștrilor și a cuvintelor este de 16 biți. Problema care apare este cum se poate obține o adresă de 20 de biți folosind cuvinte de câte 16 biți?

Memoria se adresează deci pe 20 de biți și există doar regiștri de 16 biți. Pentru rezolvarea situației a apărut conceptul de segment de memorie. *Segmentul de memorie* reprezintă o succesiune continuă de octeți care are următoarele proprietăți: începe la o adresă multiplu de 16 octeți, are lungimea multiplu de 16 octeți și are lungimea de maximum 64 Ko. Deoarece adresa de început a fiecărui

segment este un multiplu de 16, cei mai puțin semnificativi 4 biți ai adresei sunt zero! Atunci când nu sunt posibile confuzii, vom spune simplu segment unei configurații de 16 biți care localizează începutul unui segment.

Vom numi *offset* sau *deplasament* adresa unei locații față de începutul unui segment. Deoarece un segment are maximum 64Ko, sunt suficienți 16 biți pentru a reprezenta orice offset.

Vom numi *specificare de adresă* o pereche de numere de căte 16 biți, unul reprezentând adresa de început a segmentului, iar al doilea deplasamentul în cadrul segmentului. În scriere hexazecimală o adresă se exprimă sub forma:

$$S_3 S_2 S_1 S_0 : O_3 O_2 O_1 O_0$$

Deci determinarea adresei din specificarea de adresă se face conform regulii:

$$a_4 a_3 a_2 a_1 a_0 := S_3 S_2 S_1 S_0 + O_3 O_2 O_1 O_0$$

unde  $a_4 a_3 a_2 a_1 a_0$  este adresa calculată (scrisă în hexazecimal). Deci configurația de 16 biți  $S_3 S_2 S_1 S_0$  care localizează începutul segmentului este înmulțită cu 16 (îi se adaugă o cifră 0 în baza 16, sau 4 cifre 0 în baza 2) și la rezultat se adună valoarea offsetului  $O_3 O_2 O_1 O_0$ . Acest calcul este efectuat de către componenta **ADR** din BIU.

Spre exemplu, specificarea 7BC1 : 54A3 indică adresa 810B3, ca rezultat al sumei 7BC10 + 54A3.

Este ușor de observat că există mai multe specificări pentru aceeași adresă. De exemplu, adresa de mai sus poate fi specificată și prin 810B : 0003. În acest fel este posibilă suprapunerea mai multor segmente în aceeași aria de memorie.

Acest mecanism de adresare este tipic pentru 8086 și poartă numele de *mod de adresare real (Real Address Mode)*.

Începând cu 80286, mai apare *modul de adresare protejat (Protected Virtual Address Mode)*, iar începând cu 80386 mai apar încă două moduri de adresare:

- *mod paginat*;
- *mod virtual 8086*;

Ultimele trei moduri au fost introduse pentru a permite adresarea de către IBM-PC a mai mult de 1 Mo. Asupra lor vom reveni în cadrul capitolului 10.

Arhitectura 8086 permite existența a patru tipuri de segmente:

- *segment de cod*, care conține instrucțiuni mașină;
- *segment de date*, care conține date asupra cărora se acționează în conformitate cu instrucțiunile;

Prin demersu, o lăsa să mă crede că se spețuiește doar oțelul, umărul ca segment de portă numele de adresa N.E.A.R (adresa apropiată). O adresă NEAR se

- **CS :** specificare\_offset sau
- **DS :** specificare\_offset sau
- **ES :** specificare\_offset sau
- **SS :** specificare\_offset.

O adresa FAR (adresa îndepărtată). O adresa FAR se poate specifica în trei moduri:

adresa în cadrul segmentului de memorie unde se află adresa de la segmentul portată numărul

**VALORES** que põe em evidência os riscos reais (riscos) que variam de tipo duplamente curvado, care conforme o endereço F&R.

Dupa cum se obseveră, și în prezentareea adreselor respective principali reprezentanți într-o entitate administrativă, este deosebită, în capitolul I, paragraful 1.3.2.3.: Preteza cea mai puțin semnificativă are adresa cea mai mică, iar parțea cea mai semnificativă are adresa cea mai mare.

2.6.7. Calculul offsetului unui operand. Moduri de adresaare

In cadrul unei miserițijini există mai multe moduri de a calcula distanța operană care corespunde încercării de a ajunge la obiectiv.



$$addressoffset = [BX \mid BP] + [SI \mid DI] + [constant]$$

## 2.6.6. Adresa FĂRĂ SI NEAR

Așa cum am văzut în secțiunea 2.6.4, pentru a adresa o locație din memoria RAM sunt necesare două cuvinte: unul care să indice segmentul, altul care să indice offsetul în cadrul segmentului. Pentru a simplifica referirea la memorie, microprocesorul preia, în lipsa unei alternative specifică, adresa segmentului din unul dintre registri de segment CS, DS, SS sau ES. Alegerea implicită a unui registru de segment se face după năște regula proprietăți instrucțiunii folosite.

2.6.6. Adrese FABR și NEAR  
pe mail mult de un octet.  
nu este permisă folosirea unei adrese de memorie dacă celelalte operațiuni constau în reprezentarea unei permițări de mai mult de un octet. De aici se deduce o restricție suplimentară privind operațiunile:  
Operațiunile maximă sunt parțial acționare, dacă apără, însemnă să fie o adresa de membru, și o constatăre

Fizicul oceli, numai cau ademintea instrumenta inserviciu numea de executie. Al doilea odată, numit octernod specifică pentru unelte instalații natura și locul operanzilor regiștri, memorie, constanța întreagă etc.). Unele instrucțiuni folosesc acest octet pentru a reprezenta în el o constantă scurtă (short), adică o constantă care poate fi valoare între -128 și 127. Majoritatea instrucțiunilor folosesc pentru reprezentare fie numai octetul cod, fie octetul cod urmat de octetul mod.

Formatul interbelic al unei instrucțiuni este variabil, el putând ocupa între 1 și 6 octeți. Semnificația

*numetristucjine destinatice, sursa*

O instuțiunea masculină 8086 are maxim umăr de operanți. Pentru cele mai multe dimite instrucțiuni, cei doi operanți pot fi numele de *swrd*, respectiv *destinat*. Dimite cele doi operanți, maxim umăr sunt se poarte alături de memoria RAM. Ceaală se alături de într-un registrul al EU, fie este o constanță sau o instuțiune specială sub forma:

### 2.6.3. REPFREZENTATIFEA MESTUČENJUHE MASAHE

*- segment suplementar (extrasegment).*

De aici rezultă următoarele moduri de adresare la memorie:

- *directă*, atunci când apare numai *constanta*;
- *bazată*, dacă în calcul apare BX respectiv BP;
- *indexată*, dacă în calcul apare SI respectiv DI;

Cele trei moduri de adresare a memoriei pot fi combinate. De exemplu, poate să apară adresare directă bazată, adresare bazată și indexată etc.

La instrucțiunile de salt mai apar două tipuri de adresări.

*Adresa relativă* indică poziția următoarei instrucțiuni de executat, în raport cu poziția curentă. Poziția este indicată prin numărul de octeți de cod peste care se va sări, număr ce ia valori între -128 și 127. O astfel de adresă mai poartă numele de *adresă scurtă (SHORT Address)*.

*Adresarea indirectă* apare atunci când locul viitoarei instrucțiuni este indicat printr-o adresă, aflată într-o locație, a cărei adresă este dată ca operand instrucțiunii de salt. Desenul din figura 2.16 sugerează acest mecanism.

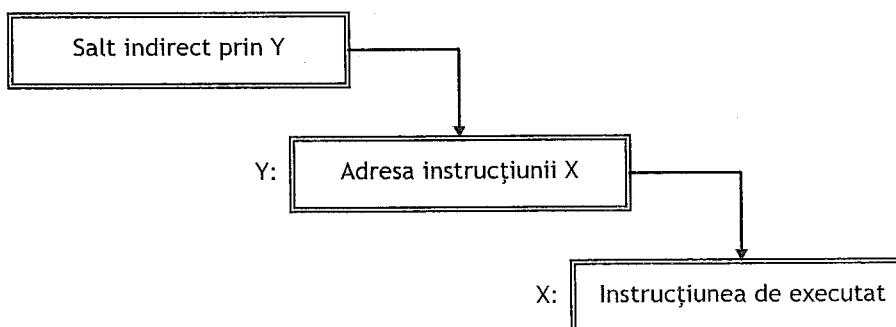


Fig. 2.16. Adresarea indirectă

Adresarea indirectă asigură o mai mare flexibilitate în controlul succesiunii instrucțiunilor. De exemplu, în figura 2.16 conținutul locației Y poate să difere de la un moment la altul a execuției programului, ceea ce permite execuția la momente diferite a unor instrucțiuni diferite în urma saltului indirect prin Y.

Adresarea indirectă poate fi la rândul ei adresare indirectă NEAR sau adresare indirectă FAR.

## CAPITOLUL 3

### ELEMENTELE LIMBAJULUI DE ASAMBLARE

*Limbajul de asamblare* al unui calculator este un limbaj de programare în care setul de bază al instrucțiunilor coincide cu operațiile mașinii și ale cărui structuri de date coincid cu structurile primare de date ale mașinii.

*Limbajul mașină* al unui sistem de calcul (SC) este format din totalitatea instrucțiunilor mașină puse la dispoziție de procesorul SC. Acestea se reprezintă sub forma unor siruri de biți cu semnificație prestabilită. În 2.6.5. am prezentat formatul instrucțiunilor mașină pentru 8086.

În ultimă instanță deci, orice program apare în calculator ca un sir (mare!) de biți. A manipula aceste structuri sub un astfel de format este extrem de dificil. Limbajul de asamblare vine să ușureze această sarcină. Astfel, codurile instrucțiunilor mașină se rescriu sub forma unor cuvinte simbol, numite *mnemonice*, suficient de sugestive pentru a indica semantica instrucțiunii respective. Adresele de memorie frecvent folosite (fie ca destinații ale unor salturi în program, fie ca denumiri asociate unor locații de memorie) sunt notate și ele prin niște cuvinte simbol, numite *etichete*.

Procesul de translatare în cod mașină a unui program scris în limbaj de asamblare se numește *asamblare* și este realizat de un program de conversie numit *asamblor*. În afară de înlocuirea instrucțiunilor simbolice ale programului sursă prin cod mașină, asamblorul asigură și prelucrarea adreselor simbolice, prin acest lucru înțelegându-se că fiecare adresă simbolică (*identificator*) este înlocuită prin adresa fizică corespunzătoare (număr).

Principalele servicii oferite de către un asamblor (și în consecință și de către asamblorul 8086) constau în:

- detectarea erorilor de sintaxă
- traducerea din scrierea cu mnemonice în cod binar
- generarea de octeți folosind pseudoinstrucțiuni
- calculul unor adrese de salt (deplasamente) folosind operații cu etichete
- evaluarea în timpul translatării a unor expresii aritmetice simple
- posibilitatea asamblării condiționate
- posibilitatea definirii și utilizării de macroinstrucțiuni

Elementele cu care lucrează un asamblor sunt:

deosebită este totuște important să punctăm semantica asociată înstării cu numărul și direcția lor dimi exemplul ce se vor prezenta prim referirea la noțiunile și concepție definite astăzi.

deosebie este faza de importanță să punctăm semantica asociată înstănciunilor și direcțiilor din exemplul ce se vor prezenta printreferirea la noțiunile și conceptele definite astăzi.

Formatul unei limbi scrise în limbajul de asamblare 8086 este următorul:

Căracterele din care poate fi constituită o elicitere sunt următoarele:

A - Z a - z — @ \$ ? 0 - 9

1) *etichete de cod*, care apar în cadrul secvențelor de instrucțiuni (deci în cadrul unor sekmene de cod) cu scopul de a defini destinația de transfer ale controlurilor în cadrul unui program. O etichetă de cod este o scrisoare de la unul la altul, care să specifică unde trebuie să se execute următoarea secvență de instrucțiuni. De exemplu, dacă în cadrul unei secvențe de instrucțiuni se întâlnește eticheta `IF A < B THEN`, atunci procesatorul va să execute următoarea secvență de instrucțiuni, după ce va să verifice dacă valoarea din registru A este mai mică decât valoarea din registru B. Dacă este astfel, atunci următoarea secvență de instrucțiuni va fi executată, înainte de a se executa următoarea secvență de instrucțiuni.

2.2. Encadrare de ambele părți, către autoritățile publice locale și naționale, să se respecte normele de variaabilitate din limba română și să se respecte normele de nivel înalt.

Vă dorarea unui etichete să vă informează efectiv ce număr într-un reprezentând adresa

Pentru un programator obisnuit cu limbajele de nivel înalt acest lucru este evident usor de acceptat în cazăul etichetelor de cod, deoarece este normal ca desemnării unui sat să fie specificată print-o adresă. Apărării în mod natural întrebarea: de ce să-l lut accesării de la intrare și pentru o adresă? Adică, de ce să hotărătiți să-l lăsați să se adreseze unei variabile ca identificatorul respectiv să fie asociat cu valoarea ei în cazul numărului său? De altă parte, dacă lucrurile stau în acest fel și de fapt simbolul p asociat unei variabile va desemna în limbajul de assemblea să cum vom

putea atunci avea acces la conținutul acelei locații? Avem oare nevoie de o altă notație specifică similară unui limbaj de nivel înalt de genul `*p` (în C) sau `p^` în Pascal (unde `p` este o variabilă pointer, deci desemnează o adresă)?

Nu, nu avem nevoie de o notație sintactică diferită sau specifică pentru a diferenția între accesarea adresei și respectiv a conținutului unei locații cu nume ci acest lucru se va face în funcție de contextul utilizării. Este foarte important să înțelegem aceste mecanisme: uneori, simbolul asociat unei variabile (numele său) va desemna NUMAI adresa sa (conform definiției) – spre exemplu cazul în care manipulăm doar adresele unor locații fără a fi interesați de conținutul acestora; alteori, când vom dori manipularea conținutului unei locații vom nota acest conținut tot cu acel nume, însă contextul utilizării aceluia simbol (adică alegerea adecvată a unei instrucțiuni care să stie să extragă acel conținut de la adresa furnizată prin specificarea numelui variabilei) va provoca interpretarea simbolului respectiv drept conținutul acelei locații (similar utilizării numelor de variabile în cazul limbajelor de nivel înalt).

Decizia în discuție s-a luat pentru a putea oferi, în funcție de contextul utilizării, acces fie la valoarea unei variabile fie la adresa sa, fără a utiliza operatori suplimentari în acest scop. Să reținem că limbajul de asamblare utilizează în mod intensiv lucrul cu adrese, deci promovarea unor mecanisme centrate pe lucrul cu adrese este absolut justificat.

Spre exemplu în cadrul instrucțiunilor:

```
lea ax, v      ; încarcă în registrul ax adresa variabilei v
mov ax, v      ; încarcă în registrul ax conținutul variabilei v
```

deși se utilizează același simbol `v` ca operand sursă, interpretarea sa va veni din contextul utilizării adică din tipul de instrucție folosit: instrucțunea `lea` va considera `v` drept adresă, în timp ce `mov` îl va interpreta drept conținut (de fapt mai exact și instrucțunea `mov` va considera într-o primă fază faptul că `v` desemnează o adresă numai că în plus, `mov` fiind o instrucție de transfer a valorilor, va provoca și extragerea valorii de la acea adresă!). Cu alte cuvinte, operația de dereferențiere (extragerea valorii de la o anumită adresă specificată) are loc implicit, în funcție de contextul utilizării aceluia simbol.

Acceptarea în cadrul instrucțiunilor și expresiilor limbajului de asamblare a numelor de variabile drept adrese oferă posibilitatea efectuării aritmetică de adrese (*pointer arithmetic* – adunare, scădere de constante la pointer sau scădere de doi pointeri). De exemplu, expresia `b-a` (cu `a` și `b` nume de variabile de memorie și `-` desemnând un operator și nu o instrucție) va desemna o aritmetică de adrese în limbajul de asamblare și nu o scădere a două “conținuturi” de variabile. Se remarcă astfel aici influența pe care limbajul de asamblare a avut-o și în deciziile de proiectare ale unor limbaje de nivel înalt, cum ar fi limbajul C spre exemplu, despre care știm de asemenea că acceptă aritmetică de adrese și că “numele unui tablou în C este de fapt adresa sa de început”! Această situație nu face decât să confirme că și în limbajul C numele de variabile (tablouri cel puțin) reprezintă și ele adrese de memorie la fel ca și etichetele de date din limbajul de asamblare.

Iată încă un argument pentru a califica limbajul C aşa cum o fac unii autori drept “*the most low-level high-level language*”!

În concluzie, este foarte important să reținem că în limbajul de asamblare, valoarea asociată simbolului ce desemnează o variabilă (etichetă de date) este prin definiție adresa sa. Totuși, acest lucru nu înseamnă că referirea la numele variabilei nu va putea însemna niciodată accesul la conținutul său! După cum am precizat și exemplificat mai sus, această distincție se face în funcție de contextul utilizării numelui respectiv. În cadrul anumitor expresii și instrucțiuni un același simbol va reprezenta adresa iar în altele va reprezenta conținutul de la acea adresă. Înțelegerea deplină și exactă a acestor diferențe de interpretare reprezintă una dintre cheile aprofundării mecanismelor de execuție la nivelul limbajului de asamblare. Studiul utilizării instrucțiunilor (capitolul 4) și experiența de programare care va fi câștigată de programator cu această ocazie vor conduce cu siguranță la o utilizare adecvată și la interpretări corecte din partea programatorului relativ la problematica de mai sus.

Există două tipuri de *mnemonice*: mnemonice de *instrucțiuni* și nume de *directive*. *Directivele* dirijează asamblorul. Ele specifică modul în care asamblorul va genera codul obiect. *Instrucțiunile* dirijează procesorul. În momentul asamblării ele sunt transformate în cod obiect.

*Operanzi* sunt parametri care definesc valorile ce vor fi prelucrate de instrucțiuni sau de directive. Ei pot fi registri, constante, etichete, expresii, cuvinte cheie sau alte simboluri. Semnificația operanzilor depinde de mnemonica instrucțiunii sau directivei asociate.

*Comentariile* sunt folosite numai de către utilizator pentru documentarea programului, fiind ignorate de către asamblor. Orice text aflat după punct și virgulă este considerat comentariu.

Parantezele drepte precizează că prezența elementului respectiv este opțională. Așadar, putem avea linii cu toate cele patru elemente prezente, după cum putem avea și linii fără nici un element (linii vidă) sau linii formate cu doar unul dintre cele patru elemente.

## 3.2. EXPRESII

O *expresie* constă din mai mulți operanzi care sunt combinații pentru a descrie o valoare sau o locație de memorie. *Operatorii* indică modul de combinare a operanzilor în scopul formării expresiei. Expresiile sunt evaluate în momentul asamblării (adică, valorile lor sunt determinabile la momentul asamblării, cu excepția celor părți care desemnează conținuturi de registri și care vor fi determinate la execuție).

### 3.2.1. Moduri de adresare

Operanzii instrucțiunilor pot fi specificați în diferite forme, numite *moduri de adresare*. Modurile de adresare indică procesorului modalitatea de a obține valoarea reală a unui operand în momentul execuției.

Chiar dacă adresele fizice de înțepătă de se mențină pot fi cunoșcută în mod logic numai după închirierea programului în memoria, depășirea unei limite de memorie, deosebit de semnificativă în cadrul acelor segmente ale eticheteelor declarate acolo (de cod sau de date) sunt și vor rămâne valoți constante pe impulsexchuiți și ce este și mai important aceste valori sunt determinabile la momentul asamblării! Aceasta se întâmplate deoarece dacă segmentele sunt generate la momentul sămbătrâii (se poate să nu

acacea și semnănd „salt cu 4 oțetii mai jos față de poziția curentă”.

↑[400]mp

Va fi evaluata la momentul asamblarii de exemplu sub forma

; salt în cadrul programului la o etichetă de cod Etjmp et

Pe măsură oricărui de descriere a variabilei și a tipurii de date (adunătură de reprezentare) se specifică la ea ocazia de declarată variabila și — în cadrul unei constante relativ la execuția programului — astfel calculabilă la momentul asamblării. În mod similar, o instrucțiune de tipul

împreună cu un alt segment de date, de la adresa ax, 0008 ; distanța de 8 octeți între de începutul segmentului de date

va putea fi evaluata la momentul asamblarii except de exemplu

• transferă în registrul axă a deplasamentului variabilei  $V_{\text{ea}} \text{ax}, V$

Este hotările importante de seminat raportul ca pe lângă constanțele numerice „clasică”, echivalente ca semnificație celor din limbajele de nivel înalt, la momentul asamblării, asamblorul poate determina și o altă categorie de valori care reprezintă constanțe pe tot parcursul execuției programului și îl numește deplasamentele echivalente de date și de cod. De aceea, de exemplu, o instrucțiune de genul celei amintite mai sus:

Deplasametul unei etichete – violare constanta determinabila la momentul asamblarii.

**Constatăre** într-o impunere zecimă constă în tipul specific de constanțe ce pot fi utilizate numai pentru înjumătățirea variabilelor codificate binar zecimal (BCD). Cu numere reale se poate opera numai în prezenta unei coprocesor matematic.

Dacă este deosebit de scurtă, se poate scrie într-un singur rând, începând cu prima linie și terminând cu ultima linie.

Pentru a lucra întrucât se poate, trebuie să cunoască și să utilizeze principiile de memorie. Întrucât există o limită maximă în ceea ce privin depozitarea de informații în memoria implicită, trebuie să se folosească metode care să îmbunătățească eficiența memoriei.

Cap.3. Elementele limbajului de asamblare.

---

[View this post on Instagram](#) [View on Facebook](#)

Necatru în modul de următoare, cele trei tipuri de operații sunt operațiuni menținere și operare și se execută imediat, în momentul încreșterii programului pentru adresaera directă și în momentul registrării în memorie. Vă ilustrăm operațiunile de calculări în momentul sămbalării pentru operațiuni de memorie. Pentru operațiuni de memorie, se calculează adresa de memorie și se scrie în memoria de adresaare, ceea ce rezultă în următoarea formă:

### 3.2.1.1. Utilizzare operazioni di mediazione

*Copelandia imbecilla* sunt tenui, diri, grise humerale costitutae cimoscute sed carunculae in moniliis.

**Constantante** sunt utilizate ca operanți în expresii. Limba SQL de asemenea renumărește patru tipuri de valori constante: **interger**, **string**, **numerice reale și constante impachetate codificăte binar secrete**.

Constatată într-un se specifică prin valori binare, octale, zecimale sau hexazecimală. Baza de numerarărie se dă prin un separator al basenilor de numere și după ultima cifră a numărului, astfel: pentru numere binare specificează 2 sau 0, pentru numere zecimale - 10 sau 11, pentru numere octale specificează 8 sau 0, pentru numere hexazemicală - 16 sau 15.

Întrucât folosind baza de numerale în specializatul metiereză și întrucât folosind baza de numerale în specializatul metiereză împlicită, în final, baza de numerale împlicită este cea decimală. De asemenea, se poate să modifice cu ajutorul directivei RADIX, care are următoarea sintaxă:

unde expresie trebuie să fie unul din numerele 2, 8, 10 sau 16.

RADIX expressive

"Acum... maestrul... este un farseur", Acum... maestrul...

sublinia în acest sens aici importanța introducerii conceptului de *contor de locații* la nivelul limbajului de asamblare!). Semantic, "constanță" acestor valori derivă din regulile de alocare adoptate de limbaje de programare în general și care statuează că ordinea de alocare în memorie a variabilelor declarate (mai precis distanța față de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanțele salturilor destinație în cazul unor instrucțiuni de tip *goto* sunt valori constante pe parcursul execuției unui program.

Adică: o variabilă odată alocată în cadrul unui segment de memorie nu își va schimba niciodată locul alocării (adică poziția sa față de începutul aceluia segment) iar această informație determinabilă la momentul asamblării derivă din ordinea specificării variabilelor la declarare în cadrul textului sursă și din dimensiunea de reprezentare dedusă pe bază informației de tip asociate.

Tipul de dată în limbaj de asamblare înseamnă în mod explicit dimensiune de reprezentare (directivele de generare a datelor specifică în mod explicit acest lucru) iar într-un limbaj de nivel înalt dimensiunea de reprezentare este dedusă implicit pe baza tipului de dată specificat: de exemplu, în Turbo Pascal o declarație de tipul **var a:real;** va provoca alocarea unei zone de 6 octeți în cadrul segmentului în care apare o astfel de declarație, deoarece `sizeof(real)` = 6, iar o declarație de tipul **var c:char;** va provoca alocarea unei zone de 1 octet, deoarece `sizeof(char)` = 1.

În concluzie, putem spune că interpretarea de nivel scăzut a noțiunii de tip de dată la nivelul arhitecturilor sistemelor de calcul actuale este cea de dimensiune de reprezentare, deoarece indiferent la ce limbaj de programare ne referim, efectul unei declarații de variabilă va duce în cele din urmă la un loc fix de alocare în cadrul unui segment de memorie pe o dimensiune de reprezentare dedusă pe baza informației de tip.

### 3.2.1.2. Utilizarea operanzilor registru

*Regiștri* sunt probabil cei mai des folosiți operanzi în cadrul instrucțiunilor limbajului de asamblare. Ei pot servi ca operanzi sursă sau ca operanzi destinație, putând conține și adrese de salt pentru instrucțiunile rezervate acestui scop. În plus, există unele instrucțiuni care pot fi folosite numai cu operanzi regiștri și instrucțiuni care pot fi folosite numai cu anumiți regiștri. Deseori, instrucțiunile au coduri mai scurte (și operațiile sunt mai rapide) dacă este specificat registratorul acumulator (AX sau AL). Regiștrii micropresorului 8086 au fost prezentate capitolul 2, iar instrucțiunile care îi vor utiliza ca operanzi vor fi prezentate în capitolul 4.

Operanzii-registru sunt formați din datele memorate în regiștri. Modul de *adresare directă* în cazul regiștrilor înseamnă folosirea valorii reale din interiorul registratorului în momentul execuției instrucțiunii (de exemplu **mov ax,bx** – provoacă transferul valorii din registrator *bx* în registrator *ax*). De asemenea, regiștrii pot fi folosiți *indirect* pentru a indica locațiile de memorie (de exemplu instrucțiunea **mov ax,[bx]** va provoca transferul cuvântului de memorie de la adresa desemnată de *bx* spre registrator *ax*), așa cum va fi prezentat în 3.2.1.4.

#### 3.2.1.3. Utilizarea operanzilor din memorie

Operanzii din memorie (termen care include și noțiunea clasică de *variabilă* cunoscută din limbajele de nivel înalt) se împart în două grupuri: operanzi cu *adresare directă* și operanzi cu *adresare indirectă*.

Când se dă un operand în memorie, procesorul calculează adresa datelor care vor fi prelucrate. Această adresă se numește *adresă efectivă*. Așa cum se va vedea în continuare calcularea adresei efective depinde de modalitatea în care este specificat operandul.

**Observație.** După cum rezultă și din 2.6.5., nu sunt admise operațiile pentru care atât sursa cât și destinația sunt operanzi din memorie.

Operandul cu *adresare directă* este o constantă sau un simbol care reprezintă adresa (segment și deplasament) unei instrucțiuni sau a unor date. Acești operanzi pot fi *etichete* (de ex: **jmp et**), *nume de proceduri* (de ex: **call proc1**) sau *valoarea contorului de locații* (de ex: **b db \$-a**).

**Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării (assembly time)** - vezi 3.2.1.1. Adresa fiecărui operand raportată la structura programului executabil (mai precis stabilirea segmentelor la care se raportează deplasamentele calculate) este calculată **în momentul editării de legături (linking time)**. Adresa fizică efectivă este calculată **în momentul încărcării programului pentru execuție (loading time)**.

Adresa efectivă este întotdeauna raportată la un registru de segment. Registrul de segment implicit pentru adresarea directă a datelor este cel specificat în directiva **ASSUME** corespunzătoare (a se vedea în acest sens și 3.3.1.). Segmentul implicit poate fi înlocuit cu ajutorul operatorului de prefixare segment (notat ":" și care se mai numește, 'operatorul de specificare a segmentului' - vom reveni în 3.2.2.6.).

#### Observație.

Dacă este omisă eticheta din adresarea directă folosită cu un index constant (de exemplu omiterea etichetei *table* din exprimarea **table[100h]**), este necesară atunci specificarea unui segment. Deplasamentul operandului este considerat drept punctul de început al segmentului specificat (care trebuie să aibă aceeași valoare cu deplasamentul etichetei *table* în cazul nostru) plus deplasamentul indexat. De exemplu, **ds:[100h]** reprezintă valoarea de la adresa 100h din segmentul referit de DS, exprimare echivalentă cu **ds:100h**.

Dacă se omite specificarea segmentului, este folosită valoarea constantă (imediată) a operandului și nu valoarea pe care o indică. De exemplu, **[100h]** desemnează chiar valoarea 100h, și nu valoarea de la adresa 100h.



11	AND
12	OR, XOR
13	SHORT, .TYPE, SMALL, LARGE
minimă	

În continuare vom descrie unii dintre cei mai utilizați operatori în cadrul instrucțiunilor limbajului de asamblare 8086 și vom da exemple de expresii formate cu acești operatori.

### 3.2.2.1. Operatori aritmetici

Limbajul de asamblare dispune de o gamă variată de operatori aritmetici pentru operațiile matematice uzuale. Ei sunt prezentati în tabelul de mai jos.

Pentru toți operatorii aritmetici, cu excepția operatorului de adunare (+) și a celui de scădere (-), expresiile asupra cărora se efectuează operațiile trebuie să fie constante întregi. Operatorii de adunare și scădere pot fi folosiți pentru efectuarea operațiilor de adunare și scădere între o constantă întreagă și un operand în memorie. Rezultatul poate fi folosit ca operand în memorie. Operatorul de scădere poate fi de asemenea folosit pentru a efectua scăderea între doi operanzi în memorie, dar numai în cazul în care operanzii adresează locații din interiorul același segment. În acest caz rezultatul va fi o constantă, reprezentând numărul de octeți dintre cele două locații desemnate.

OPERATOR	SINTAXA	SEMNIFICAȚIE
+	+ expresie	pozitiv (unar)
-	- expresie	negativ (unar)
*	expresie1 * expresie2	înmulțire
/	expresie1 / expresie2	împărțire întreagă
MOD	expr1 MOD expr2	rest (modulo)
+	expresie1 + expresie2	adunare
-	expresie1 - expresie2	scădere

De exemplu, fie A și B două etichete definite într-un același segment. Presupunem că A are valoarea 100h (deplasamentul ei în cadrul segmentului este 100h), iar B are valoarea 150h. Atunci, expresia A+5 are valoarea 105h, A-7 are valoarea 0F9h. Atât A+5 cât și A-7 pot fi folosiți ca operanzi în memorie. Expressia B-A are valoarea 50h și poate fi folosită ca și o constantă întreagă.

### 3.2.2.2. Operatorul de indexare

Operatorul de indexare ([] ) indică o adunare. El este similar cu operatorul de adunare (+). Sintaxa lui este

[expresie\_1] [expresie\_2]

Ca efect, se adună expresie\_1 cu expresie\_2. Restricțiile privind adunarea operanzilor păstrați în memorie ce se aplică la operatorul de adunare sunt valabile și pentru operatorul de indexare. De exemplu, nu se pot aduna doi operanzi în memorie adresați în mod direct. Expressia *eticheta\_1* [*eticheta\_2*] nu este admisă dacă ambii sunt operanzi în memorie.

Operatorul de indexare are o utilizare largă în specificarea operanzilor din memorie adresați indirect. Paragraful 3.2.1 a clarificat rolul operatorului [] în adresarea indirectă.

### 3.2.2.3. Operatori de deplasare de biți

Operatorii SHR (Shift Right) și SHL (Shift Left) realizează deplasări ale expresiei operand (la dreapta pentru SHR și respectiv la stânga pentru SHL) cu un număr de biți egal cu valoarea celui de-al doilea operand. Biții din dreapta (pentru SHL) și cei din stânga (pentru SHR) sunt completați cu zerouri când conținuturile lor sunt deplasate în afara pozițiilor limitrofe. Sintaxa instrucțiunilor este:

.. expresie SHR cu\_cât și expresie SHL cu\_cât

expresie este deplasată la dreapta sau la stânga cu un număr *cu\_cât* de biți. Biții deplasați dincolo de un capăt sau de celălalt al reprezentării expresiei sunt pierduți. În cazul în care *cu\_cât* este mai mare sau egal cu 16 (32 pe 80386), rezultatul este 0. O valoare negativă pentru *cu\_cât* cauzează deplasarea valorii expresiei în direcție opusă. Exemple (presupunem că expresia se reprezintă pe un octet):

01110111b SHL 3 ; desemnează valoarea 10111000b  
01110111b SHR 3 ; desemnează valoarea 00001110b

Operatorii SHR și SHL nu trebuie confundați cu instrucțiunile omonime ale procesorului (care vor fi tratate în capitolul următor). Operatorii acționează asupra constantelor întregi numai la momentul asamblării. Instrucțiunile procesorului acționează asupra valorilor păstrate în registri sau în memorie la momentul executiei. Asamblorul deosebește din context instrucțiunile SHR și SHL de operatorii SHR și respectiv SHL.

### 3.2.2.4. Operatori logici pe biți

Operatorii pe biți efectuează operații logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante. Tabelul următor conține lista operatorilor logici și semnificația acestora.

**3.2.2.7. Operatori de tip**

DATI: Variabile de tip string care reprezintă numele operatorului și adresa sa.

OPERATORI DE TIP: Structura de date care descrie un operator de tip string.

IMPLEMENTAREA: Implementarea unei clase care extindă structura de date a operatorilor de tip string și adaugă funcții specifice.

EXEMPLU: În următorul fragment de cod este prezentată implementarea unei clase care extindă structura de date a operatorilor de tip string.

```
class Operator : public OperatorString {  
public:  
    Operator(string nume, string adresa) : nume(nume), adresa(adresa) {}  
  
    void setNume(string n) { nume = n; }  
    void setAdresa(string a) { adresa = a; }  
  
    string getNume() const { return nume; }  
    string getAdresa() const { return adresa; }  
  
private:  
    string nume;  
    string adresa;  
};
```

### 3.2.2.7. Operatori de tip

Cârlile: Val ; adresa de sefment este adresa de începătă sefmentul cu numele date, iar offsetul este valoarea efectivă

‘*Je n’aurais pas dû faire ça, mais je n’aurais pas pu me résigner à ne rien faire*’

*tip PIR expertise*

Oprirea unui terminal de lucru sau a unei mașini de calcul este posibilă prin intermediul unei tastări de parolă sau prin introducerea unei chei fizice. Această parolă poate fi compusă din litere și/sau cifre și/sau simboluri speciale. În ceea ce privește cheile fizice, sunt disponibile mai multe tipuri, cum ar fi cheile cardurilor de proximitate sau cheile cu senzori de proximitate.

**Deperiori PTR** este folosit pentru a da posibilitatea multitudinii de utilizatori să-și "vadă" operațiunile ca fiind diferențiat de A. Analog **dword ptr A** indică dublu-cvantul ce începe la adresa A.

OPERATOR THIS

Sperberacion I THIS creerea un operand cu carelui tip este specificat de operator. Sintaxa lui este:

*dy* SIHL

Reținește deci că forma **THIS tip** este echivalentă cu **tip PTR \$**.

**BITLE**. Pentru elicele, tip poate fi NEAR sau FAR. Operatorii THIS se folosesc de obicei cu **WORD**, WORD sau **DWORD** pentru a opera rezultatul pasării în memorie. Deși este similar cu cel de la **MOVE**, diferența principală este că rezultatul este simbol și nu valoare numerică.

unde segmentul potrivit să specifică în mai multe moduri. El își  
defineste proprietatea de segment sau FS sau GS sau GS pe 80386. El poate fi  
acastă caz, numele trebuie să fie definită în prealabil cu o  
prezentată în 3.3.1.), și eventual asociat unui registru de segment  
continuare 3.2.2.7.). Example:

### *segment: expressive*

*Oператорul de specificare a segmentului (:) comanda calculea adresei FAB a unei variabile sau elicele în funcție de un anumit segment. Sintaxa este:*

### 3.2.2.6. Operatorul de specificare a segmentului

4 EQ 3 ; fals (0) - 4 LT 3 ; adevarat (-1)

3.2.2.3. Operatör fehlethal

O101010b AND 1111000b ;delemnizeaza valoarea 0000 1111b  
O101010b OR 1111000b ;are ca rezultat valoarea 0101000b  
O101010b XOR 1111000b ;are ca rezultat valoarea 1111010b  
O101010b NOT 1111000b ;are ca rezultat valoarea 1010010b  
O101010b NOT 1111000b ;are ca rezultat valoarea 1010010b  
tip PTR expresie  
o echivalenta de cod. Similara este

Exemplu (presupunem ca expresia se reprezinta pe un obiect):

OPERATOR	SINTAXA	SEMANTICATIE
NOT	NOT expresie	complementare bit
AND	expr1 AND expr2	SI bit cu bit
OR	expr1 OR expr2	SAU bit cu bit
XOR	expr1 XOR expr2	SAU exclsiv bit cu bit

`lg dw this word - table` este o formă de definire echivalentă cu  
`lg dw word ptr $ - table`

#### Operatorii HIGH și LOW

Operatorii **HIGH** și **LOW** întoarc octetul cel mai semnificativ, respectiv cel mai puțin semnificativ, al unei expresii constante reprezentată pe cuvânt. Sintaxa lor este:

**HIGH expresie** și **LOW expresie**

Operatorul **HIGH** întoarce cei mai semnificativi opt biți din *expresie*; operatorul **LOW** întoarce cei mai puțin semnificativi opt biți. Expresia are ca rezultat o constantă octet.

#### Operatorii SEG și OFFSET

Sintaxele acestor doi operatori sunt:

**SEG expresie** și **OFFSET expresie**

unde *expresie* adresează direct o locație de memorie.

Operatorul **SEG** întoarce adresa de segment a locației de memorie referite. Valoarea întoarsă de operatorul **OFFSET** este o constantă reprezentând numărul de octeți dintre începutul segmentului și locația de memorie referită. Valorile întoarse de acești doi operatori sunt determinate la momentul încărcării programului, ele rămânând neschimbate pe parcursul execuției. Ca exemplu, să considerăm eticheta V, a cărei adresă far este 5AFDh:0003. Atunci SEG (V+5) va avea valoarea 5AFDh iar OFFSET (V+5) va avea valoarea 0008.

Deoarece modul de adresare directă în cazul regiștrilor înseamnă folosirea valorii din regiștri, ca și caz particular operatorii SEG și OFFSET acceptă și regiștri ca operanzi, cu efectele:

SEG registru = 0	Ex: mov bx, SEG ax	;bx:=0
OFFSET registru = registru	mov bx, OFFSET ax	;bx:=ax

Pe de altă parte, deoarece numele unui segment este o etichetă având ca valoare (conform 3.1.) adresa de început a acelui segment, să reținem că avem:

SEG nume_segment = nume_segment	Ex: mov bx, SEG data	;mov bx,data
OFFSET nume_segment = 0	mov bx, OFFSET data	;mov bx,0

### 3.3. DIRECTIVE

Directivele indică modul în care sunt generate codul și datele în momentul asamblării. Majoritatea directivelor au ca operanzi constante și sau numerice și simboluri sau expresii care sunt evaluate la astfel de constante.

Tipul operandului diferă de la o directivă la alta, dar operandul este evaluat întotdeauna la o valoare cunoscută cel târziu în momentul încărcării. Prin acest aspect directivele diferă de instrucțiuni, ai căror operanzi pot fi necunoscuți în momentul asamblării și pot varia în timpul execuției.

#### 3.3.1. Directive standard pentru definirea segmentelor

La un moment dat microprocesorul poate să aibă acces la patru segmente logice: segmentul de cod curent, segmentul de date curent, segmentul de stivă și segmentul de date suplimentar. Aceste segmente logice pot să corespundă la patru segmente fizice distincte, dar pot să existe și suprapuneri.

Există două tipuri de directive segment: directive segment *standard* (**SEGMENT**, **ENDS**, **ASSUME** și **GROUP** - pe care le vom discuta în cele ce urmează) și directive segment *simplificate* (a căror prezentare completă cititorul interesat o poate găsi în [1]).

##### 3.3.1.1. Directiva SEGMENT

Începutul unui segment de program este definit cu directiva **SEGMENT**, iar sfârșitul segmentului este definit cu directiva **ENDS**. Sintaxa unei definiții de segment este următoarea:

*nume SEGMENT [aliniere] [combinare] [utilizare] ['clasa']*

[instrucțiuni]

*nume ENDS*

Numele segmentului este definit de eticheta *nume*. Acestui nume i se asociază ca valoare adresa de segment (16 biți) corespunzătoare poziției segmentului în memorie în faza de execuție (conform 3.1 – vezi și discuția de mai sus referitoare la operatorii SEG și OFFSET – 3.2.2.7). Fiind o etichetă, numele trebuie să fie unic în cadrul modulului sursă. Un nume de segment poate fi utilizat de mai multe ori într-un fișier sursă numai dacă fiecare definiție de segment ce utilizează acel nume va avea fie exact aceleași atribute, fie atribute care concordă.

Argumentele opționale *aliniere*, *combinare*, *utilizare* și *'clasa'* dău editorului de legături și asamblorului indicații referitoare la modul de încărcare și combinare a segmentelor. În absența unor argumente vor fi utilizate valori implicate.

unde ordinea specificătilor de după ASSUME nu este importantă, oricare și oricătre dimite acsestea putând să lipsească. Fiecare dintr-o număr, număr<sub>2</sub>, număr<sub>3</sub> și număr<sub>4</sub> este un nume de segment sau cuvântul rezervat NOTHING.

**ASSUME** CS:*num1*, SS:*num2*, DS:*num3*, ES:*num4*  
generable;

### 3.3.1.2. Directiva ASSUME și gestiunea sepmenelor

Directiva GROUP este utilizata pentru a combina doua sau mai multe segmente într-o singură entitate logică, astfel încât totalele componente să poată fi adreseate relativ la un singur segment.

Argumentul „clasă”, are rolul de a permite stabilitatea ordinii în care editorul de legătură poate să stabilească ordinea segmentelor de memorie. Totale segmentele având aceeași clasă vor fi plasate între-un bloc contiguu de memorie în memorie. De asemenea, ordinile de rezervat pentru segmentul stivă.

Argumēntul utilizare se folosește numai în cazul microprocesorelor 80386 și celelor ulterioare pentru specificarea marimi cuantitative.

Cap.3. Elementele limbajului de asamblare.

**Cap. 3.** Elementos de la imbaluña de asamblea.

2. În mod normal, într-un program se va termina minimum un segment de străvă (cu ajutorul tipului de combinare STACK). Dacă nu este declarat nici un segment de străvă, altunci editorul de legături va considera spațiul dintre străve în continuarea uneia dintr-o segmente existente.

1. Pentru programele care se intregioneaza a devinei programe .COM (vezi §.5.3.) nu trebuie să deservim.

Dacă nu se da nici un tip de combinare, segmentele cu același nume nu vor fi combinate, fiecare primind o zonă de memorie separată.

- MEMORY - segmentele cu acelasi atribut vor fi asezate in memorie in spatii disponibile

- STACK - Segmentele cu același nume vor fi concatenate. În fază de execuție se combină rezultatul săfii segmentul și.

- AT <expresie> - impune ca segelementul să fie înărcitat în memorie la adresa reprezentată de valoarea expresiei.

Începutul tuturor segmentelor ce au același nume. Se obține un segment având dimensiunea egală cu cea a celui mai mare segment având același nume.

segmentelor componente.

positive sum:

Argumemtul optional combinare controlereză modul în care se emite cu același nume din cardul

Dacă ar trebui să alegem o altă opțiune, ar trebui să considerăm imediata deschidere a unor noi surse de finanțare.

- WORD - multiplo de 2 (áreas de cubo)
- DWORD - multiplo de 4 (áreas de dupla palavra)
- PARA - multiplo de 16 (áreas de paralelepípedo)

se găsesc într-un răspicetiv. Aliniările posibile sunt următoarele:

Valoarea unei etichete este deplasamentul ei. Pentru completarea implicită de către asamblor a adresei sale de segment este nevoie de informația de asociere furnizată prin ASSUME. În cadrul parcurgerii textului sursă, asamblorul va întâlni o anume etichetă, o va identifica drept deplasament în cadrul acelui segment, însă pentru referirea la o adresă completă (segment:deplasament) va fi necesar ca în acel moment să prefixeze acel offset cu adresa de segment corespunzătoare. Care este însă aceasta? Tocmai aici intervine ASSUME: *asamblorul va căuta în textul sursă locul în care apare definitia etichetei, va identifica numele segmentului în care apare această definitie de etichetă după care va recurge imediat la ASSUME pentru a vedea cu ce fel de registrul de segment a fost asociat numele de segment identificat și astfel va prefixa eticheta cu registrul de segment corespunzător detectat*. Oricărei referiri (accesări) a unei etichete i se aplică mecanismul descris. Recomandăm a se urmări aplicarea acestui mecanism în exemplele care urmează pentru a înțelege pe deplin necesitatea furnizării de asociere corecte prin directiva ASSUME.

Dacă la nivelul unui program există însă numai referiri explicite de tip FAR (însă astfel de programe sunt extrem de rare!) asamblorul va asocia fiecărei etichete din program numele segmentului în care aceasta apare și astfel directiva ASSUME nu va mai fi aplicată și ca urmare nu mai este necesară. Însă acest lucru se întâmplă numai pentru acele adrese utilizate explicit ca adrese FAR (de exemplu: *FAR PTR a, ds:b sau data:y*). Celelalte adrese (adică cele NEAR) trebuie să facă obiectul unor asocieri implicate cu adrese de segment (și de aceea este nevoie de ASSUME) iar limbajul de asamblare specifică faptul că obiectul unor astfel de asocieri implicate pot fi numai registri de segment. De aceea de exemplu nu sunt posibile specificări prin ASSUME de tipul:

ASSUME data:d1                        sau                            ASSUME 07Abh:d2

adică înaintea unor precizări de nume de segment ca d1 sau d2 pot apărea numai registri de segment, nu și nume de alte segmente sau constante (chiar dacă acestea din urmă reprezintă modalități valide de specificare ale unor adrese FAR)!

Valoarea din CS (adresa de segment a segmentului de cod curent) este gestionată automat în timpul execuției, programatorul având acces doar în citire asupra acestei valori. Pentru accesarea etichetelor din cadrul unui segment de cod este necesară specificarea numelui segmentului printre directivă ASSUME. Considerăm următorul exemplu, în care folosim instrucțiunea JMP (salt necondiționat la eticheta specificată), instrucțiune pe care o vom prezenta în capitolul 4:

ASSUME CS:c	;asociază registrul CS cu segmentul de cod c
c segment	
start: jmp far ptr etd	;salt far necondiționat la eticheta etd din cadrul segmentului de cod ;d ;(aici nu e nevoie de ASSUME deoarece etd este forțată a fi ;considerată FAR)
etc: jmp x	;salt near necondiționat la eticheta locală x (aici este nevoie de ;ASSUME, pentru a se putea compune corect adresa fizică CS:x)
...	

x:  
c ends

ASSUME CS:d

;realizează o nouă asociere a registrului segment CS, de această dată cu segmentul d. Vechea ;asociere este anulată, cea curentă rămânând valabilă până la o nouă directivă ASSUME sau până la ;sfârșitul textului sursă.

d segment	
etd: jmp y	;salt near necondiționat la eticheta locală y (aici este nevoie de ;ASSUME pentru a se putea compune corect adresa fizică CS:y)
...	
y: jmp far ptr etc	;salt far necondiționat la eticheta etc din cadrul segmentului de ;cod c

d ends  
end start

**Observație.** Dacă segmentul de cod nu conține referiri la etichete locale, prezența unei directive ASSUME nu mai este obligatorie. Astfel, exemplul de mai jos conține două segmente de cod, fiecare conținând doar o instrucțiune: salt far necondiționat în celălalt segment. Se observă că spre deosebire de exemplul anterior aici nu apare nici o directivă ASSUME:

c segment	
start: jmp far ptr etd	;salt far necondiționat la eticheta etd din cadrul segmentului ;de cod d etc: ... ;nu se impune aici prezența unei directive ASSUME deoarece ;eticheta locală etc este referată doar ca etichetă FAR din segmentul ;d

c ends

d segment	
etd: jmp far ptr etc	;salt far necondiționat la eticheta etc din cadrul segmentului de ;cod c ...

d ends  
end start

După cum am mai afirmat, pentru un program scris în limbaj de asamblare este normal ca programatorul să furnizeze minimum un segment de stivă. Sunt foarte rare cazurile când elementele din stivă se accesează prin etichete. De obicei, operațiile la nivel de stivă se rezumă la introducerea sau scoaterea de cuvinte în și respectiv din vârful stivei.

91

### **Cap. 3. Elementele limbajului de asamblare.**

Dreptiva **ASSTM** pentru SS este neceasă numai în cazul în care se definesc un segment de stivă și numai dacă există accesări de elemente ale stivei prin intermediul unor elice hărțile definite în acest segment. De exemplu:

```

ASSUME CS:c, DS:d1, ES:d2
        db      ...      ,define d2
        db      ends
d2      segment

```

```
start: mov ax,d1 ;incarcarea registru si segmentului  
        mov ds,ax ;principala date  
        mov ax,d2 ;incarcarea registru si segmentului
```

“Can't address with currently ASSUMEd registrars”;  
jedac nu se specifică direcția ASSUME, la înțalnirea celor două referiri de etichete (deplasamente)  
jedac nu se specifică direcția ASSUME, la furnizarea mesajului de eroare sintactică.  
; ends  
end start

Pă de altă parte, chiar dacă programatorul înțelege nevoie să utilizeze într-o serie de puncte de definitorie a acceselor a eticheteelor definite în program și evita astfel de eroare de tip-o-simbolă sau de eroare logice. Această utilizare directivelor ASSUME se poate constitui și într-o potențială sură de eroare logică. Acum să se întâmpină situații ca îndepărtarea unei directivelor ASSUME să rezulte într-o eroare de tip-o-simbolă sau de eroare logică. Întrucât într-o directivă ASSUME se pot defini variabile și constante care să nu fie folosite în program, înlocuirea unei directivelor ASSUME cu o altă directivă sau adăugarea unei noi directivelor ASSUME va provoca o eroare de tip-o-simbolă.

90 Arhitectura de calculatoarelor. Limbaul de assembly Box86.

• în ceea ce privește micăcarea regeștilor de segment, precizăm următoarele:
- regeștiții CS este micăcăt automata
- regeștiții SS este deasemenea micăcăt automat. Dacă programatorul dorește să schimbe segmentul de stivă, trebuie el să încarce noua valoare în SS (dacă mai jos un exemplu)
- regeștiții DS și ES, în cadrul programelor .EXE (vezi 5.5.2.) trebuie încărcat de către programator.
Anticipând instația M0V (care va fi prezentată în 4.1.1) dacă mai jos un exemplu de accesare a datelor din două segmente de date. Să facem precizarea că regiștri de segment nu pot fi micăcăți direct, ci numai print-un intermediar (regeștițu sau stivă). În cauză nostru, intermediar este regeștițul a datelor din două segmente de date.
... și definiție a 1
db al segment ends d1

In ceea ce privează accesarea datelor, un program poate avea la un moment dat 0, 1 sau 2 segmente active de date. Dacă o dată este accesată prin eticheta ei, atunci segmentul în care este definita trebuie să facă obiectul unei asocieri prin direcția ASSUME cu registrul segment DS sau BS. Dacă DS-va fi segment principal din două segmente, until dintr-o asociare regisitru DS se dorește accesarea simultană a celorlalte două segmente, unde ele va fi asociat regisitru DS (vă îi se va numi și segment principal este NOTHING). În același segment de date, dacă numărul de date este mult mai mare decât numărul de segmente, atunci segmentul respectiv este anulat. Se înțelege că DS și FS să fie asociati la același segment de date.

```

ASSUME CS:c, DS:d2, ES:d1 ;aici am inversat asocierile inițiale pentru DS și ES!
c segment
start: mov ax,d1 ;încărcarea registrului DS cu adresa segmentului de date d1
       mov ds,ax ;deși ASSUME a fost modificată, acțiunile de încărcare ale
                  ;regiștrilor DS și ES nu sunt modificate corespunzător cu
                  ;ASSUME !
       mov ax,d2 ;încărcarea registrului ES cu adresa segmentului de date d2
       mov es,ax
       mov al,a1 ;accesare a1 relativ la ES (adică mov al, ES:a1, deoarece segmentul
                  ;d1 în cadrul căruia apare eticheta a1 a fost asociat prin ASSUME
                  ;cu registrul ES)
;numai că ES a fost încărcat mai sus cu adresa de început a segmentului d2 și nu cu adresa
;de început a segmentului d1 unde se află a1, deci în AL se va încărca conținutul locației de
;memorie de deplasament a1 din cadrul segmentului d2 ceea ce va reprezenta o eroare
;logică, deoarece a1 trebuie raportat la segmentul d1 și nu la segmentul d2!!
       mov ah,a2 ;accesare a2 relativ la DS (adică mov ah, DS:a2, deoarece segmentul
                  ;d2 în cadrul căruia apare eticheta a2 a fost asociat prin ASSUME
                  ;cu registrul DS)
;numai că DS a fost încărcat mai sus cu adresa de început a segmentului d1 și nu cu adresa
;de început a segmentului d2 unde se află a2, deci în AH se va încărca conținutul locației de
;memorie de deplasament a2 din cadrul segmentului d1 ceea ce va reprezenta o eroare
;logică, deoarece a2 trebuie raportat la segmentul d2 și nu la segmentul d1!!
c ends
end start

```

Cele două erori logice de mai sus se pot evita dacă după modificarea suferită de directiva ASSUME se actualizează corespunzător și acțiunile de încărcare cu adresele corespunzătoare de început de segment ale regiștrilor DS și ES, adică:

```

start: mov ax,d1 ;încărcarea registrului ES cu adresa segmentului de date d1
       mov es,ax ;deci adaptarea la asocierea furnizată prin ASSUME !
       mov ax,d2 ;încărcarea registrului DS cu adresa segmentului de date d2
       mov ds,ax ;deci și aici adaptarea la asocierea furnizată prin ASSUME !

```

În final dăm un exemplu în care se lucrează alternativ cu două stive. Stiva inițială este segmentul VS, stiva alternativă este segmentul NS, iar segmentul de date este D:

```

d segment
       vsp . . . ;definire vsp
       vss . . . ;definire vss
d ends
ns segment
       . . . ;rezervare spațiu stivă
ns ends
vs segment stack 'STACK'
       . . . ;rezervare spațiu stivă
vs ends
c segment
       assume ds:d, ss:vs
start: mov ax,d
       mov ds,ax ;este activă stiva veche
       mov vsp,sp
       mov ax,ss
       mov vss,ax
       mov ax,ns
       mov ss,ax
       mov sp,200h-2 ;este activă stiva nouă
       mov ax,vss
       mov ss,ax
       mov sp,vsp ;este activă stiva veche
c ends
end start

```

Directive ASSUME pot fi inserate în textul sursă oricând este considerat necesar de către programator. Pentru a indica faptul că unul sau mai mulți regiștri segment nu pointează spre nici un segment se folosește cuvântul rezervat **NOTHING**.

Orice program scris în limbaj de asamblare trebuie să conțină directiva END pentru marcarea sfârșitului codului sursă al programului. Eventualele linii de program ce urmează directivei END sunt ignorate de către asamblor. Sintaxa ei este

**END [adresa\_start]**

unde *adresa\_start* este o expresie sau un simbol optional indicând adresa din program de unde va începe execuția. Într-un program care constă dintr-un singur modul (fișier sursă) specificarea adresei de start în cadrul directivei END este obligatorie. Și pentru un program constituit din mai

- DD - date de tip dublu cuvânt (pointer - DWORD)
- DW - date de tip cuvânt (WORD)
- DB - date de tip octet (BYTE)
- DT - date de tip 10 octet (TWORD) - utilizează pentru memorarea constantelor reale
- DDQ - date de tip 8 octet (QWORD) - utilizează pentru memoria constantelor reale
- DTQ - date de tip 10 octet (TWORD) - utiliză pentru memorarea constantelor reale

(de la microprocesorul 80386 începând există în plus două direcțive - **DR** și **DR** - ce determină pozitiei far pe 6 octetii).

data segment	varb DB	varw DW	vard DD	vard DQ	varq DT	data ends
db	101b	32 octetii	2bfrh	4 octetii	307o	8 octetii (1 quadword)
db	100	32 octetii	2bfrh	4 octetii	100	10 octetii
db	307o	8 octetii (1 quadword)	307o	8 octetii	307o	8 octetii
db	101b	32 octetii	101b	32 octetii	101b	32 octetii

creză în tablou de 5 metri și prezentați pe curvăte având valoriile respectiv 1,2,3,4,5. Dacă valoarea de după direcțiva nu încap pe o singură linie se pot adăuga oricate limite necesare, liniile ce vor conține numai direcțiva și valoările lor. Exemplu:

**Operatormul DUP** se folosește pentru definierea unor blocuri de memorie înțializate repetitiv cu o anumita valoare. De exemplu

rezerva 256 de cuvinte pentru tabloul Tabzero initializat cu 0, iar	Tabchar	DB	80 DUP (a)
rezervă 256 de cuvinte pentru tabloul Tabzero initializată cu 0, iar	Tabzero	DW	100h DUP (0)
anumita valoare. De exemplu			
Operatorul <b>DUP</b> se folosește pentru definirea unor blocuri de memorie			
Tabpatriate DD 0, 1, 4, 9, 16, 25, 36	DD	49, 64, 81	100, 121, 144, 169

creaza un tablou de 80 de octeți inițializați fiecare cu codul ASCII al caracterului „a”.

### **3.3.2. Directive pentru determinarea datelor**

Tin data este o direcție de definire a datelor, una din următoarele:

Dacă se dorește doar rezervarea de spațiu de memorie pentru tablouri, fără inițializarea acestora cu anumite valori se va folosi caracterul '?'. Într-o astfel de situație declarațiile de mai sus ar deveni

Tabzero	DW	100h DUP (?)
Tabchar	DB	80 DUP (?)

Dacă dorim definirea de siruri de caractere este suficient să ținem cont că ele sunt de fapt ca reprezentare internă siruri de octeți, asamblorul considerând echivalente declarațiile ca:

sirchar	DB	'a','b','c','d'
sirchar	DB	'abcd'

Valoarea de inițializare poate fi și o expresie, ca de exemplu: vartest DW (1002/4+1)

Valoarea curentă a contorului de locații poate să fie referată cu ajutorul simbolului \$ sau al expresiei *this <tip>*. Ca urmare, putem avea următoarele secvențe echivalente

tabcuv	DW	50 DUP (?)	tabcuv	DW	50 DUP (?)
lungtab	DW	\$-tabcuv	lungtab	DW	this word-tabcuv
tabcuv	DW	50 DUP (?)	tabcuv	DW	50 DUP (?)
lungtab	DW	lungtab-tabcuv	lungtab	EQU	this word-tabcuv

cu deosebirea că ultima variantă (cea cu EQU – vezi și 3.3.3) față de primele 3 nu va și genera spațiu de memorie pentru lungtab (lungtab va avea un regim similar celui de constantă simbolică din limbajele de nivel înalt).

Dăm în continuare un exemplu care conține declarații și inițializări de date și care este edificator în ceea ce privește posibilitățile de combinare a facilităților de declarare descrise până aici:

```

data segment
    a1   DB  0,1,2,'xyz'
          DB  2 SHL 4, "F"+3
    a2   DB  3 DUP (44h)
    a3   DB  10 DUP (5 DUP (3), 11)
    a4   DW  a2+1, 'bc'
          DW  0ffffh, 0800h SHR 2
    a5   DW  4 DUP ('13')
    a6   DW  a4
    a7   DD  a4
    .
    .
data ends

```

Declarația pentru a1 inițializează 6 octeți cu valorile 0, 1, 2 și codurile ASCII corespunzătoare caracterelor x, y și z. Urmează doi octeți inițializați cu valori rezultate în urma evaluării de către asamblor a expresiilor constante respective. Pentru acești octeți nu a fost specificat un nume, ei putând fi însă referiți relativ la a1. Valorile lor sunt: primul octet va conține numărul 32 ( $2^2 \wedge 4$ ), iar al doilea codul ASCII al caracterului T (al treilea de după 'F'). Declarația pentru a2 inițializează 3 octeți, fiecare dintre ei cu valoarea 44h. Declarația pentru a3 rezervă 60 de octeți (de 10 ori câte 6) inițializați în ordine cu valorile 3,3,3,3,3,11,3,3,3,3,3,11,... Pentru a4 se vor rezerva 4 octeți: 2 pentru valoarea rezultată prin incrementarea adresei NEAR a etichetei a2 și încă 2 octeți pentru caracterele b și c. Urmează 4 octeți (din nou fără nume însă putând fi referiți relativ la a4) inițializați cu valorile rezultate în urma evaluării expresiilor constante corespunzătoare. Pentru a5 se vor rezerva 8 octeți (4 cuvinte) inițializați în ordine cu '1', '3', '1', '3', '1', '3', '1', '3'. Pentru a6 se vor rezerva 2 octeți (datorită directivei DW) conținând adresa NEAR (deplasament) a variabilei a4, iar pentru locația desemnată de eticheta de date a7 se vor rezerva 4 octeți (datorită specificării directivei DD) conținând adresa FAR (segment:deplasament) a variabilei a4.

### 3.3.3. Directivele LABEL, EQU, =

Directiva **LABEL** permite numirea unei locații fără alocarea de spațiu de memorie sau generare de octeți, precum și accesul la o dată utilizând alt tip decât cel cu care a fost definită data respectivă. Sintaxa este

*nume LABEL tip*

unde *nume* este un simbol ce nu a fost definit anterior în textul sursă, iar *tip* descrie dimensiunea de interpretare a simbolului și dacă acesta se va referi la cod sau la date.

Numele primește ca valoare contorul de locații. *Tip* poate să fie una din următoarele:

BYTE	NEAR	FAR
WORD	QWORD	PROC
DWORD	TBYTE	UNKNOWN

Tipurile **BYTE**, **WORD**, **DWORD**, **QWORD** și **TBYTE** etichetează respectiv date de 1, 2, 4, 8 și 10 octeți. Iată un exemplu de inițializare a unei variabile de memorie ca pereche de octeți însă accesată ca și cuvânt:

data segment	code segment
.	.
.	.
varcuv LABEL WORD	mov ax, varcuv
DB 1,2	.
.	.
.	.
data ends	

echete de

definitie prin EQU:

Expreseaza pentru echivalarea unei echete definite prin directiva EQU poate contina la randul ei

programare de nivel initial.

Se observa astfel ca echetelor echivalante prin directiva EQU cu constantele din limbajele de

Print utilizarea de astfel de echivalan textul sursa poate deveni mai lizibil.

END_OF_DATA	EQU	!.	VAR_CICLAR
BUFSIZE	EQU	1000h	INDEX_START
SIZE	EQU	(1000/4 + 2)	EQU

Example:

EQU este similar definitiei constante simbolice din casutl unui limbaj de nivel initial.  
se admitt redifinitii de echete decat in cazul echivalarii lor intitul cu sirul de caractere. Rolul  
interior, el trebuie sa fi avut ca rezultat al evaluarii expresiei in si de caractere. Cu alte cuvinte nu  
symbol EQU se utilizeaza anterior tot in-t-o directiva EQU. Daca simbolul a fost utilizat  
unde numele li este atribuita valoarea numerică a expresiei. Simbolul nume trebuie sa fie un

### nume EQU expresie

unei echete fara alocarea de spatiu de memorie sau generare de octet. Simtixa directiei EQU este  
Directiva EQU permite atribuirea, in faza de assemble, unei valori numerice sau sir de caractere

este utilizataa operatorului de conversie PTR, prezentat in 3.2.2.7.  
Reamintim ca o alta soluție pentru adresaarea unei date cu un alt tip decat cel cu care a fost declarata

data ends .  
tempvar . ;utilizare ca si octet  
add dl,tempvar . ;utilizare ca si cuvant  
mov tempvar,ax . ;utilizare ca si cuvant  
data segment .  
code segment .  
tempvar LABEL UNKNOWN .  
DB ?? .

altele ca si cuvant pota fi realizata prin declararea ei ca tip UNKNOWN:  
Tipul UNKNOWN declară un tip necunoscut și este folosit atunci cand se dorește să existe  
obiecte care să acceseze variabile temporare din secevența de mai jos înaintea ca octet și  
din limbajul C). De exemplu, accesarea variabilei temporare din secevența de mai jos înaintea ca octet și

Tipul PROC (existente folosit in cazuil utilizarii directivelor segment simplificatae.  
modelul de memorie folosit in varianta TASM) va furniza un tip NEAR sau FAR în funcție de

```
REPT contor
    secvență
ENDM
```

cu semnificația că *secvența* va fi asamblată de *contor* ori. De exemplu secvențele

```
REPT 5           dw 0
    dw 0         și   dw 0
ENDM           dw 0
                dw 0
```

generează același cod, lucru ce se poate realiza bineînțeles și cu: dw 5 DUP (0)

Exemplul următor însă nu are un echivalent atât de simplu. El generează 5 locații de memorie consecutive conținând valorile de la 0 la 4. Folosim în acest scop și directiva = :

```
Intval = 0       care va genera dw 0
REPT 10
    dw Intval
    Intval = Intval + 1
ENDM           dw 1
                dw 2
                dw 3
                dw 4
```

De asemenea, blocurile repetitive pot fi imbricate. Secvența

```
REPT 5
REPT 2
    secvență
ENDM
ENDM
```

generează de 10 ori secvența specificată.

Directiva IRP are sintaxa

```
IRP parametru, <arg1 [,arg2]...>
    secvență
ENDM
```

Se efectuează repetat asamblarea secvenței, câte o dată pentru fiecare argument prevăzut în lista de argumente, prin înlocuirea textuală în secvență a fiecărei apariții a parametrului cu argumentul curent. Argumentele pot fi siruri de caractere, simboluri, valori numerice. De exemplu,

```
IRP param,<0,1,4,9,16,25>
    db param
ENDM
generează secvența
db 0
db 1
db 4
db 9
db 16
db 25
```

```
iar   IRP reg,<ax,bx,cx,dx>
      mov reg,di
ENDM
generează secvența
mov ax,di
mov bx,di
mov cx,di
mov dx,di
```

Directivea IRPC are un efect similar, ea realizând însă înlocuirea textuală a parametrului, pe rând, cu fiecare caracter dintr-un sir de caractere dat. Sintaxa ei este

```
IRPC parametru,string
    secvență
ENDM
```

De exemplu      IRPC nr,1375
 db nr
ENDM

crează 4 octeți având respectiv valorile 1, 3, 7 și 5.

### 3.3.6. Directiva INCLUDE

Directiva INCLUDE are sintaxa

```
INCLUDE numefisier
```

Efectul ei (similar de exemplu cu cel al directivei #include a preprocesorului C) este de a insera textual fișierul *numefisier* în textul sursă curent. Inserarea se face în locul în care apare directiva INCLUDE respectivă. *numefisier* este specificarea unui nume de fișier DOS, putând aşadar să conțină specificări de unitate de disc, directoare, nume de fișier și tip. În lipsa specificării unui tip se consideră implicit .ASM. De exemplu, dacă fișierul *prog.asm* conține codul

```
cod segment
    mov ax,1
INCLUDE INSTR2.ASM
    push ax
```

Pentru înmulțirea cu 4 a valorii unei variabile rezultatul depunându-se în DX:AX) se poate scrie

Pentru înmulțirea cu 4 a valorii unei variabile (rezultatul depunându-se în DX:AX)

O utilizare a sa sub forma **imcua4 varm** va genera secvența

```
mov ax, varm
sub dx, dx
shl ax, 1
rcr dx, 1
shl ax, 1
rcr dx, 1
shl ax, 1
rcr dx, 1
imcua4 MACRO a
```

Macro-ul pot conține blocuri repetitive. În acest sens putem lua ca exemplu chiar macroul

înume4 de mai sus, care se poate re scrie astfel:

```
imcua4 MACRO a
    mov ax, a
    sub dx, dx
    rep 2
        shl ax, 1
        rcr dx, 1
    endm
ENDM
```

O posibilă problema ce apare se referă la definitia unei etichete într-un macro. Să presupunem că

```
sacde MACRO
    jcz Etich
    dec cx
    Etich:
    ENDM
```

Eticheta definită va apărea la fiecare expansiune a macroului în cadrul programului, cauzând eroare de "redifinire de etichete". Exemplu:

abia în următorul capitol. În ciuda faptului că instrucțiunile sunt în cadrul unei etichete, care vor fi prezentate

precisă. Doar dacă promovarea unei baze de discuție asupra structurii programelor pentru a putea

scăpare în evidență semnificația elementelor prezentate aici.

O posibilă problemă ce apare se referă la definitia unei etichete într-un macro. Să presupunem că

De exemplu, pentru interacțiunile a două variabile cu valori definite în următorul

macro:

Un macro este delimitat de directivele **MACRO** și **ENDM** conform următoarei sintaxe:

**nume MACRO [parametru1, parametru2...]**

Un **macro** este un text parametrizat care îl se atribuie un nume. La fiecare initializare a numărului

asamblorul pune în cadrul sursei textului cu parametrii actualizați. Operația este cunoscută și sub

numele de **expandarea macrooului**. Se poate face o analogie cu direcția INCLUDE prezintă

anterior. Fie că de fizică inclusă, macrourile prezintă un grad sporit de flexibilitate permisă în

transmisie de parametri și existența de etichete locale.

Fiecare închidere de paranteză înlocuită cu paranteză deschisă în finală să înghere codul.

Înseguția unei instrucțiuni inserată în cadrul unei etichete poate crea o nouă etichetă.

rezultatul asamblării fizierului progr.asm va fi echivalent cu cel al asamblării codduli

iar fizierul instr2.asm conține codul

rezultatul asamblării fizierului progr.asm va fi echivalent cu cel al asamblării codduli

102 Arhitectura calculatorelor. Limbajul de assembly 80x86.

```

scade      ;apare eticheta Etich
.
scade      ;și aici apare Etich!
.

```

Soluția unei astfel de probleme este oferită de către *directive LOCAL*, care, la apariția ei în cadrul unui macro forțează ca domeniul de vizibilitate al etichetelor specificate ca argumente să fie numai acel macro. Soluția pentru exemplul de mai sus este:

```

scade MACRO
LOCAL Etich
jcxz Etich
dec cx
Etich:
ENDM

```

cele două apeluri consecutive de mai sus fiind translatate în

```

jcxz ??0000
dec cx
??0000:
.
jcxz ??0001
dec cx
??0001:
.

```

Utilizarea directivei LOCAL trebuie să urmeze imediat directivei MACRO. Numărul argumentelor nu este limitat.

Să mai precizăm de asemenea că nu sunt permise pentru macrouri referințele anticipate (*forward references*), macrourile trebuind să fie definite întotdeauna înaintea invocării. De asemenea, macrourile pot fi imbricate.

O altă problemă poate să apară atunci când parametrii formali sunt amestecați cu alt text. De exemplu, să presupunem că dorim să scriem un macro care să realizeze depunerea în stivă a conținutului unuia din cei patru registri generali (AX, BX, CX sau DX) în funcție de valoarea parametrului *rlitera* care va fi a, b, c sau d respectiv ('x' fiind partea comună tuturor variantelor). Dacă scriem:

```

push_reg MACRO rlitera
push rliterax
ENDM

```

asamblorul nu va putea determina faptul că o parte din șirul operand al lui push este de fapt parametrul formal al macroului și va considera că este vorba de operandul *rliterax*.

Soluția este oferită de asamblor, care permite încadrarea în cadrul corpului macroului a numelui parametrului formal într-o pereche de caractere & (ampsand, numit operatorul de substituție). La întâlnirea textului cuprins între cele două &, asamblorul substituie acel text cu valoarea solicitată la apel. De exemplu

```

push_reg MACRO rlitera
push &rlitera&x
ENDM
.
push_reg b

```

se va asambla în **push bx**. Să facem precizarea că operatorul de substituție & poate fi utilizat și în cadrul directivelor IRP sau IRPC. De exemplu:

IRP rlitera,<a,b,c,d>	push ax
push &rlitera&x	push bx
ENDM	push cx
	generează
	push dx

După cum am văzut, macrourile pot conține blocuri repetitive. De asemenea macrourile pot invoca la rândul lor alte macrouri. În exemplul

```

push_reg MACRO      registru
                    push registru
ENDM
.
push_toate_reg MACRO
IRP reg,<ax,bx,cx,dx,si,di,bp,sp>
push_reg reg
ENDM
ENDM

```

macroul **push\_toate\_reg** conține un bloc repetitiv, care la rândul lui conține o invocare a macroului **push\_reg**.

Pentru testarea unor programe foarte simple, există posibilitatea de a elabora programe care conțin numai căte un segment de date și unul de cod, având nume predefinite (nepărat date și cod). În acest scop se pot folosi directivele simplificate (caracteristica TASM). Un astfel de program are structura:

```

;definirea punctului de intrare în program - eticheta de cod "start"
;precizarea stării initiali trebuie sărșă (direcțiva END) împreună cu
code ends
        end start
        mov ah,4ch
        int 21h
        ;apelarea funcției 4ch a interrupterii 21h pentru a provoca închiderea
        ;execuției programului curent
        ;apelarea funcției 4ch a interrupterii 21h pentru a provoca închiderea
        ;accesă registruului segment DS cu adresa segmentului de date
        mov ds,ax
        start: mov ax,data
        ;cod este aci! numele segmentului de cod – numele poate fi modificat
        ;cod este aci! numele segmentului de cod – numele poate fi modificat
        data ends
        ...
        ;data este aci! numele segmentului de date – numele poate fi modificat
        ;definiri de date utilizând directivele de definire a datelor (3,2.) – DB, DW, DD,
        ;data segment
        ;data este aci! numele segmentului de date – numele poate fi modificat
        assume cs:code, ds:data ;rolul directivelor ASSUME este detaliat în 3.3.1.2.
        ;incapăt forma generală (sau cel puțin cea mai des utilizată) a unui program ASM:
Pentru ca ceeață să poată expriemă instrucțiunile prezentate în acest capitol, prezentăm pentru
```

#### CAPITOLUL 4

### INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE

Cap.4. Instrucțiuni ale limbajului de asamblare.

```

        ;conținutul segmentului de date
        .data
        .model small
        code
        Start:
        mov ax,@Data
        mov ds,ax
        mov ah,4ch
        int 21h
        ;apelarea funcției 4ch a interrupterii 21h
        end Start

```

Indiferent de forma adoptată pentru experimentare, un program sursă este editat într-un fișier cu extensia .asm:

*nume.ASM*

cu *nume* fixat de către utilizator. În acest caz următoarea secvență de trei comenzi DOS:

```
...>TASM nume
...>TLINK nume
...>TD nume
```

realizează asamblarea, editarea de legături și execuția asistată a programului respectiv.

#### 4.1. MANIPULAREA DATELOR

Pentru o mai ușoară referire ulterioară de către cititor a instrucțiunilor și operațiilor prezentate în acest capitol vom prezenta la începutul fiecărei secțiuni câte un tabel cu trei rubrici ele reprezentând în ordine *forma generală (sintaxă)*, *efectul și respectiv flagurile afectate*. Notația *<op>* desemnează conținutul operandului *op*. Simbolurile *d* și *s* vor desemna operandul destinație și respectiv operandul sursă al instrucțiunii respective.

##### 4.1.1. Instrucțiuni de transfer al informației

###### 4.1.1.1. Instrucțiuni de transfer de uz general

<b>MOV</b> <i>d,s</i>	<i>&lt;d&gt; &lt;-&gt; &lt;s&gt;</i>	-
<b>PUSH</b> <i>s</i>	depune <i>&lt;s&gt;</i> în stivă	-
<b>POP</b> <i>d</i>	extrage elementul curent din stivă și îl depune în <i>d</i>	-
<b>XCHG</b> <i>d,s</i>	<i>&lt;d&gt; &lt;-&gt; &lt;s&gt;</i>	-
<b>XLAT</b> [tabelă de translatare]	AL <-> seg:[BX+<AL>]	-

Cea mai importantă instrucțiune de transfer de informație este **MOV**. Forma generală este

**MOV** *destinație, sursă*

unde *sursă* poate fi o constantă, un registru general sau o locație de memorie. *destinație* este un registru general, o locație de memorie sau un registru segment.

Efectul instrucțiunii este copierea valorii operandului sursă în operandul destinație cu păstrarea valorii din operandul sursă. De exemplu, secvența

```
mov ax,0
mov bx,7
mov ax,bx
```

depune întâi în registrul AX constanta 0, apoi reține constanta 7 în BX, după care copiază conținutul registrului BX în AX. În final, registrii AX și BX vor conține amândoi aceeași valoare 7.

Dacă destinația este unul dintre cei patru registri segment atunci sursa trebuie să fie unul dintre cei opt registri sau o variabilă de memorie. Cum numele segmentelor sunt valori constante (adresele de început ale acelor segmente), ele trebuie încărcate în regisitrii segment corespunzători prin intermediul unui registru general sau unei locații de memorie. De exemplu, cum am mai arătat și în 3.3.1.2., secvența de cod

```
data segment
.
.
code segment
.
.
    mov ax, data
    mov es, ax
```

încarcă registrul segment ES cu adresa de început a segmentului de date. Ceea ce se dorește de fapt dar nu este permis în mod direct este: *mov es, data*.

Instrucțiunea **MOV** poate fi folosită și pentru accesarea informației din stivă, prin intermediul modului de adresare ce utilizează BP ca registru de bază. De exemplu, instrucțiunea

```
mov ax, [bp+4]
```

încarcă AX cu conținutul cuvântului aflat la deplasamentul BP+4 în cadrul stivei. Pentru accesarea conținutului stivei se folosesc de regulă alte două instrucțiuni și anume **PUSH** și **POP**.

Instrucțiunile **PUSH** și **POP** au sintaxa

**PUSH** *s*                                   **POP** *d*

Operanții trebuie să fie reprezentați pe cuvânt, deoarece stiva este organizată pe cuvinte. Stiva crește de la adrese mari spre adrese mici, din doi în doi octeți, SP punctând întotdeauna spre cuvântul din vârful stivei. Instrucțiunea **PUSH** depune operandul (sursă) *s* în vârful stivei (implicit operandul destinație), decrementând întâi valoarea din registru SP, iar instrucțiunea **POP** extrage valoarea din vârful stivei (implicit operandul sursă aici) și o depune în operandul (destinație) *d*, incrementând ulterior valoarea din registru SP.

Varianta cu MOV se execută mai rapid, în timp ce aceasta din urmă are un cod generat mai scurt.

```
Push data
      pop es
```

**XCHG** operand1, operand2

Instrucțiunea **XCHG** permite interschimbarea conținutului a doi operaanzi de același dimensiune (ocet sau cuvânt), cel putin unul dintre ei trebuie să fie registru. Simaxa ei este

Această instrucțiune oferă o modalitate directă de a efectua o acțiune pentru care altfel s-ar impune minim trei instrucțiuni. De exemplu

xchg ax,bx

```
push bx
      pop ax
      mov ax,bx
      sau
      mov bx,ax
```

Schimbă conținutul regisitrii AX și BX, operație echivalentă cu

xchg al,varmem (sau xchg varmem,al)

**XLAT [labela\_de\_translator]**

Instrucțiunea **XLAT** "traduce" octetul din AL într-un alt octet, utilizând în acest scop o tabelă de corespondență creată de utilizator, numita *tabelă de translator*. Instrucțiunea are sintaxa

schimbă conținutul regisitrii AL cu cel al variabilei octet *varname*.

Labela de translator este adresa directă a unei și de octet. Instrucțiunea pretează la intrarea adresă

fară a tabelă de translator să subiștă sub unul din următoarele două moduri:

- DS:BX (implictit, dacă lipsesc operatorul operandului)
- registru-segment:BX, dacă operandul instrucțiunii **XLAT** este prezent, registru segmentul fiind determinat pe baza direcției ASSUME coreșpunzătoare opereandului.

Efectul instrucțiunii **XLAT** este înlocuirea octetului din AL cu octetul din tabelă ce are numărul de ordine valoarea din AL (primul octet din tabelă are indexul 0). De exemplu, se poate

```
mov bx,offset Tabela
      mov al,6
      mov Tabela
```

dată de conținutul celei de-a 7-a locații de memorie (de index 6) din *Tabela* în AL.

De exemplu, se poate face și prin intermediul instrucțiunilor PUSH și POP.

În cărare unui regisztru de segment se poate face și prin registrul ES portat în locul său

De exemplu, se poate face și în cadrul unei instrucțiuni de înlocuire cu

AX 2	996:?	BX 1	998:?	SP 1000	1000:?
------	-------	------	-------	---------	--------

iar după execuția instrucțiunii pop bx vom obține configurația

AX 2	996:?	BX 2	998:1	SP 998	1000:?
------	-------	------	-------	--------	--------

După execuția instrucțiunii pop ax vom avea

AX 1	996:2	BX 2	998:1	SP 996	1000:?
------	-------	------	-------	--------	--------

După mov bx,2 și push bx vom avea

AX 1	996:?	BX ?	998:1	SP 998	1000:?
------	-------	------	-------	--------	--------

După mov ax,l și push ax vom aveam

AX ?	996:?	BX ?	998:?	SP 1000	1000:?
------	-------	------	-------	---------	--------

La început avem situația

Registri	Stiva
----------	-------

```
pop bx
      pop ax
      push bx
      mov bx,2
      push ax
      mov ax,l
```

conținutul stivii preconiza regisitrii AX, BX și SP pe parcursul execuției sevenefiei

De exemplu, să presupunem că SP conține initial valoarea 1000h și să urmărim evoluția

Dăm un exemplu de secvență care translatează o valoare zecimală 'numar' cuprinsă între 0 și 15 în cifra hexazecimală (codul ei ASCII) corespunzătoare:

```
.data
TabHexa db '0123456789ABCDEF'

.code
mov bx,offset TabHexa
mov al,numar
xlat TabHexa ;sau doar xlat fără parametru daca s-a
               ;,specificat ASSUME ds:data
```

O astfel de strategie este des utilizată și se dovedește foarte utilă în cadrul pregătirii pentru tipărire a unei valori numerice întregi (practic este vorba despre o conversie *valoare numerică registru – string de tipărit*).

#### 4.1.1.2. Instrucțiuni de transfer de intrare-iesire

Microprocesoarele 80x86 dispun de un spațiu de memorie independent numit spațiu de adrese I/O. Există aici 65536 adrese I/O (numite *porturi*) care sunt utilizate ca și canale de date și control a resurselor externe microprocesorului (unități de disc, adaptoare video, tastatura, imprimanta). Fiecare interfață corespunzătoare unui echipament periferic are asociate niște porturi specifice, atribuirea codurilor acestor porturi pentru fiecare interfață făcându-se la proiectarea sistemului de calcul. Comunicarea între microprocesor și mediul exterior reprezentat de interfețele de intrare-iesire se face prin intermediul unor instrucțiuni specializate (IN și OUT) care accesează porturile corespunzătoare.

IN <i>acumulator, port</i>	acumulator <-> <port>, unde acumulator = AL sau AX; port = val.imediată sau DX	-
OUT <i>port,acumulator</i>	port <-> <acumulator>	-

Instrucțiunea IN copiază o valoare din portul I/O selectat în acumulator (AL sau AX). Operandul sursă (*port*) poate fi o valoare imediată (dacă este < 256) sau registrul DX. De exemplu *in ax,42h* copiază un cuvânt din portul 42h în AL, iar

```
mov dx,1000
in al,dx
```

copiază un octet din portul 1000 în AX.

Instrucțiunea OUT este complementara instrucțiunii IN, realizându-se un transfer de informație dinspre acumulator spre un port de ieșire. De exemplu, secvența

```
mov al,63
mov dx,500
out dx,al
```

scrie valoarea 63 la portul I/O 500.

#### 4.1.1.3. Instrucțiuni de transfer al adreselor

Aceste instrucțiuni sunt deosebit de utile la operațiile cu șiruri, la transmiterea de parametri către proceduri etc. În tabelul de mai jos *reg* desemnează un registrul general (deci orice registru diferit de registri segment).

LEA <i>reg,mem</i>	<i>reg</i> <-> offset( <i>mem</i> )	-
LDS <i>reg,mem</i>	<i>reg</i> <-> < <i>mem</i> >, DS <-> < <i>mem+2</i> >	-
LES <i>reg,mem</i>	<i>reg</i> <-> < <i>mem</i> >, ES <-> < <i>mem+2</i> >	-

Instrucțiunea LEA (*Load Effective Address*) transferă deplasamentul operandului din memorie *mem* în registrul destinație. De exemplu

```
lea ax,v
```

încarcă în AX offsetul variabilei v, instrucțiune echivalentă (cu excepția situațiilor prezentate mai jos) cu

```
mov ax, offset v
```

Față de această ultimă variantă, instrucțiunea LEA are avantajul că operandul sursă poate fi indexat. De exemplu, instrucțiunea

```
lea ax,[bx+v]
```

nu are ca echivalent direct o singură instrucțiune MOV cu un operator OFFSET, deoarece operatorul OFFSET impune efectuarea evaluărilor la momentul asamblării. Chiar și afirmația noastră de mai sus (*lea ax,v* echivalent cu *mov ax,offset v*) rămâne adevărată numai în condițiile în care deplasamentul variabilei v este determinabil la momentul asamblării.

Instrucțiunile LDS (*Load pointer using DS*) și LES (*Load pointer using ES*) transferă adresa far memorată la *mem* (deci continutul variabilei dublucuvânt mem!) în perechea de registri DS:reg respectiv ES:reg. Instrucțiunile LDS și LES constituie un mijloc eficient de pregătire în registri a adreselor far ale unor variabile în vederea unor prelucrări ulterioare.

Dintre cele trei instigării prezentate în cadrul acestui seficiu, instigării transferă o similitudine cu adverbată tillită specifică, fiind într-adverat o instigării ce corespundă adverbată tillită specifică, fiind într-adverat o instigării ce transferă o instigării (DSD și LDS) sunt înă net diferenție de LBA, ele transferând de fapt nu adresa ale unor memori, ci conținutul său unor locații de memorie ce au posibilitatea de a fi interpretate ulterior drept adresa (similară deci ca semantica cu noțiunea de variabilă pointer din limbajele de programare de nivel înalt). Ca urmare, în momentul apelării instigării transferă o instigării (DSD și LDS și LES) sămboluri la aceea că transferă de dublucuvinte, nefind singura instigării transferă o instigării (DSD și LDS) care să obțină că manipularea unor dublucuvinte să fie chiar în sensul de adresa fizice complete (adresă fizică, adică de formă segment:deplasament) se obține că stările celor două instigării (DSD și LDS) nu impună în nici un fel o operanță similară să fie neapărat o adresă.

In acest context, este mai natural să interpretăm intuitiv aceste instrucțiuni drept niște „superinstrucțiuni MOV”, capabile să opereze direct cu dublucvîrte (instrucțiunea MOV poate scrie o adresa sau o cetești sau cuvinte). De aceea pozitia noastră este că LDS și LES ar trebui tratate mai degrabă ca alturi de instrucțiunea MOV în cadrul secțiunii „Instrucțiuni de transfer de la general”. Pe de altă parte, este de înțeles de ce în literatura de specialitate este sunt tratate în cadrul secțiunii „Instrucțiuni de transfer al adreselor”; ele sunt într-o altă parte de instrucțiuni de transfer care să avem de manipulat global adrese fară, astfel încât trebuie neapărat ca acest lucru să îl facem interpretarea și utilizarea ultimoră a valoilor transferate drept adresă! Sau altfel spus: dacă se permite să scrie într-o instrucțiune de transfer al adreselor, atunci trebuie să fie manipulată și informația de pe adresa de transfer, astfel încât să avem de manipulat global adrese fară, astfel încât trebuie neapărat ca acest lucru să îl facem.

(goaniniutii) *{mem}* dublucuvant se transferă în dublucuvantul DS; rege sau respectiv ES; rege.

O să vedem în cadrul secțiunii dedicată operează într-un interval de viteză mai mare decât cea de la limită. În acestă secțiune vom analiza și rezolvarea unei ecuații diferențiale care descrie o funcție care crește și se comportă ca un operător de înmulțire.

operandului sursă! Aici apare greșeala de interpretare comisă de cei în cauză: LDS și LES nu transferă adresa far a operandului sursă, ci continutul operandului sursă!

Revenind și reanalizând exemplul anterior în care am avut

```
lds bx, varp ; transferul conținutului variabilei varp în DS:BX - corect!
```

este foarte important să nu facem confuzie și sub influența instrucțiunii LEA să interpretăm vreodată că am avea ca efect al instrucțiunii de mai sus "transferul adresei variabilei varp în ds:bx" (concluzie greșită!). Dacă într-adevăr am dori acest lucru, l-am putea realiza corect astfel:

```
; secvența ce urmează încarcă în ds:bx adresa far a variabilei varp!
```

```
mov ax, SEG varp ;operatorul SEG l-am prezentat în 3.2.2.7.  
mov ds,ax ;transfer adresă segment varp în DS  
lea bx, varp ;transfer offset varp în BX
```

#### 4.1.1.4. Instrucțiuni asupra flagurilor

Următoarele instrucțiuni sunt specifice flagurilor (sau *indicatorilor*, cum se mai numesc), acționând numai asupra registrului de flaguri. Din această cauză apelul lor nu necesită operanzi specificați explicit.

Următoarele patru instrucțiuni sunt *instrucțiuni de transfer* al indicatorilor:

Instrucțiunea **LAHF** (*Load register AH from Flags*) copiază indicatorii SF, ZF, AF, PF și CF din registrul de flag-uri în biții 7, 6, 4, 2 și respectiv 0 ai registrului AH. Conținutul biților 5,3 și 1 este nedefinit. Indicatorii nu sunt afectați în urma acestei operații de transfer (în sensul că instrucțiunea LAHF nu este ea însăși generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor și atât).

Instrucțiunea **SAHF** (*Store register AH into Flags*) transferă biții 7, 6, 4, 2 și 0 ai registrului AH în indicatorii SF, ZF, AF, PF și respectiv CF, înlocuind valorile anterioare ale acestor indicatori.

Instrucțiunea **PUSHF** transferă toți indicatorii în vârful stivei (conținutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectați în urma acestei operații.

Instrucțiunea **POPF** extrage cuvântul din vârful stivei și transferă din acesta indicatorii corespunzători în registrul de flag-uri.

Aceiunea unor instrucțiuni ale limbajului de asamblare este determinată de valoarea unora dintre indicatorii de condiție (de exemplu, după cum se va vedea, instrucțiunile pe șiruri acționează "crescător" sau "descrescător" în funcție de valoarea flag-ului DF). Limbajul de asamblare pune la

dispoziția programatorului niște *instrucțiuni de setare* a valorii indicatorilor de condiție, pentru ca programatorul să poată influența după dorință modul de acțiune a instrucțiunilor care exploatează flaguri.

Operanții instrucțiunilor de setare a flag-urilor sunt implicați (fiecare instrucțiune setează numai un anumit flag), apelul acestor instrucțiuni făcându-se în consecință doar prin specificarea mnemonicii. Nici una dintre aceste instrucțiuni nu afectează vreun alt flag decât cel pe care-l setează.

CLC	CF=0	CF
CMC	CF = ~CF	CF
STC	CF=1	CF
CLD	DF=0	DF
STD	DF=1	DF
CLI	IF=0	IF
STI	IF=1	IF

Instrucțiunea **CLC** (*Clear Carry flag*) poziționează la 0 indicatorul de transport CF. Instrucțiunea **CMC** (*CoMplement Carry flag*) setează valoarea flag-ului CF în valoarea sa complementară, iar instrucțiunea **STC** (*SeT Carry flag*) îl poziționează la 1.

Instrucțiunea **CLD** (*Clear Direction flag*) poziționează la 0 indicatorul de direcție DF. Instrucțiunea **STD** (*SeT Direction flag*) poziționează DF la valoarea 1.

Instrucțiunea **CLI** (*Clear Interrupt-enable flag*) poziționează la 0 indicatorul de validare a întreruperii (IF), ceea ce va produce dezactivarea întreruperilor mascabile. Activarea acestor întreruperi se face cu ajutorul instrucțiunii **STI** (*SeT Interrupt-enable flag*) care setează la 1 flagul IF. Întreruperile nemascabile sunt recunoscute indiferent de starea bitului IF (întreruperile vor fi tratate în capitolul 5).

#### 4.1.2. Instrucțiuni de conversie

Scopul unor tehnici sau instrucțiuni de conversie în cadrul limbajelor de programare este ca plecând de la valori definite într-un anumit mod (tip de dată în cazul unui limbaj de nivel înalt – dimensiune de reprezentare în cazul limbajului de asamblare) acestea să fie modificate ca reprezentare (efect distructiv) sau numai interpretate temporar sub o altă formă (efect nedistructiv).

Tehnica de conversie nedistructivă la nivelul limbajului de asamblare este oferită de utilizarea operatorului **PTR**, prezentat în 3.2.2.7. Acesta este echivalentul operatorilor de tip **cast** de la nivelul limbajelor de programare de nivel înalt.

Instructiunile de conversie prezentate in acesta secciu se realizeaza prin zeroizarea unor registri (mai precis, de pe memoria RAM) si ca urmare nu se poate folosi un simplu MOV in acest caz.

complementarea cu 8 sau 16 zerosi sau cu 8 sau 16 de cifre binare I (acest lucru depinde tocmai de semnul numarului) si ca urmare nu se poate folosi un simplu MOV in acest caz.

Dupa cum am văzut în capitolul I (secciu 1.3.2.3), arhitecturile de tip 80x86 utilizează endian la fel de cel de la care le accesează?

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definitie (ex: accesarea octetelor drept octet și nu drept secvenție de octet) interpretează ca și cum să fie adăugat la definitie.

Elaborarea în cod surșă, când și în ce fel trebuie să fiia cont acela de reprezentare little-endiană.

Instructiunea CBW converteste octetul din AL în cuvantul său semn AX (mai precis, modulatela little-endiană de reprezentare a datelor). Întrucât care se punte este căt de consistent trebuie să fie programatorul despre particularitatea de reprezentare ale unor date atunci când este utilizată little-endiană suplimentar în nici un fel pentru a asigura corectitudinea accesării și manipulării datei.

Este evident deci că accesă două instrucțiuni (CBW, CWD) sunt instrucțiuni de conversie cu semn.

...  
mov al, a ;se micărcă în AL codul ASCII al caracterului „d”  
mov bx, b ;se micărcă în BX valoarea -15642; ordinea octetelor în BX va fi însă imemoryate conform represențării little-endian în regiștri memoriale, deoarece numai reprezentarea în memorie folosește reprezentările structurale normale, echivalente unei inversări regiștrilor de memorie. Înversarea în memoria de reprezentare a ordinii octetelor în mod consistent cu dimensiunea de reprezentare definită utilizând valoari b ;corescătării la nivel de regiștri - toamă datotata a ordinii octetelor în mod de tip cuvânt).

les dx, c ;se micărcă în combinația de regiștri ES:DX valoarea dublucuvant ;12345678h, mai précis, în ES se micărcă 1234h iar în DX se micărcă 5678h, ;aceeași observație și aici; regiștrul de ordinie octetelor din memorie care este utilizată scopul dorilor Exemplu:

```
c dd 12345678h  
b dw -15642,25ah  
a db ,d,-25,120
```

mov ax, 0 ;conversie frază semn a valoři din AL în AX (byte – word)

```
mov al, 25 ;conversie frază semn a valoři din AL în AX (byte – word)  
mov ah, 0 ;conversie frază semn a valoři din AL în AX (byte – word)
```

Care sunt atunci instrucțiunile (echivalente lor) de conversie frază semn? Răspuns: nu există în dimensiunile de reprezentare în care sunt disponibile instrucțiuni simple complete cu zerouri și nenegative (de asemenea conversia metodă una simplă instrucțiune MOV realizază scopul dorilor Exemplu):

obține valoarea -10000 în DX:AX.

```
cwd  
mov ax,-10000
```

Analog, pentru conversia cu semn cuvant - dublu cuvant, instrucțiunea CWD extinde cuvantul săn semn din AX în dubluvalentul său semn DX:AX. Exemplu:

extinde valoarea octet -1 din AL în valoare cuvant -1 din AX.

```
cbw  
mov al,-1
```

Instructiunea CBW converteste octetul din AL în cuvantul său semn AX (mai precis, modificația instructiunii regiștrului AH). De exemplu,

CBW	conversie cuvant specificat în AX la dublu cuvant în DX:AX (extensie de semn)
CWD	conversie octet cuvant din AL, la cuvant din AX (extensie de semn)

Acstei instrucțiuni sunt CBW (Convert Byte to Word) și respectiv CWD (Convert Word to Doubleword).

;78h 56h 34h 12h, după acțiunea instrucțiunii les ordinea octetilor în cadrul registrelor va fi cea structurală normală, adică 12h 34h 56h 78h.

Dacă însă se dorește accesarea sau interpretarea datelor sub o formă diferită față de modalitatea de definire (ex: octeți ai cuvintelor sau dublucuvintelor, cuvinte ale dublucuvintelor, secvențe de octeți interpretate drept cuvinte sau dublucuvinte) atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să își asume însă întreaga responsabilitate a interpretării și accesării corecte a datelor. Cu alte cuvinte, în astfel de situații programatorul este obligat să constienteze particularitățile de reprezentare little-endian (este vorba despre ordinea de plasare a octetilor în memorie) și să utilizeze modalități de accesare a datelor în conformitate cu această ordine. Exemplu:

```
assume cs:code, ds:data
data segment
    a dw 1234h           ;datorită reprezentării little-endian, în memorie octeții sunt
                        ;plasati ;astfel:
    b dd 11223344h       ;34h 12h 44h 33h 22h 11h
                        ;adresa      a   a+1   b   b+1   b+2   b+3
data ends
code segment
start:
    mov ax, data
    mov ds, ax
```

;să presupunem că dorim transferul primului octet din a în AL. În primul rând trebuie să ne fie clar că la nivelul arhitecturii 80x86 primul octet din structura lui a este 12h, iar primul octet din reprezentarea lui a este 34h (datorită ordinii de plasare little-endian). Ca urmare, trebuie să fie clar ce semnificație acordăm sintagmei “primul octet al lui a”: primul octet din structură sau primul octet din reprezentare? Să presupunem că dorim primul octet din structură. Dacă am încerca să facem aceasta prin instrucțiunea

```
mov al, a             ;syntax error! a este cuvânt, iar AL este octet;
```

vom obține o eroare de sintaxă datorită dimensiunii diferite de reprezentare dintre operanții instrucțiunii **mov**. Ca urmare, va trebui să utilizăm o instrucțiune de conversie de tip prin care să selectăm doar 1 octet din cuvântul a. În acest scop folosim operatorul PTR (vezi 3.2.2.7.) prin care specificăm sub ce tip de dată dorim a fi interpretat operandul:

```
mov al, byte ptr a    ;accesarea lui a drept octet, selectarea octetului de la adresa a și
                      ;transferul aceluia octet în registrul AL
```

Dacă facem însă așa se va transfera “primul octet de la adresa a”, adică “primul octet al reprezentării lui a”, care, datorită reprezentării little-endian este 34h. Am spus mai sus însă că ne-am propus transferul “primului octet din structura lui a”, acesta fiind 12h. Datorită reprezentării little-endian acest octet se găsește la adresa a+1. Ca urmare, instrucțiunea care rezolvă ceea ce ne-am propus este:

```
mov al, byte ptr a+1 ;accesarea lui a drept octet, efectuarea calculului de adresă a+1,
                      ;selectarea octetului de la adresa a+1 (octetul de valoare 12h) și
                      ;transferul său în registrul AL.
```

Simbolul ‘+’ care apare în cadrul expresiei a+1 reprezintă un **operator** și nu o **instrucțiune**. Operatorii (vezi capitolul 3) efectuează calcule cu valori constante determinabile la momentul asamblării. Semantica sa nu reprezintă aici 12h+1 = 13h, adică nu înseamnă “conținutul lui a” +1, deoarece “conținutul unei zone de memorie” nu poate fi o valoare determinabilă la momentul asamblării (când zona destinată execuției programului nici măcar nu există!). În schimb, știm din capitolul 3 (secțiunea 3.1) că în limbajul de asamblare, valoarea asociată simbolului ce desemnează o variabilă (etichetă de date) este prin definiție adresa sa (deplasament), valoare determinabilă la momentul asamblării. Ca urmare, expresiei a+1 i se asociază de fapt ca înțeles “valoarea etichetei de date a” +1, reprezentând astfel o aritmetică de adrese.

Dacă s-ar fi dorit transferul în AL a “conținutului lui a” + 1 acest lucru nu putea fi exprimat prin utilizarea unui operator, ci trebuie făcut prin utilizarea instrucțiunii **add** corespunzătoare:

```
mov al, a             ;“conținutul lui a” se transferă în registrul AL
add al, 1             ;aici se adună 1 la “conținutul lui a”
```

Pe baza celor discutate până acum și înănd cont de ordinea de mai sus a plasării în memorie a octetelor se pot ușor verifica următoarele rezultate:

mov dx, word ptr b+2	;dx:=1122h
mov dx, word ptr a+4	;dx:=1122h deoarece b+2 = a+4 , în sensul că aceste expresii de tip pointer desemnează aceeași adresă și anume adresa octetului 22h.

mov dx, a+4	;deoarece a este de tip cuvânt această instrucțiune este de fapt echivalentă cu cea de mai sus, nefiind necesară utilizarea operatorului de conversie PTR.
-------------	--

mov bx, word ptr b	;bx:=3344h
mov bx, word ptr a+2	;bx:=3344h, deoarece ca adrese b = a+2.

les cx, dword ptr a	;es:cx:=3344h:1234h, deoarece dublucuvântul ce începe la adresa a este format din octeții 34h 12h 44h 33h care
---------------------	--

Într-o reprezentare binară, în care se pot opera cu numere întregi și frație, se va avea nevoie de un număr de biti mai mare decât cel necesar pentru reprezentarea unei frații. Dacă se folosește un număr de  $n$  biti pentru reprezentarea unei frații, atunci se va avea nevoie de un număr de  $n+1$  biti pentru reprezentarea unei frații. În acest caz, primul bit este rezervat pentru semn și următorii  $n$  biti sunt rezervati pentru reprezentarea numerelor întregi.

#### 4.2.1.1. Adunarea și scăderea

Într-o reprezentare binară, adunarea și scăderea sunt realizate folosind același set de instrucțiuni. Într-un procesor de 32 de biti, adunarea și scăderea sunt realizate folosind același set de instrucțiuni. Într-un procesor de 16 de biti, adunarea și scăderea sunt realizate folosind același set de instrucțiuni. Într-un procesor de 8 de biti, adunarea și scăderea sunt realizate folosind același set de instrucțiuni.

#### 4.2.1.2. Operațiuni aritmétice

Facilitatea de calcul aritmetică oferită de instrucțiunile masină se dovedește că surprinzător de rudimentare. De exemplu, nu se permite lucru simplu în aritmética numerică reală! Putem totuși să adunăm și să înțelepăm în mod direct că instrucțiunile de adunare și scădere sunt rezultatul unei operațiuni aritmétice.

#### 4.2. OPERAȚII

```
ldsd bx, dword ptr b           ;ax := 4412h
mov ax, word ptr a+1          ;ds:bx := 1122h:3344h
                                ;doublewordl 3441234h.
                                ;(datotita reprezentarii little-endian) înseamnă de fapt
```

122 Arhitectura calculatorelor. Limba jargonului de asamblare Box86.

ADD d,s	$d <- (d)+(s)$	AF,C,F,O,F,P,F,S,F,ZF
ADC d,s	$d <- (d)+(s)+CF$	AF,C,F,O,F,P,F,S,F,ZF
SUB d,s	$d <- (d)-(s)$	AF,C,F,O,F,P,F,S,F,ZF
DEC d	$d <- (d)-1$	AF,O,F,P,F,S,F,ZF
NEG d	$d <- (0-d)$	AF,C,F,O,F,P,F,S,F,ZF
MUL	<p>dacă s este de tip octet aritmici</p> <p><math>DX:AX &lt;- (AL)*S</math></p> <p>dacă s este de tip cuvânt aritmici</p> <p><math>DX:AX &lt;- (AX)*S</math></p>	<p>CF,O,F,modificări</p> <p>AF,P,F,S,F,ZF nedefiniți;</p> <p>AH (DX) conține valoarea unită a lui OF sunt 1, atunci</p> <p>dacă CF și OF sunt 1, atunci</p> <p>operanții sunt tratăți ca întregi</p> <p>AH (DX) conține valoarea ≠ 0</p>
IMUL	<p>dacă s este de tip octet aritmici</p> <p><math>DX:AX &lt;- (AL)*S&lt;</math></p> <p>dacă s este de tip cuvânt aritmici</p> <p><math>DX:AX &lt;- (AX)*S&lt;</math></p>	<p>CF,O,F,modificări</p> <p>AF,P,F,S,F,ZF nedefiniți;</p> <p>AH (DX) conține valoarea unită a lui OF sunt 1, atunci</p> <p>dacă CF și OF sunt 1, atunci</p> <p>operanții sunt tratăți ca întregi</p> <p>AH (DX) conține valoarea ≠ 0</p>
DIV	<p>dacă s este de tip cuvânt aritmici</p> <p><math>DX &lt;- (DX:AX)&gt;MOD&lt;s&gt;</math></p> <p>AX &lt;- (DX:AX)&gt;DIV&lt;s&gt;</p> <p>AH &lt;- (AX)&gt;MOD&lt;s&gt;</p> <p>AL &lt;- (AX)&gt;DIV&lt;s&gt;</p>	<p>AF,C,F,O,F,P,F,S,F,ZF</p> <p>nedefiniți</p> <p>La obținerea unui cat ce nu</p> <p>se mărează „de la stînga”</p> <p>se mărează „de la dreapta”</p>
IDIV	<p>dacă s este de tip octet aritmici</p> <p><math>DX &lt;- (DX:AX)&gt;MOD&lt;s&gt;</math></p> <p>AX &lt;- (DX:AX)&gt;DIV&lt;s&gt;</p> <p>AH &lt;- (AX)&gt;MOD&lt;s&gt;</p> <p>AL &lt;- (AX)&gt;DIV&lt;s&gt;</p>	<p>AF,C,F,O,F,P,F,S,F,ZF</p> <p>nedefiniți</p> <p>La obținerea unui cat ce nu</p> <p>se mărează „de la stînga”</p> <p>se mărează „de la dreapta”</p>

## INSTRUCȚIUNI ARITMETICE

Microprocesorul realizează adunarea  $C = A + B$  și obține

$$C = E6h = 11100110b (= 230 în interpretarea fără semn și -26 în interpretarea cu semn)$$

Se observă deci că simpla adunare a configurațiilor de biți (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obținut, atât în interpretarea cu semn cât și în cea fără semn.

Instrucțiunea ADD are forma

**ADD destinație, sursă**

Ea adună cei doi operanzi, rezultatul fiind depus în operandul destinație, cu actualizarea corespunzătoare a flagurilor. Cei doi operanzi trebuie să fie reprezentați ambii pe octet sau ambii pe cuvânt.

Instrucțiunea SUB are forma

**SUB destinație, sursă**

Aceasta scade operandul sursă din operandul destinație, păstrând rezultatul în operandul destinație. De exemplu, atribuirea  $E := A + B - C$  (unde A, B, C și E sunt valori reprezentate pe cuvânt) se realizează prin secvența

```
mov ax, A
add ax, B
sub ax, C
mov E, ax
```

În urma efectuării unei operații de adunare sau scădere, 80x86 setează flag-ul C (*carry flag*) cu valoarea 0 (marcând desfășurarea normală a operației) sau 1 (marcând depășirea spațiului de reprezentare a operandului destinație de către valoarea rezultată în cazul unei adunări, respectiv necesitatea "împrumutului" unei unități în cazul scăderii).

Este evident că pentru programarea unei adunări sau scăderi pe mai mult de 16 biți trebuie să ținem cont de valoarea din CF. Este ceea ce fac instrucțiunile ADC și SBB. ADC are același efect ca și ADD numai că adună în plus și valoarea din CF. SBB are același efect ca și SUB, scăzând însă și "împrumutul" făcut de o eventuală scădere anterioară, împrumut semnalat în CF. Adunarea dublucuvântului din CX:BX la dublucuvântul din DX:AX se face astfel:

```
add ax,bx
adc dx,cx
```

Să luăm acum un exemplu mai complex care să ilustreze necesitatea și utilitatea instrucțiunii ADC. Fie atribuirea  $x := x + y$ , unde  $x$  și  $y$  sunt numere fără semn reprezentate în baza 2 pe căte 48 de biți. Convenim reprezentarea astfel încât cifra binară de rang 0 să aibă adresa cea mai mică, iar cifra binară de rang 47 să aibă adresa cea mai mare. Secvența de instrucțiuni ce realizează această atribuire este:

x	dw	1234h, 2345h, 5678h, 0
y	dw	4544h, 4545h, 6766h ;reprezentările hexa ale celor doi operanzi
mov	ax, y	
add	x, ax	;efectuăm 1234h + 4544h
mov	ax, y+2	
adc	x+2,ax	; efectuăm 2345h + 4545h + CF ;(CF - cifra de transport de la adunarea anterioară)
mov	ax, y+4	
adc	x+4,ax	;efectuăm 5678h + 6766h + CF
adc	x+6,0	;necesară pentru adunarea în final a cifrei de ;transport rămasă!

Scăderea valorii dublu cuvânt CX:BX din valoarea dublu cuvânt din DX:AX (ceva de genul "DX:AX := DX:AX - CX:BX") trebuie făcută ținând cont de eventuala cifră binară "împrumutată" la prima operație de scădere :

```
sub ax,bx
sbb dx,cx
```

Atragem atenția că pentru corecta funcționare a instrucțiunilor ADC și SBB trebuie să se aibă în vedere păstrarea valorii din CF între respectivele operații compuse. De exemplu, secvența

```
add ax,bx
sub si,si ;setează CF pe 0
adc dx,cx
```

nu va aduna corect CX:BX la DX:AX deoarece instrucțiunea SUB modifică potențial valoarea din CF între cele două operații ADD și ADC.

Datorită frecvenței ridicate a apariției în cadrul programelor a adunărilor cu 1 (incrementare) precum și a scăderilor cu 1 (decrementare) limbajul de asamblare conține instrucțiunile INC și respectiv DEC cu sintaxa

INC	operand	și	DEC	operand
ADD	operand,1	și	SUB	operand,1

echivalente ca efect deci cu

Avantajele introducerii acestor instrucțiuni rezidă atât în codul compact rezultat (se reprezintă doar pe 1 octet comparativ cu variantele ADD sau SUB care necesită 3) cât și în viteza mult mai mare de execuție.

Instrucțiunea NEG are sintaxa

**NEG destinație**

care ar același rezultat de acțiune ca și MUL, cu precizarea că respectă regulile semnelor cunoscute din aritmetică. Astfel, se observă

plasează valoarea -20 (0FFECh) în AX. Dacă în loc de MUL să folosi MUL, rezultatul depășește limita de reprezentare a lui Z și va produce în AX valoarea  $254 * 10 = 2540$  (09ECh).

Vom urmări în secvență de mai jos un mod de calcul al valorii Z = X \* Y, unde X și Y sunt valori care sunt necesare 64 biți. Iată exemplu:

```
        mov ax, X      ;depuine cuvântul cel mai puin semnificativ al valorii X în AX
        mul y          ;înmulțire cu valoarea din Y, rezultatul în DX:AX
        mov z, ax      ;depuine primele 2 cifre hexa obținute (cele mai puin semnificative) în locul
        mul y          ;înmulțire cu valoarea din Y
        mov ax, x+2    ;preia următorul cuvant al sursei pentru a-l înmulți
        add dx, 0      ;se adună transportul anterior de la înmulțire
        adc dx, 0      ;se adună transportul rezultatul cu următorul cifre
        mov z+2, ax    ;se actualizează rezultatul de evenimentul transport în urma adunării
        add dx, 0      ;se adună transportul anterior de la înmulțire
        adc dx, 0      ;se adună transportul rezultatul cu următorul cifre
        mov bx, dx    ;punie în BX cuvântul superior al rezultatului pentru a servi ca transport
        mov z+4, ax    ;corespunzător celui din Y
        mul y          ;înmulțire cu valoarea din Y
        mov ax, x+4    ;preia cuvantul cel mai semnificativ al sursei
        add dx, 0      ;se adună transportul rezultatul cu următorul cifre
        adc dx, 0      ;se adună transportul rezultatul cu următorul cifre
        mov z+6, dx    ;punie cuvântul superior în BX
        mov z+4, ax    ;corespunzător celor mai semnificative 2 cifre hexa ale rezultatului final
        imparitate, astfel:
```

unde operand pot căuta în registrul de memorie sau o variabilă de memorie, el reprezentând imparitatea. Deși parțial este determinat în funcție de dimensiunea de reprezentare a operandului

### DIV operand

Împărțirea face semnă două numere întregi și face cu ajutorul instrucțiunii DIV, care are forma

`mov dx, ax ;corespunzător celor mai semnificative 2 cifre hexa ale rezultatului final`

`mov z+4, ax ;corespunzător celor mai semnificative 2 cifre hexa ale rezultatului final`

`add dx, 0 ;adică cont de transport adunat`

`adc dx, 0 ;adică cont de transport adunat`

`mov bx, dx ;punie în BX cuvântul superior al rezultatului`

`mov z+2, ax ;corespunzător rezultatului de la înmulțire`

`add dx, 0 ;adică cont de evenimentul transport în urma adunării`

`adc dx, 0 ;adică cont de evenimentul transport în urma adunării`

`mov z, ax ;punie primele 2 cifre hexa obținute (cele mai puin semnificative) în locul`

`mul y ;înmulțire cu valoarea din Z`

`mov ax, x ;depuine cuvântul cel mai puin semnificativ al valorii X în AX`

`add dx, 0 ;adică cont de transport adunat`

`adc dx, 0 ;adică cont de transport adunat`

`mov z+2, ax ;corespunzător rezultatului de la înmulțire`

`add dx, 0 ;adică cont de evenimentul transport în urma adunării`

care ar același rezultat de acțiune ca și MUL, cu precizarea că respectă regulile semnelor cunoscute

### MUL operand

Instrucțiunea `mul ax` depune în DX:AX parțial veciul valoare din AX.

```
        mov E, al
        mul B
        mov al, A
```

se observă:

`mov E, al ;calculul valorii E = A*B, cu operandii reprezentati pe octet, se poate efectua print`

`mul B ;produsul nu se poate reprezenta pe un spațiu dublu decât operandi, punând CF=1 și`

`mov al, A ;dacă îmulițera furnizată rezultatul pe un spațiu dublu decât operandi, ea semnalează dacă`

`OF=1, în caz contrar se pun CF=0 și OF=0.`

`add al, E ;dacă operand este octet, el este multiplicat cu valoarea din AL, rezultatul memorându-se în AX.`

`mov ax, al ;dacă operand este general sau o variabilă de memorie, instrucțiunea MUL realizează`

`înmulțirea face semnă a valoarii operaandului cu valoarea din AL sau AX, acționarea instrucțiunii`

`depășind de dimensiunea operaandului.`

`neg ax ;formă generală a instrucțiunii de înmulțire face semn este`

`neg bx ;înmulțirea pure la dispoziția utilizatorilor instrucțiuni de înmulțire și împărțire cu semn`

`neg bx, ax ;pot fi interpretate și ca valoare face semn, aceasta fiind în acest caz 65355.`

`neg ax, bx ;dacă interpretă ca valoare face semn, accesați fiind în acest caz -1.`

`neg ax, bx, ax ;dacă interpretă ca valoare face semn, accesați fiind în acest caz -1 (16 cifre -1)`

`neg ax, bx, bx, ax ;dacă interpretă ca valoare face semn, accesați fiind în acest caz -1 (16 cifre -1)`

exemplu, în urma execuției secvenței

și arătă schimbarea semnului operandului (registrul sau variabila de memorie). De

126 Arhitectura calculatorelor. Limbajul de assembly Box86.

\* dacă împărțitorul este octet, atunci deîmpărțitul este conținutul registrului AX. Câștul va fi depus în AL, iar restul în AH.

\* dacă împărțitorul este reprezentat pe cuvânt, atunci deîmpărțitul este DX:AX. Câștul va fi depus în AX, iar restul în DX.

De exemplu,

```
mov ax,51
mov dl,10
div dl
```

realizează împărțirea conținutului lui AX la conținutul lui DL, memorând câștul 5 în AL și restul 1 în AH, iar

```
mov ax,2
mov dx,1
mov bx,10h
div bx
```

împarte 10002h din DX:AX la 10h din BX, memorând câștul 1000h în AX și restul 2 în DX.

Să observăm că, dacă n este lungimea de reprezentare a deîmpărțitului, câștul se impune a fi reprezentat pe n/2 biți. Dacă rezultatul obținut va necesita pentru reprezentare o lungime mai mare, atunci microprocesorul va semnaliza depășire prin generarea unei întreruperi 0 (împărțire prin 0). De exemplu, următoarea secvență generează o întrerupere 0 (vezi capitolul 5):

```
mov ax,0ffffh
mov bl,1
div bl
```

Pentru împărțirea cu semn disponem de instrucțiunea

**IDIV** *operand*

cu aceleași observații și reguli de funcționare ca și DIV, înținând cont în plus de semnul operanzilor. Spre deosebire de teorema fundamentală a împărțirii din aritmetică, aici avem  $D = C * I + R$ , unde:

- semnul lui C este dat de regula semnelor D/I
- semnul lui R coincide cu semnul lui D
- $|R| < |I|$

Astfel, în urma execuției secvenței

data segment		
Divizor		DW 100
.	.	.
code segment		
.	.	.

```
mov ax,-667
cwd
idiv Divizor
```

;extinde numărul în DX:AX

se va depune -6 în AX și -67 în DX.

#### 4.2.1.3. Exemple și exerciții propuse.

a). ....

```
mov ah,0
mov al,-5 ;echivalent cu mov al,11111011b deci echivalent și cu mov al,251
;echivalent și cu mov al,0fbh în hexazecimal; f=1111b și b=1011b
```

;ansamblul celor 2 instrucțiuni de mai sus este echivalent cu mov ax,251 (echivalent în binar cu mov ax,0000000011111011b și în hexazecimal cu mov ax,00fbh) însă nu și ;cu mov ax,-5 (echivalent în binar cu mov ax,111111111111011b și în hexazecimal cu ;mov ax,0fffbh) - diferența constă în completarea conținutului lui AH cu 8 cifre binare 0 ;în cazul lui 251 – interpretare fără semn - și respectiv cu 8 cifre binare 1 în cazul lui -5 ;adică în interpretarea cu semn)

mov bx,10

```
imul bx ;dx:ax := ax*bx = 251 * 10 = 2510 = 09CEh (în AX) și DX:=0
```

(deși aici prin imul e forțată interpretarea cu semn pentru AX, deoarece AX = 0000000011111011b, bitul de semn fiind 0, rezultă că AX = 251 în ambele interpretări)

mul bx ;idem – rezultatele sunt identice datorită observației de mai sus

mov cl, -100 (=9ch = 10011100b = 156 în interpretarea fără semn!)

```
idiv cl ;AX=2510; AL (câștul):= AX idiv (-100) = -25 (=e7);AH (restul):=10 (0ah)
;conținutul lui AX este acum AX=0ae7h
```

```
imul cl ;AX:=AL*CL = (-25)*(-100) = 2500 (=09c4h)
```

```
add ax,10 ;AX:=AX+10 = 2500+10 = 2510 - refacerea valorii inițiale din AX:=2510;
```

```
div cl ;AX=2510; AL (câștul) := AX div 156 = 16 (=10h); AH (restul) := 14 (0eh)
;conținutul lui AX este acum AX=0e10h
```

Reluați discuția, analizați și justificați rezultatele furnizate în situația în care ultimele 5 instrucțiuni de mai sus devin (CL se înlocuiește cu CX):

Limbajul de assembly dispune de un set de patru instrucțiuni pentru realizarea de operații logice la nivel de bit: AND, OR, XOR și NOT.

#### 4.2.2. Operații logice pe bit

Dar dacă apoi „div cl” este înlocuită inițial cu „div cx” iar apoi cu „div ex” ? Jusnică și explicații rezultante obținute. De ce în acest caz nu se mai manifestă situația apărută în cauză „div cl”?

AND <i>d,s</i>	<i>&lt;d&gt; --&gt; &lt;d&gt; \$i &lt;s&gt;</i>	CF,OF,PF,SF,ZF modifică; AF - nedefinit
OR <i>d,s</i>	<i>&lt;d&gt; --&gt; &lt;d&gt; sau &lt;s&gt;</i>	CF,OF,PF,SF,ZF modifică; AF - nedefinit
XOR <i>d,s</i>	<i>&lt;d&gt; --&gt; &lt;d&gt; sau exclusiv &lt;s&gt;</i>	CF,OF,PF,SF,ZF modifică; AF - nedefinit
NOT <i>s</i>	<i>&lt;s&gt; --&gt; &lt;s&gt; negat complement față de I</i>	modifică; AF - nedefinit

Pentru doi biti, unul din surșă, altul din destinație, instrucțiunile logice produc nouă bit destinație având valoarea conformată reguli:

Pentru doi biti, unul din destinație, instrucțiunile logice trebuie să aibă ambițiile acesea și dimensiunea corespunzătoare de bit. Operanții suntă și destinație trebuie să aibă ambițiile acesea și dimensiunea (acetă sau ceea ce).

b111 d	b111 s	DANDs	DORs	DXORs
0	0	0	0	0
0	1	1	1	1
1	0	0	0	0
1	1	1	1	1

Astfel avem

mov al, A  
and al, B  
rezulta 00010000b în AL

mov al, A  
or al, B  
rezulta 1101111b în AL

mov al, B  
xor al, B  
rezulta 1100111b în AL

Ce observăm că se înămplinește dacă „div cl” este înlocuită cu instrucțiunea „divi cl”?

Ce se înămplinește dacă „imul bl” este înlocuită cu instrucțiunea „mul bl”? Analizati și explicați comparativ rezultatele furnizate.

div cl : divi imul une ca valoarea din ax să fie interpretată acum fară semn!

ax = ax div bl = 65036 div 255 = 255 = ff ; restul în AH = 11 = 0bh

deci continutul lui AX este acum AX = 0bff

(suma valoilor absolute ale reprezentărilor cu semn și fară semn la nivelul unității de memorie este 65536 - regulă practică !) – aici se verifică prin  $65536 = 500 + 65036$

imul bl : ax = al \* bl = -5 \* 100 = -500 = 0fe0ch = 65036

mov bl, 100

memorie este 256 - regulă practică ! – aici se verifică prin  $256 = 5 + 251$

(suma valoilor absolute ale reprezentărilor cu semn și fară semn la nivelul unității de memorie este 256 - regulă practică !) – aici se verifică prin  $256 = 5 + 251$

mov al, 251 : 251 = 0fh = -5 - deci instrucțiunea echivalentă cu mov al, -5

ce valoare se obține acum ca rezultat ? Comparați rezultatul furnizat cu cel obținut prin aplicarea operației mul bx în loc de imul bx.

imul bx : mov bx, 10  
mov ax, -5 ; echivalent cu mov al, -5  
cbw

b). Ce se înămplinește exemplul de mai sus dacă accesează devine:

div cx  
add ax, 10  
imul cx  
idiv cx  
mov cx, -100

ce se înămplinește dacă accesă devine:

div cx  
add ax, 10  
imul cx  
idiv cx  
mov cx, -100

Instrucțunea AND este indicată pentru izolarea unui anumit bit sau pentru forțarea anumitor biți la valoarea 0. Astfel, dacă dintr-o configurație pe 8 biți

$$a = \text{xxxxxx}x\ b$$

dorim izolarea bitului i ( $0 \leq i \leq 7$ ), vom crea expresia

$$\text{AND } a, 2^i$$

iar dacă dorim forțarea anumitor biți din a la valoarea 0, să zicem de exemplu biții 0, 2 și 3 (restul rămânând neschimbați) vom crea expresia

$$\text{AND } a, 11110010b \ (= 242)$$

(dacă a conține inițial valoarea 01010110b de exemplu, atunci după execuția instrucției de mai sus, în a se va afla valoarea 01010010b).

Instrucțunea OR poate fi folosită printre altele pentru forțarea anumitor biți la valoarea 1. De exemplu, presupunând că variabila a conține inițial aceeași valoare de mai sus (01010110b), dacă dorim forțarea biților 0, 2 și 3 ai acestei valori la 1 vom folosi instrucțunea OR astfel:

$$\text{OR } a, 00001101b \ (= 13)$$

Valoarea inițială din a este distrusă, în locul ei depunându-se noua valoare rezultată, adică 01011111b.

Instrucțunea XOR este indicată pentru schimbarea valorii unor biți din 0 în 1 sau din 1 în 0. De exemplu, dacă dorim ca biții 4-7 din AL să-și schimbe valorile iar ceilalți să rămână neschimbați, atunci aceasta se poate realiza prin

$$\text{XOR } al, 11110000b$$

Dacă presupunem că inițial în AL am avut valoarea 01010101b, atunci după execuția instrucției de mai sus în AL se va afla valoarea 10100101b (= A5h). Se observă că biții 1 comută valorile inițiale iar biții 0 le lasă neschimbați. De asemenea, instrucțunea XOR poate fi utilizată pentru zerorizarea unui registru:

$$\text{XOR } ax, ax$$

Instrucțunea NOT modifică biții operandului în valoarea lor complementară (0 în 1 și 1 în 0). Efectul este echivalent cu a efectua un XOR cu operandul sursă 0FFh. Exemplu:

```
mov bl, 10110001b
not bl          ;se modifică în 01001110b
xor bl, 0ffh    ;se revine la 10110001b
```

#### 4.2.3. Deplasări și rotiri de biți

Micoprocesoarele 80x86 dispun de o serie de instrucții cu ajutorul cărora pot deplasa sau roti biții din cadrul unui operand în memorie sau registru.

În cadrul octetelor sau cuvintelor biții pot fi deplasări aritmetice sau logic sau rotiri la dreapta sau la stânga cu una sau mai multe poziții. Numărul de deplasări este fie 1, fie o valoare cuprinsă între 1 și 255, valoare specificată în registrul CL.

Instrucțiunile de *deplasare* de biți se clasifică în:

##### - Instrucții de deplasare logică

- stânga - SHL
- dreapta - SHR

##### - Instrucții de deplasare aritmetică

- stânga - SAL
- dreapta - SAR

Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:

##### - Instrucții de rotire fără carry

- stânga - ROL
- dreapta - ROR

##### - Instrucții de rotire cu carry

- stânga - RCL
- dreapta - RCR

Pentru a defini deplasările și rotirile să considerăm că și configurație inițială un octet  $X = abcdefgh$ , unde a-h sunt cifre binare, h este cifra binară de rang 0, a este cifra binară de rang 7, iar k este valoarea existentă în CF ( $CF=k$ ). Atunci,

```
SHL X,1 ;rezultă X = bcdefgh0 și CF = a
SHR X,1 ;rezultă X = 0abcdefg și CF = h
SAL X,1 ; identic cu SHL
SAR X,1 ;rezultă X = aabcdefg și CF = h
ROL X,1 ;rezultă X = bcdefgha și CF = a
ROR X,1 ;rezultă X = habcdefg și CF = h
RCL X,1 ;rezultă X = bcdefghk și CF = a
RCR X,1 ;rezultă X = kabcdedfg și CF = h
```

Deplasările și rotirile pe 16 biți se fac analog. Deplasările și rotirile cu o valoare specificată în CL se fac prin aplicarea repetată a regulilor de mai sus.

**Important!** Se observă că, în toate cazurile, bitul ce părăsește configurația trece în CF.

## INSTRUCȚIUNILE DE DEPLASARE SI ROTIRE DE BITI

Deplasările logice pot fi folosite pentru izolareanumitor bit în interiorul octetelor (cuvintele), iar deplasările aritmétice se pot utiliza pentru izolareanumitor bit în interiorul aceluiași număr de biti ale lui 2.

- Flag-ul AF este măritdeunainedefiniția în urma unei operații de deplasare.

- Indicatorii PF, SF și ZF sunt actualizați ca și în cazul instrucțiunilor logice pe biti.

- Indicatorul CF conține măritdeunainedefiniția (semnul) a fost schimbată de operația de deplasare.

- Instrucțiunile de rotire afecteză numai flagurile CF și OF. CF va conține măritdeuna valoarea lui rotirea unui singur bit OF este setat la valoarea 1 dacă bitul cel mai multor biți, în cauză în care se deplasează un singur bit, OF este setat la valoarea lui rotirea unui singur bit. Pentru operațiile de rotire cu mai mulți biti valoarea flagului OF este nedefinată.

De exemplu, dacă AL conține valoarea 10010110b (=96h = 150 în interpretarea fara semn, sau -6Ah valoarea 00101100b (=2Ch = 44), CF este setat cu 1 (se depune bitul cel mai semnificativ), iar în = -106 în interpretarea lui semn), atunci după execuția instrucțiunii **SHL AL, 1** în AL se va gasi rezultatul de la figura 4.1.

O utilizare frecventă a acestei instrucțiuni apare astăzi când se doresc măritdeunain instrucțiunile SHL în acest caz poate fi operându-lui cu puteri ale lui 2. Deplasarea spre stanga obținută ca efect al instrucțiunii SHL este de la apă de tunchezieră - pierdere cea mai semnificativă (în binar).

De exemplu, dacă dorim măritdeunain instrucțiunile registrului DX cu 16 putem efectua

dor, adică **SHL DX, 4** (deși ultimul byte versiunii de TASM de exemplu permite același cum am

```
shl    dx, 1      ;DX*2
shl    dx, 1      ;DX*4
shl    dx, 1      ;DX*8
shl    dx, 1      ;DX*16
```

SAL,s,1	introduc zerouri.	OF,SF,ZF,PF,CF	modificări semnificative deplasat. La deplasare logica (aritmética) stanga cu o pozitie, CF conține ultimul bit deplasat. La deplasare logica (aritmética) stanga cu o pozitie, AF - nedefinit
SAL,s,cl	deplasare logica (aritmética) stanga cu o pozitie, AF - nedefinit	Daca CL=1, OF este nedefinit	deplasare logica (aritmética) stanga cu o pozitie, AF - nedefinit
SHR,s,1	deplasare logica (aritmética) stanga cu o pozitie, AF - nedefinit	OF,SF,ZF,PF,CF modificări semnificative este deplasat în CF.	deplasare aritmética deplasă cu o pozitie, CF va conține ultimul bit deplasat.
SHR,s,cl	deplasare aritmética deplasă cu o pozitie, AF - nedefinit	OF,SF,ZF,PF,CF modificări semnificative este deplasat în CF.	deplasare aritmética deplasă cu o pozitie, AF - nedefinit ; Daca CL=1, OF este nedefinit
SAR,s,1	deplasare aritmética deplasă cu o pozitie, AF - nedefinit	stanga se extinde bitul de semn, CF va conține ultimul bit deplasat.	deplasare aritmética deplasă cu o pozitie, AF - nedefinit ; Daca CL=1, OF este nedefinit
SAR,s,cl	deplasare aritmética deplasă cu o pozitie, AF - nedefinit	stanga se extinde bitul de semn. Bitul cel mai semnificativ este deplasat în CF.	deplasare aritmética deplasă cu o pozitie, AF - nedefinit
ROL,s,1	rotire stanga cu o pozitie.	OF,CF	rotire stanga cu o pozitie.
ROL,s,cl	rotire stanga cu o pozitie.	OF,CF ; Daca CL=1, OF este nedefinit	rotire stanga cu o pozitie.
ROR,s,1	rotire dreapta cu o pozitie.	OF,CF	rotire dreapta cu o pozitie.
ROR,s,cl	rotire dreapta cu o pozitie.	OF,CF ; Daca CL=1, OF este nedefinit	rotire dreapta cu o pozitie.
RCR,s,1	rotire dreapta cu o pozitie.	OF,CF	rotire dreapta cu o pozitie.
RCR,s,cl	rotire dreapta cu o pozitie.	OF,CF ; Daca CL=1, OF este nedefinit	rotire dreapta cu o pozitie.

În schimb, se permite utilizarea registrului CL pentru a indica numărul de poziții cu care se dorește deplasarea, ceea ce face ca secvența de mai sus să fie echivalentă cu

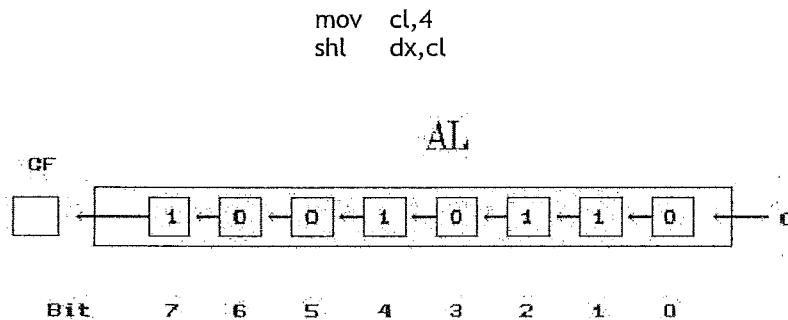


Fig. 4.1. Instrucțiunea SHL

Subliniem că variantele de multiplicare cu puteri ale lui 2 folosind SHL sunt mult mai rapide decât variantele echivalente folosind instrucțiunea MUL.

În acest sens, să urmărim secvența următoare care realizează înmulțirea cu 10 a valorii octetului *var*, fără a folosi instrucțiuni MUL sau IMUL:

```

    xor ah,ah
    mov cl,3
    mov al,var
    shl ax,cl      ;înmulțire cu opt
    add al,var
    adc ah,0        ;al = var * 9
    add al,var
    adc ah,0        ;al = var * 10
  
```

În ceea ce privește instrucțiunile de deplasare spre dreapta, presupunând că AL conține valoarea 10010110b (= 96h = -106), **shr al,1** produce 01001011b (= 04Bh = 75), iar **sar al,1** rezultă în 11001011b (= 0CBh = -53), având deci ca efect păstrarea semnului operandului. Acest lucru face ca instrucțiunea SAR să fie indicată pentru efectuarea împărțirilor cu semn la puteri ale lui 2. Spre exemplu,

```

    mov bx,-4
    sar bx,1
  
```

are ca rezultat memorarea valorii -2 în BX.

Figura 4.2. arată acțiunea instrucțiunii ROR AL,1 asupra valorii 10010110b. Rezultatul este valoarea 01001011b, în CF depunându-se valoarea inițială a ultimului bit, adică 0.

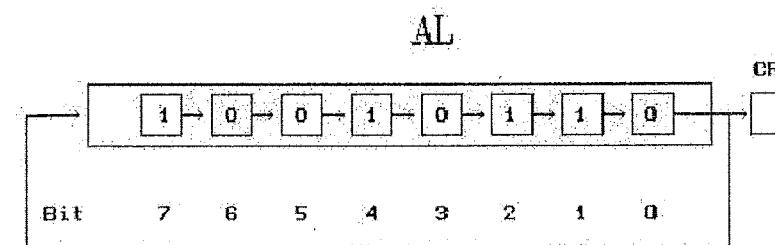


Fig. 4.2. Instrucțiunea ROR

Instrucțiunile **ROR** și **ROL** devin utile în probleme privind realinierea biților în cadrul unui octet sau cuvânt. De exemplu, secvența

```

    mov si,49f1h
    mov cl,4
    ror si,cl
  
```

face ca în SI să se afle în final 149Fh, mutând biții 0-3 în biții 12-15, biții 4-7 în biții 0-3, și.a.m.d.

Figura 4.3 arată rotirea spre dreapta a valorii 10010110b (= 96h = 150) din AL, participând și CF (care inițial conține valoarea 1). Instrucțiunea **rcr al,1** produce rezultatul 11001011b (= 0CBh = 203) care este depus în AL. CF va conține valoarea de dinainte de rotire a bitului cel mai puțin semnificativ, adică 0.

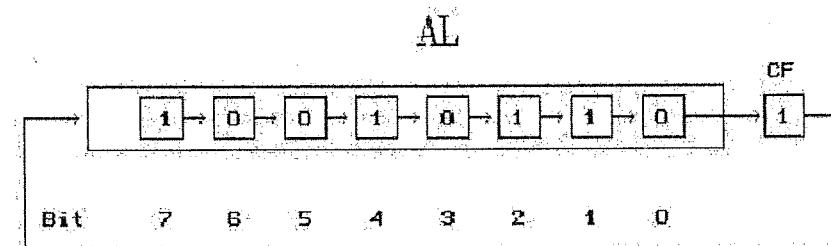


Fig. 4.3. Instrucțiunea RCR

dupa CALL).

In acesta categoria mită instrucțiunile JMP (echivalentul instrucțiunii GOTO din alte limbaje), CALL (apelul de procedură apelată) și RET (transfer control din punctul apelului la prima instrucțiune din procedură apelată) și RET (transfer control din punctul apelului la prima instrucțiune de memorie. De exemplu:

După cum am menționat, salutul poate fi facut și la o adresă memorată într-un registru sau într-o variabilă de memorie. De exemplu:

„spree happy” ca saluturi „spree” dacă este posibil (din cauză că a treceut deasă de locul unde se află „fără”).

Operatiunile SHORT impună folosirea unei adrese pe 8 biti, adresa „scurtă”, economisindu-se astfel l ocetă în reprezentarea instrucțiunii JMP. Această operarează și justifică utilizarea de către programator numai în cazul saluturilor „spree fizante”, deoarece asamblorul translatăză automat toate saluturile

### jmp SHORT PTR eticheta

există posibilitatea sănătății „salt scurt”. Un salt scurt se specifică sub forma „mod normal” instrucțiunea JMP efectuează un salt NEAR (salt în cadrul același segment). Pentru a reduce codul generat, în cauză în care destinația salutului nu este mai departe de 127 octetii, disconțum la nivel scăzut, sau îmbasează dacă discutăm la nivelul său (adică în cadrul același segment). În mod normal instrucțiunile JMP efectuează un salt NEAR (salt în cadrul același segment).

Adunătorii nu se vor executa, decât dacă se va face salt la Adunătorii de altundeva din program.

```
Adunătorii:    add    ax,2
                jmp   urmare
Adunătorii:    inc    ax
                jmp   Adunătorii
mov    ax,1
```

Înregistrul AX va conține valoarea 3. Instrucțiunile înc si jmp dintre etichetele Adunătorii și se conțină în registrul, respectiv la adresa conținută în variabilă de memorie. De exemplu, după execuția

### JMP operand

Instrucțiunea de salt necondiționat JMP are sintaxa

### 4.3.1.1. Instrucțiunea JMP

-	RET [n]	Transferă controlul instrucțiunii de după CALL
-	CALL operand	Transferă controlul proiectat de operand
-	JMP operand	Salt necondiționat la adresa determinată de operand

Cap.4. Instrucțiunile limbajului de assembleare.

în cursul executiei unui program, adresa următoarei instrucțiuni de executat este indicată microprocessorului. La o execuție sevenită CS și IP sunt modificăate automat de componenta BIU a unită de controlă (saluturi), prevedând și instrucțiuni care facilitează prelucrarea repetată a unei blocuri de controlă (controlă 80x86 disponibilă pentru a executa unele instrucțiuni speciale realizările transferelor de date între memoriile celor două unități de memorie).

O proprietate de bază pe care trebuie să o ia în acelaș context orice calculator (microprocessor dacă discutăm la nivel scăzut, sau îmbasează dacă discutăm la nivelul său) este posibilitatea de a transfera controlul la un altă instrucțiune, altă deosebită specifică seconfului celei curente. Microprocessorul 80x86 realizează transferul de date între memoriile celor două unități de memorie. Pentru a reduce codul generat, în cauză în care destinația salutului nu este mai departe de 127 octetii, disconțum la nivel scăzut, sau îmbasează dacă discutăm la nivelul său (adică în cadrul același segment). În mod normal instrucțiunile JMP efectuează un salt NEAR (salt în cadrul același segment).

În același scop sănătății structurii de control (if, goto, for, while etc.), care determină ordinea de execuție a instrucțiunilor unui program.

Un program scris într-un limbaj de programare se execută seconful, adică instrucțiunile de rezolvare „truperei” accesării ordinii, pentru a putea soluționa anumite cerințe specifice unei probleme de rezolvare. În îmbasele de programare de nivel înalt cunoscuțe (Pascal, C, FORTRAN etc.) există nevoie să se execute instrucțiunea de ordină în care este săt scrisă. Apără în mod natural programul să se execute în ordinea săt scrisă. În cadrul același segment, adică instrucțiunile de rezolvare „truperei” accesării ordinii, pentru a putea soluționa anumite cerințe specifice unei probleme de rezolvare. În cauză utilității de direcție ce permit lucru cu instrucțiunile specifice adăugate

În general, un set de n instrucțiuni de rotire (cu sau fară carry) cu căte o pozitie este mai rapidă decât o singură instrucțiune de rotire ce are cu  $crl = n$ .

rcr dx,1	;valoarea din CF se depune în bitul 0 din DX
shl dx,1	;valoarea din CF se depune în bitul 15 din AX
rcf ax,1	;bitul 15 din AX este depus în CF

Instrucțiunile RCR și RCL sunt ușor de depășită de către o valoare din 4 valoare din reprezentări pe mai multe cuvinte. De exemplu, se conține ună multă de 4 valoare din DX:AX:

### 4.3.1. Saltul necondiționat

În cursul următoarei instrucțiuni de executat este indicată instrucțiunile (cliclu).

Microprocessorul 80x86 disponibil și instrucțiuni care facilitează prelucrarea repetată a unei blocuri de controlă (saluturi), prevedând și instrucțiuni care facilitează transferurile de date între memoriile celor două unități de memorie.

În același scop sănătății structurii de control (if, goto, for, while etc.), care determină ordinea de execuție a instrucțiunilor unui program.

### 4.3. RAMIFICARI, SALUTURI, CICLURI

Practică. În cauză utilității de direcție ce permit lucru cu instrucțiunile specifice adăugate (punctul oricăre instrucțiune de depășire sau rotire). Totuși, există restricții ca accesul valoarei să nu depășească 31.

Practică. În cauză utilității de direcție ce permit lucru cu instrucțiunile specifice adăugate necesită mai puțină memoria decât o singură instrucțiune de rotire ce are cu  $crl = n$ .

rcr dx,1	;valoarea din CF se depune în bitul 0 din DX
shl dx,1	;valoarea din CF se depune în bitul 15 din AX
rcf ax,1	;bitul 15 din AX este depus în CF

Instrucțiunile RCR și RCL sunt ușor de depășită de către o valoare din 4 valoare din DX:AX:

(1) mov ax, OFFSET etich  
          jmp ax       ;operand registru  
                      etich:

```

data segment
Salt DW Dest ;Salt := offset Dest
.
.
cod segment
.
.
jmp Salt ;salt NEAR
.
.
Dest : . .
.
.
```

Să remarcăm faptul că esențial pentru exemplele date este prezentarea modalității în care offset-ul (deplasamentul) adresei destinație este permis a fi încărcat în operandul instrucțiunii JMP: în cazul (1) e vorba de o încărcare explicită prin mov a unui registru, în cazul (2) este vorba despre declararea cu initializare a variabilei de tip cuvânt *Salt*, în cadrul căreia se realizează inițializarea variabilei *Salt* cu offset-ul etichetei *Dest* (1 cuvânt de memorie = 16 biți = 2 octeți = dimensiunea de reprezentare a deplasamentului). Dacă în cazul (1) dorim înlocuirea operandului destinație regisztr cu un operand destinație variabilă de memorie, o soluție posibilă este:

```
(1')      b  dw  ?
          . . .
        mov b, offset etich
        jmp b           ; salt NEAR – operand variabilă de memorie
```

Instrucția JMP poate fi folosită de asemenea pentru saltul într-un alt segment de cod (salt FAR - far jump). Pentru aceasta este necesară determinarea unei adrese de forma *segment:offset*.

Saltul FAR se poate realiza în patru moduri:

a). declarând eticheta destinație ca etichetă far prin directiva LABEL (vezi 3.3.3). De exemplu, instrucțiunea JMP de mai jos realizează un astfel de salt:

```
Cseg1 SEGMENT
    ASSUME cs:Cseg1
    .
    .
    Dest LABEL FAR
    .
    .
Cseg1 ENDS
    .
    .
Cseg2 SEGMENT
    ASSUME cs:Cseg2
    .
    .
    jmp Dest ;salt FAR la adresa Dest în segmentul Cseg1
    .
    .
Cseg2 ENDS
```

#### Cap.4. Instructiuni ale limbajului de asamblare.

b). specificând direct în instrucțiunea JMP tipul FAR PTR pentru eticheta destinație, sub forma: JMP FAR PTR *eticheta*. Acest mod de specificare al operandului destinație a fost utilizat în exemplele din 3.3.1.2 - directiva ASSUME și gestiunea segmentelor. Operatorul PTR a fost prezentat în 3.2.2.7.

c). prefixând explicit cu un registru de segment o adresă NEAR specificată indirect (vezi 3.2.1.4 – operanți cu adresare indirectă), adică furnizarea ca operand al instrucției JMP a unei expresii de forma: *reg\_segment: specificare\_offset*. Exemplu: jmp es:[bx+di].

d). specificând ca operand o variabilă dublucuvânt care conține adresa far a destinației:

```
data segment
      Salt    DD Dest ;Salt:= segm:offset (Dest)
      .
      .

cod segment
      .
      .
      jmp Salt        ;salt FAR la eticheta Dest
      .
      .

code1 segment

Dest : . . .
```

secventă echivalentă ca efect cu

```

data segment
    Salt DD ?

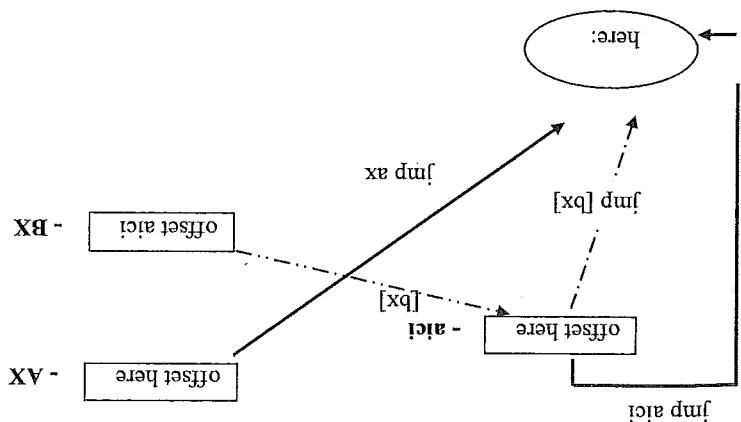
cod segment
    .
    .
    lea ax, Dest      ;ax:=offset(Dest)

    mov word ptr Salt, ax ;initializare cuvânt inferior al
                          ;variabilei dublucuvânt Salt cu offset(Dest)
                          ;— se ține cont de reprezentarea little-endian

    mov ax, cod
    mov word ptr Salt+2, ax ;initializare cuvânt superior al
                            ;variabilei dublucuvânt Salt (Salt+2 –
                            ;aritmetică de pointeri) cu seg(Dest)
    jmp Salt             ;salt FAR la adresa Dest

code1 segment
Dest : . .

```



jmp [ax] ;eroare de sintaxă: „Illegal indexing mode”  
 jmp bx ;salt (near) la adresa conținută în BX (adresare register în mod direct)  
 jmp ax ;salt la adresa conținută în AX (adresare register în mod direct), adică la here  
 jmp here ;salt la adresa lui here (sau, echivalent, salt la eticheta here)

Fig. 4.5. Modificări alternative de efectuare a saluturilor la eticheta here.

jmp [bx] ;eticheta destinație a saluturilor  
 here:  
 jmp ax ;salt la adresa conținută în AX (adresare register în mod direct)

Fig. 4.4. Inițializarea variabilei aici și a registerelor AX și BX.

jmp here ;salt la adresa lui here (sau, echivalent, salt la eticheta here)

;here), deci înstărcările echivalente cu jmp here  
 jmp aici ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui  
 ;accesată înlocuirea variabilei aici în cadrul segmentului dată (vezi mai sus cum a fost  
 ;incarcat BX) – deci corecta simetric, o astfel de înstărcare e o posibilă eroare logică în contextul  
 ;în BX avem ca valoare offset-ul lui aici în cadrul segmentului dată (vezi mai sus cum a fost  
 ;determinat însă relativ la segmentul dată);  
 ;accesul exemplu, deoarece se efectuează un salt în interiorul segmentului codă la un offset  
 ;în memoria care reprezintă valoarea de la o adresă (conformită de datele situației în  
 ;memorie care reprezintă diferența dintre operațiunii din memoria care reprezintă adrese și operațiunii  
 ;acastă exemplu ilustrază diferența dintre operațiunii din memoria care reprezintă adrese și operațiunii  
 ;interpretate la adresa, dacă să fi utilizat de exemplu sub forma mov dx, aici+2). Încheltele coduri  
 ;nume a desemnat conformită de la adresa locație de memorie (eticheta de date aici ar fi fost  
 ;nositu variabilă aici) a fost utilizată de către reprezintă cel mai frecvent valoarea de la adresa adresă. În cazul  
 ;variabila care reprezintă valoarea de la adresa (conformită de date situație în care a apărut, acest  
 ;în memoria care reprezintă valoarea de la adresa adresă). Numele de  
 ;dintr-o varabilă din segmentul de date reprezintă cel mai probabilă eroare logică în contextul

;în BX se află adresa variabilei aici, deci se accesează conformită accesului variabilei. În acestă locație  
 ;de memorie se găsește offset-ul etichetei here, deci se va efectua salutul la adresa here - ca  
 ;un salut la adresa conținută în locația de memorie a cărei adresa este conformită în

;acastă exemplu  
 ;BX (adresare registru în mod direct - singura situație de apel indirect din  
 ;salt la adresa conținută în AX (adresare registru în mod direct), adică la here

jmp ax ;salt la adresa conținută în AX (adresare registru în mod direct), adică la here

jmp here ;salt la adresa lui here (sau, echivalent, salt la eticheta here)

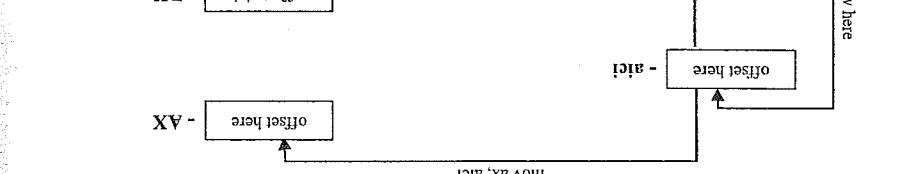
;here), deci înstărcările echivalente cu jmp here  
 jmp aici ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui

;calcul și ca urmare accesă raport este semnalată că era corectă de sintaxă;

;bază) și SI sau DI (ca registri de index). Deci AX nu poate fi parte a unei astfel de formule de  
 ;specificație a unui depășasment (offset) sau să spargă dor registri BX sau BP (ca registri de  
 ;drepăt adresa de memorie al cărei conținut va fi accesat, hînsă conform 2.6.7, §i 3.2.1.4, în formula de  
 ;justificare: orice referire de tipul „[reg]” înseamnă interpretarea conformită regisistrului specificat

;în BX, și I înlocuște variabila aici și a registerelor AX și BX.

Fig. 4.4. Inițializarea variabilei aici și a registerelor AX și BX.



mov bx,OFFSET aici ;se încarcă în BX depășasmentul lui aici în cadrul segmentului dată  
 mov ax,offset here ;here în cadrul segmentului dată (adică depășasmentul lui  
 mov ax,aici ;se încarcă în AX conținutul variabilei aici (adică depășasmentul lui  
 data ends ;echivalent cu aici := offsetul etichetei here din segmentul de cod

Exemplu 4.3.12. Prezentăm în continuare un exemplu edificator pentru modul de transfer al  
 controlului la o etichetă, purtând în evidență deosebita dintr-un transfer direct și unul indirecț.  
 data aici DW here ;echivalent cu aici := offsetul etichetei here din segmentul de cod  
 data ends ;echivalent cu aici := offsetul etichetei here din segmentul de cod

controlului la o etichetă, purtând în evidență deosebita dintr-un transfer direct și unul indirecț.

### 4.3.2. Instrucțiuni de salt condiționat

O caracteristică necesară pe care trebuie să o prezinte orice limbaj de programare este posibilitatea de a lăsa decizii, ceea ce în plan concret revine la existența unor instrucțiuni de salt condiționat. Elementele care condiționează acest tip de salt sunt valorile flagurilor și conținutul registrului CX.

#### 4.3.2.1. Comparări între operanzi

<b>CMP</b> <i>d,s</i>	comparație valori operanzi (nu modifică operanzi) (execuție fictivă <i>d - s</i> )	OF,SF,ZF,AF,PF și CF
<b>TEST</b> <i>d,s</i>	execuție fictivă <i>d AND s</i>	OF = 0, CF = 0 SF,ZF,PF - modificări AF - nedefinit

Operanții instrucțiunii **CMP** pot fi de dimensiune octet sau cuvânt, fiind supuși acelorași restricții de asociere ca și în cazul instrucțiunii **MOV**.

**CMP** scade valoarea operandului șură din operandul destinație, dar spre deosebire de instrucțiunea **SUB**, rezultatul nu este reținut, el neafectând nici una din valorile inițiale ale operanților. Efectul acestei instrucțiuni constă numai în modificarea valorii unor flaguri. Instrucțiunea **CMP** este cel mai des folosită în combinație cu instrucțiuni de salt condiționat.

Efectele execuției instrucțiunii **CMP** pot fi utilizate de exemplu atunci când dorim să facem comparații matematice. Acestea se pot face *cu semn* sau *fără semn*, în funcție de configurațiile de biți furnizate de către programator ca operanzi.

Instrucțiunile de salt condiționat se folosesc de obicei în combinație cu instrucțiuni de comparare. De aceea, semnificațiile instrucțiunilor de salt rezultă din semnificația operanților unei instrucțiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucțiune **CMP** este de multe ori necesară determinarea relației de ordine dintre două valori. De exemplu, se pune întrebarea: numărul 11111111b (= FFh = 255 = -1) este mai mare decât 00000000b (= 0h = 0)? Răspunsul poate fi și da și nu! Dacă cele două numere sunt considerate *fără semn*, atunci primul are valoarea 255 și este evident mai mare decât 0. Dacă însă cele două numere sunt considerate *cu semn*, atunci primul are valoarea -1 și este mai mic decât 0.

Este valabil și aici ceea ce am subliniat la prezentarea instrucțiunilor de adunare și scădere: interpretarea operanților și a rezultatului comparației cu sau *fără semn* este la latitudinea programatorului! Cum se poate face concret însă această distincție și cum se poate ea provoca la nivelul unui program?

Instrucțiunea **CMP** nu face distincție între cele două situații, deoarece astăzi după cum am precizat și în 4.2.1.1. adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații

### Cap.4. Instrucțiuni ale limbajului de asamblare.

binare) indiferent de semnul (interpretarea) acestor configurații. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn *operanții* scăderii fictive *d-s*, ci *rezultatul* final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine diverselor instrucțiuni de salt condiționat prezentate în secțiunea următoare (4.3.2.2) care vor prezenta categorii diferite de instrucțiuni pentru cele două tipuri posibile de interpretări.

Din semnificația flagurilor rezultă că pentru determinarea relației de ordine între operanți, după execuția instrucțiunii **CMP** sunt semnificative doar OF, SF, ZF și CF. Această problemă este detaliată în secțiunea următoare, unde vom vedea că instrucțiunile de salt condiționat diferă tocmai prin condiția asupra flag-urilor care este verificată de către fiecare instrucțiune în parte.

Instrucțiunea **TEST** este utilă pentru examinarea stării anumitor biți. În exemplul de mai jos se transferă controlul la eticheta **Acolo** dacă biții 1 și 5 ai registrului AL sunt 0. Starea celorlalți biți este ignorată:

```
test    al, 00100010b ;mascare prin AND
jz     Acolo          ;salt dacă s-a obținut 0
Nuezero: . .
Acolo: . .
```

#### 4.3.2.2. Salturi conditionate de flaguri

În tabelul 4.1. prezentăm instrucțiunile de salt condiționat împreună cu semnificația lor și cu precizarea valorilor flagurilor în urma căror se execută salturile respective. Precizăm că pentru toate instrucțiunile de salt sintaxa este aceeași, și anume

*<instrucțiune\_de\_salt> etichetă*

Semnificația instrucțiunilor de salt condiționat este dată sub forma "*salt dacă operand1 <>relație> față de operand2*" (unde cei doi operanți sunt obiectul unei instrucțiuni anterioare **CMP** sau **SUB**) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă și din condițiile ce trebuie verificate, instrucțiunile ce se află într-o aceeași linie a tabelului sunt echivalente.

Când se compară două numere *cu semn* se folosesc termenii "*less than*" (mai mic decât) și "*greater than*" (mai mare decât), iar când se compară două numere *fără semn* se folosesc termenii "*below*" (inferior, sub) și respectiv "*above*" (superior, deasupra, peste).

Instrucțiunile de salt condiționat efectuează întotdeauna salturi "scurte", adică adresa de destinație nu poate fi la distanță mai mare de 127 octeți față de instrucțiunea curentă. De exemplu, în urma asamblării sevenței

Cap.4. Înstrucțiunile limbajului de asamblare.

Tableau 4.2. Allegere correta a instruçõe similar de compararre in funciõe de relaçõe testata.

Cognitiv și acordat acest tabel cu tabelul 4.1 se poate deduce modul de pozitivare a flagului de către instituții menite CMI. Studiul acestui tabel reiesează afirmația noastră anterioră: nu instituții menite să facă diferenție între o comparativă cu semn și una fără semn! Rolul de interpretator în mod diferit (cu semn sau fără semn) rezultă din comparație cu semn și rolul de sat condiționat specificate ULTERIOR comparativi efectuate.

$d = s$	JE	JE	JNE	$d \neq s$	$d < s$	$d > s$	$d \leq s$	$d \geq s$	$d \leq s$
			JNE	JNE	JNE	JNE	JLE	JGE	JAE
					JL	JG	JLE	JGE	JBE
					JB	JA			
					JA				

folosindu-se deci o instanță de salt conditională pentru a lăsa decizia dacă trebuie său nu facă un salt necondiționat la eticheta Aici (un salt necondiționat nu este supus nici unei restricții referitoare la distanța de salt).

Tableau 4.1. Instructionnelle de salt conditionat

MEMONICA	SEMANTICATIE (sau data, <<relative>>)	Conditia verității
JBE	nu este superior sau egal	CF=1
JNC	nu este inferior	CF=0
JAE	este superior sau egal	CF=1
JNB	nu există transport	CF=0
JNC	nu există transport	CF=0
JBE	nu este superior	CF=1 sau ZF=1
JA	nu este inferior sau egal	CF=0 și ZF=0
JNE	nu este egal	ZF=0
JL	nu este mai mic decât	SF=OF
JGE	este mai mare sau egal	SF=OF
JNL	nu este mai mic decât	SF=OF
JLE	este mai mic sau egal	ZF=1 sau SF=OF
JGE	nu este mai mare decât	ZF=0 și SF=OF
JNL	este mai mare sau egal	SF=OF
JLE	nu este mai mic decât	ZF=1 sau SF=OF
JG	este mai mare decât	ZF=0 și SF=OF
JPE	nu are paritatea este pară	PF=1
JPO	nu are paritatea este impară	PF=0
JS	nu are semn negativ	SF=1
JNS	nu are semn negativ	SF=0
JO	există depasire	OF=1
JNO	nu există depasire	OF=0

Tabelul de mai sus îl considerăm foarte util pentru interpretarea rezultatelor instrucțiunilor aritmetice în general. Fără a mai efectua o instrucțiune CMP, considerând *d* rezultatul ultimei instrucțiuni aritmetice executate și punând *s*=0, tabelul rămâne valabil.

#### 4.3.2.3. Exemple comentate

##### Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

mov al,80h	;al := 128 = 10000000b = -128 ! (Interesant! – remarcăm faptul că datorită regulilor de reprezentare în cod complementar 128 și -128 au aceeași reprezentare binară și anume 10000000b!)
(*) cmp al,0	;instrucțiunea cmp nu interpretează în nici un fel valoarea din AL (ca fiind ;cu semn sau fără semn) ci doar realizează scăderea fictivă al-0 și afectează ;corespunzător flagurile: SF=1, CF=ZF=OF=PF=AF=0.
jl et	;utilizarea instrucțiunii JL (Jump if Less than) provoacă interpretarea ;comparației al<0 <u>cu semn</u> (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă SF≠OF și cum SF=1 iar OF=0 se decide îndeplinirea condiției ;și saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a ;stat la latitudinea programatorului care prin utilizarea instrucțiunii JL a ;decis că dorește să compare -128 cu 0 și cum -128 este “less than” 0 ;condiția a fost îndeplinită (echiv. cu jnge et). În contrast, jnl et sau jge et ;(care vor testa dacă SF=OF) NU vor fi îndeplinite și NU vor provoca saltul ;la eticheta specificată.
jb et	;utilizarea instrucțiunii JB (Jump if Below) provoacă interpretarea ;comparației al<0 <u>fără semn</u> (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă CF=1 și cum CF=0 se decide neîndeplinirea condiției deci nu ;se va face saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JB a decis că dorește să compare 128 cu 0 și cum 128 NU este “below” 0 condiția NU a fost îndeplinită (echivalent cu jnae et sau jc et).
jae et1	;se testează <u>fără semn</u> dacă al $\geq$ 0 (128 $\geq$ 0?) - CF=0 deci condiție ;îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la ;eticheta et1
jbe et2	;se testează <u>fără semn</u> dacă al $\leq$ 0 (128 $\leq$ 0?) – CF = ZF = 0 deci condiția ;(CF=1 sau ZF=1) NU este îndeplinită și ca urmare nu se va face saltul la ;eticheta et2 – rezultat consistent cu jb et, deoarece jbe implică jb ;(echivalent cu jna et2)

ja et3	;se testează <u>fără semn</u> dacă al > 0 (128 > 0?) – CF = ZF = 0 deci condiția ;(CF=0 și ZF=0) este îndeplinită și ca urmare se va face saltul la eticheta et3 ;(echivalent cu jnbe et3) și rezultat consistent cu jbe et2, deoarece dacă ;jbe nu este îndeplinită atunci ja trebuie să fie.
je et4	;se testează dacă al = 0 (128 = 0 ?) – nu se pune problema semnului dacă se ;testează egalitatea! – cum ZF=0, condiția ZF=1 nu este îndeplinită deci nu ;se va efectua saltul la eticheta et4 (echivalent cu jz et4). În contrast, jne ;et4 sau jnz et4 (care vor testa dacă ZF=1) vor fi îndeplinite și vor provoca ;saltul la eticheta specificată.
jle et5	;se testează <u>cu semn</u> dacă al $\leq$ 0 (-128 $\leq$ 0?) – OF = ZF = 0 și SF=1 deci ;condiția (ZF=1 sau SF≠OF) este îndeplinită și ca urmare se va face saltul la ;eticheta et5 (echivalent cu jng et5) și rezultat consistent cu jl et, deoarece ;jle implică jl.
jg et6	;se testează <u>cu semn</u> dacă al > 0 (-128 > 0?) – OF = ZF = 0 și SF=1 deci ;condiția (ZF=0 și SF=OF) NU este îndeplinită și ca urmare NU se va face ;saltul la eticheta et6 (echivalent cu jnle et6) și rezultat consistent cu jle ;et5, deoarece dacă jg nu este îndeplinită atunci jle trebuie să fie.
jp et7	;se testează dacă PF=1 - PF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (echivalent cu jpe et7 – Jump if Parity Even). În contrast, jnp et7 ;(care testează dacă PF=0 – echivalentă cu jpo et7 – Jump if Parity Odd) va ;fi îndeplinită și saltul se va efectua.
jo et8	;se testează dacă OF=1 - OF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (nu există depășire). În contrast, jno et8 (care testează dacă OF=0) va ;fi îndeplinită și saltul se va efectua.
js et9	;se testează dacă în interpretarea <u>cu semn rezultatul</u> comparației are semn ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucțiunii ;CMP, nu este vorba de a interpreta cu semn sau fără semn <u>operanții</u> ;scăderii fictive <i>d-s</i> , ci <u>rezultatul</u> final al acesteia !) adică testăm dacă SF=1 - ;condiție îndeplinită în cazul nostru și ca urmare saltul se va efectua !. În ;contrast, jns et9 (care testează dacă SF=0) NU va fi îndeplinită și saltul ;NU se va efectua.
cmp 0,al	;eroare de sintaxă : “Illegal immediate” deoarece sintaxa instrucțiunii cmp ;interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuși dorim forțarea unei comparații de acest tip (0-al) putem utiliza ;pe post de prim operand un registru inițializat cu valoarea 0.
mov bl,0	

De exemplu, într-un decifer, dacă vom considera  $100 + 30 = 130$  vom obține așa ceva (figura 1001100b) = 150 (= 96h = 1001100b). Justificare: 100 (= 64h = 01100100b) + 50 (= 32h = 0011000b) = 150 (= 96h = 1001100b), unde 100 și 50 sunt paralele la 100 și 30, respectiv la 150 și 64. Întrucât, de către form RDAs vom avea OF=1, Operațiunile au același semn dar rezultatul este de semn diferit, deci conformat RDAs vom avea OF=-1. Întruire, deși diferența se poate justifica prin raport cu 150 (= 128,127] deci se obține o eroare de tip "out of range". Deși și astăzi replica fizică la 150 = 1001100b = -106 (în interpretarea cu semn), iar -106 = [-128,127], această ultimă interpretare nu poate fi acceptată deoarece operează cu numere pozitive (în interpretare în semnătatea lui 0). Ca urmare, suma a două numere pozitive nu poate da un număr negativ și astfel imaginării interpretează ce poate fi acceptată în acest context pentru 1001100b este 150 ≠ [-128,127].

Analoga, în interpretarea cu semn, suma a două numere negative nu poate furniza un număr pozitiv.

**Quaternary example:**  
10010110 +  
10000010  
—————  
10001100

Băiețelul său nu poate să se întâlnească cu prietenii săi și să joace împreună. El este singur și triste. În următoarele zile, băiețelul va fi în stare să se simtă foarte slab și să nu mai poată să meargă sau să joace. Va fi nevoie să ia un tratament medical și să rămână acasă pentru o perioadă de timp.

Atagam misă extenziă asupra a ceea ce se ignorează de multe ori în acest context și anume faptul că o situație de tipul CF=1 (cu  $Q_F=0$ ) semnalizează la randul ei o depășire, înăsă pentru ca și numerele

Cum să determină situația de dezvoltare în cază și operațiile de adunare și scadere? Care sunt rezultatele practicei de aplicare pentru a înțelege și să pună judecata corectă searăriile de flag-wi pe care le remarcăm în cadrul programelor rutinale? În discuție ce urmăza ne vom concreta în principiu pe judecătorele modului de setare a flag-ului OF (overflow flag) deoarece și datorei numelei său acesta este principial factor raspunzător de caracterizarea unei situații din partea programatorilor ca fiind deosebit de sănătu.

In cauză operatiilor/operanților care semn deținător va fi semnalată prin setarea imdicatorului CF (carry flag). În cauză operațiilor/operanților cum se poate semnalată prin setarea imdicatorului CF (carry flag).

Ultimile două rețele regăsiti derivă de fapt din modul de implementare a conceptului de **depassire** (overflow) la nivelul procesorului 80x86.

- CF ia valoarea critiei de transport : daca e vorba despre o adunare se analizeaza daca rezultatul obtinut a provocat ( $CF=1$ ) sau nu ( $CF=0$ ) un transport in afara spatiului de reperezentare; daca e vorba despre o scadere -d-s, avem: daca  $|d| \geq |s|$  atunci  $CF=1$  este nevoie de citta de impunut penitul efectuarea scaderii) iar daca  $|d| < |s|$  atunci  $CF=0$  (nu e nevoie de citta de impunut penitul efectuarea scaderii) si acast lucru se reflecta in CF)
- OF este setul la valoarea 1 daca exista deparamentul pentru caracterarea scaderii si acast lucru se reflecta in CF)
- is set of three exists a signed overflow, "adică dacă rezultatul obținut nu se încadrăza în intervalul de interpretare admis (acasta fiind [-128...+127] dacă este vorba despre octetii și respectiv [-32768...+32767] pentru cuvinte interpretate cu semn).

Pentru a justifica modurile de setare diferențiale ale flag-urilor trebuie să lăsăm în discuție regrile practice de setare a acestor flag-uri. Aceste regule generale sunt:

Care ar fi măsă justificarea raportului că în casuță comp bl, al avem CF = OF = SF = 1 iar în casuță comp al doar SF=1 iar CF = OF = 0 ?

**Exercitiu 10.10** Rezolvă diferența discutată efectuând un exercițiu de mai sus (analizarea penituu) și un exercițiu de mai jos (analizarea rezultatelor unei comparații).

cmp bl, al ;realiza la scadre de flciva  
 ;10000000b) si efectua bl-al = 0-80h = 0-10000000b =  
 ;ZF=PF=Af=0.

Nu putem avea deci  $-106 + (-126) = 24$ ! (pentru că  $00011000b = 18h = 24$  în ambele interpretări) Acesta este sensul în care se aplică RDA aici. Un alt mod de justificare intuitivă a depășirii în acest tip de situație este:

În interpretarea cu semn avem  $-106 + (-126) = -232 \notin [-128..127]$  deci OF=1.

Această ultimă motivație este mai intuitivă pentru justificarea depășirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne "cea mai rapid aplicabilă regulă practică din punct de vedere algoritmic" dacă ne putem exprima așa... (și iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depășire. De asemenea, dacă adunăm două numere de același semn dar rezultatul are același semn cu operanții nu se va semnala nici în acest caz depășire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeași dimensiune ca și cea a operanților). Se poate verifica ușor din punct de vedere matematic că în nici unul din aceste cazuri nu ieșim din intervalul de interpretare admis.

**Pentru SCĂDERE:** se interprează operanții respectiv cu semn, se efectuează scăderea solicitată asupra configurațiilor corespunzătoare de biți și dacă rezultatul obținut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul [-128..127] pentru octeții cu semn și respectiv [-32768..32767] pentru cuvinte interpretate cu semn) atunci se semnalează depășire (*overflow*) și astfel OF=1. Această formulare o putem numi *regula depășirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depășirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea CF=1, pe care o putem interpreta semantic drept "depășire la scădere în interpretarea fără semn".

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum și impactul lor asupra modului de setare al flag-urilor.

Exemple:

- mov ah,82h ; $82h = 130$  (interpretarea fără semn) =  $-126$  (interpretarea cu semn)  
; =  $10000010b$  (bitul de semn fiind 1 cele două interpretări diferă)  
mov bh,2ah ; $2ah = 42$  (atât în interpretarea cu semn cât și în cea fără semn)  
; =  $00101010b$  (bitul de semn fiind 0 cele două interpretări coincid)  
cmp ah,bh ;se realizează scăderea fictivă  $ah-bh = 10000010b - 00101010b = 01011000b$   
; =  $58h = 88$  (atât în interpretarea cu semn cât și în cea fără semn deoarece bitul de semn este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul  $58h = 01011000b$  este 0)

CF = 0 (deoarece  $|82h| > |2ah|$  nu se pune problema unei scăderi cu împrumut de cifră; deci nu vom avea depășire în interpretarea fără semn care se efectuează:  $130 - 42 = 88$ )  
OF = 1 (se efectuează scăderea în interpretarea cu semn, adică  $ah-bh = -126 - 42 = -168$  și cum  $-168 \notin [-128..127]$  se semnalează *signed overflow* și ca urmare OF=1)

cmp bh,ah ;se realizează scăderea fictivă  $bh-ah = 00101010b - 10000010b = 10101000b$   
; =  $A8h = 168$  (în interpretarea fără semn) = -88 (în interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul  $A8h = 10101000b$  este 1)

CF = 1 (deoarece  $|2ah| < |82h|$  se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine  $42 - 130 = 168$  (!) provenită de fapt din necesitatea unei scăderi de tipul  $(256 + 42) - 130 = 168$  și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn, înțeleasă aici ca "nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut")

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică  $bh-ah = 42 - (-126) = +168$  și cum  $+168 \notin [-128..127]$  se semnalează *signed overflow* și ca urmare OF=1)

- ii. mov ah,126 ;echivalent cu mov ah,7eh deoarece  $126 = 7Eh = 01111110b$  (bitul de semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este 126 atât în interpretarea cu semn cât și în cea fără semn)  
mov bh,2ah ; $2ah = 42$  (atât în interpretarea cu semn cât și în cea fără semn)  
; =  $00101010b$  (bitul de semn fiind 0 cele două interpretări coincid)  
cmp ah,bh ;se realizează scăderea fictivă  $ah-bh = 01111110b - 00101010b = 01010100b$   
; =  $54h = 84 = 126 - 42$  (atât în interpretarea cu semn cât și în cea fără semn deoarece bitul de semn al rezultatului este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul  $54h = 01010100b$  este 0)

CF = 0 (deoarece  $|126| > |42|$  nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnala depășire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică  $ah-bh = 126 - 42 = 84$  și cum  $84 \in [-128..127]$  NU se semnalează *signed overflow* și ca urmare OF=0)

cmp bh,ah ;se realizează scăderea fictivă  $bh-ah = 00101010b - 01111110b = 10101100b$   
; =  $42 - 126 = ACh = 172$  (în interpretarea fără semn) = -84 (în interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul  $ACh = 10101100b$  este 1)

CF = 1 (deoarece  $|42| < |126|$  se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine  $42 - 126 = 172$  (!) provenită de fapt din necesitatea unei

Baștana și-a realizat în cadrul unei expoziții de artă contemporană din 2010, în cadrul festivalului "Caleidoscop" organizat de Muzeul Național de Artă Contemporană din București.

Pe l ocet se pot reprezenta 256 de valori, indiferent ca vorbim despre interpretarea cu semn sau interpretarea fara semn. In interpretarea fara semn aceste valori sunt cele din intervalul [0..255]. Cu toate sunt insa cele 256 de valori reprezentabile in interpretarea cu semn ? Este vorba despre intervalul [-128..127] sau despre intervalul [-127..128] ? Pentru ca nu poate fi vorba despre intervalul [-128..128] deoarece in acest interval sunt doar valori negative si totodata sa fie pozitive. Debutul sa alega una dintre cele doua variante si totodata sa fie pozitive. Debutul sa alega una dintre cele doua variante si totodata sa fie pozitive. Debutul sa alega una dintre cele doua variante si totodata sa fie pozitive. Debutul sa alega una dintre cele doua variante si totodata sa fie pozitive.

În acest sens este de obiceiă să impactul acestui mod de reprezentare asupra limbajelor de nivel mai mult ca pe acela de a observa că în modul de interpretare este într-o formă mai scurtă și mai ușoară de citit.

Ca urmare, s-a lăsat decizia că intervalele să fie acoperite cu semnul reprezentabil pe 1 octet și unele intervale [−128, +127] care este exact domeniul de valori și la tipul de date shortint din Turbo Pascal); deci +128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet!

pentru a obține după cum putem verifica totare user-ului, **128 si mov ah, -128 sunt mandoua acceptate de către asamblator, efectul fiind în ambele cazuri închiderea într-o anumită configurație binară a unității de memorie**. Într-un alt caz, se va obține 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fară semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea fară semn cu o anumită configurație binară și necesitatea imposibilității reprezentării rezpectivă într-un anumit format.

Bara a fi nevoie de o cîrtă de transport înporturi pe la un port, să se efectueze o călătorie cu un vas de la portul de unde se încearcă să se aducă la portul unde se urmărește să se desemneze.

Cf = 1 în cazul cmp u, unde deoarece se efectueaza o scadere cu imprumut de tipul:

$F = 1$  (se efectueaza scaderea în interpretarea cu semn, adică  $b - ah = 0 - (-128)$ )

$CF = 1$  (dearece  $|oh| < |oh|$ ) se pune problema unei scaderi cu impozit de interpretarea fără semn scaderă devine  $0 - 128 = 128$  (!) provinția de răpt din necesitatea de tipul  $(256 + 0) - 128 = 128$  și ca urmare a neexistării împozitului se va scăda de la interpretarea fără semn scaderă  $|oh|$  și că interpretarea fără semn scaderă  $|oh|$  este corectă.

SF = 1 (deoarceea btitl de semn pentru rezultaatul 80h = 10000000b este 1)

se realizaza scaderea fizica  $b-h = 000000000-1000000000 = 10$  cmp  $b,h$

Porte f vorba despre deplasăriile în interpretarea semnului

SF = 1 (deoorice bîtil de semn Pentru rezultatul 80h = 100000000 este 1) CF = 0 (deoorice [80h] < [0] nu se pune problema unei scadere cu împărtut de cîteva

Se realizaza scaderile fictive  $Ah - bh = 100000000b - 000000000b = 100$  cmp  $ah, bh$

mov bh,0 ;bh=0  
;10000000b (bitul de semn fiind 1 cele două interpretari diferențiale)

**cmp a,b** și **cmp b,a** vor furniza în totdeauna același valoare pentru OF.

scaderii a-b, în [-127..127] amici și b-a în [-127..127] (situația particulațiilor în care a-b trămați mai mult decât cuvântul întravaleului [ -32768..32768]. Ca urmare se poate concluziona faptul că într-o discuție asupra pozitiei particulelor de tip cuvântul întravaleului [-32768..32768],

OR = 0 (se efectuarea se va deruta in imprejurata cu somn, adica D1-A1 = 42-126 = -84 E [-128..127] NU se semnalaza siigned overflow si ca urmare O

seceder de tipul  $(256 + 42) - 126 = 172$  și ca urmare a neceșitării impunutării se va depăși în imprestirea prima secera (la găuri cu cată)

154 Arhitectura Calculatoarelor. Limbaul de assembly 80x86.

Pe de altă parte, avem evident în ambele cazuri SF=1. Justificarea *intuitivă*: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea *tehnică*: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

**iv).** Să analizăm în continuare modurile în care putem compara valorile 0 și 1 (și apoi 0 și -1) și ce efecte are asupra flagurilor instrucțiunea cmp în fiecare dintre situații.

Situată **cmp 1,0** (evidențiată la nivelul unui text sursă de exemplu prin cmp ah,0 cu ah=1) va efectua scăderea fictivă  $1-0 = 1 = 00000001b$ . Efectul asupra flag-urilor va fi CF = SF = OF = ZF = PF = AF = 0. Justificările sunt evidente pe baza discuțiilor din exemplele anterioare.

Situată **cmp 0,1** (evidențiată la nivelul unui text sursă de exemplu prin cmp ah,1 cu ah=0) va efectua scăderea fictivă  $0-1 = -1 = 11111111b$ :

$$\begin{array}{r} 0 - 00000001b = 100000000 - \\ \underline{00000001} \\ 01111111 \end{array}$$

Efectul asupra flag-urilor va fi CF = SF = PF = AF = 1 și ZF = OF = 0. Justificarea valorilor din CF și SF este și aici evidentă pe baza discuțiilor din exemplele anterioare iar OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar  $-1 \in [-128..127]$ .

Situată **cmp -1,0** (evidențiată la nivelul unui text sursă de exemplu prin cmp ah,0 cu ah = -1) va efectua scăderea fictivă  $-1-0 = -1 = 11111111b$ . Efectul asupra flag-urilor va fi SF = PF = 1 și CF = OF = ZF = AF = 0. SF=1 deoarece bitul de semn este 1. OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar  $-1 \in [-128..127]$ . CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situată **cmp 0,-1** (evidențiată la nivelul unui text sursă de exemplu prin cmp ah,-1 cu ah = 0) va efectua scăderea fictivă  $0 - (-1) = +1 = 00000001b$ :

$$\begin{array}{r} 0 - 11111111b = 100000000 - \\ \underline{11111111} \\ 00000001 \end{array}$$

Efectul asupra flag-urilor va fi CF = AF = 1 și OF = SF = ZF = PF = 0. SF = 0 deoarece bitul de semn este 0. OF=0 deoarece  $0 - (-1) = +1 \in [-128..127]$ . CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica și așa: în interpretarea fără semn această scădere înseamnă de fapt  $0 - 255 = 1$  (!), care trebuie justificată prin  $(256+0) - 255 = 1$ , deci e nevoie de cifră de împrumut și astfel se semnalează depășire în cazul interpretării fără semn, deci CF = 1.

v). Cazurile studiate anterior (i-iv) s-au referit la operații de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucțiunii cmp. Să analizăm în continuare și cazul unei depășiri furnizate de operația de adunare revenind astfel la discuția asupra aplicării regulii RDA:

```
mov ah,126 ;126 = 01111110b = 7eh (aceeași valoare 126 în ambele interpretări)
add ah, 2    ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b = 7eh + 02h =
            ; 10000000b = 80h (= 128 fără semn = -128 cu semn)

CF = 0 deoarece: 01111110 +
                  00000010
                  10000000 - nu există transport în afara spațiului de reprezentare al rez.
```

SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operației efectuate este strict negativ = -128).

OF = 1 deoarece:

- justificare *tehnică* - conform RDA se adună două numere de același semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).
- justificare *intuitivă* - adunăm două numere fără semn a căror sumă este  $126 + 2 = 128$ . Însă numărul  $+128 \notin [-128..127]$  deci se semnalează *signed overflow* și ca urmare OF=1.

**vi).** Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situația în care programatorul își inițializează operanții cu anumite valori inițiale dorite (cu semn sau fără semn, conform necesităților problemei în cauză) și se așteaptă la obținerea unor rezultate sau reacții în conformitate cu valorile furnizate. Atenție însă! De obicei aceste valori au o dublă interpretare posibilă și nu vor fi interpretate în orice situație sub forma furnizată la inițializare!

Utilizarea ulterioară a unor instrucțiuni care forțează prin modul de lor de acțiune interpretarea complementară (cu semn/fără semn) celei de la inițializare poate provoca apariția unor situații în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare... Aceasta se întâmplă dacă nu se ține cont în permanență de dubla interpretare posibilă a configurațiilor binare manipulate. Să luăm un exemplu:

```
mov al, 200 ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (cu semn)
mov bl, -1   ; bl = 11111111b = 0FFh = 255 (fără semn) = -1 (cu semn)
cmp al, bl   ; al-bl = 11001001b = C9h = -55 (cu semn) = 201 (fără semn)
              ; și se setează corespunzător OF=ZF=0 și CF=SF=1)
```

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 așa cum precizează valorile de la inițializare? Sau poate pe 200 cu 255? Sau pe -56 cu -1? Sau pe -56 cu 255?

Răspuns: comparăm întotdeauna pe 0C8h cu 0FFh sau în exprimare binară pe 11001000 cu 11111111. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparației efectuate în baza 10 nu este

Cap.4. Instrucțiunile limbajului de asamblare. 159

urire secrete la valoarea 1 cu semnificația de „deținute la imobilă” în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biți:

**mul bl** ;AX := AL \* BL = 5 \* 170 = 850 = 0352h si vom avea CF=1 și OF=1  
;deoarece octetul superior AH = 03, ≠ 0.

```

        mov bl,1/U ;J/U = Uah ;BL = 5 * (-86) = -430 = 0f52h și vom avea CF=1 și
        imul bl ;AX = AL * BL = 5 * (-86) = -430 = 0f52h și vom avea AH = 0feh ≠ 0.
        ;OF=1 deoarece octetul superior AH = 0feh ≠ 0.

```

mul val2 .;DX:AX = 00200000h și vom avea CF=1 și OF=1 deoarece summa este  
superioră la producături DX:AX, adică reșiduul DX conține valoarea 0020h ≠ 0.

In cauză imobilării cu semn (instituțiiunile MUD), instituția care este similară:  $127/ * 127/ = 16129 < 32767$  (numărul maximul cu semn ce poate fi reprezentat pe 1 cuvânt), iar  $32767 * 32767 = 1073676289 < 2147483647$  (numărul maximul cu semn reprezentabil pe un dublucuvânt).

Depsătirea în cauză lămâului în la nivelul limbajului de operanți ocașie este doar o semnala reacție (respective într-un curvăt) ci este realemente nevoie de o dimensiune dublă pentru a face tot într-un rezultatul. În acest sens, a se vedea și captiolul I, în care din punct de vedere matematică specifică clar că lămâuirea nu provoacă de fapt despră, tocmai din cauza locurii nici spațiu-materiale.

Distribuitor în imunitate. Instanță judecătoare MUL și IMLU. Se cerea  $CH=1$  și  $QH=1$  dacă jumătatea superioară a produsului (ocultă) supără deosebită daca este vorba despre produs-cuvant sau cuvantul superioară din imunitate. Instanță judecătoare MUL și IMLU. Se cerea  $CH=1$  și  $QH=1$  dacă jumătatea superioară a produsului (ocultă) supără deosebită daca este vorba despre produs-cuvant sau cuvantul superioară din imunitate. În cazăul arhitecturii  $80 \times 86$ , să remarcăm faptul că nu se face distincție între MUL și IMLU și de aceea nici într-o CF și OF. Oî vor fi amândouă flag-  
de mărită nouănumi de „depărtare la imunitate” în cazăul arhitecturii  $80 \times 86$ . Aceasta este superioară daca este vorba despre produs-dublu-cuvant), este o valoare diferențială de zero. Aceasta este definită nouănumi de „depărtare la imunitate” în cazăul arhitecturii  $80 \times 86$ . Să remarcăm faptul că nu se face distincție între MUL și IMLU și de aceea nici într-o CF și OF. Oî vor fi amândouă flag-

Am studiat în exemplul anterioare modalitatea de reacție (de interprétere) a procesorului 80x86 legată de noțiunea de depășirea cauzală operabilă de adunare și de scadere. Cum semnalarea lăsată procesorului în familia 80x86 depășește la limită cea să respecte la imparitate?

Nu putem să vedem că comparația de pe 200 cu -1 să căuta o soluție la minitălizerii și nici pe -56 cu 255 deoarece interpretarea este ortă cu semn ori triplă semn pentru ambi

Că urmare din cele 4 situații teoretice posibile de mari sus, vom înțelege concret numai două:

- comparativă fără semn (-56 cu -1) – impulsă de „less than” sau „below”
- comparativă cu semn (-56 cu -1) – impulsă de „greater than” sau „above”

Ja etă; deoarece  $200 < -1$  în acest caz ne-am așteptă că salutul să se efectueze... Însă utilizarea instrucțiunii **Ja** (jump if Above) impune interpretarea frază semn, deci varianta de comparatie corectă ar fi  $200 \geq 255$  și cum  $200 > 255$  condiția nu este îndeplinită și deci salutul nu se va efectua. În plus, în cauză nostru  $CF=1$  deoarece este îndeplinită: ar trebui să avem  $CF=Z=0$ , însă în același moment  $Z=1$  și  $CF=1$ . Ca o contumace, se poate vedea că nici condiția tehnică impusă de **Ja** nu superioară valoarii  $-1$  își. Într-o situație similară, să vedem cum se poate demonstra că  $200$  nu este îndeplinită și deci salutul nu se va efectua (în loc să comparați  $200$  cu  $255$  să comparați  $200$  cu  $255$  și cum  $200 > 255$  condiția nu este îndeplinită și deci salutul nu se va efectua). În acest caz trebuie să dezvoltăm un program care să verifice dacă  $200 < -1$ .

**Jf eti** : evidență că 200-*i* deci la prima vedere pare că nu este îndeplinită condiția neceasă pentru efectuarea salutului... să nu uităm însă faptul că Jf (Jump If Less) interpretează rezultatul comparării ca fiind cu semn (deci -55) acesta înseamnă implicit că scaderea este interpretată ca (-56 -(-1)) deci și operanții vor fi amândoi interpretati și faptul că scaderea este inițială comparată cu semn (decit -55) aceasta înseamnă implicit că scaderea este inițială comparată cu semn (decit -56) și astfel se va obține rezultatul că Jf este îndeplinită condiția se verifica (pe lângă jurnalul de interpretare) și faptul că este inițială comparată cu semn... cum -56 < -1 astă că și inițială condiția se verifica (pe lângă jurnalul de interpretare) și faptul că este inițială comparată cu semn... cum 200 < -1 (exemplifică de aci și jurnalul de calcul se va efectua și portă să fie "demonstrativ")

debut din acțiunea instanței CMP (care nu distinge absolut de loc într-o varianta cele 4 de comparație de mai sus) și pe baza unor evenimente instanțăului într-o rolă de a interpreta în moduri de mai multe moduri de mai sus comparată efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instanțăului correspunzător de la final.

Situatiile in care produsul incapsuleaza pe dimensiunea operanzilor vor fi caracterizate de setarile CF=OF=0 (nu avem deci depasire la inmultire). Iata un exemplu:

```
mov al, 5
mov bl, 51
mul bl      ; AX := AL * BL = 5 * 51 = 255 = 00ffh si vom avea CF=0 si OF=0
              ; deoarece octetul superior AH = 0.
```

**Depasirea la impartire.** In cazul impartirii, specificarea acestei operații sub forma

(IDIV operand

presupune că operandul specificat este împărțitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biți) iar deîmpărțitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărțitorul este cuvânt). Efectuarea operației are ca efect:

AX : operand pe 8 biți = cîtuil în AL și restul în AH;  
DX:AX / operand pe 16 biți = cîtuil în AX și restul în DX;

În cazul împartirii depasirea apare atunci când rezultatul împartirii nu încapsează în spațiul rezervat conform definiției pentru reprezentare, mai exact, când cîtuil nu încapsează în AL sau respectiv AX. Într-o astfel de situație, procesorul 80x86 emite o întrerupere 0, execuția terminându-se cu un mesaj furnizat de către rutina de tratare a întreruperii 0, de genul "Divide by zero", "Zero divide" sau "Divide overflow" (în funcție de tipul de procesor și/sau de SO instalat). Pare ciudat la prima vedere că o împartire prin 0 (de genul *div bh* cu *bh* = 0) ce practic nu se poate efectua din punct de vedere matematic este tratată similar ca efect din punct de vedere al limbajului de asamblare cu o împartire care matematic se poate efectua. Secvența

```
mov ax,60000
mov bl,2
div bl
```

ar trebui să furnizeze din punct de vedere matematic cîtuil 30000. Însă conform definiției împartirii DIV acest cîtuil trebuie memorat în registrul AL, de dimensiune octet. Cum cea mai mare valoare reprezentabilă pe 1 octet este 255, este evident astfel că din punct de vedere al limbajului de asamblare împartirea de mai sus nu se poate efectua (similar cu o situație de tip *div 0*) și ca urmare înțelegem acum decizia proiectanților de a trata tot prin emiterea unei întreruperi 0 și o situație de genul celei de mai sus. Să remarcăm în acest sens și faptul că mesajul "*Divide overflow*" (depasire la împartire) este acceptat în acest context ca similar unui "*Divide by zero*".

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentați este de a confunda exprimările "*numere cu semn*" și "*numere fără semn*" cu exprimările "*numere negative*" și respectiv "*numere pozitive*". Numere cu semn nu înseamnă automat numere negative! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.

Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunțul unei probleme care cere efectuarea unei anumite acțiuni "dacă numărul v este (strict) negativ"? În primul rînd vom conchidona că este vorba despre interpretarea **cu semn**. Se pune însă întrebarea: cum vom testa practic dacă un număr **cu semn** este **negativ** sau nu? (să presupunem că v este octet). Fiind vorba despre interpretarea **cu semn**, dacă primul bit al configurației binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra primului bit din reprezentarea numărului. Iată două alternative pentru realizarea unui astfel de test:

a). Realizăm o deplasare a primului bit în CF și testăm valoarea sa printr-o instrucție adecvată de salt condiționat. Secvența

mov al,v	;pentru a nu afecta deosebitiv conținutul variabilei v
shl al,1	;shift stînga cu 1 poziție pentru ca primul bit să treacă în CF.
jc este_negativ	;dacă CF=1 atunci salt la eticheta este_negativ

asigură testarea faptului dacă variabila v este sau nu un număr negativ.

b). Utilizăm instrucția **cmp** pentru o comparație în raport cu 0:

cmp v,0	;scădere fictivă v-0
jl este_negativ	;dacă v<0 atunci salt la eticheta este_negativ

sau alternativ

cmp 0,v	;scădere fictivă 0-v
jg este_negativ	;dacă 0>v atunci salt la eticheta este_negativ

ix). Am văzut că la nivelul efectuării operațiilor de adunare sau scădere procesorul 80x86 nu diferențiază între adunări/scăderi cu semn sau fără semn (tehnic vorbind ele se efectuează drept operații binare cu **rezultat interpretabil ulterior** drept cu semn sau fără). Totuși, în momentul în care se pune problema exprimării în baza 10 a unei operații de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect **operanții** operației respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obținut? Mai concret:

00000101 +	(= 5 în ambele interpretări)
11111110	(= 254 fără semn și -2 în interpretarea cu semn)
(1) 00000011	(= 3 în ambele interpretări ale configurației pe 8 biți)

reprezintă  $5 + 254 = 259 (= 1\ 00000011 - \text{configurație pe 9 biți} !)$  sau reprezintă  $5 + (-2) = 3 ?$  După cum vom vedea și aici răspunsul este că putem interpreta în ambele moduri și să justificăm astfel ca două reacții separate modul de setare al flag-urilor CF și respectiv OF.

Datorită cifrei de transport vom avea CF=1 (independent de interpretarea operanzilor sau a rezultatului final drept cu semn sau fără semn, deoarece este vorba despre o consecință tehnică a

Test DB	'Accessa este un exemplu'	SF LABEL BYTE
data segment	.	.
cod segment	.	.
mov cx, SF-Test	mov bx, OFFSET Test	mov bx, cx
Tipareste:	mov dl,[bx]	:preia urmatorul caracter
inc bx	inc bx	:puntem pe urmatorul caracter
mov ah,2	mov ah,2	:afisarea tiparitii
int 21h	int 21h	:acelui care a urmat
Loop Tipareste	loop Tipareste	:daca mai sunt caractere
endif	endif	

Înstructiunile **LOOP** și **DO WHILE** sunt de tip **repetitie**. În cadrul unei instructiuni **LOOP**, se execută o secvență de instrucțiuni de terminare, cînd împreună cu condiția de parcare formă un ciclu. În cadrul unei instructiuni **DO WHILE**, se execută o secvență de instrucțiuni de parcare, cînd împreună cu condiția de terminare, formă un ciclu.

În cadrul unei prezentări de informații se pot apăra și argumentațiile care susțin că un anumit lucru este adevărat sau că un alt lucru este fals.

```

data segment
    Vector DB 128 DUP(?)
Vector DB 128 DUP(?)
cod segment
    Bucle:
        mov cx,128
        lea bx,Vector
        mov ah,1
        int 21h
        ;Funcia DOS pentru citire caracter
        ;aciiara functie
        ;memorarea la seti apasate
        mov [bx],al
        inc bx
        cmp al,0dh
        jne bucla
        ;daca nu, preda controlul instigurii de dupa
        ;afost <ENTER>?
        loopne bucla

```

modulii de efectuare a operațiilor binare de adunare). Cu un mare interes maxim reprezentabil pe 1 octet).

Ce se întâmplă cu UE? Rularea secvenței

mov bl, 254 ; -2 în interpretarea cu semn  
add al, bl ; AL := AL+BL = 5+(-2) = 3

În interpretarea sa regăzit OF la variabila x<sub>1</sub>, deci situarea de mai sus nu este considerată „adeasă” întrucât de valoarea lui x<sub>1</sub> depinde de valoarea lui x<sub>2</sub>. În schimb, dacă ar fi scrisă:

```
add al, bl ; deci 5 + (-2) = 3  
mov bl, -2  
mov al, 5
```

și este evident că în acestă intervale nu este vorba despre nici o depășire (și de aceea și OF = 0). Să ne remitem în acest context și exemplul date la prezentarea RDA și RDS de la paginile 77-78; adunarea 100 + 50 = 150 va semnala depășirea (signed overflow - conform RDA) și că paginile 77-78 au interpretat ca -126 - 42 = -168 ([-128..127]) și 42 - 130 (interpretată ca scaderă 42 - (-126) = +168 și [-128..127]) produc la randul lor signed overflow și că numerele OF=1.

### 4.3.3. Instrucțiuni de ciclare

Cu ajutorul instrucțiunilor de salt condiționat se pot construi cicluri (bucle). Un ciclu nu este altceva decât un bloc de instrucții care se termină cu o instrucție de salt condiționat ( în acest sens se cumonsco foarte bine constatăcă ilie de tip for, while și repeat din unele limbaje de nivel înalt).

Limbajul de asamblare 80x86 prevede instrucțiuni speciale pentru realizarea ciclurii. Ele sunt LOOP, LOOPNE, LOOPNE \$I JCZ. Sintaxa lor este

Ca exemplu să luăm tipările caracterelor unui și următori:

**LOOP** comanda reținarea excepțiilor blocați în de instrucțiuni ce începe la `elcheta`, astăzi înlocuită cu `try`.

<instruction> etichetta

**LOOPE** mai este cunoscută și sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută și sub numele de **LOOPNZ**. Aceste instrucțiuni se folosesc de obicei precedate de o instrucțiune CMP sau SUB.

O altă instrucțiune folosită pentru controlul ciclării este **JCXZ** (*Jump if CX is Zero*). Această instrucțiune realizează saltul la eticheta operand numai dacă CX=0, fiind utilă în situația în care se dorește testarea valorii din CX înaintea intrării într-o buclă. Exemplul următor se referă la inițializarea cu 0 a unui șir de octeți, CX conținând lungimea acestui șir. Instrucțiunea JCXZ se folosește pentru a se evita intrarea în ciclu dacă CX=0:

```

jcxz MaiDeparté ;dacă CX=0 se sare peste buclă
Bucla:
    Mov  BYTE PTR [si],0 ;inițializarea octetului curent
    inc  si               ;trecere la octetul următor
    loop Bucla            ;reluare ciclu sau terminare
MaiDeparté:

```

Dar să vedem de ce se acordă totuși atenție sporită situației în care se începe execuția unei bucle cu CX=0. La întâlnirea instrucțiunii LOOP cu CX=0, CX este decrementat, obținându-se valoarea 0FFFFh (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în CX, adică de încă 65535 ori! Iată deci de ce este necesar să ne asigurăm că nu intrăm într-un ciclu cu CX=0, iar instrucțiunea JCXZ acționează rapid și eficient exact în acest sens.

Este important să precizăm aici faptul că nici una dintre instrucțiunile de ciclare prezentate nu afectează flag-urile.

```

loop Bucla  și  dec cx
                jnz Bucla

```

desi semantic echivalente, nu au exact același efect, deoarece spre deosebire de LOOP, instrucțiunea DEC afectează indicatorii OF, ZF, SF și PF.

#### 4.3.4. Instrucțiunile CALL și RET

Apelul unei proceduri se face cu ajutorul instrucțiunii **CALL**, acesta putând fi *apel direct* sau *apel indirect*. Apelul direct are sintaxa

**CALL operand**

Asemănător instrucțiunii JMP și instrucțiunea CALL transferă controlul la adresa desemnată de operand. În plus față de aceasta, înainte de a face saltul, instrucțiunea CALL salvează în stivă adresa următoarei instrucțiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalența

#### Cap.4. Instrucțiuni ale limbajului de asamblare.

<b>CALL operand</b>	[ push CS ] (numai dacă este operand far)
A: . . .	push offset A
	jmp operand

Terminarea execuției secvenței apelate este marcată de întâlnirea unei instrucțiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucțiunea de la această adresă. Sintaxa instrucțiunii RET este

**RET [n]**

unde *n* este un parametru optional. El indică eliberarea din stivă a *n* octeți aflați sub adresa de revenire. Vom detalia acest mecanism în capitolul 8. Instrucțiunea RET poate fi far sau near. Ca efect, avem echivalențele

<b>RET n</b> (revenire near)	B dw ? pop B add sp,n jmp B
<b>RET n</b> (revenire far)	B dd ? pop word ptr B pop word ptr B+2 add sp,n jmp B

De cele mai multe ori, instrucțiunile CALL și RET apar în următorul context

```

nume PROC
    .
    .
    .
    ret n
nume ENDP
    .
    .
    .
    CALL nume

```

Directivele PROC și ENDP au fost prezentate în 3.3.4. Apelul și revenirea sunt implicit far sau near, după cum procedura *nume* este declarată FAR sau respectiv NEAR.

Instrucțiunea CALL poate de asemenea prelua adresa de transfer dintr-un registru (pentru un apel intrasegment) sau dintr-o variabilă de memorie. Un asemenea gen de apel este denumit *apel indirect*. Exemple:

```

call  bx  ;adresă preluată din registru
call  vptr ;adresă preluată din memorie

```



#### 4.4.2. Instrucțiuni pe siruri pentru transferul de date

Acstea instrucțiuni sunt asemănătoare instrucțiunii MOV, dar ele realizează mai mult și operează mai rapid.

Instrucțiunea LODS se prezintă sub două forme: LODSB și LODSW. Ea încarcă un octet sau respectiv un cuvânt din memorie în registrul acumulator.

<b>LODSB</b>	AL <-> <DS:SI>	if DF=0 inc(SI) else dec(SI)	—
<b>LODSW</b>	AX <-> <DS:SI>	if DF=0 SI<-SI+2 else SI<-SI-2	—
<b>STOSB</b>	<ES:DI> <-> AL	if DF=0 inc(DI) else dec(DI)	—
<b>STOSW</b>	<ES:DI> <-> AX	if DF=0 DI<-DI+2 else DI<-DI-2	—
<b>MOVSB</b>	<ES:DI> <-> <DS:SI>; if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}	—	—
<b>MOVSW</b>	<ES:DI> <-> <DS:SI>; if DF=0 {SI<-SI+2; DI<-DI+2} else {SI<-SI-2; DI<-DI-2}	—	—

Instrucțiunea **LODSB** încarcă octetul de adresă DS:SI în AL și apoi incrementează sau decrementează SI, aceasta depinzând de starea flagului DF (*Direction Flag*): dacă DF=0 (valoare ce se poate seta după cum am văzut prin instrucțiunea CLD) atunci SI este incrementat, iar dacă DF=1 (setarea acestuia cu 1 făcându-se cu instrucțiunea STD) SI este decrementat. Această regulă impusă de valoarea din DF este valabilă pentru toate instrucțiunile pe siruri ce afectează registri pointer. De exemplu,

```
cld
mov si,0
lodsb
```

va încărca AL cu conținutul octetului de deplasament 0 din cadrul segmentului de date și apoi se va incrementa SI cu 1, acțiuni echivalente cu

```
mov si,0
mov al,[si]
inc si
```

Instrucțiunea LODSB este însă mai rapidă și cu 2 octeți mai scurtă decât echivalentul

```
mov al,[si]
inc si
```

Instrucțiunea LODSW încarcă în AX cuvântul adresat de DS:SI, incrementând sau decrementând apoi SI cu 2 (deoarece este vorba de valori reprezentate pe cuvânt). De exemplu,

```
std
mov si,10
lodsw
```

#### Cap.4. Instrucțiuni ale limbajului de asamblare.

încarcă AX cu valoarea cuvântului de memorie de deplasament 10 din cadrul segmentului de date, decrementând apoi SI cu 2.

Instrucțiunea STOS este complementara instrucțiunii LODS, ea transferând valoarea octet sau cuvânt din acumulator la locația de memorie de adresă ES:DI, incrementând sau decrementând apoi corespunzător pe DI. și instrucțiunea STOS se prezintă sub două forme: STOSB și STOSW.

Instrucțiunea STOSB copiază octetul din AL la octetul de adresă ES:DI, incrementând sau decrementând DI în funcție de valoarea din DF. De exemplu,

```
std
mov di,0ffffh
mov al,55h
stosb
```

depune valoarea 55h la octetul de deplasament 0FFFFh din cadrul segmentului pointat de ES, decrementând apoi DI la valoarea 0FFEh.

Instrucțiunea STOSW este asemănătoare, copiind valoarea cuvânt din AX în cuvântul adresat de ES:DI, incrementând sau decrementând apoi DI cu 2. De exemplu,

```
cld
mov di,0ffeh
mov ax,102h
stosw
```

copiază valoarea cuvânt 102h din AX la adresa ES:0ffeh, incrementând pe DI la valoarea 1000h.

Instrucțiunile LODS și STOS funcționează eficient împreună pentru copierea de siruri. Subrutina COPIERE de mai jos realizează copierea sirului terminat cu 0 care începe la DS:SI în sirul ce începe la adresa ES:DI :

```
;Subrutină pentru copierea unui sir terminat cu 0 în altul
;Intrări:      adresa de început a sirului sursă    DS:SI
;            ; adresa de început a sirului destinație ES:DI
;Ieșiri : -
;Registri afectați : AL, SI, DI
```

```
Copiere      PROC
    cld
                ;parcurgerea se va face crescător deci se va impune incrementarea
```

SCASB	CMP AL, <b>{ES:DI}</b> if DF=0 <b>inc(DI)</b> else <b>dec(DI)</b> OF, SF, ZF ,AF, PF, CF
SCASW	CMP AX, <b>{ES:DI}</b> if DF=0 <b>DI&lt;-DI-2</b> else <b>DI&lt;-DI-2</b> OF, SF, ZF ,AF, PF, CF
CMPSB	CMP {DS:SI}, <b>{ES:DI}</b> , if DF=0 <b>inc(DI)</b> else <b>dec(SI); dec(DI)</b> OF, SF, ZF ,AF, PF, CF
CMPSW	CMP {DS:SI}, <b>{ES:DI}</b> , if DF=0 <b>{SI&lt;-SI+2; DI&lt;-DI-2}</b> OF, SF, ZF ,AF, PF, CF

De exemplu, secvența următoare căuta prima literă 'a' din cardul săru lui Text:

**L**a ca cuhodasystem direct numarul de octetii (cu valoare) ce trebuie copiatii (copiate) se poate proceda astfel:

da cada nu, se continua  
a fast 0 ?  
memorareaza la destinatife

MOV \$memoria, [adresa] Modifică memoria la adresa [adresa] cu valoarea de \$memoria. Adresa poate fi imediată sau indirectă prin folosirea instrucțiunii MOV.

călăunea SCAS (care are și ea două variante: SCASB și SCASW) căuta în memoria o sumă în

4.4.3. Înstrucțiunile de servicii pentru consultarea și compararea datelor

**SCASB** compara valoarea din AL cu valoarea acelăiași adresații de ES-**DL**, pentru instituțiile sale de stat și pentru CMB), Dl și apoi

În cadrul unei interacțiuni cu un utilizator, se poate întâlni următoarea scenă: utilizatorul spune "Îmi place că suntem conectați la același rețea de rețele sociale", și programul răspunde "Da, suntem conectați la același rețea de rețele sociale". În acest caz, programul nu a înțeles propria întrebare și a generat o răspunsă nesensicală.

```

segment Text DB ,comparare date',0
Text equ ($-Text)
LungiMesir mov ax,dx
mov al,' '
mov es,ax
mov di,OFFSET Text
mov cx,LungiMesir
jungimedesir
;se stabilește direcția de consultare "spate înainte"
;lungimea și înălțimea
;se potrivește AL cu ES:DI?
;dacă da, s-a terminat căutarea
;dacă nu, se continuă
loop Cautarea
je Gasta
scasb
Cautarea:
(DI se va micșora)
;lungimea și înălțimea
;se stabilizează direcția de căutare
;se deosemnează DI pentru a indica
;offestul caracterului 'a' găsit
dec di
Gasta:

```

singura diferență ca ordine de efectuare a acțiunilor apărând în faptul că SCASB realizează incrementarea lui DI înaintea execuției instrucțiunii JE. Dacă aici incrementarea o facem explicit, atunci trebuie să o efectuăm după instrucțiunea JE pentru a nu afecta flagurile setate de către CMP!

Legat de aceasta să facem aici precizarea că instrucțiunile pe șiruri NU setează flagurile în urma acțiunii asupra reștrierilor SI, DI sau CX. Instrucțiunile LODS, STOS și MOVS nu afectează nici un flag, iar SCAS și CMPS modifică flagurile doar ca rezultat al comparațiilor pe care le efectuează.

Instrucțiunea SCASW compară conținutul registrului AX cu cuvântul de adresă ES:DI, incrementând sau decrementând DI cu 2, în funcție de valoarea din DF. Secvența de mai jos utilizează instrucțiunea SCASW cu prefixul REPE (vezi 4.4.4.) pentru a căuta ultima valoare nenulă dintr-un vector de întregi:

```

mov ax, SEG Tablou
mov es, ax
mov di, OFFSET Tablou+((NrElem-1)*2)
          ;ES:DI punctează astfel spre ultimul element al tabloului
mov cx, NrElem
sub ax, ax      ;pone 0 în AX pentru a căuta un element nenul
std             ;căutarea începe de la sfârșit
repe scasw     ;se repetă căutarea până la primul element nenul sau
               ;până la epuizarea elementelor tabloului
jne AmGasit
               ;dacă se ajunge aici, tabloul conține numai elemente nule
AmGasit: inc di
           ;se actualizează DI pentru a puncta spre elementul găsit
inc di

```

Instrucțiunea CMPS are ca rol efectuarea de comparații de șiruri de octeți sau cuvinte. Execuția unei instrucțiuni CMPS are ca efect comparația locațiilor de memorie de adrese DS:SI și respectiv ES:DI, urmată de incrementarea sau decrementarea reștrierilor SI și DI. Flagurile vor fi actualizate corespunzător pentru a reflecta rezultatul comparației.

Instrucțiunea CMPSB realizează comparația la nivel de octet, iar CMPSW la nivel de cuvânt, prima incrementând sau decrementând SI și DI cu 1 iar ultima cu 2. În exemplul următor se compară două tablouri ce conțin elemente reprezentate pe cuvânt pentru a se decide dacă primele 100 de elemente sunt sau nu identice:

```

mov si, OFFSET Tablou1
mov ax, SEG Tablou1
mov ds, ax
mov di, OFFSET Tablou2
mov ax, SEG Tablou2
mov es, ax

```

```

mov cx, 100
cld
repe cmpsw
jne TabDiferite

```

TabDiferite:

```

dec si      ;se actualizează reștrierii SI și DI pentru
dec si      ;a puncta spre elementul prin care diferă
dec di
dec di

```

#### 4.4.4. Execuția repetată a unei instrucțiuni pe șiruri

Pentru execuția repetată a unei instrucțiuni pe șiruri, limbajul de asamblare dispune de variante echivalente instrucțiunii de ciclare LOOP. Acestea sunt oferite de-a lungul *prefixelor de instrucțiune*. Sintaxa utilizării lor este

*prefix\_de\_instrucțiune instrucțiune\_pe\_sir*

În principiu este vorba despre un singur prefix de instrucțiune, și anume REP. Vom vedea mai jos însă, că în cazul utilizării instrucțiunii SCAS sau CMPS apar două variante posibile de REP.

Prefixul de instrucțiune REP impune execuția repetată a instrucțiunii pe care o prefixează, până când valoarea din CX devine 0. Dacă de la început avem CX=0 atunci instrucțiunea respectivă este inoperantă. Astfel, ciclul

Bucla: *instrucțiune\_pe\_sir*  
loop Bucla

este echivalent cu instrucțiunea

rep *instrucțiune\_pe\_sir*

În cazul utilizării prefixelor cu instrucțiunile SCAS sau CMPS condiția verificată se completează, ținându-se cont de valoarea flagului ZF. Acest lucru face ca prefixul REP să fie prezent în două variante.

Forma REP (echivalentă cu formele REPE - REPeat while Equal și REPZ - REPeat while Zero) provoacă execuția repetată a instrucțiunilor SCAS sau CMPS până când CX devine 0 sau până când apare o nepotrivire (caz în care ZF va primi valoarea 0).

#### 4.3. UN EXEMPLU COMPLET DE PROGRAM

Am prezentat acesta pentru a evidenția faptul că prezenta o operanță numărătură într-o instrucțiune pe său nu încarcă automat adresa lui *Dw* la *DS:SI*, acest lucru ramânând în sarcina programatorului. În acestă secțiune vom prezenta un exemplu complet de program. În acest scop ne-am optat asupra program prime; constă în următoarele linii de cod:

```

begin
    for i := 0 to n-1 do S[i] := i;
    for i := 0 to n-1 do
        j := i + 3;
        while j <= n do begin
            pas := i + j;
            j := j + 3;
            while S[j] = 0 do i := i + 1;
            S[j] := 0;
        end;
        i := i + 1;
    end;
    writeln(j+3:10);
end.

```

Limbajul de assembly 80x86 acceptă utilizarea instrucțiunilor neexplicite dacă se prevad corectă numărările într-o instrucțiune de executat (de exemplu, LODSB sau LODSW).

Intr-o instrucțiune de dimensiună 4 byte, am folosit LODSB și LODSW dar în exemplul de mai jos este echivalentă cu MOVSB:

```

data segment
    S12 DB 50 DUP (?)
    LungsS11 EQU ($-S11)
    S11 LABEL BYTE DB 'Stil surșa'
    cod segment
        mov ax, data
        mov ds, ax
        mov si, OFFSET S11
        mov di, OFFSET S12
        mov cx, LungS11
        rep moves es:[S12], [S11]
        cld
    ends

```

Asemănător, *REPNE* (*REP* at *while* *Not Equal*, forma echivalentă cu *REPZ-REP*) provoacă execuția unei operațiuni *not* pe fiecare byte din cadrul *ZF*, permitând răfinarea condițiilor pe baza că sunt singurele care afectează vreun flag (îl anume *ZF*), ceea ce în cazul instrucțiunilor LODS, STOS și MOVs totale că micșorează numărările într-o instrucțiune pe său.

De ce există deosebite dorințe să se folosească variantele *REP*? Foarte simplu: pentru că apar o potrivire (căză în care *ZF* va primi valoarea 1). De asemenea, *REPNE* (*REP* at *while* *Not Equal*, forma echivalentă cu *REPZ-REP*) provoacă execuția unei operațiuni *not* pe fiecare byte din cadrul *ZF*, permitând răfinarea condițiilor pe baza cădă și se aplică pe fiecare byte din cadrul *ZF*.

Asemănătorul va avege varianta de instrucțiune MOVs în funcție de dimensiunea de reprezentare a operațiunilor, astfel că ea va fi:

Vom transpune acum acest program într-unul echivalent scris în limbaj de asamblare. Evident, vom face uz de toate facilitățile oferite de acesta din urmă. De aceea, în loc de a reține câte un octet pentru fiecare poziție din ciur, vom reține doar câte un bit. Variabila *i* din programul Pascal este înlocuită aici cu registrul **bx**, care va conține numărul bitului curent. Numărul cuvântului curent este conținut în registrul **di**. Variabila *j* este înlocuită cu registrul **dx**. Tipărirea este realizată cu ajutorul funcției 9 a întreruperii 21h, despre care vom vorbi în capitolul următor. Aici trebuie doar remarcat faptul că operația de conversie în sir ASCII a unui număr cade în sarcina programatorului.

Cu aceste precizări, programul ASM este următorul:

```
.model small

n    equ 65535      ;definire de constante
ni   equ (n-1)/2
ni2  equ ni/2
nc   equ (ni+15)/16

.data

rez  db  '      ,13,10,'$'
pas  dw ?
zece dw 10
s    dw nc dup (?)

.code
Start:
    mov ax,@data
    mov ds,ax
    mov es,ax

    mov ax,0ffffh
    cld
    mov cx,nc
    lea di,S
    rep stosw      ;for i
    xor bx,bx

whilei:
    cmp bx,ni2
    jb peste        ;echivalent cu jnb exit dar eticheta exit
    jmp exit         ;este prea departe pentru un salt condiționat
peste:
    mov di,bx
```

```
mov cl,4
shr di,cl
shl di,1
mov ax,[S+di]    ;ax cuvântul curent

mov cx,bx
and cx,0fh
shr ax,cl
shl ax,cl
cmp ax,0          ;în cuvântul curent au mai rămas biți 1
jnz AreBit1
mov cx,nc
add di,2 + offset S
repe scasw        ;s-a găsit un cuvânt nenul
sub i,2 + offset S
mov x,[S+di]
```

## AreBit1:

```
mov cl,3
mov bx,di        ;bx = numărul de biți din cuvintele precedente
shl bx,cl
iar:
    shr ax,1
    jc gasit
    add bx,1      ; se adaugă până la primul bit 1
    jmp iar
```

```
gasit:
    mov pas,bx
    add pas,bx
    add pas,3      ;pas := i + i + 3
```

```
    mov dx,bx
    add dx,pas      ;j := i + pas
```

## whilej:

```
    cmp dx,ni
    jae Alti

    mov di,dx
    mov cl,3
    shr di,cl
    mov al,byte ptr [S+di]
    mov cx,dx
```



## CAPITOLUL 5

### ÎNTRERUPERI

#### 5.1. PROBLEME GENERALE PRIVIND ÎNTRERUPERILE

O întrerupere este o acțiune a microprocesorului prin care acesta anunță apariția unui eveniment. Mai concret, întreruperea este un semnal electric transmis sistemului de calcul (SC) prin care acesta este anunțat de apariția unui eveniment particular.

Acțiunile pe care le efectuează sistemul de calcul la apariția unei întreruperi sunt:

1. suspendarea programului în curs de desfășurare;
2. lansarea în execuție a unei rutine specializate, numită *Rutină de Tratare a Întreruperii (RTI)* sau *Handler de întrerupere*, care deservește întreruperea;
3. eventual, reluarea execuției programului suspendat (depinzând de tipul de întrerupere).

Cauzele apariției acestor evenimente pot fi de 2 tipuri:

- a). *externe* (apăsarea unor combinații de taste, inițierea sau terminarea unor operații de I/O);
- b). *interne* (împărțirea la 0, tentativa de adresare a unei zone de memorie inexistente, tentativa de execuție a unei instrucțiuni având un cod inexistență, depășirea capacitatei de reprezentare a unui rezultat).

De obicei, după tratarea unei întreruperi externe programul se reia, după o întrerupere internă nu!

La apariția unei întreruperi, SC trebuie, în ordine:

- 1) să determine tipul evenimentului care a generat întreruperea (intern, extern);
- 2) să afle care este cauza întreruperii;
- 3) să determine adresa RTI (rutinei de tratare a întreruperii);

Există 3 categorii de rutine de tratare a întreruperilor (RTI):

- furnizate odată cu sistemul de calcul;
- scrise de proiectanții sistemului de operare (SO);
- scrise de utilizatori;

În documentații recomandătem interupeptorul hardware prim preferirea la IRQ (Interupt Request),

Un excelent tutorial oferind detalii despre funcțiile unei interupeptori hardware sunt prezentate în cele mai cunoscute și frecvent utilizate (mesie) interupeptor hardware.

INT 8 este interupeptor hard de ceas (the system timer). Este recomandată prim IRQ 0. Se produce de 18.2 ori/seunda. Nu se determinază de obicei, dacă se cauză de deturarea soft (ICh). RTI

refine nr. de cicluri de ceas (timer ticks) la adresa 0000:046Ch pentru a gestiona în bună

condiții ora sistem.

- decreremontare un counter de la adresa 0000:0440h (Diskette Drive Motor Off Counter) adresă 0000:043Ch este moment în care motorul de antrenare este pus pe OFF și ocetul de tip

de interupeptor hardware – interupeptor generale în mod automat ca răspuns la apariția unor cauze concrete ale RTI corespunzătoare sună în sensul: BIOS-ul răspunde prim către sec-an-codul

interupeptor este generată de tastatura, la fiecare apasare și eliberare a unei taste. Acestui interupeptor este specifică de tastatura, care să la adresa 0000:041Ch. Începând de la dezactivarea Scan code/ASCII code în documentația sa la adresa 0000:043Ch, interupeptor (a se vedea

corespunzător tastei, convertirea sa la codul ASCII corespunzător de memorarea acestui caracter de atribută în bufferul de tastatură, aceasta fiind localizată la adresa 0000:041Ch. Începând de la

accesătă apăsat și prima la dezplasamente 043ch (adică 20h = 32 octet) se pot răfie 16 caractere numai a unui cod ASCII pentru l-character, deoarece există mai multe tipuri de tastatură și poziția unui caracter în configurația tastaturii trebuie caracterizată prin atributul scan code. Aceasta este

adresăt un cod asociat unei taste în funcție de pozitia de tastatură. Acesta este adresăt de la adresa 0000:0417h (Shift Status byte) și 0000:0418h (Extended Shift Status byte).

Atenție! – nu se dezacorda interupeptore, deoarece atunci SC nu va mai răspunde nici la CTRL + ALT + DELTE !!!.

Dacă s-a apăsat una din teastele Ctrl, Alt sau Shift, se actualizează corespunzător ocafei de la

Vom prezenta și analiza în continuare cele mai importante interupeptori din fizice categorie.

Interupeptori ce pot fi inițiate numai de către programator prin INT sunt interupeptori soft.

Datorita acestui specific putem da o definitie alternațiva acestui tip de interupeptor: accele

explicat de programator.

Interupeptori software tocmăi pentru că ele sunt invocate soft print-o instanță specifice

programatorului inițiază o astfel de acțiune este instrucțiunea INT. Aceste interupeptori se numesc al execuției, inițiat de programator către o rutina specială (handler). Mulțumit printr-o car

c), interupeptori software (software interrupts sau traps) – acestea preseupun în transfer de control

accesarea unei zone de memorie interioare (memory protection fault).

Exemplu de cauze: împărțirea prin 0, încercarea de execuție a unei cod de instrucțiune înexisten-

b), execuție - interupeptori generate în mod automat ca răspuns la apariția unor cauze de tip inter-

BIOS.

corespunzătoare sunt înărcicăte în memorie la porțile sistemu de calcul din fizice ROM.

de intrare-iesire (I/O system) și fiind astfel interupeptori BIOS (Basic Input Output System), RTI

microprocesorului, cum ar fi de exemplu semnalul de la periferice, fiind asadar de sistemul de tip extem. Acestea sunt deci cauzate de un eveniment extem hardware (extrem

a). interupeptori hardware – interupeptori generale în mod automat ca răspuns la apariția unor cauze

La nivelul arhitecturii 80x86 apar trei tipuri de evenimente numite în documentații interupeptori:

## 5.2. CLASIFICAREA INTERUPEPILOR

interupeptori, și ne vom ocupa de aceasta în capitolul 6.

sistemului, fiind necesară remobilizarea lui. Modificarea accidentala conduce de cele mai multe ori la blocarea

funcționalită sistemu, este posibila modificarea unor direcție adresă. Modificările pot apărea fie

acces tablou cu adrese se inițializază în momentul încarcării sistemu de opera. În timpul

interupeptor K, adresa RTI(K) se gaseste la adresa 0000 : K\*4.

1024 octetii conțin adrese (zona se numește tabla vectorilor de interupeptore - TVI). Pentru

Tabelă RTI (vectorul de interupeptor) se află în memorie la adresa 0000:0000. Primi 256 x 4 =

RTI corespunzătoare interupeptori.

Pentru locația rapida a RTI se folosește vectorizarea interupeptor: asocieră fizice

RTI corespunzătoare interupeptori.

**INT 0Bh** și **INT 0Ch** – intreruperi ce deservesc porturile seriale. Există o gamă de dispozitive fizice ce se conectează la SC prin intermediul porturilor seriale, cum ar fi un plotter, un modem, USB (Universal Serial Bus) sau mouse-ul de exemplu.

Ce este însă transmisia serială a informației comparativ cu cea paralelă? În transmisia serială există o singură linie de date iar informația este transmisă succesiv bit cu bit (un singur bit odată). Deși lentă în comparație cu o transmisie paralelă ce permite transmiterea simultană a mai multor biți (8,16,32 sau 64) această tehnică a transmisiei seriale este adecvată transmiterii informației la distanțe mari sau în cazul utilizării ca medii de transmisie a cablurilor telefonice, cablurilor coaxiale, undelor radio sau fibrei optice.

INT 0Bh (IRQ 3) se activează când este vorba despre o comunicare serială pe interfața (portul) COM2, iar INT 0Ch (IRQ 4) se activează când este vorba despre o comunicare serială pe interfața (portul) COM1.

Gestionarea comunicației seriale la nivelul arhitecturii 80x86 este asigurată de către dispozitivul UART (*Universal Asynchronous Receiver/Transmitter*) 8250 (sau compatibil cu acesta) care generează o intrerupere (IRQ 3 sau IRQ 4, depinzând de linia serială pe care are loc comunicarea) într-o din următoarele patru situații: un caracter sosește pe o linie serială, dispozitivul UART a terminat transmisia unui caracter și solicită un altul, apariția unei erori sau solicitarea unei modificări de stare. RTI corespunzătoare va trebui să determine cauza exactă a intreruperii prin interogarea dispozitivului UART.

**INT 0Dh** și **INT 0Fh** – intreruperi ce deservesc porturile parallele. După cum aminteam mai sus interfețele parallele permit transmisia simultană a mai multor biți (8,16,32 sau 64). Principalul avantaj al interfețelor parallele este astfel viteza de transmisie. Aceasta este însă obținută prin costuri suplimentare reprezentate de necesitatea unor cablări adiționale pentru a asigura extinderea canalelor de date. Din cauza acestor costuri, interfețele parallele sunt de obicei limitate la conexiuni scurte ce au sub 1 m lungime.

Înțial, intreruperile INT 0Dh (IRQ 5) și INT 0Fh (IRQ 7) au fost proiectate pentru a deservi porturile parallele LPT1, LPT2 (*Line Printer*), însă imediat după aceasta, IBM a proiectat o interfață pentru imprimantă (*printer interface card*) care nu este compatibilă cu aceste intreruperi! Ca urmare, astăzi ele nu mai sunt utilizate pentru imprimante ci preponderent pentru plăci SCSI și plăci de sunet.

**INT 0Eh** → intreruperea de dischetă (IRQ 6 - *Diskette Drive interrupt*)

**INT 76h** → intreruperea de hard-disk (IRQ 14 - *Hard Disk Controller*)

Unitățile de dischetă și harddisk generează fiecare câte o intrerupere la încheierea unei operații de citire/scriere. Aceste intreruperi sunt utile pentru gestiunea aplicațiilor în sistemele de operare multitasking (OS/2, Linux, Windows): în timp ce are loc o operație I/O cu discheta sau hard-disk-ul, micropresorul poate executa părți din alte procese; când un disc și-a încheiat operația

currentă de citire/scriere va întrerupe acțiunea curentă a micropresorului pentru a-i semnala acestuia posibilitatea revenirii la procesul anterior.

**INT 70h** - The Real-Time Clock Interrupt (IRQ 8). Această intrerupere este activată de CMOS de 1024 ori/secundă în vederea asigurării bunei funcționări a ceasului real al sistemului.

**INT 75h** - intreruperea de unitate în virgulă flotantă (*FPU Interrupt* - IRQ 13) este o intrerupere generată de coprocesorul matematic la orice situație de excepție de tip virgulă flotantă (*floating-point exception*).

### 5.2.2. Exceptii

Datorită faptului că și acestea sunt intreruperi BIOS sunt clasificări care încadrează excepțiile la întreruperi hard (care sunt și ele BIOS). Nu este însă corect pentru că după tratarea unei intreruperi hard programul se reia întotdeauna, în timp ce după tratarea excepțiilor, de obicei programul NU se reia!

**INT 0** - intreruperea împărțirii la zero (*Zero Divide interrupt*). Întreruperea 0 este generată de fiecare dată când apare o așa numită *condiție de împărțire la zero*. INT 0 poate fi emisă în trei situații distincte:

i). depășirea rezultatului (câțulu) la împărțire atunci când utilizăm DIV sau IDIV;

```
mov ax,600
mov bh,2
(i) div bh ; se efectuează ax/bh, cu câtul în AL și restul în AH
```

Câtul ar trebui să fie 300 și să fie obținut în AL, însă valoarea 300 nu începe pe 1 octet în AL. Ca urmare, se va emite INT 0 cu mesajul de eroare “*Divide by zero*”. Într-un astfel de caz se recomandă efectuarea unei conversii prin lărgire care să asigure efectuarea corectă a împărțirii indiferent de valorile considerate:

```
mov ax,600
mov dx,0 ; sau cwd dacă se face conversie cu semn
mov bx,2
div bx ; se efectuează dx:ax/bx, cu câtul în AX și restul în DX
```

Acum valoarea 300 începe în AX și nu se mai emite INT 0.

ii). încercarea de efectuare a unei împărțiri la zero:

```
mov ax, 600
mov bh,0
div bh ; se va emite INT 0 deoarece se încearcă împărțirea la BH=0!
```

Instrucțiunea INTO are următorul efect:

Scrierea unui handler pentru INT 4 oferă programatorilor o modalitate facila pentru gestiunea aritmetică expusă de peisajii, se poate asigura tratarea adecvată a situației de depeșare.

**INT 4 (Overflow interrupt).** Aceasta interupează execuția în cadrul unor operațiuni aritmice. Mai précis, se emite când se execuțiază INTO (interrupt on overflow) și OF = 1. Dacă OF = 0, INTO se tragește în cursul depanării programelor.

**INT 3 (Breakpoint interrupt).** Aceasta interupează execuția (breakpoints) în cursul stabilită de puncte de intrupere a cărora de la începută de către depanătorul programelor.

**INT 2 - Interupeerea nemascabila (Non-Maskable Interrupt - NMI).** Interupere pot fi dezactivate (mascate) prin instrucțiunea CLI (Clear Interrupts). INT 2 este singura interupere ce nu poate fi mascată, ea fiind generată de fiecare dată când apare o condiție nemascabilă, ca de exemplu o eroare de memorie (memory parity error).

**INT 1 (Single Step).** Aceasta interupează imediată de procesor după fiecare instrucțiune săracă. Când depanătorul trimite deosebită emisiuni INT 1 după fiecare linie de cod mașină, dacă flagul TF = 1, se folosește la depanare, execuție pas cu pas ale programelor în reversire din rutina (spre deosebită de revizuire, astfel încât se va folosi instrucțiunea RET), aceasta scoate din stivă și regăsești de la începutul rutinei de revizuire.

**Observație:** De fapt totale interupeuri din care face parte pot fi activate și numărul interuperei expecifică la nivelul codului surșă a instrucțiunii INT n, unde n este deținătoarea alternativă pe care am dat-o acestora; interuperele software pot fi invocate și software, însă caracteristica lor de bază este că ele sunt emise în mod automat la numărul interuperei expecifică din care face parte interuperei softwarei. A se vedea și interuperele hardware pot fi invocate și interuperele hardware pot fi invocate și software, însă caracteristica lor de bază este că ele sunt emise în mod automat la numărul interuperei expecifică la nivelul codului surșă a instrucțiunii INT n, unde n este SOFT prin specificația „Divide by zero” și preda controlul SO MS-DOS.

.....

INT 0 ; se emite explicit de către programator INT 0

add ax,2

.....

iii). Emitează explicită accesă interuperei prin invocare software sub forma INT 0:

**INT 6 - (invalid opcode) – cod de instrucțiune ilegal.** Aceasta interupează se emite la încercarea de executie unui cod de instrucțiune inexistent. De exemplu:

a db 199 ; accesați linie de cod, deși la prima vedere ar trebui să aparțină numai

numi segment de date, poate apărea și în segmentul de cod, semnificația ei fiind în acest caz

interpretată valoari întregi (adică 199) de pe cod de instrucțiune, și că urmare se va emite la accesați linie INT 6, al cărei reprezentă în cod de instrucțiune validă și că urmărește un cod de instrucțiune ilegal. De exemplu:

add ax,2 ; ok, se executa fara problema

de executie unei valori întregi (adică 199) de pe cod de instrucțiune ilegal. Aceasta interupează se emite la încercarea

**Cap.5. Interupere.**

**12h** Returnează dimensiunea memoriei RAM. Apelul acestei întreruperi avea sens atunci când calculatoarele aveau o memorie de până la 64K. Acest lucru nu mai este valabil în zilele noastre, aşadar această întrerupere este oarecum depăşită.

**13h** Pune la dispoziție servicii de lucru cu harddisk-ul și cu discheta: resetarea unui disc (funcția 00h), obținerea stării unei operații făcute asupra discului (funcția 01h), citirea și scrierea de sectoare de pe un disc fix sau dischetă într-un (respectiv dintr-un) buffer din memoria internă (funcția 02h, respectiv 03h), verificarea sectoarelor (funcția 04h) și.a.m.d.

**14h** Permite accesul la porturile seriale. Această întrerupere pune la dispoziție funcții de inițializare a unui port serial (funcția 00h), transmitere sau primire a unui caracter (funcțiile 01h și 02h) și obținere a stării unui port serial (funcția 03h).

**15h** Pune la dispoziție funcții de acces la memoria extinsă, de citire a dispozitivelor de tip joystick și.a.m.d.

**16h** Această întrerupere oferă servicii de manipulare a tastaturii, dintre care: citirea unui caracter din buffer-ul tastaturii (funcția 00h), returnarea stării unor taste (Caps Lock, Scroll Lock, Num Lock, ... – funcția 02h).

**17h** Această întrerupere oferă servicii de manipulare a imprimantei: tipărire unui caracter (funcția 00h), inițializarea imprimantei (funcția 01h) și returnarea stării imprimantei (funcția 02h).

**18h** Activează interpretorul ROM BASIC. Astăzi mai puțin folosită, deoarece calculatoarele compatibile IBM nu mai includ acest interpretor.

**19h** Oferă servicii de încărcare a sistemului de operare. Efectul apelului acestei întreruperi este echivalent cu cel al apăsării combinației de taste Ctrl-Alt-Del.

**1Ah** Servicii legate de ceasul sistem. Există două astfel de servicii: citirea ceasului (funcția 00h) și setarea ceasului (funcția 01h).

**1Bh** Această întrerupere este apelată la apăsarea combinației de taste <CTRL/BREAK>. Ca efect al RTI corespunzătoare, în bufferul tastaturii se va pune CTRL-C ca următorul caracter. La citirea acestuia se va invoca întreruperea 23h, care termină execuția programului curent și redă controlul sistemului de operare.

**1Ch** Această întrerupere este apelată de 18.2 ori / secundă de RTI 8. Rutina de tratare a acestei întreruperi nu face nici o acțiune, lăsând posibilitatea utilizatorului de a scrie propria rutină de tratare. Aceasta este o întrerupere utilizator.

Adrese ale unor structuri de date BIOS (vezi Norton Guide pentru detalii):

**1Dh** Adresa zonei de parametri video.

**1Eh** Adresa zonei de parametri a unităților de dischetă.

**1Fh** Parametrii adaptorului grafic.

**41h** Parametrii harddisk-ului.

**50h** Accesarea memoriei CMOS.

#### Principalele întreruperi DOS sunt:

Principala întrerupere DOS este **21h**. Ea înmagazinează practic întreaga componentă BDOS a sistemului de operare DOS. Secțiunea următoare va fi destinată exclusiv acestei întreruperi.

Alte întreruperi DOS:

**20h** Aceasta este unul dintre apelurile care pot termina execuția unui program. Memoria ocupată va fi eliberată ca efect al acestui apel.

**25h** Permite citirea fizică de pe disc de la o anumită locație de memorie, începând cu un anumit sector, într-o anumită locație de memorie.

**26h** Permite scrierea fizică pe disc dintr-o anumită locație de memorie, începând cu un anumit sector.

**27h** Termină execuția programului curent lăsând rezidentă în memorie o parte sau întreg programul, astfel încât această zonă de memorie să nu fie suprascrisă de un alt program.

**28h** Întrerupere DOS *nedocumentată* pentru partajarea timpului. Asupra întreruperilor nedокументate vom reveni ulterior cu explicații.

**2Eh** (nedocumentată). Execută o comandă DOS ca și când ar fi dată de la prompter.

**2Fh** Funcțiile acestei întreruperi se ocupă cu: multiplexarea resurselor sistemului, gestiunea memoriei extinse (XMS) dacă aceasta există, controlul programelor TSR și.a.m.d.

**67h** Această întrerupere grupează toate serviciile necesare gestiunii memoriei expandate dacă SC este dotat cu o astfel de componentă hard.

**33h** Această întrerupere grupează toate funcțiile necesare lucrului cu mouse-ul.

Următoarele întreruperi furnizează o serie de adrese ale unor structuri DOS:

**22h** Adresa de terminare a programului.

**23h** Adresa handlerului care tratează tastarea <CTRL/BREAK>.

**24h** Adresa handlerului care tratează erorile critice apărute în execuție.

**33h** Returnaza codul (numarul) discutii folosit la incarcarea SO.

**19h** Returnaza codul (0=A, 1=B, ...) discutii implicit.

**Functii specifice discutii:**

**62h** Ofigne adresa de exceptie a PSP-ului in memorie.

**26h** Copiază PSP-ului programul curent la o adresa in memorie, apoi actualizează nouă PSP pentru a fi folosit de către un nou program.

**4Dh** Accesează funcția este folosita pentru obiectele codului de return al unui proces în lansat de către un program print apelul funcției 4Bh.

**31h** Termina execuția unui program lăsatându-l rezidient în memorie. Print intermediul lui AL se va transmite codul de return.

**4Ch** Termina execuția unui program introdându-se în comand.com sau în rutina apelantă, cu un număr cod de return (error level) setat în AL.

**4Bh** Încarcă un program pentru a fi executat sub controlul unui program existent. La terminarea executării programului apelat, controlul se întoarcе în programul apelant.

**Functii de gestiune a proceselor:**

**4Ah** Ajustează spațiul de memorie alocat.

**49h** Eliberează o zonă de memorie pentru a face disponibilă altor programe.

**48h** Aloca un bloc de memorie și returnează un pointer spre începutul acelui zone de memorie.

**Functii de gestiune a memoriei:**

Vom exemplifica, pe categoria, unde dintr-unicele DOS mai importante. O prezintare exhaustivă a lor ar necesita mult spațiu tipografic. Documentația DOS fac o astfel de prezenter, ca și o serie de programe de tip HELP, accesibile pe orice calculator.

În primul rând, trebuie să menționăm faptul că interfața DOS este 21h. Ea managerul de către funcție care să se interneze într-o altă.

Asa cum am mai arătat, principala interfață DOS este 21h. Ea managerul de către funcție care să se interneze într-o altă.

**Principalele funcții DOS**

**190 Arhitectura Calculatoarelor. Limbaul de assembly 80x86.**

**191 Cap.5. Interupeți.**

**1Ah** Selezază adresa zonei de transfer disc (DTA), aceasta zona urmăndă să folosească de operațiile care dimensiunile acelui zone este suficient de mare pentru o operatiile ce urmează a fi efectuate.

**1Bh** Furnizează informații despre tabelă de alocare a fizicelor (FAT).

**Functii specifice directorelor și fizicelor:**

**39h** Creează un nou director folosind discul și cala sa specificată.

**3Ah** Sterge un director specificând cala sa specificată.

**3Bh** Schimbă directorul curent în cel specificat.

**3Ah** Sterge un director specificând cala sa specificată.

**3Bh** Schimbă directorul curent în cel specificat.

**47h** Ofigne un string ASCII reprezentând cala sa specificată.

**56h** Schimbă numărul unui fizic.

**4Eh** Caută în directorul specificat sau în cel curent ultimul nume de fizic care se potrivește cu o specificare generică.

**4Fh** Caută în directorul specificat sau în cel curent ultimul nume de fizic care se potrivește cu o specificare generică.

**41h** Sterge un fizic din directorul specificat sau din cel curent.

**3Dh** Deschide un fizic din directorul specificat sau din cel curent. Dacă deschiderea să-a facut cu succes, returnează un identificator al fizicului (handle) pe 16 biți pentru accesarea fizicului de acum înainte.

**3Eh** Inchide un fizic care a fost deschis cu succes. Identificatorul fizicului devine astfel liber pentru a fi folosit ca identificator al altui fizic. Dacă fizicul a fost modificat, date și ora ultimei modificări vor fi actualizate.

**3Fh** Creează un anumit număr de octetii într-un fizic deschis cu succes.

**40h** Scrie un anumit număr de octetii într-un fizic deschis cu succes.

**19h** Intră în /iese în cadrul unei fizice.

**01h** Citește un caracter de la intrarea standard și îl afisează la ieșirea standard.

02h Afisează un caracter la ieșirea standard.

09h Tipărește un sir de caractere la ieșirea standard. Acest sir de caractere trebuie să conțină ca și marcat de sfârșit de sir caracterul '\$'. Acest caracter nu va fi tipărit.

0Ah Citește de la intrarea standard un sir de caractere până la tastarea lui *Enter*.

#### Informatii despre sistem:

30h Furnizează versiunea sistemului de operare.

38h Tratează parametrii dependenți de regiunea geografică.

2Bh Setează data curentă reținută de ceasul sistem.

2Dh Setează ora curentă reținută de ceasul sistem.

#### Diverse alte funcții:

35h Obține adresa unui handler de întrerupere, sub forma adresă de segment:offset

25h Modifică adresa unui handler de întrerupere.

44h Ansamblu de funcții destinat lucrului la nivel fizic cu diverse tipuri de periferice.

34h Returnează numărul proceselor curente active (nedокументată).

52h Atribuie valori unor variabile DOS (nedocumentată).

### 5.3. CÂTEVA OBSERVAȚII ASUPRA ÎNTRERUPERILOR 8086

a) Înșiruirea principalelor întreruperi 8086, precum și a principalelor funcții DOS relevă următorul fapt. Numai o parte dintre ele deservesc apariția unui eveniment neobișnuit, intern sau extern. Acestea, evident că vor fi activate la apariția evenimentului, fără a se putea prevedea momentul apariției. Vom spune despre handler-ele din această categorie că sunt activate prin evenimente.

În schimb, cealaltă parte a handler-elor oferă aceleasi servicii ca o bibliotecă de subprograme, apelabile de către programele utilizator. Activarea lor are loc atunci când programele o cer. Handler-ele din această categorie sunt activate prin instrucțiuni speciale de apel de întreruperi. În 5.4 vom trata aceste instrucțiuni.

#### Cap.5. Întreruperi.

b) Funcțiile DOS ale întreruperii 21h execută unele sarcini pe care le execută și alte întreruperi. De exemplu, funcția 4ch a întreruperii 21h - terminarea unui program, este realizată și de către întreruperea 20h. Din punct de vedere istoric, funcțiile DOS au apărut ultimele. Ele sunt mai performante și mai fiabile decât întreruperile similare, motiv pentru care în prezent sunt folosite aproape exclusiv numai acestea.

c) După cum s-a văzut deja, există așa-numitele *întreruperi nedocumentedate*. Acestea sunt rezervate spre folosire doar de către proiectanții DOS. Unele dintre ele sunt efectiv rezervate, altele au o serie de sarcini intermediare pentru alte întreruperi, iar altele au roluri mai mult sau mai puțin "obscure" pentru muritorii de rând. Aceste întreruperi sau funcții constituie ținta preocupărilor pentru mulți programatori pasionați. Nu trebuie uitat însă faptul că proiectantii sistemului de operare își rezervă dreptul de a folosi aceste întreruperi pentru dezvoltări ulterioare.

d) O serie de numere de întreruperi sunt în prezent neocupate. Ele pot fi ocupate de către utilizatori prin handlere proprii, după cum vom vedea în capitolul 6. Instrumentele de contaminare a programelor, cunoscute sub numele de "viroși", folosesc de multe ori aceste numere de întreruperi pentru scopurile lor distructive.

#### 5.4. INSTRUCȚIUNI SPECIFICE LUCRULUI CU ÎNTRERUPERI

După cum am arătat în 5.2, un handler de întrerupere poate fi apelat și direct prin intermediul instrucțiunii INT. Instrucțiunea INT, cu sintaxa:

INT n

provoacă activarea handlerului corespunzător întreruperii cu numărul n.

Ea realizează patru acțiuni succesive:

- punе în stivă flagurile;
- punе în stivă adresa FAR de revenire;
- punе 0 în flagurile TF și IF;
- apelează, prin adresare indirectă, handlerul asociat întreruperii.

De multe ori, în aplicații este utilă simularea acestei instrucțiuni sau a unei părți din ea. Iată o secvență care realizează simularea ei.

AdRTI dd ... ;Presupunem că conține adresa FAR a handlerului

```
-----  
pushf      ;Depune flagurile în stivă  
push cs    ;Segmentul adresei de revenire  
lea ax,REV  
push ax    ;Offsetul adresei de revenire  
xor ax,ax
```

Deplasament	Lungime	Semnificație	
00h	2	Codul instrucțiunii INT 20h - terminare program	
02h	2	Adresa de start a memoriei ocupate de program	
04h	1	rezervat	
05h	1	Codul instrucțiunii INT 21h - funcții DOS	
06h	2	Memoria disponibilă (număr octet) în cardul segmentului	
08h	2	rezervat	
0Ah	4	Adresa FAR a RTI 22h - adresa fizică a segmentului de revizuire din	
0Eh	4	Adresa FAR a RTI 23h - se utilizează pentru restituirea veciului handler dacă programul curent a destruit ceea ce a interrupterea 23h (CTRL+Break)	

După cum se observă, în PSP există o serie de zone care în prezent sunt mai puțin folosite. Ele au versiunile mai noi cu cele din prima versiune.  
folosește, dar sună destul de bună să asigure compatibilitatea programelor exechutabile DOS din CPM (de unde s-a inspirat MICROSOFT pentru DOS V1.0). Începând cu DOS V2.0, ele nu se mai folosesc, deoarece în primă versiune de DOS, pentru asigurarea compatibilității cu sistemul de operare folosindu-se la introducere în sistemă, începând cu DOS V3.0, și până la versiunea 4.0, încă nu există o soluție similară.

*Segment Prez.* În momentul închecării programului, însemna că în memorie este completată cu datele de la adresa DOS și în direct de către utilizator. În figura 5.1 este indicată structura tabelului de operare DOS care are 256 octet. Informațiile din PSP sunt utilizabile direct de către sistemul acest tabel special care sunt înregistrate în memoria lui în modul în care începe cu un anumit PSP (program

### 5.5.1. Preluxul unui program executabil (PSP)

Fisierul EXE sună foarte marți. Cu ajutorul lor se pot deschide programe complexe, de aceea elă trebuie să devină generală, pasărandu-se căre de tip COM numai acolo unde elă devin indisponibile.

Din punct de vedere istoric primul fisier este apărat de sistemul maxim 64 de octet.

În general fisierul mică având lungimea de maxim 64 de octet. În instația în limba maghiară, în ceea ce se referă la extensia de fișier, este extensia .COM. Fisierul de tip COM sună în general fisierul semnificativ de maximă ne vom ocupa de acesta după din urmă.

### 5.5. FORMATELE COM și EXE

Sub sistemul de operare MS-DOS există trei tipuri de fisieri care sunt lansabile în execuție: fisier

REV: - - - - -

Cap.5. Interuperi.

195

jmp ADRTI ;Salut indirect la handler

push ax ;Se amulează totale flagurile pentru a anula și IF și IF

popf ;Refacă flagurile

add ax,b ;dacă se banuitește că operanții adunării

jmp OF ;precedente ar putea provoca depășire la depășire.

De aceea, orice instrucțiune care ar putea provoca depășire este indicată se programă astfel:

- dacă OF = 0, atunci este echivalentă cu NOP (instrucțiunea inefективă, nu a apărut de departe arimetică);

- dacă OF = 1, atunci este echivalentă cu INT 4 (se apărăza handerul interuperei de flagul OF);

instrucțiunea INT0 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

- dacă OF = 0, atunci este echivalentă cu INT 4 (se apărăza handerul interuperei de

depeșării arimetică);

instrucțiunea INT1 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT2 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT3 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT4 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT5 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT6 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT7 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT8 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INT9 (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTA (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTB (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTC (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTD (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTE (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTF (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTG (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTH (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTI (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTJ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTK (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTL (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTM (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTN (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTO (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTP (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTQ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTR (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTS (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTT (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTU (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTV (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTW (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTX (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTY (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTZ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTD (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTL (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTM (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTN (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTP (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTQ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTR (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTS (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTT (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTU (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTV (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTW (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTX (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTY (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTZ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTD (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTL (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTM (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTN (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTP (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTQ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTR (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTS (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTT (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTU (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTV (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTW (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTX (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTY (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTZ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTD (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTL (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTM (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTN (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTP (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTQ (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTR (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTS (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTT (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTU (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTV (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTW (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTX (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTY (care nu are operații) se comportă în două moduri, în funcție de valoarea

depeșării arimetică;

instrucțiunea INTZ (care nu are operații) se comportă în două moduri, în funcție de valoarea

12h	4	Adresa FAR a RTI 24h - se utilizează pentru restaurarea vechiului handler dacă programul curent a deturnat cumva întreruperea 24h ( <i>Critical error interrupt handler</i> )
16h	22	rezervat
2Ch	2	Adresa de segment a mediului DOS, unde se găsesc variabilele de sistem MS-DOS
2Eh	46	rezervat
5Ch	32	FCB1 și FCB2, câte 16 octeți pentru accesarea fișierelor standard de intrare și ieșire (se evita utilizarea lor la momentul actual – păstrați pentru compatibilitate)
7Ch	4	rezervat
80h	1	Lungimea cozii liniei de comandă
81h	127	Coada liniei de comandă - astfel se pot accesa din limbaj de asamblare parametrii transmiși în linia de comandă la lansarea în execuție a programului

Fig. 5.1. Structura unui PSP

Fiecare program, pe lângă codul lui propriu-zis mai are o zonă de memorie în care este descris contextul în care lucrează programul. Acest context include, printre altele, informații referitoare la:

- numele discului implicit;
- numele directorului implicit;
- calea spre interpretorul de comenzi COMMAND.COM etc.

Această zonă de context este un segment numit *mediu (environment)* și PSP-ul conține un pointer la începutul acestuia (la deplasament 2ch). Segmentul de mediu conține siruri de caractere ASCII de forma:

NumeParam	=	ValoareParam	0
-----------	---	--------------	---

*NumeParam* este un identificator reprezentând o variabilă de mediu (PATH, PROMPT, ABC etc). Semnul "=" separă identificatorul de sirul de caractere *Valoareparam*. Construcția se termină cu un octet conținând valoarea zero.

Se știe că o comandă DOS are forma:

...>numecomanda arg1,...,argn

Porțiunea arg1,...,argn se numește *coada liniei de comandă* și ea este memorată în jumătatea a două a tăbelei PSP.

### 5.5.2. Structura unui program EXE

Un program de tip EXE poate avea oricără segmente de tip cod, date sau stivă. Acestea pot fi prezente toate în memorie sau numai o parte dintre ele. Prezența optională a unor segmente se realizează prin acesta-nunțul mecanism *overlay*, implementat de principalele medii de programare. În continuare vom trata numai cazul când toate segmentele sunt prezente în memorie.

În fiecare moment al execuției unui program este activ un segment de cod, un segment de date și un segment de stivă. Toate segmentele sunt plasate de regulă după PSP, însă ordinea lor nu este importantă. Stiva nu are implicit 64 octeți ca la programele de tip COM, ci poate fi dimensionată de către programator. În figura 5.2 este ilustrat un program EXE care are căte un singur segment din fiecare tip și sunt plasate în ordinea cod, date, stivă.

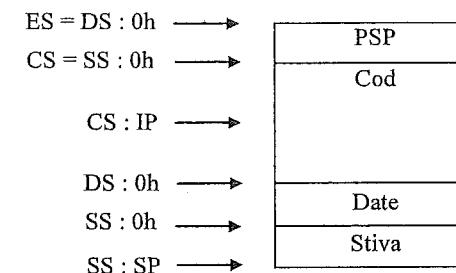


Fig. 5.2. Structura unui program EXE în memorie

În momentul încărcării în memorie a unui program EXE, se creează PSP, și se depun în memorie segmentele. Se încarcă CS cu adresa segmentului de cod ce trebuie să fie activ primul. Prima instrucție executabilă poate să apară oriunde în segmentul de cod punctat de CS, CS:IP va puncta spre această instrucție.

La încărcare ES și DS se inițializează și punctează la începutul PSP. Ulterior, programatorul are obligația să încarce (cel puțin) registrul DS cu adresa segmentului de date curent. De aceea în figura 5.2, DS:0h apare în două poziții.

Registrul SS se va încărca automat cu adresa de început a segmentului de stivă, dacă este declarat un astfel de segment. Dacă nu, atunci SS va primi aceeași valoare ca și CS, considerându-se o stivă de lungime 64 Ko. Registrul SP va puncta spre ultimul cuvânt al segmentului de stivă. Din această cauză, în figura 5.2 SS:0h apare în două poziții.

Dăm în continuare programul E.ASM, care tipărește numărul unităților de dischetă din sistem. El va avea declarată o stivă de 512 octeți (200h).

tipărire pe ecran a mesajului t2

mov ah, 9 ;afficher le message

```
Start:    mov    ax,4C00h
          int    21h
          end   code
```

La lansarea în execuție cu TD a lui E.EXE, registril au avut valoile:

După execuția primei rând două instrucțiuni, registru DS a primit valoarea 62BB.

Un fel de program, în format EXE are dimensiuni mult mai mari decât echivalența lui în format COM. Această faptă se petrece deoarece fiecare fișier EXE trebuie să conțină și segmentele de date și de instrucțiuni ale programului EXE, pentru a fi executat în memoria la interfață cu un alt fișier EXE. Acestea sunt segmentele curente la interfață și sunt definite ca fișiere separate.

Deplasament	Lumegime	Semnificație	Deplasament	Lumegime	Semnificație
00h	2	Semnatura EXE: 5A4Dh (codul MZ, Marek Zukrowski)	02h	2	Lumegime fără mod 512
04h	2	Lumegime fără div 512	06h	2	Numerul total al adreselor relocalabile (NAR)
08h	2	Lumegime în paragrafe (multiplii de 16) a antetului	0Ah	2	Numerul minim de paragrafe cunoscute în plus (0000)
0Ch	2	Numerul maxim de paragrafe cunoscute în plus (FFE)	0Eh	2	Pozitia relativă a segmentului SS (paragrafe)
10h	2	Va lăsa multă a registrului SP	12h	2	Suma de control (Checksum) a fizierului
14h	2	Va lăsa multă a registrului DP	16h	2	Pozitia relativă a segmentului CS (paragrafe)

start:	assume cs:code, ds:date, ss:stiva	code segment
int 11h	mov ax, date	mov ax, ds
int 0 din configurația lui AX precizează dacă există unități de	;dacă există instalație (valoarea 1) sau nu (valoarea 0)	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
dischete în instalație	;dacă valoarea la zero valoarea bitului 0	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
jeftinim în bx valoarea lui AX, pentru a testa configurația obținută	;jeftinim la zero valoarea bitului 1-15 din configurația lui AX,	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de
pasăram neșchiință valoarea bitului 0	;jeftinim la zero valoarea bitului 1-15 din configurația lui AX,	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
dischete în instalație	;dacă valoarea la zero valoarea bitului 0	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
int 21h	lea dx, t1	int 21h
int 9 ah, 9	mov dx, t1	int 21h
NUExista	z	and bx, ax
bx, ax	mov bx, ax	mov bx, ax
and bx, ax	and bx, ax	and bx, ax
jeftinim în bx valoarea lui AX, pentru a testa configurația obținută	;jeftinim la zero valoarea bitului 1-15 din configurația lui AX,	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de
dischete în instalație (valoarea 1) sau nu (valoarea 0)	;dacă valoarea la zero valoarea bitului 0	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
int 0 din configurația lui AX precizează dacă există unități de	;dacă există instalație (valoarea 1) sau nu (valoarea 0)	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
dischete în instalație	;jeftinim la zero valoarea bitului 0	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de
int 10h	mov ax, 0	int 10h
bx, ax	mov bx, ax	mov bx, ax
bx, ax	and bx, ax	and bx, ax
jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de
dischete în instalație (valoarea 1) sau nu (valoarea 0)	;dacă valoarea la zero valoarea bitului 0	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
int 0 din configurația lui AX precizează dacă există unități de	;dacă există instalație (valoarea 1) sau nu (valoarea 0)	;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
dischete în instalație	;jeftinim la zero valoarea bitului 0	;jeftinim la zero valoarea bitului 0, înseamnă că nu există nici o unitate de
int 21h	int 21h	int 21h
mov ah, 02h	mov dl, al	mov dl, al
ah, 02h	;vom tipări caracterul astfel obținut cu funcția 02h a interrupterii 21h	;vom tipări caracterul astfel obținut cu funcția 02h a interrupterii 21h
code segment	assume cs:code	code segment

18h	2	Adresa disc a tabelii de relocare (TR) (uzual 1Ch)
1Ah	2	Număr de suprapunere (pentru overlay) (2 octeți)
1Ch	?	Zonă rezervată, parte a antetului
TR		Tabela de relocare: Offset1 Segment1 ... OffsetNAR SegmentNAR

Fig. 5.3. Structura antetului EXE pe disc

Este util să detaliem puțin mecanismul de încărcare și lansare în execuție a unui program EXE. Ordinea segmentelor este fixată de către editorul de legături. Prima acțiune a încărcătorului este crearea PSP. Urmează apoi încărcarea segmentelor. Ele vor fi plasate în ordinea dată de către editorul de legături. De regulă, segmentele sunt plasate începând de la PSP+100h (deci imediat după PSP). Să notăm cu StartSeg adresa de unde începe încărcarea segmentelor.

Așa cum am văzut în capitolul 3, unele instrucțiuni citează ca operand un nume de segment. De exemplu, secvența:

```
mov ax, SegmentDeDate
mov ds, ax
```

Încarcă registrul DS cu adresa segmentului cu numele SegmentDeDate. Editorul de legături plasează în codul primei instrucțiuni mov valoarea care localizează segmentul respectiv. În urma încărcării segmentelor în memorie, valorile prin care se identifică segmente trebuie mărite cu valoarea StartSeg.

Acest proces poartă numele de operatie de relocare. Aceasta este necesară deoarece deplasamentele sunt determinabile ca și constante la momentul asamblării, însă adresele de segment nu. Ca urmare, elementele relocabile sunt de fapt operanții din cadrul programului care reprezintă adrese de segment și care trebuie acum ajustate cu o valoare rezultată din plasamentul concret în memorie al programului EXE (valoarea StartSeg). De exemplu în cadrul instrucțiunilor:

```
mov ax, data           sau          mov bx, seg a
```

elementele relocabile sunt data și respectiv seg a.

Octeții 6 și 7 din antet conțin numărul total de astfel de citări ale numerelor de segmente (număr notat NAR – Numărul Adreselor de Relocare - în figura 5.3). Octeții 24 și 25 (adresa 18h) din antet indică poziția în antet (notată TR în figura 5.3), identică cu poziția în fișierul disc, a începutului unei așa numite tabele de relocare. În această tabelă există câte o adresă FAR pentru fiecare citare a unui nume de segment.

Checksum are sarcina de a verifica și păstra integritatea antetului disc. Astfel, se consideră întregul fișier ca o succesiune de cuvinte. Conținuturile negate ale acestor cuvinte sunt însumate modulo 65536, rezultatul obținut fiind acest checksum. La încărcarea programului în memorie, sistemul de operare DOS repetă acest calcul și compară cu ceea ce este în checksum, refuzând încărcarea în caz de neconcordanță.

După acest control, se creează PSP. Apoi sunt încărcate segmentele și se actualizează adresele relocabile. În sfârșit, sunt încărcate din antet regiștrii CS, SS, IP și SP așa după cum am arătat mai sus. Ca efect al încărcării CS:IP programul este lansat în execuție.

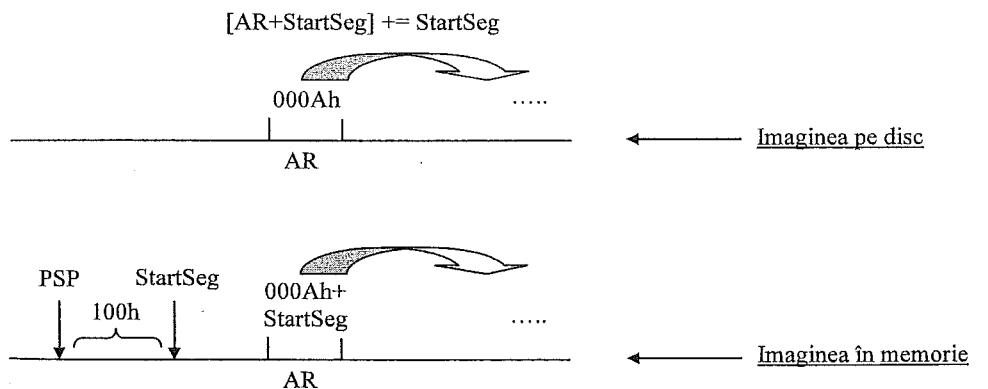
Descriind într-o manieră algoritmică procesul de încărcare în memorie a unui program EXE, putem identifica următorii pași:

1. Se creează în memorie o structură de date PSP (Program Segment Prefix) de lungime 256 de octeți.
2. Se alege o adresă de la care începând se va încărca programul (de obicei aceasta este sfârșitul PSP). Fie această adresa StartSeg:

$$\text{StartSeg} := \text{adresa PSP} + 100h$$

3. Se încarcă la StartSeg porțiunea din fișierul EXE de după antetul de pe disc.

4. Se efectuează operația de relocare: pentru fiecare adresă de relocare AR (adică pentru fiecare adresă unde se găsește un element ce trebuie relocat), la conținutul adresei AR+StartSeg se adună StartSeg; în notație C acest lucru se poate exprima



2.imediat după declarări de segment trebuie să apară o direcțivă:

unde <nume> este același astăzi pentru CS căt și pentru DS.

6. Se înfișează registri astfel:

3. Datele pot fi plasate oriunde între instrucțiuni, singura condiție (depinzând numai de izolarea zonei de date prin instrucțiuni de salt corespunzătoare).

4. Registri de segment sunt inițializati automat (sa în figura 5.4), deci utilizatorul nu mai trebuie să încarcă cu valori inițiale (în sensul celor discutate în 3.1.2).

5. În casă programul nu trebuie să apară elemente relocabile, adică operații numere de tiparete numărăți unităților de dischete din sistem. Textul programului COM cară face acest lucru este dat mai jos.

Să considerăm programul echivalent COM al celui EXE din secțiunea anterioră, care să fie:

```
start:
    code segment
        assume cs:code,ds:code
        org 100h
    db 'Nu există unitate de dischete în sistem $.
    db 'Unitate de dischete $.
    db 'In sistem există $.
    jmp incrupt
incrupt:
    mov bx,1
    mov dx,ax
    and bx,1
    jz NUexistă
    mov ax,1
    int 11h
    mov bx,ax
    and bx,1
    jz NUexistă
    mov ah,9
    int 21h
    mov ah,4ch
    int 21h
    db 0dh,0ah,'Aşa se evita interferarea datelor cu codul
    db 0dh,0ah,'Nu există unitate de discheta instalată în sistem $.'
```

În momentul înărcirării în memoria a unui program COM se crează PSP, se inițializează toți registrii de segment ca în figura 5.4 și se dă controlul instrucțiunii altărie la adresa 100h, adică primul octet care se adă după PSP. Poziția de stivă SP are cea mai mare adresă posibilă, adică FFHh, în ideea că programul are 64 kocetă iar stiva spore adresele FFFh, în cadrul aceluiași segment; în consecință directiva assume este de forma:

„Programul trebuie să conțină un singur segment (toate refeririile la cod, date și stivă se fac pentru ca un program scris în limbaj de assembler să fie de tip COM trebuie îndepărtate ultimă rețire):

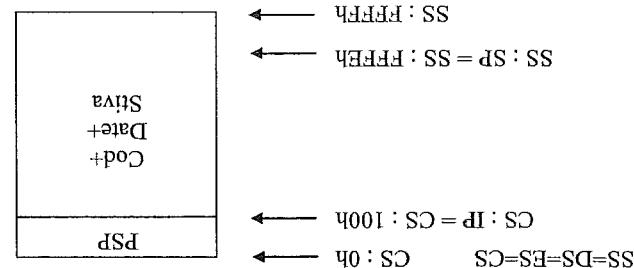
In momentul înărcirării în memoria a unui program COM se crează PSP, se inițializează toți registrii de segment ca în figura 5.4 și se dă controlul instrucțiunii altărie la adresa 100h, adică primul octet care se adă după PSP. Poziția de stivă SP are cea mai mare adresă posibilă, adică FFHh, în ideea că programul are 64 kocetă iar stiva spore adresele FFFh, în cadrul aceluiași segment; în consecință directiva assume este de forma:

assume cs:<nume>,ds:<nume>

„Programul trebuie să conțină un singur segment (toate refeririile la cod, date și stivă se fac în cadrul aceluiași segment); în consecință directiva assume este de forma:

assume cs:<nume>,ds:<nume>

Fig. 5.4. Imaginea unui program COM în memorie



Un fizier de tip COM are o structură simplă. El conține imaginea (binară) a configurației de pe disc! Numai programul EXE au ilustrat în memoria după PSP pentru a obține un program lansabil în excepție. În figura 5.4 este înscrădit în memoria după PSP care trebuie să fie prima instrucție executabilă. Numai accesul de la segment este similar cu valoarea inițială (în sensul celor discutate în 3.1.2).

### 5.3. Structura unui program COM

CS = CS relativ (valoarea din antetul EXE) + StartSeg  
IP = IP inițial (valoarea din antetul EXE)  
SS = SS relativ (valoarea din antetul EXE) + StartSeg  
SP = SS inițial (valoarea din antetul EXE)  
ES = DS = adresa de încrepută PSP

5. Se aloca dacă este posibil memoria necesară în plus în funcție de valoare din antet (de obicei pentru heap).

6. Se înfișează registri astfel:

7. Se înlocuiează calculatorul.

8. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

9. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

10. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

11. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

12. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

13. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

14. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

15. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

16. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

17. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

18. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

19. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

20. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

21. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

22. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

23. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

24. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

25. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

26. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

27. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

28. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

29. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

30. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

31. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

32. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

33. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

34. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

35. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

36. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

37. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

38. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

39. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

40. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

41. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

42. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

43. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

44. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

45. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

46. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

47. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

48. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

49. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

50. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

51. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

52. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

53. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

54. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

55. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

56. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

57. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

58. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

59. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

60. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

61. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

62. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

63. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

64. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

65. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

66. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

67. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

68. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

69. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

70. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

71. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

72. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

73. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

74. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

75. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

76. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

77. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

78. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

79. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

80. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

81. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

82. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

83. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

84. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

85. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

86. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

87. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

88. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

89. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

90. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

91. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

92. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

93. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

94. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

95. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

96. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

97. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

98. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

99. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

100. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

101. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

102. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

103. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

104. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

105. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

106. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

107. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

108. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

109. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

110. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

111. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

112. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

113. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

114. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

115. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

116. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

117. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

118. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

119. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

120. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

121. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

122. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

123. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

124. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

125. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

126. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

127. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

128. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

129. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

130. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

131. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

132. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

133. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

134. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

135. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

136. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

137. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

138. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

139. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

140. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

141. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

142. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

143. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

144. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

145. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

146. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

147. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

148. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

149. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

150. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

151. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

152. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

153. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

154. Înainte de a porni calculatorul, trebuie să se aplice următoarele directive:

```

and al, 0C0h
mov cl, 6
shr al, cl
add al, 1+'0'
mov dl, al
mov ah, 02h
int 21h

```

```

mov ah, 9
lea dx, t2
int 21h
jmp Sfarsit

```

NuExista:

```

mov ah, 9
lea dx, t3
int 21h

```

Sfarsit:

```

mov ax, 4c00h
int 21h

```

code ends

end start

Compilarea programului se face cu comanda ...>TASM C

Editarea de legături se face cu comanda ...>TLINK C/T

Un program de tip EXE poate fi transformat într-unul de tip COM echivalent (dacă sunt respectate condițiile de mai sus), cu ajutorul comenzi DOS EXE2BIN:

...>EXE2BIN C.EXE C.COM

Dacă programul EXE nu respectă condițiile impuse unui COM și are spre exemplu elemente relocabile, comanda Tlink va furniza eroarea:

*"Fatal: Cannot generate COM file: segment relocatable items present".*

#### 5.5.4. Depanarea programelor .EXE și .COM

În ceea ce privește depanarea unui program .EXE, folosirea opțiunilor /zi pentru asamblare și /v pentru linkeditare,

```

>tasm /zi prb
>tlink /v prb

```

provoacă includerea de informații simbolice pentru depanare la nivelul codului obiect și respectiv la nivelul celui executabil. Includerea informației simbolice înseamnă refinarea asocierii adresă de memorie - nume simbolic pentru fiecare etichetă din program. Acest lucru înseamnă că în cadrul unei sesiuni de depanare putem identifica referirea la o etichetă prin numele său și nu doar prin adresa care o reprezintă. În mod implicit, fără specificarea opțiunilor de mai sus, asamblorul nu include informațiile simbolice în formatul obiect pe care îl generează.

De exemplu, dacă avem definită variabila

```
a dw 5
```

fișierelor .obj și .exe vor conține informația simbolică ce face asocierea între numele a și deplasamentul la care se află această variabilă (de exemplu 0005h). Desigur că vom prefera ca în momentul depanării să vedem simbolul a mai degrabă decât deplasamentul acestuia. Pentru instrucțiunea:

```
mov ax, a
```

vom obține astfel la depanare:

```

mov ax, a
în loc de
    mov ax, [0005h]

```

Includerea acestor informații simbolice pentru depanare în fișierul .exe face ca dimensiunea acestuia să fie mai mare decât dacă nu ar conține aceste informații.

În ceea ce privește depanarea unui program .COM, care nu poate avea o dimensiune mai mare de 64 kocete (din care 256 octeți sunt folosiți pentru PSP), dimensiunea acestuia nu permite includerea informațiilor simbolice pentru depanare. Aceasta este motivul pentru care depanarea unui program .COM nu se poate face la nivel simbolic.

Prezentăm în continuare două programe echivalente semantic care efectuează suma a două numere. Pe o parte avem fișierul pexe.asm din care vom obține un program .EXE și fișierul pcom.asm din care vom obține un program .COM. Informațiile obținute la depanare sunt prezentate comparativ.

```
assume cs:code, ds:code
```

```
a db 5
```

```
b db 7
```

```
data segment
```

```
code segment
```

```
pc0m.asm
```

```
code segment
```

```
assume cs:code
```

```
org 100h
```

```
start:
```

```
jmp RealStart
```

```
a db 5
```

```
code segment
```

```
assume cs:code
```

```
end ends
```

```
data ends
```

```
start:
```

```
jmp RealStart
```

```
a db 7
```

```
code segment
```

```
assume cs:code
```

```
mov ds, ax
```

```
RealStart:
```

```
mov al, a
```

```
add al, b
```

```
mov al, b
```

```
add al, a
```

```
mov ax, 4C00h
```

```
int 21h
```

```
code ends
```

```
end starts
```

```
code ends
```

```
RealStart:
```

```
assume cs:code, ds:code
```

```
org 100h
```

```
jmp RealStart
```

```
a db 5
```

```
b db 7
```

```
mov ax, 4C00h
```

```
int 21h
```

```
code segment
```

```
assume cs:code
```

```
org 100h
```

```
jmp RealStart
```

```
a db 5
```

Secvența de cod anterioară dezactivează intreruperile pe durata modificării adresei handler-ului pentru a preveni un apel accidental pe durata modificării acesteia. Aceasta deoarece în cazul în care se generează un apel la intreruperea pe care o modificăm, exact între cele două instrucțiuni MOV WORD PTR, transferul se face la noul deplasament, însă în cadrul vechiului segment. Dezactivarea intreruperilor (sti) este necesară atunci când se modifică rutinele de tratare a intreruperilor ce pot fi generate asincron. În această categorie intră de obicei intreruperile ce deservesc un echipament hardware, intreruperile de ceas, etc. Acestea sunt generate automat de către sistem la apariția unui eveniment (ceas, tastatură etc). Ele pot apărea în orice moment și de aceea se numesc *asincrone*. După actualizarea noii adrese a rutinei de tratare a intreruperii se reactivează intreruperile pentru a permite funcționarea corectă a sistemului. Având în vedere că adresa unui handler se reprezintă pe 4 octeți (un cuvânt pentru offset și un cuvânt pentru adresa de segment), rezultă că deplasamentul în cadrul tabelei pentru intreruperea 77h se află la adresa 4\*77h. Conform convenției de reprezentare a datelor în memorie (octetul cel mai nesemnificativ se află la adresa mai mică) înseamnă ca începând cu deplasamentul 4\*77h vom stoca offset-ul nouui handler, iar la adresa 4\*77h+2 adresa de segment a acestuia.

Modificarea adresei RTI cu ajutorul funcției DOS 25h se face printr-o secvență de cod similară cu cea prezentată mai jos. Vom modifica din nou RTI 77h:

```
push ds           ;salvează registrul DS întrucât îl vom modifica
mov ax, 2577h    ;AH – 25h, AL – numărul intreruperii
mov dx, seg NewHandler
mov ds, dx
mov dx, offset NewHandler
int 21h
pop ds           ;reface registrul DS (înapoi spre segmentul de date)
```

Secvența de cod de mai sus depune în registrul DS adresa de segment a nouui handler, în DX deplasamentul nouui handler, în AL numărul intreruperii și apeleză funcția DOS 25h. În acest caz funcția DOS 25h dezactivează intreruperile în decursul modificării adresei RTI. Chiar dacă această secvență de cod pare mai complicată decât cea care accesează direct tabela vectorilor de intrerupere, ea este mai sigură. Aceasta deoarece multe dintre aplicațiile DOS monitorizează modificările aduse tabelii vectorilor de intrerupere prin intermediul DOS (funcția 25h). Modificarea directă a tabelii de vectori scurt-circuitează mecanismul de monitorizare și astfel aceste aplicații nu își mai pot da seama de modificările apărute, fapt ce poate duce la funcționarea lor incorrectă.

Pentru a putea refa, sau atunci când este nevoie apela handler-ul original al unei intreruperi, este necesar să salvăm adresa acestuia înainte de a face redirectarea spre rutina noastră de tratare. Următoarele două secvențe de cod ilustrează modul în care putem realiza acest lucru, atât prin accesul direct la tabela vectorilor de intrerupere cât și prin utilizarea funcției DOS 35h:

```
oldint77 dd ?      ;rezervare dublu cuvânt pt. memorarea adresei vechii RTI
...
...
```

```
mov ax, 0
mov es, ax
cli
mov ax, word ptr es:[4*77h]      ;depune offset-ul handler-ului original în AX
mov word ptr cs:oldint77, ax      ;salvează offset-ul handler-ului original
mov ax, word ptr es:[4*77h+2]
mov word ptr cs:[oldint77+2], ax  ;salvează adresa de segment a handler-ului original
sti
```

Presupunând că am rezervat spațiu de stocare pentru adresa vechiului handler în segmentul de cod (oldint77), secvența de mai sus salvează în dublul cuvânt de la adresa oldint77, adresa rutinei de tratare a intreruperii 77h. Același lucru se poate face folosind funcția DOS 35h:

```
oldint77 dd ?
...
...
mov ax, 3577h
int 21h
mov word ptr cs:oldint77, bx      ;se obține în ES:BX adresa handler-ului curent
mov cs:[oldint77+2], es
```

În general există puține cazuri în care înlocuim complet funcționalitatea unei rutine de tratare de intrerupere. De cele mai multe ori modificăm doar comportamentul unei funcții a intreruperii sau modificăm comportamentul în anumite cazuri speciale (de exemplu rescriem handlerul intreruperii de tastatură 09h pentru a putea intercepta activarea unor anumite combinații de taste). În toate aceste cazuri vom trata prin codul rescris cazul care ne interesează, iar pentru restul scenariilor posibile vom apela handler-ul original al intreruperii. Acesta este un alt motiv pentru care este necesară salvarea adresei handler-ului original. Procesul duce la un mecanism de *înlănțuire* a apelurilor către handler-ele de intrerupere. Acest mecanism este extrem de util pentru buna funcționare a programelor *TSR* (*Terminate and Stay Resident*) după cum vom vedea în continuare. Această partajare/înlănțuire a vectorilor de intrerupere este simplu de implementat în cazul în care fiecare RTI salvează adresa handler-ului vechi de intrerupere. Pentru a apela vechiul handler de intrerupere atunci când acesta este salvat în variabila dublucuvânt oldint77 vom proceda în felul următor:

```
pushf
call dword ptr cs:oldint77      ;plasarea flag-urilor pe stivă
...
iret                          ;cod executat după apelul handler-ului original
                               ;terminarea execuției handler-ului curent (cel nou) și
                               ;întoarcerea în contextul aplicației intrerupte
```

în cazul în care dorim să apelăm vechiul handler și apoi să continuăm execuția handler-ului nostru (folosind eventual efectul produs de handler-ul original), sau:



Nativ MS-DOS nu poate rula mai multe aplicații simultan (este un sistem de operare monotask). MS-DOS întotdeauna încarcă în memorie programul de executat la adresa indicată de pointerul spre *Spațiul de Memorie Liberă* și îi alocă întreaga zonă de memorie liberă, până la adresa 0BFFFh. Practic întreaga memorie RAM disponibilă este alocată programului în curs de execuție. Harta de alocare a memoriei MS-DOS atunci când în sistem rulează o aplicație este prezentată în figura 6.2.

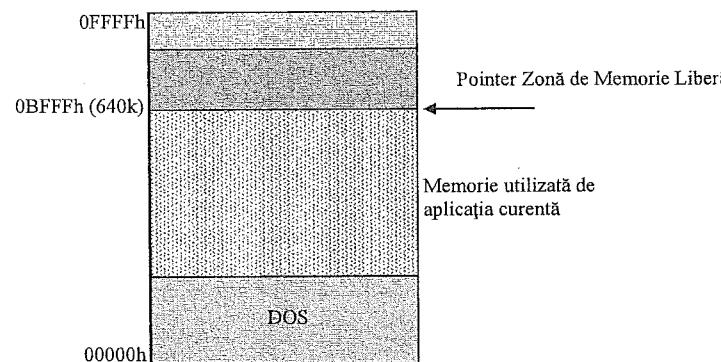


Fig. 6.2. Harta de memorie a sistemului de operare DOS cu o aplicație activă în sistem.

Atunci când o aplicație se termină prin invocarea funcției DOS 4Ch, sistemul de operare eliberează întreaga zonă de memorie alocată acesteia și resetează pointerul *Zonă de Memorie Liberă* imediat deasupra componentei MS-DOS din zona joasă de memorie. Funcția DOS 31h schimbă comportamentul la terminarea unei aplicații în modul ilustrat în figura 6.3.

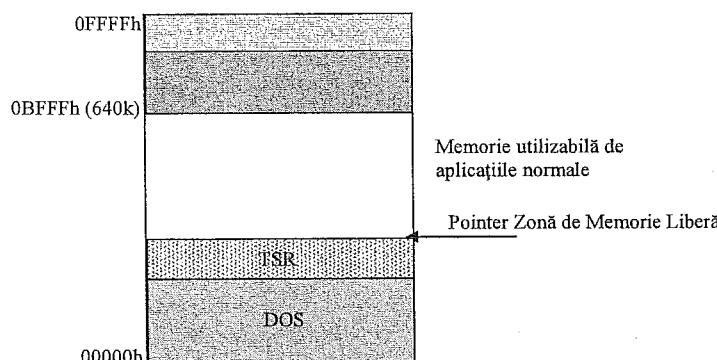


Fig. 6.3. Harta de memorie DOS după terminarea unei aplicații rezidente.

În acest caz DOS face o excepție: nu eliberează întreaga cantitate de memorie. Acest apel de tip *terminate and stay resident* menține alocat în memorie blocul de la începutul zonei alocate aplicației având dimensiunea stabilită în registrul DX. Dimensiunea acestui bloc de memorie este specificată în paragrafe (un paragraf = 16 octeți). La apelul funcției 31h, DOS setează de fapt pointerul *Zonă de Memorie Liberă* astfel încât să indice locația de memorie aflată la  $DX * 16$  octeți „deasupra” PSP-ului programului. La execuția unei alte aplicații în sistem, DOS va aloca doar memoria care începe la noua poziție a pointerului spre *Zona de Memorie Liberă* indicată în figura 6.3. Acest lucru permite protejarea spațiului de memorie în care se află încărcat programul TSR. Harta alocării memoriei după încărcarea unui program TSR și a unei aplicații normale DOS este ilustrată în figura următoare:

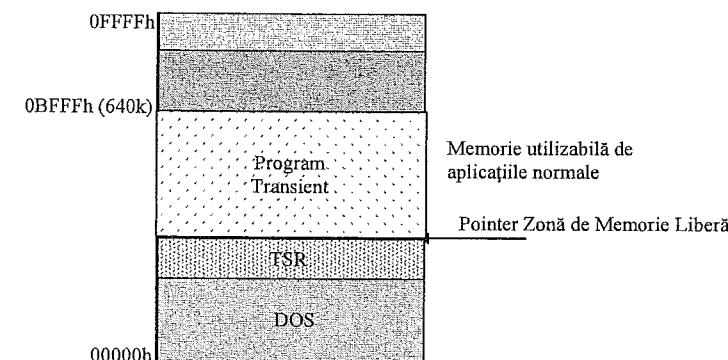


Fig. 6.4. Harta alocării memoriei DOS după încărcarea unui TSR și a unei aplicații tranziente.

La terminarea execuției aplicației tranziente (care este încărcată „peste” TSR), DOS va elibera doar memoria alocată acesteia, protejând încă o dată spațiul de memorie alocat TSR-ului.

Aspectul cel mai important la utilizarea funcției DOS 31h este calcularea corectă a numărului de paragrafe care trebuie să rămână rezidente. Cea mai mare parte a programelor TSR conțin două secțiuni: o secțiune *tranzientă* și o secțiune *rezidentă*. Partea tranzientă conține datele, programul principal și rutinele de suport care se execută atunci când lansăm programul în linie de comandă. Această porțiune de cod nu se mai execută în general niciodată după instalarea programului TSR în memorie. În consecință ea nu trebuie să rămână alocată în memorie odată faza de instalare terminată. Fiecare octet care rămâne inutil în memorie la instalarea unui TSR este de fapt un octet mai puțin pentru restul programelor care vor rula în sistem.

Partea rezidentă este cea care rămâne în memorie și implementează funcționalitățile oferite de TSR. Întrucât PSP-ul este construit în memorie de către sistemul de operare imediat înaintea primului octet al programului, dimensiunea acestuia trebuie considerată atunci când calculăm numărul de paragrafe rezidente. Harta memoriei unui program TSR încărcat trebuie să fie una similară cu cea ilustrată în figura 6.5. Pentru ca un TSR să poată funcționa corect este necesar să organizăm codul

SR-SR-unit sunt cu mic excepții hardware separate de tramele de interrupție. IFSK-unit active sunt în general mult mai puțin complexe decât cele pasive sunt handlele ale unor interrupții hardware separate de tip *latch*.

SR-Ultile pasvive reprezentanta echivalență implementare la nivel DOS sau BIOS. De exemplu dacă dorm să edificăm locate caracterele trimise de o aplicație imprimantă vom redirecționa interoperația cu vom intercepța toate datele destinate imprimantei. Putem să adăugăm funcționalitatea noi pentru înlocuirea în lanțul de handle ale vechiorului de exemplu putem implementa o funcție BIOS nouă pentru interfață corespunzătoare. De exemplu în lanțul de handle ale vechiorului de interrupție corespunzător. Deoarece interpretarea unei RTT este nouă cod de funcție BIOS nouă implementată în BIOS și vom înlocui interoperația cu interfața de la altă parte TSR-ură poate fi implementată în BIOS. Un exemplu de servicii implementate în BIOS este interfața unei RTT pe care se utilizează un vector neutralizat de BIOS. În acestă situație este implementarea serviciilor de gestiune a mouse-ului. Acestea sunt implementate în BIOS în același mod ca și celelalte servicii.

Întrupere software de timp) care este apărată automat de către sistem de 18.2 ori pe secundă.

om da în contimură un exemplu de program ISR activ care stimulează activitatea unui creen-saver (economizor de ecran). El va detecta perioadele de timp în care tătaitura este activă și va lansa un mesaj correspunzător pe ecran. Designul său va dezacțiva imediat monitorul calculatorului.

Punem în general că un ISR este activ dacă el îloosește macar o componentă activă; cel puțin să dințe rutinile ISR-ului este activă.

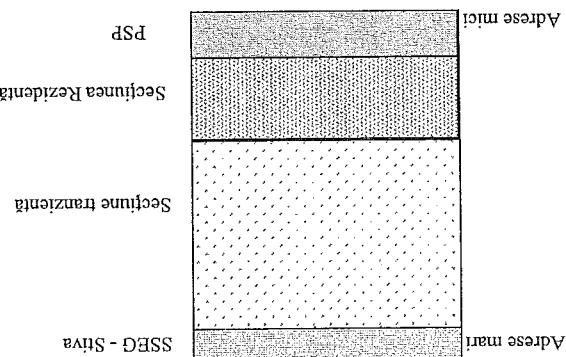
Microsoft identifică două tipuri mai de programe TSR: active și pasive. Un TSR pasiv este activat printr-un apel explicit call din applicația care rulează în sistem. Un TSR activ este acela care răspunde la interrupție hardware generată de sistem pe parcursul funcțiilor acestuia.

#### 6.4. TSR-URI ACTIVE SI TSR-URI PATIVE

Inainte de a trecere la altă secțiune legată de aspectele programelor TSR mai trebuie să amintim acesta în ultim detaliu legat de gestiunea memoriei – accesarea datelor în secțiunea rezidențială după instalarea programului. Procedurile din cadrul unității program TSR devin active ca urmare a apelului direct dîntr-un alt program sau prin intermediul unei interrupții hardware. După intrarea în secțiunea rezidențială, ultima apelată poate presupune că suntem în regiunea conturi parameți cu anumite valori. Nu rezidindă, ultima apelată poate presupune că suntem în regiunea conturi parameți cu anumite valori. Nu putem însă face acest lucru odată privatizare la valoare stocate în regiunea de segment.

Sigurul rezultat de segment care conține o valoare semnificativă semantic este rezultatul de cod CS. Restul rezultatelor de segment vor trebui înțelese ca valori semnificative pentru aplicația TSR. La termenare rutinei TSR, aceasta va rezulta o valoare registrilor de segment pentru a permite execuția corectă a aplicației care a fost interrupțiată.

**Fig. 6.5.** Organizarea în memorie a unui program TSR.





#### b.4.1. Problema funzione DOS non reentratte

apeluri *reentrant*. În realitate, de fiecare dată când un TSR apelează alte rutine decât cele implementate direct în TSR trebuie să simă conștienți de posibilitatea apariției problemelor legate de codul non reentrant.

În general TSR-urile pasive nu suferă de această problemă. Aceasta deoarece apelul la rutinele TSR în acest caz se face în contextul apelatorului. Dacă nu există posibilitatea ca TSR-ul să fie activat și printr-un eveniment asincron (generarea unei intreruperi hardware) atunci nu există probleme din punct de vedere al reentrantării codului TSR-ului.

Așa cum am descris în exemplul de mai sus, DOS este probabil problema cea mai spinoasă din punct de vedere al programării de TSR-uri și reentrantării codului. Aceasta deoarece DOS nu este reentrant, însă furnizează o multitudine de servicii de care un TSR are nevoie în mod normal. Atunci când au realizat acest lucru specialiștii de la Microsoft au adăugat suport DOS pentru TSR-uri astfel încât acestea să poată să determine atunci când DOS este activ (există apeluri DOS în curs) și când nu. Motivul care stă la baza rezolvării în acest mod al problemei: problema reentrantării DOS se pune doar atunci când un TSR apelează DOS în timp ce există deja un apel DOS în curs. Dacă DOS nu este activ, atunci orice TSR poate apela funcțiile DOS fără probleme.

MS-DOS furnizează un flag special pe un octet (denumit flag InDOS) care conține valoarea zero dacă DOS nu este activ și o valoare diferită de zero dacă DOS se află în decursul procesării unei cereri. Prin testarea valorii flag-ului InDOS o aplicație TSR poate să își „dea seama singură” dacă poate apela în siguranță funcții DOS. Dacă flag-ul InDOS conține valoarea zero TSR-ul poate face apel la funcții DOS. Dacă valoarea InDOS conține o altă valoare atunci există pericolul de a ne suprapune apelul cu o funcție DOS în curs. Există o funcție DOS (GetInDosFlagAddress) care returnează adresa flag-ului InDOS. Pentru utilizarea acestei funcții trebuie să cărăcătez registru AH cu valoarea 34h și apelați DOS, care va returna adresa flag-ului InDOS în perechea de regiștri ES:BX. Prin salvarea acestei adrese vom putea apoi în orice moment să verificăm valoarea flag-ului InDOS.

```
mov ah, 34h
int 21h
mov word ptr cs:InDOS, bx
mov word ptr cs:InDOS+2, es
```

Dacă există două flag-uri care trebuie testate: InDOS și CritError (flag-ul de eroare critică DOS). Ambele trebuie să conțină valoarea zero pentru a putea apela în siguranță funcții DOS. Flag-ul de eroare critică este setat atunci când un apel DOS se termină cu o eroare critică. Detaliile legate de eroare critică sunt stocate de DOS la o adresă fixă și sunt suprascrise la fiecare apel al unei funcții DOS. Dacă alegem să facem un apel DOS atunci când acest flag este setat, cel mai probabil din cauza faptului că aplicația ce rulează în prim-plan a apelați o funcție DOS care nu s-a putut executa corect, vom suprascrie detaliile legate de eroare critică și vom însela în acest fel aplicația intreruptă. Aceasta nu își va da seama că apelul DOS executat a eşuat, fapt care va duce cu siguranță la funcționarea ei incorrectă în continuare. Flag-ul de eroare critică DOS se află stocat pentru versiunile DOS 3.1 și mai noi în octetul imediat precedent flag-ului InDOS.

Întrebarea care se pune este: "Ce trebuie să facem atunci când cele două flag-uri au valori nenule?". Răspunsul verbal este unul extrem de simplu: "Înțoarce-te mai târziu pentru a executa TSR-ul, atunci când funcția MS-DOS activă întoarce controlul la programul utilizator". Cum putem însă implementa în practică acest lucru? Dacă TSR-ul nostru este activat de intreruperea de tastatura la apăsarea unei combinații de taste și redăm controlul handler-ului original din cauza faptului că DOS este activ, cum putem reactiva TSR-ul mai târziu când DOS nu mai este activ? Cea mai simplă soluție ar fi aceea de a obliga utilizatorul să activeze în mod repetat combinația de taste până când prinde un interval de timp între două apeluri DOS. Această soluție nu este însă aplicabilă în practică din motive pe care cititorul le poate bănuia cu ușurință. A doua soluție este aceea de a redirecta în cadrul TSR-ului intreruperea de ceas, pe lângă cea de tastatură. Atunci când la apăsarea combinației de taste descooperim în handler-ul intreruperii de tastatură că DOS este activ setăm un flag (CerereTSR) la nivelul TSR-ului care ne indică că am avut o cerere de activare a TSR-ului și redăm controlul handler-ului original de tastatură. În acest timp RTI de ceas va verifica periodic valoarea flag-ului din TSR. Dacă flag-ul nu este setat apelează handler-ului original al intreruperii de ceas. Dacă flag-ul este setat atunci verifică flag-ul InDOS și CritError. Dacă DOS este activ/ocupat RTI de ceas redă controlul handler-ului original. După ce DOS redă controlul aplicației utilizator prima intrerupere de ceas generată va determina faptul că DOS nu este activ va putea activa codul principal al programului TSR, care poate în acest caz face apel la funcții DOS. Desigur, primul lucru pe care TSR-ul îl va face la activare este acela de a reseta flag-ul intern CerereTSR astfel încât următoarele intreruperi de ceas să nu restarteze TSR-ul.

Soluția de mai sus este valabilă în toate cazurile cu o singură excepție. Există unele funcții DOS care necesită un interval de timp nedefinit de execuție. Este vorba de exemplul de funcția de citire a unui caracter de la tastatură. Aceasta se poate termina într-o secundă dacă utilizatorul apasă o tastă, sau în 10 ore dacă nimeni nu apasă vreo tastă. Este cazul apelurilor blocante de funcții DOS. Implementarea DOS a acestor funcții se bazează pe o buclă internă DOS care așteaptă ca utilizatorul să apese o tastă. Își până când utilizatorul face acest lucru flag-ul InDOS va rămâne nenu. Dacă TSR-ul pe care l-am implementat trebuie să scrie date într-un fișier la intervale regulate de timp, acest lucru nu va fi posibil dacă utilizatorul tocmai a plecat să ia masa în timp de DOS așteptă apăsarea unei taste sau introducerea unui sir de caractere de la tastatură.

Din fericire DOS furnizează o soluție și pentru această problemă prin implementarea unei intreruperi de *temp de inactivitate*. Cât timp DOS este într-un ciclu de așteptare a unei operații cu un dispozitiv I/O (intrare-iesire), apelează în mod continuu intreruperea 28h. Prin redirectarea vectorului intreruperii 28h, TSR-ul nostru poate determina cazurile în care DOS se află în bucle de așteptare. Atunci când DOS apelează int 28h putem face în siguranță apeluri DOS către funcții ale căror număr de identificare este mai mare decât 0Ch. Pentru funcțiile care au identificator mai mic de 0Ch DOS folosește stiva aflată la o adresă fixă. Pentru a nu distruga stiva unui apel în curs printr-un alt apel este interzisă apelarea acestor funcții DOS. De cele mai multe ori folosim intreruperea 21h pentru a afișa siruri de caractere pe ecran (funcția 09h). Conform remarcilor de mai sus, utilizarea acestei funcții nu este posibilă în timpul unei intreruperi 28h. Pentru a face posibil apelul oricărei funcții DOS atunci când DOS se află într-o buclă de așteptare putem salva conținutul anterior al stivei DOS într-o zonă de memorie din cadrul TSR-ului și să refacem conținutul stivei DOS după terminarea execuției curente a acestuia, sau atunci când nu mai facem



### 6.5. ÎNTRERUPERA MULTIPLEX (INT 2Fh)

Atunci când instalăm un TSR care conține componente pasive trebuie să alegem un vector de întrerupere pe care îl vom utiliza pentru a comunica cu rutinile pasive din cadrul TSR-ului. Pentru aceasta avem la îndemână două soluții:

1. Alegerea la întâmplare a unui vector de întrerupere;
2. Alegerea unui vector de întrerupere deja definit în cadrul sistemului - care implementează o funcție specifică;

Prima soluție este nepotrivită din mai multe considerente. Dacă vectorul respectiv este deja folosit de către sistem, atunci rescrierea lui poate să ducă la disfuncționalități în funcționarea sistemului de operare. Chiar dacă introducem doar o funcție suplimentară întruperii alese, este posibil ca acea funcție să mai fie aleasă și de alți implementatori de TSR-uri sau este posibil ca funcția să fie aleasă de chiar implementatorii sistemului de operare în viitor pentru implementarea unui serviciu specific.

Pentru a două soluție în unele cazuri alegerea este clară. Atunci când dorim să extindem serviciile de tastatură ale întreruperii 16h este clar că vom redirecta această întrerupere căreia îl vom redirecta/adăuga una sau mai multe funcții. Pe de altă parte, dacă implementăm un driver de dispozitiv sau un TSR care nu extinde servicii existente problema alegерii vectorului de întrerupere prin care să comunicăm cu TSR-ul rămâne. Din fericire MS-DOS furnizează o soluție și în acest caz: întreruperea multiplex 2Fh. Această întrerupere a fost rezervată pentru a furniza un mecanism general de instalare, testarea prezenței și comunicare cu un TSR.

Pentru a utiliza întreruperea multiplex, aplicațiile stochează un număr de identificare în registrul AH și fac apel la INT 2Fh. Fiecare TSR din lanțul de handleuri asociate va verifica numărul de identificare cu numărul de identificare stocat intern la nivelul TSR-ului. Dacă numerele de identificare coincid se va executa comanda specificată în registrul AL, altfel TSR-ul va transmite controlul următorului handler din lanțul vectorilor întreruperii 2Fh.

Desigur, această metodă rezolvă doar o parte a problemei, aceea a alegării unui vector de întrerupere. Numărul de identificare trebuie să fie cunoscut atât de entitatea care verifică cât și de TSR-ul instalat. Cum nu există un organism de standardizare care să aloce numerele de identificare, desigur nu vor putea fi eliminate conflictele: alegerea același număr de identificare de către mai multe TSR-uri. Dacă numărul de identificare este ales dinamic atunci se pune întrebarea: „Cum poate fi el cunoscut de aplicația care testează prezența TSR-ului?”

Problema enunțată mai sus poate fi rezolvată printr-un mic artificiu. Folosim prin convenție funcția (comanda) zero pentru a testa dacă un TSR este instalat sau nu. Orice aplicație va executa întotdeauna această funcție pentru a determina dacă TSR-ul este instalat în memorie înainte de a executa alte funcții ale TSR-ului. În mod normal funcția zero va returna valoarea zero în registrul AL dacă TSR-ul nu este prezent în memorie sau OFFh dacă TSR-ul este prezent. Desigur prezența unei valori OFFh în AL ne indică doar prezența *unui* TSR instalat, dar nu ne garantează că TSR-ul

respectiv este cel pe care îl căutăm noi. Prin extinderea convenției făcute până acum însă, vom putea determina corect și exact prezența TSR-ului care ne interesează în memorie. Pentru aceasta să presupunem că funcția zero întoarce pe lângă valoarea din AL un pointer la string de identificare în regiștri ES:DI. Prin compararea string-ului de la adresa ES:DI cu semnătura cunoscută de aplicația care testează prezența TSR-ului se poate detecta exact prezența *unui* TSR anume în memorie.

### 6.6. INSTALAREA UNUI TSR

Deși am văzut deja în primul exemplu de program TSR din acest capitol modalitatea de instalare a unui TSR în memorie, mai sunt câteva aspecte de discutat legate de această problemă. Întrebările care se pun în acest context sunt:

1. Ce se întâmplă dacă utilizatorul instalează un TSR care este deja instalat în memorie fără a dezinstala mai întâi copia existentă?
2. Cum putem afecta dinamic un număr de identificare unui TSR astfel încât să nu fie în conflict cu TSR-urile gata instalate?

Prima întrebare se referă la instalarea de mai multe ori a unui TSR în memorie. În general nu există aplicații TSR care să funcționeze cu mai multe copii simultan în memorie. Aceasta deoarece fiecare copie redirecțiază aceleași întreruperi și îndeplinește aceleași sarcini. Mai mult, instalarea de  $N$  ori,  $N > 1$  a unui TSR în memorie implică consumul unei cantități de  $N$  ori mai mare de memorie. În majoritatea cazurilor un astfel de scenariu nu poate decât să ducă la risipă de memorie și chiar la blocarea sistemului. În consecință fiecare TSR trebuie să verifice la instalare existența unei copii deja instalare anterior. În cazul în care se detectează o copie deja instalată de obicei se afișează un mesaj informativ și TSR-ul refuză instalarea.

Secvența de cod prezentată în continuare determină prezența unui TSR în memorie și furnizează numărul de identificare al acestuia. Această secvență de cod ne dă răspunsul și la două întrebări. Rutina determină în cazul în care TSR-ul nu este activ un număr de identificare liber și resetează CF la terminare.

; Scanează toate ID-urile posibile între 255 și 0. Dacă vreunul este instalat verifică dacă coincid ; semnăturile

...

Sign db 'Semnatura TSR test'

Sign\_len EQU \$-Sign

FuncID db 0

...

check\_installed proc

mov cx, OFFh

;verifică ID-urile posibile de la 255 la 0

- Să deținăce memoria alocația astfel încât să se poate să utilizeze în altă aplicație;

- Sa dispunea totate activitatile in curs de programare ISX;
  - Sa reprezinta locuri de vectori de intersectie de TSR;
  - Sa dea loc memoriei alocaata astfel incat aceasta sa poata fi utilizata de alte aplicatii;

Dezinstalarea unui TSR este mai complicata decat procesul de instalare. Cu mult de dezinstalare se duceti sa faci la mijlocul turei.

Deosebita de masoara o constiuie unimare doua puncte. In acest lucru se introducunam posibilitati de interrupere sau sunt multe posibile cele doua activitati.

In figura 6.6 am reprezentat grafic două TRS-uri care au fost instalație după "TRS-ul nosțu", în figura 6.6 am reprezentat grafic două TRS-uri care au fost instalată la instalație, conformării finale va fi cea prezentată în figura 6.7.

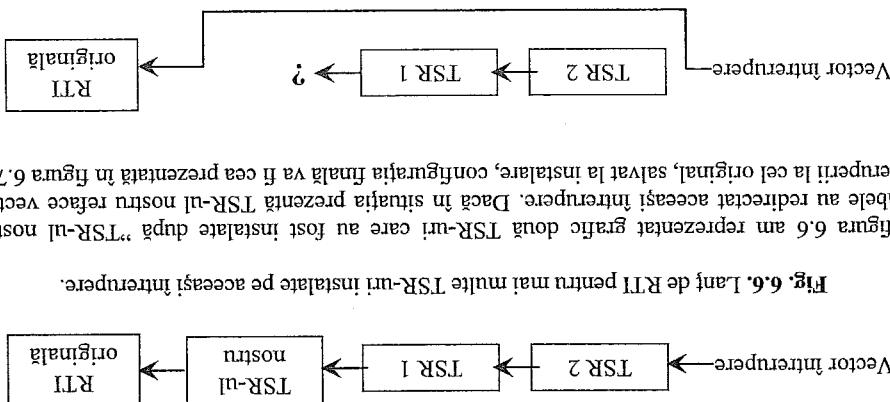


Fig. 6.7. Refacebook RTI orthogonal multilatera mai multor TRS-uri de același interpuere.

In figura 6.6, în care TSR a salvat adresa hanlder-ului original, din punctul său de vedere, fizică sănătatea RIT salvată, astfel încât orice interrupție generată trage prin hîrge lanțul de hanlder sau interrupție vectorială în care TSR-ului nu poate să reducă rata de interrupții de la generează ulterioră a interrupției, rămânând în final să primească din urmă cea ce este și mai rau să fie apărată de către TSR 1 și TSR 2 în schimbănd activele. Această lucru ducă în general la funcționarea defectuoasă a celor două TSR-uri implicate. Ceea ce rezultă în final este că un interrupțor redirecțiat în TSR 1 și TSR 2 nu vor mai fi capabile de a efectua o acțiune de la adresa originală a interrupției, rămânând în final să primească din urmă cea ce este și mai rau să fie apărată de către TSR 1 și TSR 2 în schimbănd activele. Bestă că TSR-uri vor fi dezafectate, rezultă interrupții redirecțiate în TSR 1 și TSR 2 rămașând active.

greu de prezis care va fi comportamentul sistemului într-o astfel de situație. Depinde de funcționalitățile implementate de cele două TSR-uri rămase.

Cum putem rezolva această problemă? Soluția cea mai simplă și cea mai eficace atunci când detectăm un astfel de caz este să tipărim un mesaj de atenționare și să refuzăm dezinstalarea TSR-ului. Aceasta este o hibă bine cunoscută legată de TSR-uri și în general utilizatorii care instalează TSR-uri sunt conștienți (sau ar trebui să fie) de faptul că dezinstalarea lor trebuie făcută în ordine inversă instalării. Desigur se mai pune încă problema modalității prin care putem detecta dacă utilizatorul a mai instalat și alte TSR-uri care redirectează întreruperi comune cu TSR-ul nostru. Răspunsul la această întrebare este destul de simplu. Este suficient să comparăm, înainte de eventuala dezinstalare, vectorul curent al întreruperii respective cu adresa salvată la instalarea TSR-ului. Dacă cele două adrese coincid putem fi siguri că nici un alt TSR nu a mai redirectat respectivă întreupere după noi. În caz contrar nu putem permite dezinstalarea TSR-ului.

Desigur, faptul că nici o întreupere nu a fost redirectată *după* TSR-ul nostru nu înseamnă că nu există și alte TSR-uri încărcate în memorie după el. Acest fenomen duce la o altă anomalie. Chiar dacă eliberăm memoria alocată TSR-ului nostru ea nu va putea fi folosită de către aplicațiile care vor rula mai târziu în sistem. Aceasta datorită schemei de alocare și gestiune a memoriei de către MS-DOS. Situația descrisă mai sus este prezentată în figura 6.8. După încărcarea TSR-ului nostru, pointerul spre zona de memorie utilizabilă este deplasat la adresa superioară imediat următoare acestuia. Încărcarea programului TSR1 va deplasa pointerul zonei de memorie liberă la adresa imediat superioară ultimului octet alocat de acesta. În consecință după dealocarea memoriei utilizate de către TSR-ul nostru, pointerul spre zona de memorie liberă (utilizabilă) va rămâne neschimbăt, pentru a proteja TSR1. Fenomenul descris mai sus se numește *fragmentarea memoriei* și poate să apară de fiecare dată când procesele de alocare și dealocare a memoriei alocate TSR-urilor nu sunt făcute în regim LIFO (*Last In First Out*).

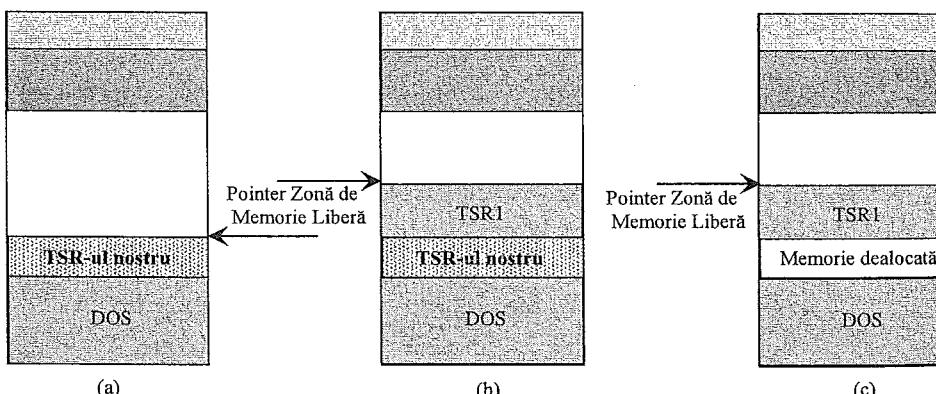


Fig. 6.8. Pointerul de Memorie Liberă în timpul alocării și dealocării TSR-urilor.

În figura 6.8 (a) este prezentată adresa pointerului de memorie liberă după instalarea TSR-ului nostru. În figura (b) este prezentată adresa de memorie a zonei libere după încărcarea în memorie a programului TSR1, iar în figura (c) adresa zonei de memorie libere după dezinstalarea TSR-ului nostru fără a dezinstala în prealabil TSR1. După cum se poate observa și în figură, zona de memorie devenită liberă după dezinstalarea TSR-ului nostru nu poate fi folosită până când nu se dezinstalează și TSR1.

Pentru dealocarea memoriei programului TSR avem nevoie de apeluri ale funcției DOS 49h. Acest apel preia din registrul ES adresa blocului de memorie care trebuie eliberat. Primul apel este necesar pentru eliberarea blocului de variabile de mediu alocat la crearea programului. Pointerul spre acest bloc de memorie se găsește la deplasamentul 2Ch în PSP. Acesta este de altfel și unul dintre motivele pentru care orice program TSR dezinstalabil trebuie să salveze adresa sa PSP. Urmatul bloc de memorie este cel al programului rezident însăși. Acesta începe la adresa de început a PSP-ului în memorie. Secvența de cod care permite dealocarea memoriei unui TSR este prezentată în continuare:

; Secvența de cod de mai jos presupune că variabila PSP a fost inițializată cu adresa PSP-ului ; acestui program înainte de apelul care lasă TSR-ul rezident

mov es, PSP ;încarcă în ES adresa de segment a variabilelor de mediu

mov es, es:[2Ch] ;deallocă blocul variabilelor de mediu

mov ah, 49h ;încarcă în ES adresa de segment a PSP-ului (programul însuși)

int 21h ;deallocă programul TSR din memorie

mov es, PSP ;încarcă în ES adresa de segment a variabilelor de mediu

mov ah, 49h ;deallocă blocul variabilelor de mediu

int 21h ;deallocă programul TSR din memorie

La fel cum codul de instalare a unui TSR este înglobat în același cod sursă cu programul TSR, codul de dezinstalare face parte tot din partea tranzientă a aceluiși program. De obicei același program apelat cu parametrul /u în linia de comandă execută sarcina de dezinstalare a TSR-ului, atunci când acesta a fost instalat în prealabil.

## 6.8. TSR MONITOR TASTATURA

În cele ce urmează vom exemplifica noțiunile prezentate în paragrafele precedente printr-un program TSR complet. TSR-ul denumit KeybMon permite monitorizarea tastaturii și înregistrarea tuturor tastelor apăsate pe parcursul rulării sale. La apăsarea unei combinații de taste de activare (F9), KeybMon va afișa pe ecran toate tastele apăsate de la ultima activare. TSR-ul verifică la instalare dacă există deja o copie în memorie, caz în care refuză o nouă instalare. El poate fi dezinstalat prin lansarea aplicației cu parametrul /u în linia de comandă. TSR-ul verifică dacă procesul de dezinstalare este posibil și nu intră în conflict cu alte TSR-uri instalate după el.

casă de segment Pară asumă CS:CSeg, DS:CSeg, ES:CSeg

RezStart equ \$  
jmp begin  
start:  
;se înstalată în memorie  
;semnatura TSR-ului – utilizată de către copia tranzientă a KeyBoard Monitor  
;sign db Keyboard Monitor

maxh EQU 2000 ;numărul maxim de taste apăsată care îl înregistrează

keyCodes db 1111\$, EC\$;, 1\$, 2\$, 3\$, 4\$, 5\$, 6\$, 7\$, 8\$, 9\$, 0\$, -, \$, db ENT\$, TAB\$, q\$, w\$, e\$, t\$, y\$, u\$, o\$, p\$, [\$, ], \$, db BKSP\$, TAB\$, a\$, d\$, s\$, f\$, g\$, h\$, j\$, k\$, l\$, i\$, db 1111\$, SP\$, 1111\$, F1\$, F2\$, F3\$, F4\$, F5\$, F6\$, F7\$, F8\$, F9\$, F10\$

keys db mach dup(0) ;tabel cu tuturor indexelor după codul SCAN al fiecarui tastă (L=ESC, etc)

msg db „Taste înregistrate: 13, 10, \$.” ;mesajul afisat la apăsarea combinării de taste (F9) de activare a TSR-ului inițiată de utilizator apăsată.

index dw -1 ;controllul curent de taste apăsată

intrl proc far ;tastatura

in al, 60h push ax

je read-key cmp al, 80h

nu este nevoie să le treacă. Să îl rutina originală

;codurile scană > 80h semnifica eliberarea unei taste.

;nu este nevoie să le treacă. Să îl rutina originală

;verificăm dacă mai avem spațiu de stocare

;dacă nu readam controloul handelor-ului într-o originală

;mai avem spațiu => memorăm codul scan în keys

;salvăm codul scan în tabeloul keys pe prima pozitie

;libera

;verificăm regiștri modificați

;apăsel rutină originală a int'g revine

;activăm interrupție și apăsăm oldint9

;în general apăsarea unei taste generaază apelul int'g. Handler-ul originală al int'g crește codul

;scan al tastei apăsată din portul 60h și îl transformă în codul ASCII corespunzător. Apoi plasează

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care vom salva 64 de cuvinte din stiva DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

oldint2F dd ?

PSp db 0 ;codul de identificare al TSR-ului

FuncID db 0 ;variabilă în care salvăm adresa PSP-ului și vectorii

save\_ss dw ? ;originală ai interrupților redirecțiate

stiva\_sp dw ? ;acție salvăm regiștri de adresa a vecchi stive DOS

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

flag db 0 ;stiva\_cere de activare TSR-ului și DOS

indos\_OFS dw ? ;adresa indicatorului ImDOS

indos\_SEGdw ? ;indica căcerere de activare TSR-ului și DOS

stiva(dw) ;o vom folosi atunci când TSR-ul este activat

stiva\_ss dw ?

stiva\_sp dw ?

stiva\_ssdw cseg dw 256 dup(0) ;stiva în care salvăm regiștri de adresa a vecchi stive DOS

oldint9 dd ?

oldint28 dd ?

</

;atât codul scan cât și codul ASCII în buffer-ul de tastatură. Acestea vor putea fi apoi citite din buffer-ul de tastatură cui ajutorul întreruperii 16h. Handler-ul întreruperii 9h execută următorii pași:

- ; 1) Citește codul scan din controller-ul de tastatură (portul 60h)
- ; 2) Setează bitul 7 în portul 61h
- ; 3) Resetează bitul 7 în portul 61h indicând astfel controller-ului de tastatură consumarea codului scan
- ; 4) Convertește codul scan la corespondentul ASCII și plasează ambele coduri în bufferul de tastatură
- ; 5) Trimit secvența EOI – 20h (End of Interrupt – Sfârșit de întrerupere) controller-ului programabil de întreruperi 9259 (portul 20h). Atât timp cât acesta nu primește secvența EOI nu mai generează alte apeluri int 9.

```
read_key:
    cli
    in al,61h
    mov ah,al
    or al,80h
    out 61h, al
    xchg ah,al
    out 61h, al
    mov al, 20h
    out 20h, al
    push ds
    push dx
    lds dx, dword ptr cs:inDos_OFFSET
    cmp byte ptr dx, 0
    jne later

    call write_keys
    jmp end_int9

later:
    mov byte ptr cs:flag,1
```

```
end_int9:
    pop ds
    pop dx
    pop ax
    sti
    iret
int9 endp
```

;dacă tastă apăsata este F9  
;dezactivăm întreruperile  
;consumăm codul scan din controller-ul de tastatură  
;Simulăm acțiunea standard a int 9 = consumare  
;a codului scan cu ștergerea acestuia din controller  
;Controller-ul nu va mai semnaliza apăsarea acestei  
;taste. Ea a fost citită deja dpdv al controller-ului de  
;de către sistemul de operare.  
;trimitem secvența de sfârșit de întrerupere către  
;controller-ul programabil de întreruperi 8259.

;verificăm dacă DOS este activ  
;dacă DA setăm variabila flag și terminăm execuția  
;handler-ului în ideea de a fi reactivați după.  
;dacă NU, apelăm rutina de afișare a tastelor  
;memorate și terminăm execuția handler-ului

; Int28 Noul handler al întreruperii 28h (DOS Idle).Sistemul apelează această rutină de fiecare dată când DOS este într-o buclă de așteptare I/O. Rutina apelează handler-ul original, cu revenire, pentru a permite și celorlalte aplicații să detecteze starea DOS și verifică apoi starea variabilei flag. În cazul în care aceasta este setată, salvăm stiva DOS pentru a permite revenirea corectă la apelul DOS întrerupt și apelăm rutina de afișare a tastelor memorate de la ultimul apel *write\_keys*. Se resetează variabila *flag* (am tratat cererea de activare). După afișarea tastelor memorate refacem stiva DOS și redăm controlul aplicației întrerupe

```
int28 proc
    pushf
    call dword ptr cs:oldint28
    cli
    cmp byte ptr cs:flag,1
    jne int28_end

    mov word ptr cs:save_sp, sp
    mov word ptr cs:save_ss, ss
    mov ss, cs:stiva_ss
    mov sp, cs:stiva_sp
    push cx
    push si
    push ds
    mov cx, 64
    mov ds, cs:save_ss
    mov si, cs:save_sp
rep_save_stiva:
    push word ptr [si]
    inc si
    inc si
loop rep_save_stiva
    mov byte ptr cs:flag,0
    call write_keys

    mov cx,64
    mov ds, cs:save_ss
    mov si, cs:save_sp
    add si, 128
rep_restore_stiva:
    dec si
    dec si
    pop word ptr [si]
loop rep_restore_stiva
```

;simulare apel de tip *call* al int28 originală  
;dezactivare întreruperi.  
;verificăm dacă avem cerere de activare  
;dacă NU redăm controlul aplicației întrerupe

;dacă DA – salvăm 64 de cuvinte din stiva DOS și  
;adresa acestieia

;resetăm variabila *flag*  
;apelăm rutina de afișare a tastelor memorate

;refacem primele 64 de cuvinte din stiva DOS

