Alexandru Vancea
Darius Bufnea
Adrian Dărăbant
Florian Boian
Anca Andreica
Andreea Navroschi

Arhitectura calculatoarelor. Limbajul de asamblare 80x86.

Editura RISOPRINT
Cluj-Napoca • 2014

© 2014 RISOPRINT

Toate drepturile rezervate autorilor

*Editura **RISOPRINT** este recunoscută de C.N.C.S. **(Consiliul Național al Cercetării Științifice)**. Pagina web a CNCS: www.cncs-uefiscdi.ro*

Opiniile exprimate în această carte apartin autorului si nu reprezinta punctul de vedere al Editurii Risoprint. Autorul îsi asuma întreaga responsabilitate pentru forma si continutul cărții si se obliga sa respecte toate legile privind drepturile de autor.

Toate drepturile rezervate. Tipărit în România. Nicio parte din această lucrare nu poate fi reprodusă sub nicio formă, prin niciun mijloc mecanic sau electronic, sau stocată într-o bază de date fără acordul prealabil, în scris, al autorului.
All rights reserved. Printed in Romania. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the author.

VANCEA, ALEXANDRU
ARHITECTURA CALCULATOARELOR. Limbajul de asamblare 80x86 /
Alexandru Vancea, Florian Boian, Darius Bufnea, Anca Gog, Adrian Dărăbant, Andreea Sabău - Cluj-Napoca; Risoprint, 2005
404 p.; 17 x 24 cm.
Bibliogr.

ISBN 973-651-037-2

I. BOIAN, FLORIAN
II. BUFNEA, DARIUS
III. GOG, ANCA
IV. DĂRĂBANT, ADRIAN
V. SABĂU, ANDREEA

Director editură: GHEORGHE POP
Consilier editorial: VOICHIȚA-MARIA CLINCI

Coperta: ILEANA ŞERBĂNUȚ

Tiparul executat la:
S.C. ROPRINT® S.R.L.
400 188 Cluj-Napoca • Str. Cernavodă nr. 5-9
Tel./Fax: **0264-590651** • roprint@roprint.ro
430 315 Baia Mare • Piața Revoluției nr. 5/1
Tel./Fax: **0262-212290**

PREFAȚĂ

Lucrarea de față se adresează în primul rând studenților facultăților cu profil informatic care studiază arhitectura calculatoarelor și limbajul de asamblare 80x86. Volumul urmărește prezentarea și aprofundarea graduală de către studenți a noțiunilor de bază de reprezentare a informației la nivelul sistemelor de calcul, a celor mai importante elemente structurale din componența arhitecturilor acestora și a funcționalităților lor, precum și înțelegerea modului nativ de lucru al unui calculator.

În acest context, programarea în limbaj de asamblare nu reprezintă neapărat un scop în sine, ci se constituie în mecanismul fundamental la care trebuie să recurgem atunci când se impune controlul deplin al resurselor unui sistem de calcul. De aceea, pe lângă prezentarea absolut necesară a elementelor și tehnicilor de programare în limbaj de asamblare descrise pe parcursul lucrării, am pus un accent deosebit pe modul în care programarea în limbaj de asamblare se integrează cu programarea în cadrul limbajelor de nivel înalt la nivelul arhitecturilor de tip 80x86.

Volumul are un caracter pronunțat didactic constituindu-se în suportul de curs al materiei "**Arhitectura calculatoarelor**" predată la Facultatea de Matematică și Informatică a Universității "Babeș-Bolyai" din Cluj Napoca.

Mulțumim în mod deosebit foștilor noștri colegi Simona Horea și Marius Iurian, alături de care s-a materializat prima lucrare legată de limbajul de asamblare 80x86, pentru acordul de a introduce în lucrarea de față unele dintre rezultatele muncii noastre de atunci.

Pornind de la premisa că nici un programator nu va putea să dezvolte o aplicație cu adevărat eficientă atâta timp cât nu este conștient de particularitățile limbajului nativ de comunicare cu procesorul unui sistem de calcul, sperăm ca această carte să fie de un real folos tuturor celor care și-au propus să înțeleagă pe deplin cum poate fi realizat acest lucru.

Autorii

CUPRINS

1. REPREZENTAREA DATELOR 1
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 octeților 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

2. ARHITECTURA SISTEMELOR DE CALCUL 35
2.1. Definiții. Organizarea unui sistem de calcul 35

2.2. Unitatea centrală (*Central Processing Unit* – CPU) 38
2.2.1. Ceasul sistem (*The System Clock*) 39
2.2.2. "Dimensiunea" unui microprocesor sau răspunsul la întrebarea "ce înseamnă calculator pe n biţi?" 41
2.3. Memoria 42
2.3.1. Memoria internă 43
2.3.2. Memoria externă / secundară 45
2.3.3. Ierarhia memoriei 48
2.4. Dispozitive periferice 50
2.4.1. Magistrale – structuri de interconectare 51
2.4.2. Magistrale I/O interne şi interfeţele asociate 53
2.4.3. Magistrale I/O externe şi interfeţele asociate 55
2.4.4. Componentele unui calculator personal 57
2.5. Performanţele unui sistem de calcul 61
2.6. Arhitectura microprocesorului 8086 61
2.6.1. Structura microprocesorului 61
2.6.2. Regiştrii generali EU 62
2.6.3. Flagurile 63
2.6.4. Regiştrii de adresă şi calculul de adresă 64
2.6.5. Reprezentarea instrucţiunilor maşină 66
2.6.6. Adrese FAR şi NEAR 66
2.6.7. Calculul offsetului unui operand. Moduri de adresare 67

3. ELEMENTELE LIMBAJULUI DE ASAMBLARE 69
3.1. Formatul unei linii sursă 71
3.2. Expresii 73
3.2.1. Moduri de adresare 73
3.2.1.1. Utilizarea operanzilor imediaţi 74
3.2.1.2. Utilizarea operanzilor registru 76
3.2.1.3. Utilizarea operanzilor din memorie 77

3.2.1.4. Operanzi cu adresare indirectă 78
3.2.2. Utilizarea operatorilor 79
3.2.2.1. Operatori aritmetici 80
3.2.2.2. Operatorul de indexare 81
3.2.2.3. Operatori de deplasare de biţi 81
3.2.2.4. Operatori logici pe biţi 81
3.2.2.5. Operatori relaţionali 82
3.2.2.6. Operatorul de specificare a segmentului 82
3.2.2.7. Operatori de tip 83
3.3. Directive 85
3.3.1. Directive standard pentru definirea segmentelor 85
3.3.1.1. Directiva SEGMENT 85
3.3.1.2. Directiva ASSUME şi gestiunea segmentelor 87
3.3.2. Directive pentru definirea datelor 94
3.3.3. Directivele LABEL, EQU, = ... 97
3.3.4. Directiva PROC 99
3.3.5. Blocuri repetitive 99
3.3.6. Directiva INCLUDE 101
3.3.7. Macrouri 102

4. INSTRUCŢIUNI ALE LIMBAJULUI DE ASAMBLARE 107
4.1. Manipularea datelor 108
4.1.1. Instrucţiuni de transfer al informaţiei 108
4.1.1.1. Instrucţiuni de transfer de uz general 108
4.1.1.2. Instrucţiuni de transfer de intrare-ieşire 112
4.1.1.3. Instrucţiuni de transfer al adreselor 113
4.1.1.4. Instrucţiuni asupra flagurilor 116
4.1.2. Instrucţiuni de conversie 117
4.1.3. Impactul reprezentării little-endian asupra accesării datelor 119
4.2. Operaţii 122

4.2.1. Operaţii aritmetice 122
4.2.1.1. Adunarea şi scăderea 122
4.2.1.2. Înmulţirea şi împărţirea 126
4.2.1.3. Exemple şi exerciţii propuse 129
4.2.2. Operaţii logice pe biţi 131
4.2.3. Deplasări şi rotiri de biţi 133
4.3. Ramificări, salturi, cicluri 138
4.3.1. Saltul necondiţionat 138
4.3.1.1. Instrucţiunea JMP 139
4.3.2. Instrucţiuni de salt condiţionat 144
4.3.2.1. Comparaţii între operanzi 144
4.3.2.2. Salturi condiţionate de flaguri 145
4.3.2.3. Exemple comentate 148
4.3.3. Instrucţiuni de ciclare 162
4.3.4. Instrucţiunile CALL şi RET 164
4.4. Instrucţiuni pe şiruri 166
4.4.1.Generalităţi privind şirurile şi instrucţiunile pe şiruri 166
4.4.2. Instrucţiuni pe şiruri pentru transferul de date 168
4.4.3. Instrucţiuni pe şiruri pentru consultarea şi compararea datelor 170
4.4.4. Execuţia repetată a unei instrucţiuni pe şiruri 173
4.4.5. Utilizarea de operanzi pentru instrucţiuni pe şiruri 174
4.5. Un exemplu complet de program 175

5. ÎNTRERUPERI 181
5.1. Probleme generale privind întreruperile 181
5.2. Clasificarea întreruperilor 182
5.2.1. Întreruperi hardware 183
5.2.2. Excepţii 185

5.2.3. Întreruperi software 187
5.3. Câteva observaţii asupra întreruperilor 8086 192
5.4. Instrucţiuni specifice lucrului cu întreruperi 193
5.5. Formatele COM şi EXE 195
5.5.1. Prefixul unui program executabil (PSP) 195
5.5.2. Structura unui program EXE 197
5.5.3. Structura unui program COM 202
5.5.4. Depanarea programelor .EXE şi .COM 205

6. REDIRECTAREA ÎNTRERUPERILOR 207
6.1. Redirectarea întreruperilor 207
6.2. Programe TSR 210
6.3. Harta de memorie DOS şi programele TSR 210
6.4. TSR-uri active şi TSR-uri pasive 214
6.4.1. Problema funcţiilor DOS non reentrante 219
6.4.2. Problema întreruperilor BIOS non reentrante 223
6.5. Întreruperea multiplex (INT 2Fh) 224
6.6. Instalarea unui TSR 225
6.7. Dezinstalarea unui TSR 227
6.8. TSR monitor tastatură 229
6.9. Depanarea programelor TSR 244
6.10. TSR-uri şi redirectare întreruperi în cadrul SO Windows 249

7. IMPLEMENTAREA APELULUI DE SUBPROGRAME 251
7.1. Cod de apel, cod de intrare, cod de ieşire 251
7.1.1. Cod de apel 251
7.1.2. Cod de intrare 252
7.1.3. Cod de ieşire 255
7.2. Implementarea subprogramelor în Turbo Pascal 256
7.2.1. Modele de memorie 256

7.2.2. Harta memoriei Turbo Pascal ... 257
7.2.3. Cod de apel al subprogramelor Pascal ... 259
7.2.3.1. Transmiterea parametrilor ... 259
7.2.3.2. Tipuri de apel al subprogramelor ... 260
7.2.3.3. Exemplu ... 261
7.2.4. Cod de intrare în subprogramele Pascal ... 262
7.2.4.1. Întoarcerea rezultatului de către funcţii ... 262
7.2.4.2. Exemplu ... 262
7.2.5. Cod de ieşire din subprogramele Pascal ... 264
7.2.6. Proceduri şi funcţii imbricate în Turbo Pascal ... 265
7.3. Implementarea subprogramelor în Borland C ... 267
7.3.1. Pointeri far şi near ... 267
7.3.2. Cod de apel al subprogramelor C ... 268
7.3.3. Cod de intrare în subprogramele C ... 269
7.3.4. Cod de ieşire din subprogramele C ... 270

8. PROGRAMAREA MULTIMODUL ... 271
8.1. Directiva MODEL. Directive de segment simplificate ... 271
8.2. Cerinţele unui modul asamblare la legarea cu un alt modul ... 274
8.2.1. Directiva PUBLIC ... 274
8.2.2. Directiva EXTRN ... 275
8.2.3. Directiva GLOBAL. Legarea de module asamblare ... 276
8.3. Legarea de module asamblare cu module scrise în limbaje de nivel înalt ... 278
8.3.1. Etapele legării unui modul asamblare cu un modul scris în limbaj de nivel înalt ... 278
8.3.1.1. Cerinţe ale editorului de legături ... 278
8.3.1.2. Intrarea în procedură ... 279
8.3.1.3. Nealterarea valorilor unor regiştri ... 279
8.3.1.4. Transmiterea şi accesarea parametrilor ... 279
8.3.1.5. Alocarea de spaţiu de memorie pentru datele locale ... 281

8.3.1.6. Întoarcerea unui rezultat ... 282
8.3.1.7. Revenirea din procedură ... 282
8.3.1.8. Directivele ARG şi LOCAL ... 283
8.3.2. Interfaţa dintre Turbo Assembler şi Turbo Pascal ... 285
8.3.2.1. Directiva de compilare $L şi subprogramele *external* ... 285
8.3.2.2. Reguli de utilizare a regiştrilor ... 288
8.3.2.3. Transmiterea şi accesarea parametrilor ... 288
8.3.2.4. Întoarcerea rezultatului de către funcţii ... 289
8.3.2.5. Alocarea de spaţiu pentru datele locale ... 289
8.3.2.6. Utilizarea directivelor de segment simplificate ... 290
8.3.2.7. Exemplul 1 ... 291
8.3.2.8. Exemplul 2 ... 293
8.3.3. Interfaţa dintre Turbo Assembler şi Borland C++ ... 301
8.3.3.1. Cerinţe ale editorului de legături privind definirea modulului asamblare ... 301
8.3.3.2. Transmiterea parametrilor ... 304
8.3.3.3. Întoarcerea de valori ... 306
8.3.3.4. Convenţii privind utilizarea regiştrilor ... 306
8.3.3.5. Exemplu ... 306

9. PROGRAMAREA LOW LEVEL ÎN LIMBAJELE PASCAL ŞI C ... 309
9.1. Inserarea de cod maşină în textul sursă Pascal ... 309
9.1.1. Instrucţiunea inline ... 309
9.1.2. Directiva inline ... 311
9.2. Asambloare inline ... 312
9.2.1. Asamblorul inline al lui Borland Pascal 6.0 ... 312
9.2.1.1. Instrucţiunea asm ... 312
9.2.1.2. Instrucţiunile asamblorului ... 312
9.2.1.3. Expresii ... 315
9.2.1.4. Proceduri şi funcţii assembler ... 321
9.2.1.5. Exemple ... 322

9.2.2. Asamblorul inline al lui Borland C++ 327
9.2.2.1. Instrucţiunea asm 327
9.2.2.2. Exemplu 330
9.3. Proceduri şi funcţii imbricate în Borland Pascal 331
9.4. Accesarea regiştrilor şi apelarea de întreruperi 335
9.4.1. Borland Pascal 6.0 335
9.4.2. Borland C++ 336
9.5. Scrierea de rutine de tratare a întreruperilor în limbajele Pascal şi C 337
9.5.1. Proceduri interrupt în Pascal 337
9.5.2. Funcţii interrupt în C 340

10. EXTENSII x86 341
10.1. Memoria înaltă (*high memory*) şi memoria extinsă 342
10.2. Procesorul 80386 şi modul de lucru protejat 355
10.3. Noi instrucţiuni aduse de urmaşii procesorului 8086 361
10.4. Exemplu de program care lucrează cu procesorul în mod protejat 365

11. PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS 371
11.1. Modul de adresare protejat şi microprocesoare pe 32 de biţi 371
11.2. Programare în limbaj de asamblare sub Windows 372
11.3. Microsoft Macro Assembler 373
11.4. Modalităţi 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 (studiu de caz – Visual C++) 382
11.4.3. Programare multimodul 384
11.5 Limbaj de asamblare vs. limbaje de nivel înalt 390

CAPITOLUL 1

REPREZENTAREA DATELOR

1.1. TIPURI DE DATE ELEMENTARE

Un element esenţial în utilizarea unui sistem de calcul este cunoaşterea tipurilor de date primitive cu care lucrează sistemul şi modul în care acesta şi 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 şi calcule în care numerele iniţiale, rezultatele intermediare şi finale nu sunt neapărat întregi. Deşi a doua categorie de calcule o include pe prima, în sistemele de calcul ele sunt tratate ca şi două categorii distincte, din raţiuni care vor fi lămurite mai târziu. Să exemplificăm aceste trei categorii de date.

Crearea şi întreţinerea listei studenţilor dintr-un an de studii nu reclamă, cel puţin în primă instanţă, nici un fel de prelucrări aritmetice. Prelucrarea unei astfel de liste presupune numai introducere, ştergere şi 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 operaţia 7 : 3, o operaţie simplă de împărţire. După caz, noi oamenii interpretăm rezultatul în unul din următoarele două moduri:

7 : 3 = 2.3333 . . . şi vom numi aceasta *împărţire reală*

7 : 3 dă câtul 2 şi restul 1 şi vom numi aceasta *împărţire întreagă*

Dacă oamenii dau o interpretare adecvată a rezultatelor împărţirii, nu acelaşi lucru se poate spune despre un calculator. Acesta trebuie să ştie, apriori, dacă va efectua o împărţire reală sau una întreagă. Astfel de situaţii au stat la baza separării calculelor numerice în calcule întregi şi 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 împărţiri întregi cu obţinerea de câturi şi de resturi întregi. Datele primare cu care se fac astfel de operaţii vom spune că sunt *date de tip întreg* sau în argoul informaticienilor *numere întregi.*

În multe aplicaţii inginereşti apar tot felul de calcule în care se operează cu diverşi coeficienţi care nu sunt neapărat numere întregi. De exemplu aflarea unei soluţii a unui sistem de ecuaţii presupune intrinsec efectuarea de operaţii, în particular împărţiri 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.*

Practica a dovedit că majoritatea prelucrărilor în sistemele de calcul se fac asupra numerelor întregi. Operaţiile asupra caracterelor, ca şi asupra numerelor reale, sunt efectuate folosindu-se operaţiile asupra numerelor întregi. Din acest motiv vom acorda o atenţie deosebită datelor de tip întreg.

1.2. NUMERE ÎNTREGI

1.2.1. Baze de numeraţie

În viaţa de zi cu zi folosim *scrierea zecimală* sau reprezentarea în *baza 10*. De exemplu, numărul **39549** înseamnă, de fapt, numărul:

$3*10^4 + 9*10^3 + 5*10^2 + 4*10^1 + 9*10^0 = 30000 + 9000 + 500 + 40 + 9 = 39549$

După cum bine se ştie, numerele pot fi reprezentate şi în alte baze de numeraţie. În ştiinţa şi practica informatică, pe lângă baza 10 sunt frecvent folosite bazele de numeraţie 2, 8 şi 16.

Dacă se lucrează în *baza 8* atunci sunt folosite doar cifrele 0, 1, 2, 3, 4, 5, 6 şi 7, pe care le vom numi *cifre octale*. Să considerăm numărul **115175** reprezentat în baza 8. Valoarea acestuia este:

$1*8^5 + 1*8^4 + 5*8^3 + 1*8^2 + 7*8^1 + 5*8^0 = 32768 + 4096 + 2560 + 64 + 56 + 5 = 39549$

Pentru reprezentarea în *baza 2* sunt folosite doar cifrele 0 şi 1, pe care le vom numi *cifre binare*. Să considerăm numărul **1001101001111101** reprezentat în baza 2. Valoarea lui este:

$1*2^{15} + 0*2^{14} + 0*2^{13} + 1*2^{12} + 1*2^{11} + 0*2^{10} + 1*2^9 + 0*2^8 + 0*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 32768 + 4096 + 2048 + 512 + 64 + 32 + 16 + 8 + 4 + 1 = 39549$

Pentru utilizarea bazei 16 sunt folosite cifrele 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 şi literele A, B, C, D, E, F. Toate aceste 16 simboluri le vom numi în continuare *cifre hexazecimale*. Prin A se notează cifra hexazecimală cu valoarea 10, B este cifra cu valoarea 11, C este cifra cu valoarea 12, D este cifra cu valoarea 13, E este cifra cu valoarea 14 şi F este cifra hexazecimală cu valoarea 15. Să considerăm numărul **9A7D** reprezentat în baza 16. Valoarea lui este:

$9*16^3 + 10*16^2 + 7*16^1 + 13*16^0 = 36864 + 2560 + 112 + 13 = 39549$

Se observă că toate cele patru reprezentări de mai sus, în bazele 10, 8, 2 şi 16 reprezintă acelaşi număr: **39549** (reprezentat în baza 10!)

Apare firesc întrebarea: cum se converteşte un număr natural dintr-o bază în alta? Pentru o prezentare completă a acestor conversii se poate consulta lucrarea [Boian96]. Pentru scopurile noastre este suficient să prezentăm o formă simplificată de conversie folosind ca intermediar baza 10 şi care constă din doi paşi:

1) Conversia unui număr dintr-o bază alta decât 10 în baza 10;
2) Conversia unui număr din baza 10 într-o altă bază.

Înainte de a merge mai departe, ne simţim obligaţi să precizăm că alegerea ca intermediar a bazei 10 s-a făcut pentru că suntem obişnuiţi să facem calcule în această bază! Dacă cineva sau ceva (un calculator) ştie să facă calcule în altă bază, o poate lua ca intermediar pe aceea!
Pentru uşurinţa operării cu bazele de numeraţie 10, 16, 2, 8, recomandăm însuşirea tabelului de mai jos cu reprezentările primelor 16 numere naturale.

Baza 10	Baza 16	Baza 2	Baza 8
0	0	0000	00
1	1	0001	01
2	2	0010	02
3	3	0011	03
4	4	0100	04
5	5	0101	05
6	6	0110	06
7	7	0111	07
8	8	1000	10
9	9	1001	11
10	A	1010	12
11	B	1011	13
12	C	1100	14
13	D	1101	15
14	E	1110	16
15	F	1111	17

1.2.2. Conversii între baze de numeraţie

Pentru a prezenta ***conversia dintr-o bază dată în baza 10*** să analizăm exemplele de mai sus cu valorile în diverse baze. În fiecare dintre ele apare valoarea unei expresii polinomiale în care coeficienţii polinomului sunt valorile cifrelor numărului în baza din care se converteşte, iar argumentul este valoarea bazei în care este reprezentat numărul. Atât valorile cifrelor, cât şi valoarea bazei se vor reprezenta în baza 10 (tabelul de mai sus ne ajută). Numărul obţinut din evaluarea expresiei polinomiale este reprezentarea dorită.

Să reluăm exemplul de mai sus pentru conversia numărului **9A7D** din baza 16 în baza 10. Scriem expresia polinomială:

$(9A7D)_{16} = (9*16^3 + 10*16^2 + 7*16^1 + 13*16^0)_{10} =$
(rescriem sub o formă mai uşoară de calculat)
$= (((9*16 + 10)*16 + 7)*16 + 13) = ((154*16 + 7)*16 + 13) = (2471*16 + 13) = 39549$

Conversia unui număr întreg din baza 10 într-o bază oarecare. Pentru fixarea ideilor, să notăm cu ***d*** numărul de convertit şi cu ***i*** baza în care se doreşte conversia. Regula fundamentală pentru aceste conversii se poate deduce imediat dacă se scrie teorema împărţirii cu rest:

$$d = i * c + r$$

unde d este deîmpărţitul, i este împărţitorul (baza în care se doreşte conversia), c este câtul, iar r este restul împărţirii. Din această scriere se deduce faptul că valoarea cifrei unităţilor în baza i (coeficientul lui i^0 în scrierea polinomială) este restul r al împărţirii lui d la i. Aplicând lui c aceeaşi reţetă se obţine valoarea cifrei unităţilor lui c care este coeficientul lui i^1 în scrierea lui d ş.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 opreşte la ultimul deîmpărţit 0

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

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

Reluăm aceleaşi 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 împărţirilor, vor apare resturi între 0 şi 15. Drept urmare, pentru resturile cu valorile 10 , ..., 15 vom folosi în scrierea în noua

bază simbolurile A, ..., F corespunzătoare. Şi 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 şi 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 aceluiaşi număr în baza 8. Primele trei linii de la conversia în baza 2 au acelaşi efect ca şi prima linie de la conversia în baza 8. Acest lucru este normal, deoarece o împărţire la 8 este echivalentă cu trei împărţiri la 2. Deci, luând în ordine inversă şi concatenând cele trei resturi din baza 2 se obţine $(101)_2$, adică restul $(5)_{10}$ de la conversia în baza 8. Continuând comparaţia, cea de-a doua linie de la conversia în

baza 8 cu următoarele trei linii de la conversia în baza 2, restul în baza 8 se obţine prin concatenarea celor trei resturi de la conversia în baza 2 ş.a.m.d.

Un efect similar se poate observa dacă se compară prima linie de la conversia în baza 16 cu primele patru linii de la conversia în baza 2 etc. Aceste similarităţi nu sunt întâmplătoare, ele decurg din faptul că $8 = 2^3$ şi $16 = 2^4$. Plecând de la aceste observaţii, se pot defini nişte reguli simple de conversie între bazele 2 şi 8, respectiv 2 şi 16.

Orice grup de 3 cifre binare determină, în mod unic, o cifră octală. Reciproc, o cifră octală se reprezintă în binar printr-un grup de 3 cifre binare, completând zerourile necesare la stânga. Un astfel de grup de 3 cifre binare îl numim triadă.

Analog, **orice grup de 4 cifre binare determină, în mod unic, o cifră hexazecimală.** Reciproc, o cifră hexazecimală se reprezintă în binar printr-un grup de 4 cifre binare, completând cu zerourile necesare la stânga. Un astfel de grup de 4 cifre binare îl vom numi tetradă.

Sintetizând cele de mai sus, avem următoarele două reguli practice de trecere între bazele 2 şi 8:

1. Pentru trecerea din baza 2 în baza 8, se grupează cifrele reprezentării binare în triade, pornind de la cifra binară de rang 0 spre stânga. Dacă cel mai din stânga grup al părţii întregi nu are exact trei cifre, se completează cu zerouri la stânga pentru partea întreagă. Apoi se înlocuieşte fiecare triadă cu cifra octală corespunzătoare.
2. Pentru trecerea din baza 8 în baza 2, pornind de la cifra octală de rang 0 spre stânga, se înlocuieşte fiecare cifră octală cu triada binară corespunzătoare ei (fiecare cifră octală se va înlocui cu exact trei cifre binare!).

Să considerăm câteva exemple:

$(115175)_8 = (001\ 001\ 101\ 001\ 111\ 101)_2$
$(100\ 101\ 110)_2 = (456)_8$
$(11\ 100\ 111\ 000\ 001\ 101)_2 = (347015)_8$
$(1153)_8 = (001\ 001\ 101\ 011)_2$

În acelaşi spirit, avem următoarele două reguli practice de trecere între bazele 2 şi 16:

1. Pentru trecerea din baza 2 în baza 16, se grupează cifrele reprezentării binare în tetrade, pornind de la cifra binară de rang 0 spre stânga. Dacă cel mai din stânga grup al părţii întregi nu are exact patru cifre, se completează cu zerouri la stânga pentru partea întreagă. Apoi se înlocuieşte fiecare tetradă cu cifra hexazecimală corespunzătoare.
2. Pentru trecerea din baza 16 în baza 2, pornind de la cifra hexazecimală de rang 0 spre stânga, se înlocuieşte fiecare cifră hexazecimală cu tetrada binară corespunzătoare ei (fiecara cifră hexazecimală se va înlocui cu exact patru cifre binare!).

Să considerăm câteva exemple:

$(9A7D)_{16} = (1001\ 1010\ 0111\ 1101)_2$
$(1\ 0010\ 1110)_2 = (12E)_{16}$
$(1\ 1100\ 1110\ 0000\ 1101)_2 = (1CE0D)_{16}$
$(26B)_{16} = (0010\ 0110\ 1011)_2$

O regulă practică de trecere între bazele 8 şi 16 este aceea al folosirii bazei 2 ca intermediar. Pentru a nu opera cu şiruri nesfârşite de cifre binare, facem următoarele recomandări:

1. Pentru trecerea din baza 8 în baza 16, se grupează la stânga, câte 4 cifre octale. Acestea vor fi transformate mai întâi în 12 cifre binare, care apoi vor fi transformate în 3 cifre hexazecimale.
2. Pentru trecerea din baza 16 în baza 8, se procedează analog: se grupează la stânga 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, $(27354357)_8 = (5DD8EF)_{16}$. Grupările intermediare în baza 2 se pot organiza astfel:

$(\ 2735\ \ 4357\ )_8 =$
$(010\ 111\ 011\ 101\ \ 100\ 011\ 101\ 111\ )_2 =$
$(0101\ 1101\ 1101\ \ 1000\ 1110\ 1111\ )_2 =$
$(\ 5DD\ \ 8EF\ )_{16}$

Lăsăm pe seama cititorului să generalizeze, pentru baze de numeraţie puteri ale lui 2, mecanismele de conversie mai sus prezentate. De asemenea, amatorii de aritmetică pot generaliza mai departe, la cazul bazelor care sunt puteri ale unui număr natural oarecare.

1.3. REPREZENTĂRI BINARE ŞI ORDINI DE PLASARE

Prin particularităţile ei, aritmetica binară se pretează la automatizare mai bine decât aritmetica în orice altă bază de numeraţie. Acesta este motivul pentru care în calculatoare se foloseşte aritmetica în baza 2. În continuare vom folosi alternativ termenul de **bit** ca sinonim pentru **cifră binară.**

1.3.1. Dimensiune a reprezentării

Fiind vorba de calcule efectuate cu o maşină, se impun firesc o serie de restricţii legate de reprezentarea întregilor (şi nu numai). Cea mai importantă dintre ele este dimensiune a reprezentării, adică numărul maxim de cifre binare (numărul de biţi) din reprezentarea unui număr întreg. Să notăm cu n această dimensiune de reprezentare. Numărul n este o constantă a calculatorului, fixată la proiectarea sistemului de calcul respectiv.

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

Tot în categoria restricţiilor intră şi faptul că dimensiunile a doi operanzi care participă la o operaţie, precum şi dimensiunile rezultatului sunt de asemenea constante ale calculatorului, indiferent de tipul de codificare a numerelor. Pentru a preciza regulile de dimensionare în operaţiile binare asupra numerelor întregi, vom folosi sintagma "operaţie pe n biţi". Regulile de dimensionare în urma operaţiilor sunt:
Operaţiile de adunare pe ***n*** biţi şi scădere pe ***n*** biţi presupun că ambii termeni sunt reprezentaţi pe câte ***n*** biţi, iar rezultatul, suma sau diferenţa, se va reprezenta tot pe ***n*** biţi, vezi figura 1.1.

Fig. 1.1. Dimensiuni de reprezentare la adunare şi scădere

Înmulţirea pe ***n*** biţi presupune că ambii factori sunt reprezentaţi pe câte ***n*** biţi, iar produsul lor va fi reprezentat pe **2 * *n*** biţi, vezi figura 1.2.

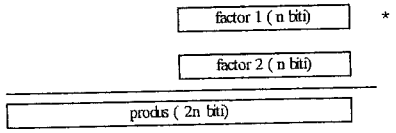


Fig. 1.2. Dimensiuni de reprezentare la înmulţire

Împărţirea pe ***n*** biţi (oarecum invers faţă de înmulţire), impune condiţia ca deîmpărţitul să fie reprezentat pe **2 * *n*** biţi, iar împărţitorul pe ***n*** biţi. Operaţia furnizează două rezultate: câtul reprezentat pe ***n*** biţi şi restul reprezentat tot pe ***n*** biţi, vezi figura 1.3.

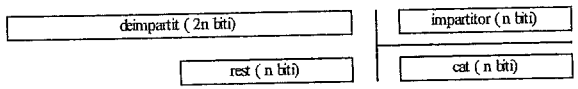


Fig. 1.3. Dimensiuni de reprezentare la împărţire

Dacă rezultatul unei operaţii nu încape în dimensiunea de reprezentare, atunci se vor pierde biţii cei mai semnificativi, rămânând biţii mai puţin semnificativi: 0, 1, 2 ş.a.m.d, calculatorul semnalând fenomenul de ***depăşire.***

1.3.2. Organizarea şi memorarea datelor

1.3.2.1. Bit, octet, locaţie, adresă

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

Unitatea elementară de informaţie este ***bitul.*** Într-un bit se poate reprezenta o informaţie care poate să aibă doar două valori posibile. Tradiţional, aceste valori vor fi 0 şi 1. De exemplu, în funcţie 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 volţi poate însemna 0, în timp ce o tensiune de +5V poate însemna 1.

Definiţie: Un octet (byte) este o succesiune de 8 biţi, numerotaţi de la 0 la 7, ca în figura 1.4.

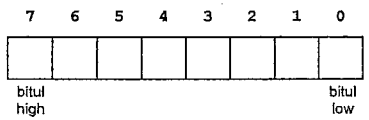


Fig. 1.4. Numerotarea biţilor în cadrul unui octet

Octetul este unitatea elementară de adresare a memoriei. Fiecare octet are ataşat un număr întreg şi nenegativ numit *adresa octetului* respectiv. Primul octet din memorie are adresa 0, al doilea octet are adresa 1, al treilea are adresa 2 ş.a.m.d. Sistemul poate referi / identifica fiecare octet din memorie folosind adresa acestuia. Intuitiv, ne putem imagina memoria ca o mulţime de căsuţe (aşa cum sunt cele de la post-restant). Căsuţele sunt numerotate începând de la 0 şi în fiecare căsuţă poate fi un singur număr. În momentul în care depunem în căsuţă un alt număr, vechiul număr se pierde (ca şi la înregistrările audio / video pe bandă, rămâne doar ultima înregistrare). Numărul de pe uşa căsuţei reprezintă *adresa / referinţa,* iar numărul din căsuţă reprezintă *conţinutul.*

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 poziţia lui faţă de un alt octet. În primul caz vorbim de *adresa absolută a unui octet,* iar în al doilea caz vorbim de *adresa relativă faţă de un alt octet.* În al doilea caz adresa absolută se obţine adunând la adresa octetului de referinţă adresa relativă. De exemplu, dacă un octet **A** are adresa absolută **5643** şi un octet **B** are adresa relativă **5** faţă de **A**, atunci adresa absolută a octetului **B** este **5648**. Ca şi terminologie, spunem că octetul **B** este cu **5** octeţi *mai la dreapta* decât **A**. Vezi în figura 1.5. ilustrarea acestei situaţii.

Conţinut octet 0	Conţinut octet 1	Conţinut octet 2	---	Conţinut octet	---	Conţinut octet	---
Adresa 0	Adresa 1	Adresa 2		Adresa 5643		Adresa 5648	

Fig. 1.5. Succesiunea octeţilor în memorie

Ca şi terminologie privind conţinutul unui octet, vorbim de *bitul 0* al octetului sau bitul *cel mai puţin semnificativ*, bitul de *rang minim*, bitul *cel mai din dreapta*, bitul *low* etc. Bitul 1 are rangul / semnificaţia următoare bitului **0** ş.a.m.d. Bitul 7 este bitul *cel mai semnificativ*, bitul de *rang maxim*, bitul cel mai *din stânga*, bitul *high* etc. De ce s-a adoptat numerotarea biţilor în octet de la dreapta spre stânga ca în figura 1.4.? Pentru faptul că în scrierea numerelor, cifrele (binare) se scriu, în scrierea obişnuită de la dreapta spre stânga în ordinea crescătoare a puterilor bazei.
Aici este momentul să clarificăm două noţiuni fundamentale: **adresă – adresare** şi **conţinut – numerotare conţinut.** În ceea ce priveşte **adresarea**: este unanim acceptată convenţia că adresele octeţilor în memorie cresc de la stânga la dreapta (vezi figura 1.5.); deci la adresare relativă, octetul de referinţă este *la stânga* octetului curent. În ceea ce priveşte **conţinutul** unei locaţii de memorie: numerotările entităţilor dintr-un conţinut se fac de la dreapta spre stânga, aşa cum am văzut la numerotarea biţilor unui octet (vezi figura 1.4); în cele ce urmează vom mai vedea astfel de numerotări şi pentru alte tipuri de conţinuturi.

Entitatea octet este folosită practic de către toate instrucţiunile de prelucrare şi de schimb cu exteriorul ale unui sistem de calcul. În contextul comunicaţiilor trebuie reţinut postulatul că: octetul are aceeaşi reprezentare, indiferent de sistemul de calcul. Cu alte cuvinte, se spune că octetul este portabil. În acelaşi context, terminologia de mai sus este valabilă pentru octet indiferent de sistemul de calcul.

Accesul la biţii unui octet se poate face prin intermediul unor instrucţiuni specializate. În particular, sunt situaţii în care se foloseşte ca şi entitate de execuţie *semioctetul (nibble)*. Un semioctet este format din patru biţi, alăturaţi. Vorbim de *semioctet low*, sau *semioctet mai puţin semnificativ*, sau *semioctet drept*, sau cifră hexazecimală dreaptă, sau *nibble drept* etc. Analog, vorbim de *semioctet high*, sau *semioctet semnificativ*, sau *semioctet stâng*, sau cifră hexazecimală stângă, sau *nibble stâng* etc. Schematic, această împărţire apare ca în figura 1.6.

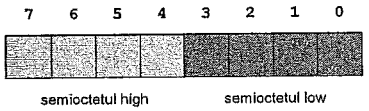


Fig. 1.6. Semiocteţii componenţi ai unui octet

Prelucrările fundamentale sunt efectuate pe octeţi şi pe grupuri de octeţi consecutivi. În particular, operaţiile cu numere întregi se pot efectua fie pe octeţi, fie pe grupuri de octeţi consecutivi.

Prin definiţie, o succesiune de octeţi consecutivi de dimensiune fixată, privită ca o entitate de sine stătătoare formează o ***locaţie*** sau ***unitate de prelucrare***.

Prin definiţie, ***adresa unei locaţii*** este egală cu adresa primului octet component al locaţiei (cu cea mai mică adresă a octeţilor ce o compun).

Dimensiunea unei locaţii este egală cu numărul de octeţi care o compun.
Dimensiunea şi denumirea pe care o poartă o locaţie, variază de la un tip de sistem de calcul la altul. Spre exemplu, la unele sisteme, cum ar fi cele din familia IBM-PC:

- doi octeţi consecutivi formează un *cuvânt*;
- patru octeţi consecutivi formează un *dublucuvânt*.

La astfel de sisteme vorbim de locaţii octet, locaţii cuvânt şi locaţii dublucuvânt. În figura 1.7. prezentăm un cuvânt cu subdiviziunile (subentităţile) lui, iar în figura 1.8. prezentăm un dublucuvânt cu subdiviziunile (subentităţile) lui.

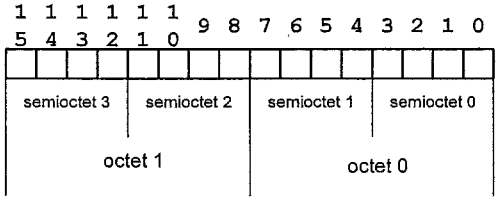


Fig. 1.7. Un cuvânt IBM-PC

Evident, pentru cuvântul din figura 1.7., semioctetul 0 şi octetul 0 sunt respectiv *semioctetul low* şi *octetul low* din cadrul locaţiei. Similar, semioctetul 3 şi octetul 1 sunt respectiv *semioctetul high* şi *octetul high* din cadrul locaţiei.

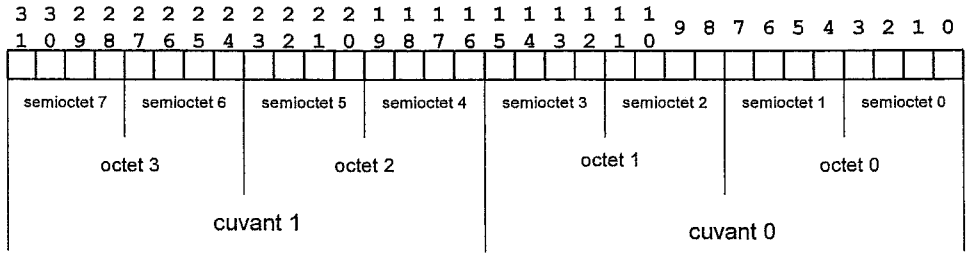


Fig. 1.8. Un dublucuvânt IBM-PC

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

La alte tipuri de sisteme de calcul, printre care şi la supercalculatoare, se vorbeşte de locaţii octet, locaţii semicuvânt, locaţii cuvânt şi locaţii dublucuvânt:

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

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

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

Lăsând detaliile de reprezentare pentru mai târziu, enumerăm câteva dimensiuni ale locaţiilor de reprezentare:

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

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

Pentru ca cititorul să îşi facă o idee despre ce înseamnă interpretarea conţinutului unei locaţii, vom da câteva exemple de reprezentări ale unor date elementare. Conţinuturile locaţiilor le vom scrie în hexazecimal:

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

- Numere **1** şi **-1**, dar de această dată interpretate ca şi numere *reale* se reprezintă:
  - **3F800000** respectiv **BF800000** – standardul IEEE simplă precizie,
  - **3FF0000000000000** respectiv **BFF0000000000000** – standardul IEEE dublă precizie,
  - **000000000081** respectiv **800000000081** – standardul Turbo Pascal.

Conform cu cele prezentate la numerotarea octeţilor într-o locaţie, 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**.
- ş.a.m.d.

1.3.2.3. Ordinea octeţilor într-o locaţie; maşini little-endian şi maşini big-endian

La nivelul unei locaţii, privită ca o entitate de sine stătătoare cu conţinut interpretat în funcţie de standardul de reprezentare, se disting două abstracţiuni:

- Numerotarea octeţilor în cadrul unei locaţii fixată de standardul de reprezentare;
- Adresele octeţilor care compun locaţia.

Aşa cum am precizat deja în secţiunea 1.3.2.1, numerotările de conţinuturi se fac de la dreapta spre stânga, iar adresele cresc de la stânga spre dreapta. În mod normal, se pune problema unei corespondenţe între aceste două elemente. Pe de o parte vorbim de *reprezentarea structurală* a unui tip de date, impusă de standardul de reprezentare. Pe de altă parte vorbim de *memorarea concretă* a datei în locaţie: în care octet al locaţiei memorăm octetul 0 al reprezentării, în care octet al locaţiei memorăm octetul 1 ş.a.m.d. Cu alte cuvinte trebuie stabilită o *corespondenţă* între ordinea octeţilor impusă de reprezentarea structurală şi adresele din locaţie în care se memorează valorile acestor octeţi.

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

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

Să alegem, spre exemplu, numărul $(1025)_{10}$ pe care să îl reprezentăm într-o locaţie de patru octeţi. Pentru această dimensiune a locaţiei, reprezentările lui în bazele 16 şi 2 sunt $(00000401)_{16}$,

respectiv **(00000000 00000000 00000100 00000001)**$_2$. Să presupunem că **B** este adresa locației în care numărul este plasat în ordinea big-endian, iar **L** este adresa locației în care același număr este plasat în ordinea little-endian. Cele două plasări sunt ilustrate în figura 1.9., cu conținuturile octeților scrise atât în hexazecimal, cât și în binar:

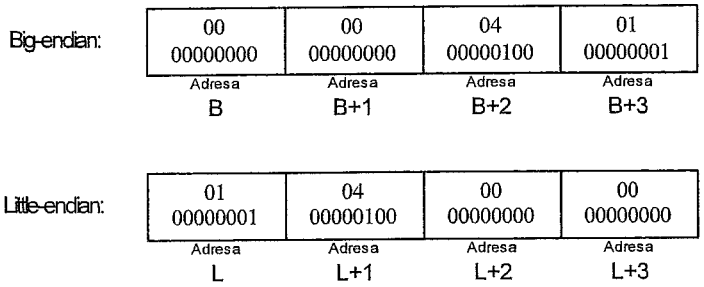


Fig. 1.9. Plasările big-endian și little-endian

În secțiunea precedentă am prezentat câteva exemple de reprezentări, cu diverse standarde, a caracterelor 'M' și 'm' și a numerelor 1, -1, 2 și -2. Tabelele care urmează prezintă, în hexazecimal, plasarea big-endian și little-endian a acestor reprezentări.

Exemple de plasări big-endian									
Dată	Standard de reprezentare	Adresele relative ale octeților în locație							
		0	1	2	3	4	5	6	7
'M'	ASCII	4D							
'M'	UNICODE	00	4D						
'm'	ASCII	6D							
'm'	UNICODE	00	6D						
1	întreg, 2 octeți	00	01						
-1	întreg, 2 octeți	FF	FF						
1	întreg, 4 octeți	00	00	00	01				
-1	întreg, 4 octeți	FF	FF	FF	FF				
2	întreg, 2 octeți	00	02						
-2	întreg, 2 octeți	FF	FE						
2	întreg, 4 octeți	00	00	00	02				
-2	întreg, 4 octeți	FF	FF	FF	FE				
1	IEEE, simplă precizie	3F	80	00	00				
-1	IEEE, simplă precizie	BF	80	00	00				
1	IEEE, dublă precizie	3F	F0	00	00	00	00	00	00
-1	IEEE, dublă precizie	BF	F0	00	00	00	00	00	00
1	Turbo Pascal	00	00	00	00	00	81		
-1	Turbo Pascal	80	00	00	00	00	81		

Plasarea little-endian sau big-endian este o caracteristică a sistemului de calcul. De multe ori se folosește termenul de *arhitectură big-endian*, respectiv *arhitectură little-endian*. Un procesor are instrucțiuni specializate care să opereze cu fiecare tip de locație, instrucțiuni care "știu" standardul de reprezentare și ordinea de plasare. Utilizatorul trebuie doar să comande procesorului operația dorită și adresa locației de unde aceasta să își ia reprezentarea datei.

Exemple de plasări little-endian									
Dată	Standard de reprezentare	Adresele relative ale octeților în locație							
		0	1	2	3	4	5	6	7
'M'	ASCII	4D							
'M'	UNICODE	4D	00						
'm'	ASCII	6D							
'm'	UNICODE	6D	00						
1	Întreg, 2 octeți	01	00						
-1	Întreg, 2 octeți	FF	FF						
1	Întreg, 4 octeți	01	00	00	00				
-1	Întreg, 4 octeți	FF	FF	FF	FF				
2	Întreg, 2 octeți	02	00						
-2	Întreg, 2 octeți	FE	FF						
2	Întreg, 4 octeți	02	00	00	00				
-2	Întreg, 4 octeți	FE	FF	FF	FF				
1	IEEE, simplă precizie	00	00	80	3F				
-1	IEEE, simplă precizie	00	00	80	BF				
1	IEEE, dublă precizie	00	00	00	00	00	00	F0	3F
-1	IEEE, dublă precizie	00	00	00	00	00	00	F0	BF
1	Turbo Pascal	81	00	00	00	00	00		
-1	Turbo Pascal	81	00	00	00	00	80		

Fiecare dintre cele două moduri de plasare are avantajele și dezavantajele ei. Disputa big-endian – little-endian este lungă și exotică, există argumente pro și contra de ambele părți. De exemplu, forumul Internet [http://www.rdrop.com/~cary/html/endian_faq.html] întreține o dezbatere permanentă pe acest subiect. Pentru scopurile noastre, vom prezenta două criterii de comparare, fiecare dintre ele fiind avantajos pentru o arhitectură și dezavantajos pentru cealaltă. Aceste criterii sunt:

- Conversia unui întreg de la o reprezentare mai mare la una mai mică. De exemplu, dacă se cere ca un întreg dintr-o locație de 4 octeți, dar care încape de fapt pe 2 octeți, să fie prelucrat, cu aceeași valoare, ca și când ar face parte dintr-o locație pe 2 octeți. Comparativ, acest criteriu reprezintă:
  - avantaj little-endian, dezavantaj big-endian, argumentarea în paragraful următor.

- Depistarea bitului de semn. Este unanim acceptat faptul că semnul unui număr, întreg sau real, se reprezintă pe un bit, cu valoarea 0 pentru număr pozitiv și cu valoarea 1 pentru

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 şi exemplu de comparare reprezentările lui 2 pe 4 octeţi, atât în plasarea big-endian, cât şi în plasarea little-endian, ilustrate în figura 1.10.

Pentru conversie, în cazul unei maşini little endian adresa locaţiei rămâne **L**, indiferent că aceasta are dimensiunea de 4 octeţi, de 2 octeţi, sau chiar de 1 octet, modificându-se doar dimensiunea locaţiei (deci locaţia ce conţine numărul 2 va avea aici aceeaşi adresă **L**, indiferent dacă el este reprezentat pe 1, 2 sau 4 octeţi!). La maşina big-endian, adresele locaţiilor trebuie modificate în funcţie de dimensiunea acestora: **B+2** pentru locaţia de 2 octeţi (conţinând octeţii de adrese **B+2** şi **B+3**) şi **B+3** pentru locaţia 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	Adresa B+1	Adresa B+2	Adresa B+3
Little-endian:	02 00000010	00 0000000	00 00000000	00 00000000
	Adresa L	Adresa L+1	Adresa L+2	Adresa L+3

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

Pentru bitul de semn, în cazul unei maşini big-endian adresa octetului cu bitul de semn coincide cu adresa locaţiei. Cu notaţiile din figura 1.10., bitul de semn, la maşina big-endian, se află la adresa **B**, indiferent de faptul că se va prelucra o locaţie de 4 octeţi, de 2 octeţi sau de 1 octet. În cazul unei maşini little-endian, bitul de semn se află în ultimul octet al locaţiei (cel cu cea mai mare adresă), aşa că pentru a-l obţine procesorul trebuie să calculeze adresa aceşti octet: ea este **L** pentru locaţia de 1 octet, **L+1** pentru locaţia de 2 octeţi şi **L+3** pentru cea de 4 octeţi.

Utilizatorul poate să interpreteze în mod diferit conţinutul aceleiaşi locaţii! De exemplu, poate să memoreze într-un şir de 4 octeţi un număr întreg, după care să comande operarea asupra aceleiaşi arii de memorie prin patru instrucţiuni care să utilizeze 4 locaţii consecutive de tip caracter reprezentat pe octet. În astfel de situaţii, când la momente diferite se interpretează diferit aceeaşi arie de memorie, trebuie să se ţină cont de ordinea de plasare şi de standardele de reprezentare ale datelor elementare.

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

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

Şi acum câteva cuvinte "exotice" legate de arhitecturile big-endian şi 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 şir de 4 octeţi se doreşte încărcarea cuvântului **'UNIX'**. Un programator "isteţ", în loc să comande încărcarea pe rând în cele patru locaţii succesive a câte unui caracter pentru a obţine succesiunea **('U','N','I','X')**, face numai două încărcări a câte unui întreg pe doi octeţi: mai întâi încarcă şirul **'UN'** privit ca un întreg, apoi în locaţia întreagă următoare încarcă şirul **'IX'** privit tot ca un întreg! În acest mod a făcut "economie" de două instrucţiuni.

Toate bune şi frumoase, el speră că a obţinut **('UN','IX')**, ceea ce coincide cu ceea ce doreşte. Într-adevăr este aşa dacă maşina este una big-endian.
În schimb dacă este una little-endian, atunci primii doi octeţi vor conţine **'NU'** şi următorii doi **'XI'**. Deci, el va obţine **('NU','XI')** privind ca un şir de patru caractere, deci de fapt obţine **'NUXI'** !

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

Adjectivele *big-endian* şi *little-endian* au şi 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 Guliver". Acolo se vorbeşte de facţiunea politică "Big Endians" care susţinea că după fierbere oul trebuie spart pe la capătul mai larg (big-endian), în opoziţie cu facţiunea rebelă "Little Endian" potrivnică regelui liliputanilor, care susţine că oul trebuie spart la capătul mai ascuţit (little-endian)!

1.3.2.4. Unităţi de capacitate a memoriei

Prin capacitatea de memorare a unui sistem de calcul înţelegem *numărul total de octeţi ai unităţii de memorie.*

În practică se folosesc o serie de multipli ai numărului de octeţi. Spre deosebire de multiplii folosiţi în activitatea cotidiană, unităţile multipli ale capacităţii de memorare sunt exprimate sub formă de puteri ale lui 2, astfel:

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

Spre a avea o idee asupra capacităţilor de memorare actuale, un calculator personal obişnuit are memoria internă între 256 Mo şi 4 Go. Capacitatea unui hard – disc actual este între 20 Go şi 512 Go. Pentru comparaţie, o pagină de format A4 – echivalentă cu o pagină dintr-o carte, conţine cu puţin peste 3000 de caractere: litere mari / mici, cifre, semne speciale. Aşa cum vom vedea mai târziu, memorarea unui text se face un caracter pe octet. Rezultă că pentru o pagină sunt necesari 3 Ko de memorie. Drept urmare, într-un Mo de memorie se pot memora cam 350 pagini de carte.
O altă comparaţie: un CD obişnuit (nu DVD) are capacitatea de 700 Mo. Într-un astfel de CD se poate memora lista tuturor abonaţilor telefonici Romtelecom din Romania, cu toate informaţiile necesare: nume, prenume, adresa completă, telefon.

1.4. CODIFICAREA CARACTERELOR

Să considerăm mulţimea *caracterelor tipăribile*, existente la fiecare imprimantă, tastatură, care apar pe ecran etc. Este vorba de literele (cele 26 din alfabetul englez) mari şi mici, cifrele, semnele speciale etc. Pentru codificarea acestor caractere în vederea prelucrării lor automate, organizaţiile internaţionale de profil au definit o serie standarde de reprezentare. Aceste standarde atribuie câte un număr întreg fiecărui caracter, iar valorile de codificare a caracterelor dintr-o anumită grupă verifică anumite condiţii. După cum se va vedea, existenţa acestor condiţii este benefică pentru prelucrarea automată a caracterelor.

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

În prezent, cel mai folosit sistem de codificare este ASCII (American Standard Code for Information Interchange). ASCII ca şi standard este un cod pe 7 biţi, folosind numerele întregi din intervalul [0,127]. În ASCII este codificat un caracter pe un octet, iar bitul 7 (cel de-al 8-lea) este automat 0. Practic, toate calculatoarele actuale folosesc ASCII pentru codificarea caracterelor. Firma IBM a propus (şi la propunere au aderat practic toate marile case de software şi hardware), extinderea ASCII folosind şi cel de-al 8-lea bit din octet, pentru codificarea unor caractere grafice speciale.

În contextul cerinţelor de internaţionalizare impuse de existenţa Internet, în ultimii 10 ani se impune din ce în ce mai mult standardul de codificare UNICODE. Acesta codifică un caracter pe doi octeţi, pentru a se permite codificarea simbolurilor şi a caracterelor folosite în scrierile majorităţii limbilor de pe planetă.

În cele ce urmează ne rezumăm la prezentarea standardului de codificare ASCII. Standardul ASCII împarte caracterele în următoarele cinci grupe:

1. *Literele mici* ale alfabetului a, b, ... , z.
2. *Literele mari* ale alfabetului A, B, ... , Z.
3. Cifrele zecimale 0, 1, ... , 9.
4. O serie de *caractere speciale*: spaţiul, virgula, punctul, +, -, _, $, & ş.a.m.d

5. Un set de *caractere funcţionale* care nu apar la tipărire / afişare, ci doar dirijează tipărirea / afişarea.

Caracterele funcţionale apar în tabelele de definiţie sub forma unor grupuri de litere, ca de exemplu CR, LF, TAB, FF, BEL, BS ş.a.m.d. CR provoacă deplasarea dispozitivului de afişare (tipărire) la început de rând (Carriage Return). LF sau NL provoacă deplasarea dispozitivului cu un rând mai jos (Line Feed sau New Line), păstrându-se poziţia în cadrul rândului. În funcţie de sistem, pentru a separa două linii dintr-un text se foloseşte fie LF, fie succesiunea de caractere CR LF. TAB este caracterul de tabulare, deci avansul dispozitivului la poziţia următorului stop de tabulare (de obicei peste 5-8 caractere). FF (Form Feed) provoacă trecerea la pagina (ecranul) următoare (următor). BEL provoacă emiterea unui semnal sonor, iar BS (backspace) provoacă deplasarea dispozitivului de afişare (tipărire) cu o poziţie spre stânga, în vederea ştergerii (supraimprimării) ultimului caracter.

Condiţiile de codificare pe care le respectă standardul ASCII sunt:

- Toate caracterele funcţionale au codul mai mic decât codul caracterului spaţiu.
- Codul caracterului spaţiu este mai mic decât codurile celorlalte caractere tipăribile.
- Literele mici sunt codificate prin 26 numere consecutive, în ordine alfabetică.
- Literele mari sunt codificate prin 26 numere consecutive, în ordine alfabetică.
- Cifrele zecimale sunt codificate prin 10 numere consecutive, în ordinea valorilor.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Fig. 1.11. Codul ASCII standard

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

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	ÿ	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	¢	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
141	ì	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Source: www.LookupTables.com

Fig. 1.12. Codul ASCII extins

1.5. CODIFICAREA NUMERELOR ÎNTREGI

1.5.1. Convenţie cu semn şi convenţie fără semn

Aşa 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 conţinuturile locaţiilor de numere întregi cu care operează în una din următoarele două convenţii:

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

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

Tabelul următor prezintă intervalele de numere reprezentabile într-o locaţie, atât în convenţia fără semn, cât şi în convenţia cu semn, în funcţie de dimensiunea acesteia.

Nr. octeţi	Convenţia fără semn	Convenţia cu semn
1	$[0, 2^8-1] = [0, 255]$	$[-2^7, 2^7-1] = [-128, 127]$
2	$[0, 2^{16}-1] = [0, 65535]$	$[-2^{15}, 2^{15}-1] = [-32\,768, 32\,767]$
4	$[0, 2^{32}-1] = [0, 4\,294\,967\,295]$	$[-2^{31}, 2^{31}-1] = [-2\,147\,483\,648, 2\,147\,483\,647]$
8	$[0, 2^{64}-1] = [0, 18\,446\,824\,753\,389\,551\,615]$	$[-2^{63}, 2^{63}-1] = [-9\,223\,412\,376\,694\,775\,808, 9\,223\,412\,376\,694\,775\,807]$

Acestea sunt convenţiile impuse constructorilor de procesoare. Implementările operaţiilor 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 operaţiilor fundamentale peste numere întregi, indiferent de convenţia de reprezentare. Modul în care se realizează aceste implementări şi măsura în care se vor folosi algoritmi comuni ambelor convenţii de reprezentare se va vedea în secţiunea următoare.

1.5.2. Bitul de semn; codul complementar

Dacă interpretăm o anumită configuraţie de biţi drept întreg cu semn, atunci prin convenţie, pentru reprezentarea semnului unui număr se foloseşte un singur bit. Este vorba de *bitul high* (bitul 7) *din octetul high* al locaţiei 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 locaţie pe doi octeţi (16 biţi), dăm câteva exemple de configuraţii (scrise în hexazecimal şi în binar), pe care dacă le interpretăm ca şi 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.

Şi acum întrebarea (pe care ne-o punem de câteva secţiuni încoace): cum se reprezintă numerele întregi în convenţia cu semn?

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

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

cod direct. Soluţia, foarte apropiată de cea naturală – pe care o folosim în calculele manuale, s-a dovedit a fi mai puţin eficientă decât altele. Azi se foloseşte codul direct numai la reprezentarea numerelor reale.

O a doua direcţie este aceea a reprezentării valorii absolute a numărului pe n-1 biţi din cei n ai locaţiei, iar în cazul în care numărul este negativ, să se inverseze toţi cei n biţi ai reprezentării. În acest mod bitul de semn va deveni automat 1. Această reprezentare poartă numele de cod invers, sau complement faţă de 1. A fost, de asemenea, o încercare de codificare la care s-a renunţat. Pentru detalii privind codul direct şi codul invers, cititorul poate consulta [Boian96].

A treia direcţie, care de fapt s-a impus pentru reprezentarea numerelor întregi cu semn poartă numele de cod complementar, sau complement faţă de 2. Pe tot restul prezentei lucrări vom discuta despre acest cod.

Pentru început vom defini operaţia de complementare, numită de multe ori şi operaţia de schimbare de semn.

***Definiţie**: Pentru **complementarea** unui număr întreg reprezentat pe n biţi, mai întâi se inversează valorile tuturor biţilor (valoarea 0 devine 1 şi valoarea 1 devine 0) din locaţia de reprezentare, după care se adaugă 1 la valoarea obţinută.*

Înainte de a prezenta câteva exemple de complementare, mai prezentăm regula de complementare în alte variante, convenabile mai ales când se complementează manual:

1.5.2.1. Reguli alternative de complementare:

Se lasă neschimbaţi biţii începând din dreapta reprezentării binare până la primul bit 1 inclusiv; restul biţilor se inversează până la bitul n-1 inclusiv.

sau

Se scade binar conţinutul (evident binar) al locaţiei de complementat din 100 ...00, unde numărul de după cifra binară 1 are atâtea zerouri câţi biţi are locaţia de complementat.

sau

Se scade hexazecimal conţinutul (evident hexazecimal) al locaţiei de complementat din 100...00, unde după cifra hexazecimală 1 apar atâtea zerouri câte cifre hexazecimale are locaţia de complementat.

Lăsăm pe seama cititorului să verifice că aceste trei variante sunt echivalente cu regula de complementare, dată mai sus prin definiţie.

De exemplu, dacă vrem să complementăm o locaţie de un octet şi care conţine numărul $(18)_{10}$:

Locaţia iniţială:	00010010
După inversarea biţilor	11101101
Se adaugă 1	11101101+
	00000001
Complementul:	11101110

Deci numărul $(18)_{10}$, adică $(12)_{16}$, adică $(00010010)_2$, are ca şi complement numărul $(11101110)_2$ adică $(EE)_{16}$, adică $(238)_{10}$.
Aplicând regula de scădere binară, avem:

	100000000-
Locaţia iniţială:	00010010
Complementul:	11101110

Aplicând regula de scădere hexazecimală, avem:

	100-
Locaţia iniţială:	12
Complementul:	EE

Natural, regulile de complementare sunt valabile şi la locaţii de 2 octeţi, si la locaţii de 4 octeţi etc. Lăsăm pe seama cititorului să complementeze şi să verifice, folosind una (sau mai multe) din regulile de mai sus că într-o locaţie de 2 octeţi numerele, scrise hexazecimal, **9A7D** şi **7583** sunt complementare. De asemenea, într-o locaţie de 4 octeţi, numerele **000F095D** şi **FFF0F6A3** sunt complementare.

De asemenea, se poate verifica că pe 1 octet numerele **7F** şi **81** sunt complementare şi pe 2 octeţi **7FFF** şi **8001** sunt complementare.

Să complementăm acum locaţia cu conţinutul $(EE)_{16}$, adică rezultatul unei complementări precedente. Avem:

	100-
Locaţia iniţială:	EE
Complementul:	12

Deci tocmai numărul de la care am plecat! Într-adevăr, complementarea complementului este numărul iniţial supus complementării.

Şi acum să adunăm pe 8 biţi un număr cu complementul său. Printr-o adunare pe **n** biţi înţelegem că se ignoră transportul de cifră semnificativă începând cu cifra binară de rang **n**. Avem:

	00010010+
Se ignoră acest transport:	11101110
1	00000000

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

Locaţia iniţială:	00000000
După inversarea biţilor	11111111
Se adăugă 1	11111111+
(se ignoră ultimul transport)	00000001
Complementul:	00000000

Locaţia iniţială:	10000000
După inversarea biţilor	01111111
Se adăugă 1	01111111+
	00000001
Complementul:	10000000

În sfârşit, prezentăm **regula de reprezentare a numerelor întregi cu semn**: Un număr întreg între -2^{n-1} şi $2^{n-1}-1$ se reprezintă într-o locaţie de **n** biţi astfel:

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

Înainte de a trece la exemple, trebuie să clarificăm situaţia reprezentării numărului -2^{n-1}. Valoarea lui absolută nu poate fi reprezentată pe **n-1** biţi ca să rămână loc şi pentru bitul de semn, ci el se reprezintă pe **n** biţi şi 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 convenţie s-a stabilit că numărul -2^{n-1} se reprezintă în cod complementar pe n biţi prin **100...0.** Aceeaşi configuraţie interpretată fără semn reprezintă numărul 2^{n-1}.
Tabelul următor prezintă reprezentările mai multor numere, în locaţii de 8 biţi – 1 octet, 16 biţi – 2 octeţi şi 32 de biţi – 4 octeţi.

Dim. locaţie (octeţi)	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	0000000001111111
1	-128	80	10000000
2	-128	FF80	1111111110000000
2	128	0080	0000000010000000
2	32767	7FFF	0111111111111111
2	-32767	8001	1000000000000001
2	-32768	8000	1000000000000000
4	-32768	FFFF8000	11111111111111111000000000000000
4	32768	00008000	00000000000000001000000000000000
1	18	12	00010010
2	18	0012	0000000000010010
1	-18	EE	11101110
2	-18	FFEE	1111111111101110
4	39549	00009A7D	00000000000000001001101001111101
4	-39549	FFFF6583	11111111111111111011001011000011
4	985437	000F095D	00000000000011110000100101011101
4	-985437	FFF0F6A3	11111111111100001111011010100011

1.5.3. Operaţii aritmetice; conceptul de depăşire

1.5.3.1. De ce codul complementar?

Spuneam într-o secţiune precedentă că implementările operaţiilor 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 operaţiilor fundamentale peste numere întregi, indiferent de convenţia de reprezentare.

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

- Operaţia de adunare se execută la fel, indiferent de faptul că avem de-a face cu convenţia de reprezentare fără semn sau cea de reprezentare cu semn. Operaţia executată este o adunare simplă, pe n biţi (n – dimensiunea locaţiei), cu ignorarea ultimului transport.
- Operaţia de scădere se reduce la operaţia de adunare a descăzutului cu complementul scăzătorului.

După cum se apreciază, procentul operaţiilor aditive – adunări şi scăderi este mult mai mare în aplicaţii decât cel al operaţiilor multiplicative. De aici şi preferinţa proiectanţilor pentru adoptarea codului complementar pentru reprezentarea întregilor cu semn. În schimb, operaţiile de înmulţire şi împărţire sunt efectuate cu algoritmi separaţi pentru reprezentările fără semn şi reprezentările cu semn.
În consecinţă, programatorul îşi alege convenţia cu semn sau fără semn în funcţie de specificul problemei. El utilizează aceleaşi operaţii pentru adunări şi scăderi şi operaţii specifice cu semn sau fără semn pentru operaţiile de înmulţire şi împărţire.

1.5.3.2. Conceptul de depăşire

Aşa după cum am arătat mai sus, reprezentarea numerelor întregi se face în locaţii cu dimensiune fixată apriori. În mod natural, trebuie să ne punem problema: ce se întâmplă când rezultatul nu încape în spaţiul care-i este rezervat?
Generic vorbind, *depăşirea* apare atunci când rezultatul unui calcul nu încape în spaţiul care-i este rezervat.

Specific vorbind, condiţiile de apariţie a unei depăşiri apar în mod diferit în funcţie de context. Contextul depinde, pe de o parte dacă se utilizează convenţia fără semn sau convenţia cu semn. Pe de altă parte, contextul depinde de tipul operaţiei: aditivă sau multiplicativă.

Procesoarele sunt astfel construite încât să semnaleze, în mod specific, fiecare situaţie de depăşire. Rămâne în sarcina utilizatorului dacă ia în calcul semnalele procesorului şi pe care anume.

În cele ce urmează vom aborda, pe rând tipurile de operaţii şi în cadrul acestora vom semnala situaţiile de depăşire.

1.5.3.3. Adunări şi scăderi

Operaţiile de adunare şi de scădere sunt efectuate de calculator exact după algoritmii cunoscuţi din clasa I primară, cu singura deosebire că operaţiile sunt efectuate în baza 2. În schimb, datorită regulilor de dimensionare şi a faptului că operaţia este efectuată de maşină şi nu de om, există posibilitatea de apariţie a depăşirilor.

Pentru cazul operaţiilor fără semn, există două reguli care provoacă depăşire:

*R1) Dacă rezultatul adunării nu încape pe **n** biţi, atunci apare depăşire la adunare şi rămân în rezultat numai biţii de la ordinul 0 la ordinul **n-1**, iar bitul de ordin **n** se pierde.*

R2) Dacă într-o operaţie de scădere descăzutul este mai mic decât scăzătorul, atunci are loc depăşire la scădere, dar operaţia se execută, făcându-se "împrumut fictiv" de la un rang inexistent.

Aceste reguli sunt intuitiv clare (justificabile) şi se pot aplica şi tehnic ca atare.

Dăm, în continuare, câteva exemple de operaţii de adunare şi scădere analizate în convenţia fără semn. Pentru fiecare operaţie, vom scrie mai întâi operanzii în baza 10, apoi în baza 16 şi apoi în baza 2. Dimensiunea locaţiilor este fie de un octet, fie de doi octeţi.

Baza 10	Baza 16	Baza 2	Observaţii
18+ 18 36	12+ 12 24	00010010+ 00010010 00100100	
243- 18 225	F3- 12 E1	11110011- 00010010 11100001	
243+ 18 261	F3+ 12 05	11110011+ 00010010 00000101	Depăşire la adunare, conform regulii R1!
18- 243 -225	12- F3 1F	00010010- 11110011 00011111	Depăşire la scădere, conform regulii R2!
575+ 650 1225	023F+ 028A 04C9	0000001000111111+ 0000001010001010 0000010011001001	
650- 575 75	028A- 023F 004B	0000001010001010- 0000001000111111 0000000001001011	
65535+ 90 65625	FFFF+ 005A 0059	1111111111111111+ 0000000001011010 0000000001011001	Depăşire la adunare, conform regulii R1!
575- 650 -75	023F- 028A FFB5	0000001000111111- 0000001010001010 1111111110110101	Depăşire la scădere, conform regulii R2!

Să ne ocupăm în continuare de operaţiile de adunare şi scădere în convenţia cu semn. Toate numerele se vor considera ca fiind reprezentate în cod complementar.

Mai întâi trebuie să reamintim faptul că în cod complementar scăderea înseamnă, de fapt, adunarea descăzutului cu complementul scăzătorului (vezi secţiunea 1.5.3.1.). Pentru a surprinde situaţiile de depăşire este suficient să studiem efectuarea de sume ale unor numere din intervalul $[-2^{n-1}, 2^{n-1}-1]$ şi de a depista aici situaţiile în care poate să apară depăşire, adică neîncadrarea sumei în intervalul admis $[-2^{n-1}, 2^{n-1}-1]$ pentru suma pe n biţi.

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

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

Inegalităţile din cazul 1) ne asigură că în primul caz suma algebrică aparţine aceluiaşi interval. Într-adevăr, din inegalităţile: $-2^{n-1} \leq x < 0$ şi $0 \leq y \leq 2^{n-1}-1$ avem, pe de o parte, că $x \leq x + y$, deci $-2^{n-1} \leq x + y$. Pe de altă parte, din aceleaşi inegalităţi rezultă că $x + y \leq y$, deci $x + y \leq 2^{n-1}-1$. În concluzie, $x + y$ aparţine intervalului $[-2^{n-1}, 2^{n-1}-1]$. De aici se deduce că <u>dacă două numere sunt de semne contrare, atunci suma lor nu poate produce depăşire.</u> Aşa cum se va vedea din exemplele care urmează, atunci când se face suma codurilor complementare este posibil să apară <u>transport de cifră semnificativă</u>, dar acest fapt este ignorat, deoarece nu produce depăşire.

În cazurile 2) şi 3) este posibilă apariţia depăşirii. În cazul 2) $x + y$ poate avea valoarea maximă 2^n-2, iar depăşire apare dacă $x + y > 2^{n-1}-1$. Deoarece codurile complementare se adună ca şi numere fără semn, rezultă că depăşire apare atunci când pe poziţia 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 depăşire apare dacă $x + y < -2^{n-1}$. Analog ca mai sus, rezultă depăşire dacă pe poziţia bitului de semn apare cifra 0, ceea ce în cod complementar înseamnă număr pozitiv!

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

R3) Suma a două numere reprezentate în cod complementar provoacă depăşire dacă şi numai dacă sunt de acelaşi semn şi rezultatul sumei lor este de semn <u>contrar.</u>

Din această regulă se poate deduce şi regula depăşirii 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) Diferenţa a două numere reprezentate în cod complementar provoacă depăşire dacă şi numai dacă scăzătorul şi diferenţa sunt de acelaşi semn şi descăzutul este de semn <u>contrar.</u>

Practic, putem identifica două tipuri de situaţii ce vor semnala depăşire la scăderea cu semn, în situaţia b) fiind necesar un "împrumut fictiv".

```
a)                b)
  1--------        0--------
  0--------        1--------
  0--------        1--------
```

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

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

Aşa cum se va vedea din exemplele care urmează, atunci când se face diferenţa codurilor complementare este posibil să apară <u>împrumut fictiv de cifră semnificativă</u>, dar acest fapt este ignorat, deoarece nu produce depăşire.

În continuare prezentăm câteva exemple de adunări şi scăderi ale unor <u>numere reprezentate în cod complementar</u>. Pentru simplitatea expunerii, vom utiliza operanzii reprezentaţi în cod complementar <u>pe 4 biţi</u>. Conform celor de mai sus, pentru patru biţi intervalul de reprezentare este $[-2^3, 2^3-1]$, adică $[-8, 7]$. Operanzii îi vom transcrie din baza 10 direct în cod complementar. Vom semnala cazurile de transport, împrumut fictiv şi depăşire.

Suma în baza 10	Suma în cod complementar	Transport sau depăşire	Diferenţa în baza 10	Diferenţa în cod complementar	Împrumut sau depăşire
(-7) + 5 = -2	1001+ 0101 1110		5 – 7 = - 2	0101- 0111 1110	
(-4) + 4 = 0	1100+ 0100 0000	Transport	4 – 4 = 0	0100- 0100 0000	
2 + (-7) = -5	0010+ 1001 1011		2 – 7 = -5	0010- 0111 1011	Împrumut
5 + (-2) = 3	0101+ 1110 0011	Transport	5 – 2 = 3	0101- 0010 0011	
3 + 4 = 7	0011+ 0100 0111				
(-4) + (-1) = -5	1100+ 1111 1011		(-4) – 1 = -5	1100- 0001 1011	
5 + 2 = 7	0101+ 0010 0111				

(-5) + 2 = -3	1011+ 0010 1101		2 – 5 = -3	0010- 0101 1101	Împrumut
(-8) + 7 = -1	1000+ 0111 1111		Nu se poate scrie 7-8 pe 4 biţi, deoarece 8 nu aparţine intervalului [-8,7].		
5 + 4 = 9	0101+ 0100 1001	Depăşire!			
(-7) + (-6) = -13	1001+ 1010 0011	Depăşire!	(-7) - 6 =-13	1001- 0110 0011	Depăşire!
(-6) + (-4) = -10	1010+ 1100 0110	Depăşire!	(-6) – 4 =-10	1010- 0100 0110	Depăşire!
7 + 7 = 14	0111+ 0111 1110	Depăşire!			
(-8) + (-8) = -16	1000+ 1000 0000	Depăşire!			

1.5.3.4. Înmulţiri şi împărţiri

Înmulţirea şi împărţirea pe **n** biţi impun următoarele restricţii de dimensiune a locaţiilor:

Înmulţirea pe ***n*** biţi presupune că ambii factori sunt reprezentaţi pe câte ***n*** biţi, iar produsul lor va fi reprezentat pe **2 * *n*** biţi. În cazul convenţiei cu semn, factorii înmulţirii vor fi reprezentaţi în cod complementar. Produsul va fi reprezentat tot în cod complementar şi va respecta regula semnelor.

Ca o consecinţă imediată a acestei dimensionări, rezultă că operaţia de înmulţire nu provoacă depăşire! Într-adevăr, în convenţia fără semn cea mai mare valoare posibilă pe n biţi este 2^n-1, pătratul acestei valori este 2^{2n}- 2^{n+1}+1, număr care încape pe 2n biţi. În convenţia cu semn, numărul -2^{n-1} are valoarea absolută cea mai mare, iar pătratul acesteia este 2^{2n-2}, număr care se poate reprezenta pe 2n-1 biţi, deci în cod complementar acest număr se poate reprezenta pe 2n biţi.

Împărţirea pe ***n*** biţi (oarecum invers faţă de înmulţire), impune condiţia ca deîmpărţitul să fie reprezentat pe **2 * *n*** biţi, iar împărţitorul pe ***n*** biţi. Operaţia furnizează două rezultate: câtul reprezentat pe ***n*** biţi şi restul reprezentat tot pe ***n*** biţi. În cazul convenţiei cu semn, deîmpărţitul şi împărţitorul se vor reprezenta în cod complementar. Atât câtul, cât şi restul vor fi, de asemenea,

reprezentate în cod complementar. Câtul împărţirii va respecta regula semnelor. Important! Restul împărţirii va fi, în valoare absolută mai mic decât valoarea absolută a împărţitorului şi va avea acelaşi semn ca şi deîmpărţitul! De exemplu, -7 : 3 dă câtul -2 şi restul -1, adică -7 = (-2) * 3 + (-1). Rugăm cititorul să compare acest rezultat cu cel impus de teorema împărţirii cu rest din aritmetică, care impune ca restul să fie un număr pozitiv, deci în aritmetică avem (-7) = (-3) * 3 + 2, deci câtul -3 şi restul 2!

Operaţia de împărţire semnalează ***eroare*** la împărţitor zero (Divide by zero!). În cazul neîncadrării în dimensiuni semnalează ***depăşire***! Spre exemplu, împărţirea fără semn 1000 : 3 se poate, formal, efectua, deoarece deîmpărţitul se poate reprezenta pe 16 biţi şi împărţitorul se poate reprezenta pe 8 biţi. La această operaţie va apare depăşire, deoarece câtul este 333 şi nu se poate reprezenta pe 8 biţi. Restul împărţirii este 1 şi se poate reprezenta pe un octet. Pentru a evita o astfel de situaţie, programatorul poate decide să facă operaţia pe 16 biţi, adică să reprezinte deîmpărţitul pe 32 de biţi şi împărţitorul pe 16 biţi, urmând să obţină câtul şi restul pe câte 16 biţi (333 încape pe 16 biţi şi astfel nu vom mai avea depăşire).

Să exemplificăm mai întâi operaţiile de înmulţire şi împărţire fără semn.

Înmulţirea fără semn a numerelor întregi reprezentate binar se desfăşoară conform algoritmului cunoscut, doar că se foloseşte baza 2. Pentru comoditate, vom considera **n** = 4. Să considerăm factorii 11 şi 13. Înmulţirea lor în baza 2 este:

```
   1011 ×
   1101
   1011
  0000
 1011
1011
10001111
```

Într-adevăr, 11 × 13 = 143 = $(10001111)_2$

Plecând de la acest exemplu, se poate observa că:

1) Înmulţirea generează o serie de produse parţiale care sunt adunate. Există câte un produs parţial pentru fiecare cifră a înmulţitorului.
2) Produsele parţiale sunt fie deînmulţitul, fie 0.
3) Produsele parţiale nenule pot fi adunate unul câte unul. Ele sunt obţinute deplasând deînmulţitul spre stânga cu câte o poziţie.
4) Înmulţirea a două numere de câte **n** biţi generează un produs care ocupă **2*n** biţi (motiv pentru care s-a impus restricţia de dimensiune amintită mai sus).

Împărţirea fără semn se efectuează, de asemenea, după regulile cunoscute ale aritmeticii. Să împărţim pe 147 la 11. Împărţirea în baza 2 este:

```
10010011 | 1011
1011     |------
=1110    |1101
 1011
 =0111
  0000
  =1111
   1011
   =100
```

Într-adevăr, 147 împărţit la 11 dă câtul 13 şi restul 4.
Ca şi la înmulţire, putem face o serie de observaţii. Acestea evidenţiază faptul că, în ultimă instanţă, împărţirea în baza 2 se reduce la o succesiune de scăderi succesive combinate cu operaţii de deplasare.

Să reamintim câteva observaţii privind înmulţirile şi împărţirile cu semn.

1) Toţi operanzii implicaţi în operaţii sunt reprezentaţi în cod complementar, în conformitate cu cerinţele de dimensionare expuse mai sus.
2) Atât la înmulţirea cu semn cât şi la împărţirea cu semn, se respectă regula semnelor.
3) Pentru operaţia de împărţire, restul este în modul mai mic decât modulul împărţitorului, iar semnul restului este acelaşi cu semnul deîmpărţitului.

Algoritmii de înmulţire şi împărţire în cod complementar nu pot fi preluaţi natural de la cei fără semn, aşa cum stau lucrurile la adunare şi scădere. Dacă ar fi aşa, atunci în exemplul de mai sus, interpretând **1011** ca **-5** şi **1101** ca **-3** ar rezulta produsul **-113 în loc de -15**! Normal, deoarece configuraţia de biţi **10001111** care este rezultatul înmulţirii binare, este reprezentarea în cod complementar a numărului **-113**! La fel, dacă interpretăm cu semn exemplul dat la împărţirea fără semn, avem "egalitatea" în cod complementar:

10010011 = 1011 * 1101 + 0100 adică -109 = (-5) * (-3) + 4 !

Nu vom detalia algoritmii specifici înmulţirii şi împărţirii cu semn. Cititorii pot obţine detalii din [Boian96].

1.5.4. Conversia la o locaţie de alte dimensiuni

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

În fapt este vorba de patru operaţii:

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

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

8 biţi:	**16 biţi: extensie cu semn**	**32 biţi: extensie cu semn**	**16 biţi: extensie cu zero**	**32 biţi: extensie cu zero**
80 10000000	FF80 1111111110000000	FFFFFF80 1111111111111111 1111111110000000	0080 0000000010000000	00000080 0000000000000000 0000000010000000
28 00101000	0028 0000000000101000	00000028 0000000000000000 0000000000101000	0028 0000000000101000	00000028 0000000000000000 0000000000101000
9A 10011010	FF9A 1111111110011010	FFFFFF9A 1111111111111111 1111111110011010	009A 0000000010011010	0000009A 0000000000000000 0000000010011010
7F 01111111	007F 0000000001111111	0000007F 0000000000000000 0000000001111111	007F 0000000001111111	0000007F 0000000000000000 0000000001111111
--	1020 0001000000100000	00001020 0000000000000000 0001000000100000	----	00001020 0000000000000000 0001000000100000
--	8088 1000000010001000	FFFF8088 1111111111111111 1000000010001000	----	00008088 0000000000000000 1000000010001000

Operaţiile de contracţie nu se pot executa întotdeauna. Spre exemplu, într-o locaţie pe 16 biţi există numărul **-448** în baza 10, care în cod complementar se reprezintă **FE40**. Dorim să efectuăm o contracţie la 8 biţi. Eliminând pur şi simplu primul octet se obţine **40**, adică numărul **64** în baza 10! Avem, evident, o situaţie de depăşire. Cu alte cuvinte, contracţiile (conversii prin îngustare) se pot executa numai dacă NU se provoacă pierderea de informaţie.

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

16 biţi:	8 biţi: contracţie cu semn	8 biţi: contracţie cu zero
FF80 1111111110000000	80 10000000	Depăşire! Se pierd 8 biţi 1
0028 0000000000101000	28 00101000	28 000101000
FF9A 1111111110011010	9A 10011010	Depăşire! Se pierd 8 biţi 1
FE40 1111111001000000	Depăşire! Se schimbă bitul de semn	Depăşire! Se pierd 8 biţi 1
0100 0000000100000000	Depăşire! Se schimbă bitul de semn	Depăşire! Se pierd 8 biţi 1
0088 0000000010001000	Depăşire! Se schimbă bitul de semn	88 10001000

CAPITOLUL 2

ARHITECTURA SISTEMELOR DE CALCUL

2.1. DEFINIŢII. ORGANIZAREA UNUI SISTEM DE CALCUL.

Numim sistem de calcul (SC) un dispozitiv care lucrează automat, sub controlul unui program memorat, prelucrând date în vederea producerii unor rezultate ca efect al procesării.

O definiţie similară a unui sistem de calcul (*The American Heritage Dictionary of the English Language*, 2000) este: un dispozitiv care efectuează calcule, în special o maşină electronică programabilă care execută operaţii aritmetice, logice sau care asamblează, stochează, corelează sau efectuează un alt tip de procesare a informaţiei, cu viteză ridicată.

Funcţiile de bază ale unui SC sunt:
- procesarea de date;
- memorarea de date;
- transferul de informaţii;
- controlul tuturor componentelor SC.

Arhitectura unui sistem de calcul poate fi analizată la nivel structural (care sunt componentele fizice şi căile de comunicare între acestea) sau la nivel logic (funcţiile fiecărei componente în cadrul structurii).

Structura unui sistem de calcul este:
- hardware - partea de echipamente:
  - unitatea centrală de procesare (*Central Processing Unit* – CPU);
  - memoria;
  - dispozitivele periferice;
- software - partea de programe:
  - soft sistem (aplicaţii destinate sistemului de calcul şi sistemului de operare);
  - soft utilizator (restul aplicaţiilor);
- firmware - partea de microprograme.

Hard-ul şi soft-ul se prezintă sub forma unor nivele (componente) ierarhice, fiecare componentă inferioară ascunzând detaliile de implementare faţă de componenta superioară imediată. O astfel de abordare este reflectarea **principiului abstractizării**, el constituind maniera de atac a proiectanţilor hard şi soft asupra complexităţii sistemelor de calcul.

Printre nivelele ierarhice ale hardware-ului se pot identifica structuri din siliciu sau alte materiale, componente electronice (tranzistori ş.a.), componente logice, circuite logice, unităţi

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

Arhitectura (organizarea) unui sistem de calcul se referă la acele atribute ale sistemului care sunt vizibile programatorului şi care au un impact direct asupra execuţiei unui program: setul de instrucţiuni maşină, caracteristicile de reprezentare a datelor, modurile de adresare şi sistemul de intrare / ieşire (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*) / ieşire (*output*) = sistemul de I/O;
- structuri de interconectare a componentelor de mai sus (magistrale);

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

Această organizare este independentă de tehnologia hard adoptată pentru construcţia 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 *mulţimea instrucţiunilor maşină* şi *organizarea maşinii*.

Mulţimea instrucţiunilor maşină (*Instruction Set Arhitecture* – ISA) este o interfaţă cheie între nivelele de abstractizare, fiind interfaţa dintre hard şi soft-ul de nivel scăzut (*low-level software*). O astfel de interfaţă 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 acelaşi hardware, dar răspund aceleiaşi ISA).

ISA defineşte:

- organizarea SC, modul de stocare a informaţiei (regiştri, memorie);
- tipurile şi structurile de date (codificări, reprezentări);
- formatul instrucţiunilor;
- setul de instrucţiuni (codurile operaţiilor) pe care microprocesorul le poate efectua;
- modurile de adresare şi accesare a datelor şi instrucţiunilor;
- condiţiile de excepţie;

Organizarea unei maşini se referă la:

- implementarea, capacitatea şi performanţa unităţilor funcţionale;
- interconexiunile dintre aceste unităţi;
- fluxul de informaţie dintre unităţi;
- controlul fluxului de informaţie.

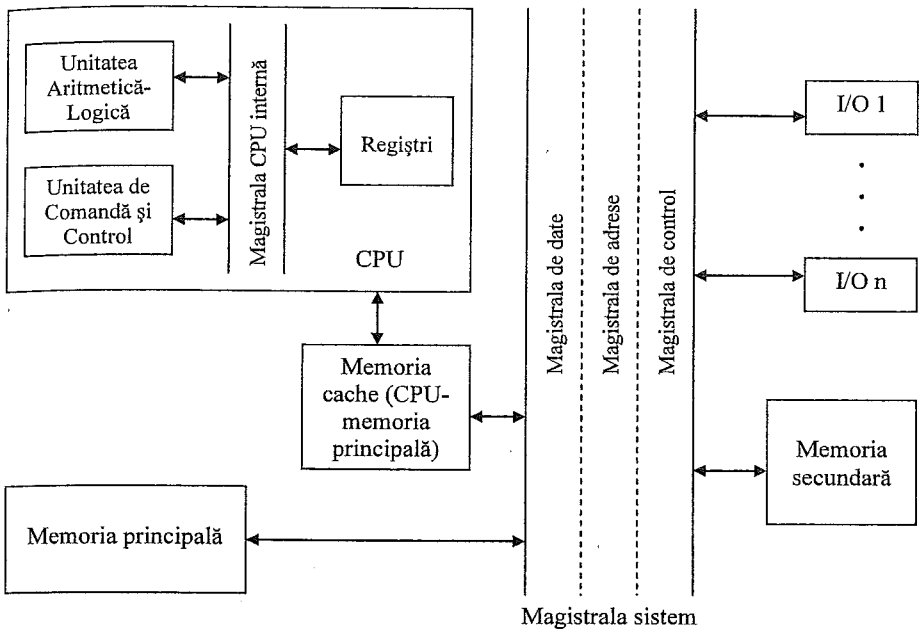


Fig. 2.1. Arhitectura unui sistem de calcul

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

Caracteristicile arhitecturii von Neumann sunt:

- atât datele, cât şi instrucţiunile sunt reprezentate ca şiruri de biţi şi sunt stocate într-o memorie read-write; e important de subliniat faptul că nu se poate face diferenţa între date şi instrucţiuni prin simpla citire a unei locaţii de memorie - trebuie cunoscut ce reprezintă pentru a i se stabili semnificaţia (de exemplu: valoarea 98h - poate reprezenta o valoare a unei variabile de dimensiune un cuvânt sau codul instrucţiunii maşină corespunzător instrucţiunii asamblare 8086 CBW);
- conţinutul memoriei se poate accesa în funcţie de locaţie (adresă), indiferent de tipul informaţiei conţinute;
- execuţia unui set de instrucţiuni se efectuează secvenţial, prin citirea de instrucţiuni consecutive din memorie.

2.2. UNITATEA CENTRALĂ (CENTRAL PROCESSING UNIT – CPU)

Procesorul controlează modul de operare a calculatorului şi execută funcţiile de procesare a datelor.

Funcţiile unui CPU sunt:
- obţinerea instrucţiunilor care trebuie executate;
- obţinerea datelor necesare instrucţiunilor;
- procesarea datelor (execuţia instrucţiunilor);
- furnizarea rezultatelor obţinute.

Putem identifica două componente de bază la nivelul unui CPU:
- Unitatea Aritmetică-Logică (*Arithmetic Logic Unit* – ALU);
- Unitatea de Comandă şi Control (*Control Unit* – CU);

iar pentru îndeplinirea funcţiilor de mai sus, CPU mai are nevoie de:
- regiştri – aceştia sunt dispozitive de stocare temporară a datelor şi informaţiilor de control (instrucţiunile), de capacitate mică şi viteză de acces mare;
- magistrale interne CPU – dispozitive pentru comunicare între componentele CPU şi comunicare cu exteriorul, pentru traficul de informaţii.

Unitatea Aritmetică-Logică execută operaţii aritmetice şi logice asupra datelor.

Unitatea de Comandă şi Control este componenta CPU care dirijează toate componentele sistemului de calcul prin controlul direct asupra acestora. Pentru aceasta, trimite semnale de control în interiorul CPU şi către magistrala sistem, sau captează semnale dinspre magistrala sistem. Nu execută instrucţiuni, ci le decodifică şi comandă altor componente execuţia efectivă a acestora. Regulile după care funcţionează CU sunt codificate în *Programmable Logic Array* (PLA - dispozitiv programabil utilizat pentru implementarea de circuite logice combinaţionale AND / OR) sau într-o memorie de tip ROM (*Read-Only Memory*).

Pentru execuţia unei instrucţiuni, aceasta şi datele necesare trebuie aduse din memoria secundară sau de la un dispozitiv de intrare în memoria principală. CU coordonează *ciclul de execuţie a instrucţiunii*:
- citirea instrucţiunii din memoria principală (FETCH);
- decodificarea acesteia (DECODE – tipul instrucţiunii, numărul de argumente, etc);
- obţinerea operanzilor necesari instrucţiunii (READ MEMORY);
- execuţia ei (EXECUTE; dacă e cazul, ALU primeşte controlul pentru efectuarea operaţiei aritmetice sau logice implicate);
- furnizarea rezultatelor în regiştri sau în memorie (STORE).

Apoi, CU poate iniţia transferul acestor date rezultate din memoria internă către un dispozitiv de ieşire sau către un dispozitiv de stocare secundar.

Cunoscând structura unui CPU şi paşii din ciclul de execuţie a unei instrucţiuni, putem contura structura funcţională a procesorului: *modulul de control* şi *calea de date* (*datapath*). Modulul de

control este răspunzător de comunicarea cu exteriorul CPU (preluarea şi interpretarea instrucţiunilor şi de transmiterea rezultatelor), precum şi de controlul execuţiei instrucţiunilor (emiterea semnalelor de control către calea de date şi recepţionarea semnalelor de stare dinspre aceasta). Din calea de date fac parte componente de stocare (regiştri de date), unităţi funcţionale (ALU) şi căi de comunicare. Practic, calea de date defineşte mulţimea operaţiilor efectuate asupra datelor în vederea execuţiei unei instrucţiuni şi drumul parcurs de datele prelucrate în decursul execuţiei operaţiilor.

2.2.1. Ceasul sistem (The System Clock)

Fiecare CPU are un ceas intern care produce şi trimite semnale electrice pe magistrala de control pentru a sincroniza operaţiile sistemului. Semnalele alternează valori 0 şi 1 cu o anumită frecvenţă numită frecvenţa ceasului sistem. Timpul necesar trecerii din starea 0 în 1 şi apoi iar în 0 se numeşte *perioada ceasului* sau *ciclu de ceas* (*clock cycle*).

Frecvenţa de ceas este de fapt numărul de cicluri de ceas pe secundă şi este măsurată în hertzi.

Observaţie: 1 Hz = 1 ciclu de ceas / secundă
frecvenţa = 1 / perioada

Toate operaţiile efectuate de către microprocesor sunt sincronizate cu ceasul sistem; aceasta înseamnă că procesorul nu poate efectua de o manieră secvenţială operaţii mai rapid decât frecvenţa de tact a ceasului (a unităţii de timp a ceasului). Altfel spus, această unitate de ceas este răspunzătoare de viteza la care rulează procesorul respectiv. Astfel, un procesor cu o viteză de ceas de 200 MHz are un ceas intern care este capabil să *ticăie* de exact 200.000.000 ori pe secundă. Un ciclu de ceas pentru un astfel de procesor are o durată de 1 / 200.000.000 secunde.

Un ciclu de ceas este în general, pentru un procesor, unitatea de bază pentru măsurarea timpului. Este cea mai mică cuantă de timp sesizabilă de către procesor. Toate activităţile şi instrucţiunile executate de procesor sunt executate în general în multipli ai ciclului de ceas (din punct de vedere al duratei de execuţie). Fiecare instrucţiune maşină necesită un număr fix, cunoscut (în general) de cicluri procesor. Există diferenţe în durata execuţiei instrucţiunilor maşină în funcţie de natura parametrilor şi mediul lor de stocare. Astfel o instrucţiune **MOV destinaţie, sursă** se execută între 2 şi 14 cicluri de ceas procesor în funcţie de natura argumentelor destinaţie şi sursă care pot fi: regiştri generali, regiştri de segment, constante, adrese de memorie. Fiecare combinaţie posibilă a acestor tipuri de parametri va rezulta într-o durată de execuţie specifică în cicluri de ceas procesor. Pentru fiecare combinaţie posibilă, durata de execuţie în cicluri procesor este cunoscută. Pentru a afla numărul de cicluri procesor pentru fiecare instrucţiune maşină, împreună cu derivaţiile rezultate din tipul parametrilor se poate consulta documentaţia oferită de *Norton Guide*.

Timpul de execuţie al unei instrucţiuni măsurat în cicluri de ceas procesor poartă denumirea *Cycles per Instruction (CPI)*. Încă de la primele procesoare apărute pe piaţă, modul de execuţie

al instrucţiunilor maşină a fost unul secvenţial. Procesorul preia o instrucţiune, *petrece un număr de cicluri de ceas la execuţia completă a acesteia*, după care trece la următoarea instrucţiune ş.a.m.d. În general CPI se referă la numărul mediu de cicluri procesor per instrucţiune executată de procesor – în raport cu toate instrucţiunile ce compun un program sau în raport cu întreg setul de instrucţiuni al procesorului. De remarcat aici, însă, că deşi o instrucţiune **MOV** se execută în acelaşi număr *N* de cicluri procesor, durata de execuţie în timp (secunde) este diferită de la un procesor la altul. Astfel un procesor ce rulează la viteza de 200 MHz va executa o instrucţiune **MOV** în 20 nanosec, în timp ce un procesor ce rulează la 1GHz va executa exact aceeaşi instrucţiune **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 frecvenţei de ceas a procesoarelor, se urmăreşte creşterea vitezei acestora prin alte metode. Cea mai des întâlnită este paralelizarea execuţiei instrucţiunilor maşină. Această paralelizare la nivel de instrucţiune maşină se referă la paralelizarea etapelor execuţiei unei instrucţiuni (vezi paragraful anterior). În loc să execute toate cele 5 faze ale unei instrucţiuni şi să treacă abia apoi la următoarea, ne putem închipui o arhitectură în care, după execuţia fazei FETCH pentru o instrucţiune I1, aceasta trece în faza DECODE, iar procesorul preia următoarea instrucţiune a programului în faza FETCH. Putem avea astfel în arhitectura prezentată mai sus maximum cinci instrucţiuni 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) şi I5 în faza FETCH (F) (vezi figura 2.2.). Această tehnică de paralelizare se numeşte *pipelining* (de la cuvântul *pipeline* – conductă).

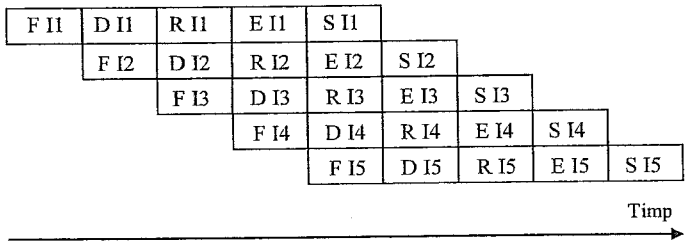


Fig. 2.2. Mecanismul de pararelizare în execuţia instrucţiunilor (pipeline)

De remarcat faptul că prin această tehnică nu se creşte viteza de execuţie a unei instrucţiuni (ea se execută în acelaşi număr de cicluri de ceas), ci se măreşte numărul de instrucţiuni executate de către procesor per unitate de timp.

Folosind tehnica de pipeline ajungem la un caz cu totul surprinzător: putem să ajungem, pentru unele secvenţe de instrucţiuni să măsurăm numărul de instrucţiuni / 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 acţiune / instrucţiune în fiecare ciclu de ceas. Tehnica de pipeline permite însă execuţia *virtual paralelă* a mai multor instrucţiuni în acelaşi timp, fapt care

poate conduce la situaţii cu valori CPI medii subunitare. Cu cât pipeline-ul este mai lung şi numărul de cicluri de ceas pentru instrucţiuni mai mic, cu atât creşte factorul IPC pentru procesorul respectiv.

O problemă spinoasă de-a lungul timpului a fost măsurarea vitezei calculatoarelor. Pentru aceasta s-a pornit iniţial cu frecvenţa de ceas a CPU. Aceasta s-a dovedit rapid a fi o măsură neadecvată, întrucât două procesoare (de exemplu – unul de 1GHz şi unul de 500 MHz) pot rezolva o problemă în acelaşi timp, timp care depinde şi 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 noţiunea de *MIPS (Millions of Instructions per Second)* – care nu mai depinde de ciclul de ceas. Aceasta măsoară numărul (exprimat în milioane) de instrucţiuni (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 instrucţiuni 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 instrucţiuni pe care le poate executa (vezi ISA). Ca şi criteriu de performanţă se consideră deseori viteza de lucru a CPU, care depinde atât de frecvenţa ceasului intern, cât şi de capacitatea de paralelizare (organizarea execuţiei instrucţiunilor), dimensiunea regiştrilor interni şi a magistralei de date, tipul microprocesorului sau dimensiunea memoriei cache a CPU (ca factor de influenţă a vitezei de comunicare cu exteriorul).

Viteza de lucru a CPU, ca şi în cazul altor componente ale SC, a cunoscut în timp o continuă evoluţie. 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 biţi?"

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 biţi = 64 linii de date, astfel că la fiecare "memory cycle" procesorul poate accesa 8 octeţi din memorie);
- perspectiva soft (punctul de vedere software): dimensiunea unui cuvânt de memorie (dimensiunea regiştrilor CPU);

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

Microprocesor	Data Bus	Regiştri
8088	8 biţi	16 biţi
80386sx	16 biţi	32 biţi
Pentium	64 biţi	32 biţi

2.3. MEMORIA

Memoria este un dispozitiv de stocare a datelor pentru un anumit interval de timp. Se disting diverse criterii de clasificare a tipurilor de memorie:

a. din punct de vedere al accesării datelor:

a.1. *memorie cu acces aleator* (*Random Access Memory*): locaţiile pot fi accesate (în citire sau scriere) în orice ordine (aleator) indiferent de ultima locaţie accesată. Exemple: oricare chip de memorie (memorie "*on-chip*"), precum memoria principală (DRAM, SRAM), memoria flash ş.a.;

a.2. *memorie cu acces asociativ*. Exemplu: memoria cache.

Observaţie: La încărcarea sau căutarea în memoria cache a unei informaţii, pe baza adresei acesteia se determină care este poziţia din cache în care trebuie să fie încărcată sau la care ar trebui să se găsească dacă ar exista deja acolo. Strategia de determinare a locaţiei din memoria cache depinde de modul de organizare a acesteia. De exemplu: *maparea directă* – unei informaţii de la o anumită adresă îi corespunde o locaţie bine determinată din memoria cache, locaţie calculată cu ajutorul unei funcţii de transformare (poate fi funcţie de dispersie de tip modulo); *maparea asociativă pe mulţime* – o anumită informaţie poate fi încărcată în memoria cache în oricare din poziţiile dintr-o mulţime bine determinată.

a.3. *memorie cu acces secvenţial*: pentru a accesa a n-a înregistrare, trebuie parcurse primele n-1 înregistrări => timpul de accesare a datelor este variabil, depinzând de locaţia accesată. Exemplu: benzi magnetice;

a.4. *memorie cu acces direct*: spre deosebire de accesul secvenţial, poziţionarea pe o anumită înregistrare se face în mod direct pe baza unui calcul de adresă. Exemplu: dispozitivele de tip disc, precum hard disk, floppy disk, CD-ROM (vezi secţiunea 2.3.2.).

Observaţie: Modul de accesare a datelor din memoria cu acces aleator este similar cu cel folosit pentru memoria cu acces direct (direct, pe baza unei adrese cunoscute). Diferenţa constă în timpul de acces: în cazul memoriei cu acces aleator timpul de acces este acelaşi, indiferent de adresa datelor accesate; în schimb, memoria cu acces direct (vezi dispozitivele disc – secţiunea 2.3.2.) foloseşte un mecanism fizic de poziţionare la o anumită adresă, fapt care introduce un

timp de întârziere în accesarea datelor => timp de accesare variabil, în funcţie de poziţia curentă a mecanismului de accesare şi de poziţia datelor pe disc.

b. din punct de vedere al volatilităţii:

b.1. *memorie volatilă* (de scurtă durată): conţinutul său se pierde la îndepărtarea sursei de curent. Cel mai elocvent exemplu îl constituie în acest sens memoria principală a SC (care conţine datele şi instrucţiunile utilizate curent de CPU);

b.2. *memorie non-volatilă sau remanentă* (de lungă durată): conţinutul se păstrează şi după deconectarea de la sursă. Exemple: memoria ROM, hard disk, CDROM, memoria Flash.

c. din punct de vedere al accesului CPU:

c.1. *memorie internă*: accesată direct de către CPU;

c.2. *memorie secundară* sau *dispozitiv de stocare periferic*: memorie externă, cu acces indirect al CPU. Exemple: HD, floppy disk, CDROM.

d. din punct de vedere al tipurilor de acces permise:

d.1. *memorie read/write*: permite acces la date în citire sau scriere. Exemple: memoria principală, hard disk, floppy disk;
d.2. *memorie read-only*: permite doar citirea datelor. Exemple: ROM, CDROM.

2.3.1. Memoria internă

Memoria internă reprezintă toate spaţiile de stocare de date accesibile CPU fără utilizarea canalelor de comunicaţie de intrare / ieşire. Din această categorie fac parte memoria principală, memoria cache (dintre CPU şi memoria principală), ROM şi regiştrii, toate aceste dispozitive putând fi direct accesate de către CPU.

Memoria principală (*main memory*, *primary memory*), cunoscută în general sub numele de memorie RAM, conţine date care sunt utilizate curent de către procesor – instrucţiuni ale programelor care sunt executate şi date cu care acestea operează. Aceste informaţii sunt aduse în memoria principală de pe un suport de stocare extern sau de la un dispozitiv de I/O. Tipurile de memorie care sunt folosite ca memorie principală sunt cele din clasa memoriilor RAM, precum DRAM sau SRAM. În general, capacitatea memoriei principale a unui sistem de calcul este cuprinsă între 1 MB şi 4 GB.

Memoria DRAM (Dynamic RAM) face parte din clasa memoriilor volatile. Datorită modului în care este construită, este necesară reactualizarea conţinutului la un anumit interval de timp, de

exemplu de ordin μs (de aici denumirea *dinamică*). Datele nu sunt disponibile în timpul operaţiilor de reactualizare. Deşi timpul consumat de aceste operaţii constituie aproximativ 1% din timpul de funcţionare, acestea contribuie la viteza de acces mai redusă faţă de alte tipuri de memorie (vezi SRAM). Conţinutul unei asemenea memorii este organizat ca tablou bidimensional de biţi. La citirea unui element al tabloului, se citeşte întreg rândul, care apoi este rescris (*refresh*). Pentru operaţia de scriere a unui element, se citeşte întreg rândul, se modifică elementul, apoi se rescrie întreg rândul înapoi. Elementele unei memorii DRAM sunt mai mici şi 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 îmbunătăţită a DRAM, *Double Data Rate SDRAM* (DDR SDRAM) – o îmbunătăţire ulterioară a SDRAM, *Direct Rambus DRAM* (DRDRAM sau RDRAM), *Synchronous Graphics RAM* (SGRAM) – o formă a SDRAM specializată pentru adaptoare grafice.

Observaţie: 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-volatilă, dar de viteză mai mică la citire / scriere.

Memoria SRAM (Static RAM) este un tip de memorie semiconductor, volatilă. După cum indică denumirea, conţinutul unei memorii SRAM se păstrează atâta timp cât sistemul este conectat la o sursă, spre deosebire de DRAM care necesită reactualizări periodice ale conţinutului. Structura SRAM permite un acces mai rapid la locaţiile acesteia, în comparaţie cu DRAM, motiv pentru care este utilizată ca memorie cache a CPU. Memoriile SRAM de viteză şi capacitate mai mici sunt folosite atunci când se cere un consum de energie şi cost scăzut, de exemplu pentru backup RAM cu sursă de tip baterie. Deoarece este mai puţin densă faţă de DRAM (conţine mai puţini biţi pe unitate de suprafaţă), în general capacitatea unei memorii SRAM este mai mică faţă de a unei memorii DRAM.

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

Deoarece nu poate fi (uşor) scrisă, memoria ROM este utilizată în general ca spaţiu de stocare al *firmware*-ului, care nu necesită actualizări frecvente. Memoria ROM a multor sisteme de calcul din generaţiile trecute (anii '80) conţinea încă de la furnizare sistemul de operare, iar o parte din acestea includeau şi un interpretor al limbajului de programare BASIC. Era cea mai practică alternativă, dischetele nefiind utilizate încă pe scară largă. În prezent, tendinţa este de a stoca cât mai puţine informaţii în memoriile ROM şi o cantitate tot mai mare de date pe dispozitivele de memorare externe. Deşi memoria ROM este de capacitate mică, avantajul principal este viteza mare de accesare a datelor. În general este întâlnită ca şi componentă CPU, caz în care conţine 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 iniţializarea sistemului, verificarea echipamentelor periferice din sistem şi accesul primar la acestea. BIOS-ul poate fi considerat şi *firmware*-ul plăcii de bază (vezi 2.4.4.), rutinele sale fiind printre primele care se

execută la pornirea unui calculator. De asemenea, poate să conţină rutinele cu funcţiile de bază pentru dispozitive precum telefoane mobile, controlere de reţea, controlere video.

Memoria ROM este în general cunoscută ca memorie scrisă în faza de producţie, al cărui conţinut nu poate fi modificat ulterior. Mai există, însă, câteva alte tipuri de memorie ROM al căror conţinut 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 ştergerea conţinutului (prin expunere la ultraviolete) şi rescrierea acestuia cu ajutorul unui 'programator EPROM'; numărul de rescrieri este însă limitat din cauza degradării progresive din faza de ştergere;
- 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 conţinutului, bit cu bit;
- EEPROM (*Electrically Erasable Read-Only Memory*) – permite ştergerea electrică a întregului conţinut sau doar a unui bloc din conţinutul memoriei şi rescrierea acestuia; exemplu – memoria flash a camerelor digitale, MP3 player-elor ş.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 şi are o viteză mai mică de accesare a datelor faţă de memoria internă şi face parte din categoria memoriilor 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. Informaţia poate fi înregistrată magnetic pe una sau ambele *feţe* ale unui disc. Pentru accesarea informaţiei, fiecare suprafaţă de memorare are asociat un *cap de citire / scriere*. Braţele capetelor de accesare corespunzătoare discurilor sunt situate pe un suport unic al dispozitivului. Mişcarea acestora permite deplasarea capetelor de acces radial pe suprafaţa discului, permiţând astfel accesarea informaţiei indiferent de localizarea acesteia faţă de axul central.

Din punct de vedere logic, o suprafaţă de memorare a discului este divizată în benzi concentrice numite *piste*. Pentru un dispozitiv ce deţine mai multe suprafeţe de memorare, numărul de piste de pe aceste suprafeţe este acelaşi, iar pistele de aceeaşi rază formează un *cilindru*. O pistă este divizată în porţiuni numite *sectoare*. Numărul de sectoare este acelaşi pentru fiecare pistă a discului şi fiecare sector are aceeaşi dimensiune. Un sector reprezintă în general unitatea de

transfer de date între disc şi memoria internă. Structura unui volum disc este reprezentată în figura 2.3.

Din cele menţionate mai sus rezultă următorii parametri (constante) ai unui dispozitiv disc: numărul de discuri, numărul capetelor de citire / scriere pentru un disc (numărul de feţe active ale unui disc), numărul de piste de pe o faţă, numărul de sectoare de pe o pistă şi numărul de octeţi dintr-un sector.

Cunoscând valorile parametrilor unui dispozitiv disc, o metodă de adresare a informaţiilor (a unui sector) de pe disc în general utilizată este *CHS* (*cylinder-head-sector*). După cum sugerează numele, identificarea sectorului accesat se realizează prin specificarea numărului de ordine al capului de citire / scriere, numărul cilindrului şi numărul sectorului din cadrul pistei (unde pista e determinată de cap şi cilidru). De exemplu, o dischetă de 3.5 inch este configurată în general la următorii parametri: 2 capete de citire / scriere (numerotate 0 şi 1), 80 cilindri / disc (numerotaţi 0 – 79), 18 sectoare / pistă (numerotate 0 – 17), 512 octeţi / sector.

Fig. 2.3. Structura unui volum disc

Un criteriu de evaluare a performanţei unui dispozitiv disc este timpul de accesare a informaţiei, ce poate fi calculat după formula:

TimpAcces = TimpCăutare + TimpRotire + TimpTransfer

unde:

- TimpCăutare depinde de numărul de piste peste care trebuie să de deplaseze capul de accesare şi viteza de căutare a discului;
- TimpRotire este în funcţie de viteza de rotaţie a discului şi de distanţa dintre sectorul care trebuie accesat şi capul de accesare;
- TimpTransfer depinde de rata de transfer a datelor (bandwidth) caracteristică dispozitivului.

La TimpAcces mai putem adăuga TimpControler ce reprezintă întârzierea produsă de controlerul dispozitivului, ca interfaţă de comunicare între CPU şi disc.

Performanţele dispozitivelor disc au rate de evoluţie particulare, raportat la diferitele criterii de evaluare. De exemplu, capacitatea de stocare poate să crească cu aproximativ 100% în 1-1.5 ani, rata de transfer cu 40% / an, timpul de acces cu doar 8% / an, în timp ce raportul cost / capacitate scade de aproape 2 ori / an.

Cele mai utilizate volume disc sunt:

- hard disk-urile - dispozitive de stocare a datelor pe suport magnetic, alcătuite din mai multe discuri; capacitatea de memorare a acestora este mare, ajungând în prezent până la sute de gigabytes (1 GB = 2^{30} bytes); evaluări ale unor performanţe ale acestor dispozitive în prezent: 7200-10000-15000 RPM (rotaţii pe minut), timp de acces mediu = 8-15 ms, 50-150 operaţii I/O / secundă;
- dischetele (floppy disk) - permit acces direct la date, au preţ de producţie şi achiziţionare foarte scăzut şi sunt portabile; cele mai utilizate sunt cele de 1.44 megabytes (1 MB = 2^{20} bytes);
- benzile magnetice - permit acces secvenţial la date, asigură o capacitate mare de stocare, sunt ieftine şi sunt folosite în general pentru arhivare şi memorarea cópiilor de siguranţă (suport de backup);
- discurile optice:
  - CD (*Compact Disc*) - sunt dispozitive de capacitate de memorare relativ mare (650-700 MB), de tip Read-Only sau Read / Write, pentru care producţia şi duplicarea nu sunt costisitoare; de exemplu: WORM CD (*Write Once, Read Many*) - permit înregistrarea permanentă a unui volum mare de date (CD-ROM), CD-RW (CD Read / Write) - permit rescrierea datelor de pe suport;
  - DVD (*Digital Versatile Disc*) - sunt dispozitive disc similare CD-urilor, însă prezintă un mod de codificare a datelor diferit şi o densitate a acestora mai mare; capacitatea de stocare a acestora este în prezent de 4.7 GB până la 17.1 GB; pot fi de tip Read Only (DVD-ROM) sau Read / Write (DVD-RW);

- 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

Motivaţie

Pe măsură ce sistemele de calcul se dezvoltă, diferenţa de performanţă dintre diferitele componente poate să crească tot mai mult. Cel mai grăitor exemplu este diferenţa dintre performanţa CPU şi cea a memoriei interne de tip DRAM.

Din cauza diferenţei timpului de acces al CPU şi al memoriei principale, CPU este nevoit să aştepte destul de mult pentru a primi datele din memorie. O asemenea diferenţă este defavorabilă şi în cazul interacţiunii dintre memoria principală şi memoria secundară.

Pentru a gestiona cât mai eficient accesul la date, un sistem de calcul deţine un sistem complex al memoriei, în care combină memorie de capacitate mică, dar rapidă, şi 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:

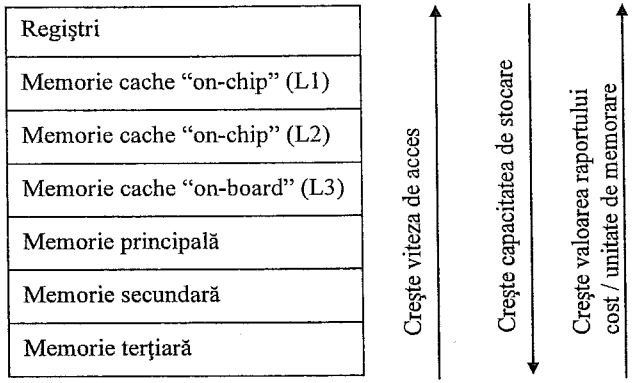


Fig. 2.4. Ierarhia memoriei

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 submulţime de informaţii 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 funcţii de transformare a adreselor de pe nivelul inferior către cel imediat superior. Întotdeauna datele sunt copiate numai între două nivele adiacente.

Eficienţa 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 reţină data respectivă pentru o perioadă de timp (de exemplu: instrucţiunile dintr-o structură repetitivă sau ale unei subrutine)
- *localizare spaţială*: dacă se accesează o locaţie, sunt mari şanse să urmeze accesarea unor locaţii din vecinătatea primeia => ar trebui ca la accesarea datei curente să se aducă un întreg bloc de informaţie care să conţină atât informaţia necesară în momentul curent, cât şi informaţia conţinută la adrese învecinate (de exemplu: variabile locale unei subrutine sau elementele unui şir)

În urma acestor observaţii statistice s-au dezvoltat aşa-numitele memorii de tip cache.

Memoria cache

O *memorie de tip cache* este o colecţie 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 faţă 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 operaţiei de copiere, scăzând astfel semnificativ timpul mediu de accesare.

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

Ca exemplu de funcţionare a unei memorii cache, considerăm interfaţa dintre CPU şi 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 şi L2). Primul nivel de cache este integrat pe chip-ul CPU (cache "*on-chip*") şi asigură o viteză de accces similară CPU. Capacitatea acestei memorii poate varia de la 16, 32 până la 64, chiar 128 KB. Al doilea nivel asigură interfaţa dintre primul nivel şi memoria internă şi în general este memorie de tip SRAM. Este plasată de obicei tot pe chip-ul CPU, însă viteza 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

există şi un al treilea nivel de memorie cache (L3), aceasta este amplasată pe placa de bază (cache "*on-board*"), capacitatea acesteia putând ajunge la 2-4 MB. Secvenţa de căutare este:

- CPU solicită o instrucţiune sau un operand; fie I informaţia solicitată.
- Se caută în primul nivel cache, cel mai apropiat de CPU.
- Dacă I este găsit => *cache hit* (succes); se opreşte căutarea pe acest nivel.
- Dacă I nu este găsit => *cache miss* (eşec); se continuă căutarea pe nivelul cache secundar.
- În cazul în care pe toate nivelele cache s-a raportat eşec, se caută I în memoria principală.

Ca şi criterii de măsură a performanţei unei memorii cache se folosesc

hit rate = nr.hit / nr.referinţe la memorie (procentul de accesări cu succes din totalul de accesări)

miss rate = nr.miss / nr.referinţe la memorie (procentul de tentative de acces eşuate din totalul de cereri),

unde *miss rate = 1 – hit rate.*

În general se întâlnesc următoarele tipuri de memorie cache:

- cache al memoriei principale, ca interfaţă între aceasta şi CPU: poate fi pe unul, două sau trei nivele; acest cache este gestionat de către hardware
- memoria cache între memoria principală şi memoria secundară (disc) este memoria virtuală; transferul datelor de pe disc în memoria principală (gestiunea memoriei virtuale) este responsabilitatea sistemului de operare
- cache *Translation Lookaside Buffer* (TLB) al tabelei de pagini folosită pentru realizarea corespondenţei de adrese între memoria principală şi memoria virtuală
- memorii cache gestionate de componente soft: cache DNS (pentru corespondenţe dintre nume de domenii şi adrese IP); cache al unui web browser (pentru ultimele pagini accesate); cache al unui SGBD (de exemplu: Oracle, SQL-Server, pentru ultimele date citite sau ultimele planuri de execuţie dezvoltate).

În general, performanţa unei memorii cache depinde de parametrii de organizare şi funcţionare ai acesteia, precum: algoritmul de plasare a blocurilor aduse în cache (*block placement strategy*), modalitatea de identificare a unui bloc din cache (*block identification policy*), politica de selectare a unui bloc care să fie înlocuit în cazul în care nu mai este spaţiu liber în cache (*block replacement policy*), scrierea datelor modificate (*cache write policy* – imediat sau la dealocarea blocului).

2.4. DISPOZITIVE PERIFERICE

Dispozitivele periferice asigură interfaţa dintre utilizator şi sistemul de calcul sau dintre sistemul de calcul şi alte sisteme fizice. Un asemenea dispozitiv este caracterizat de tipul de comportament (intrare, ieşire, stocare), partener (utilizator uman sau sistem fizic) şi performanţă.

Performanţa unui dispozitiv de intrare / ieşire (*Input / Output* – I/O) depinde în general atât de tipul său, cât şi de alte componente ale sistemului (CPU, memorie cache, memorie principală, magistrale de comunicare, controler I/O, software I/O, sistem de operare) şi este reprezentată de rata de transfer a datelor (*I/O bandwidth* = cantitatea de date transmisă şi recepţionată într-un interval de timp) şi timpul de răspuns al dispozitivului periferic (*latency*).

Tipurile de dispozitive periferice des întâlnite sunt:

- dispozitive de intrare: tastatură, mouse, scanner
- dispozitive de ieşire: imprimantă, monitor
- dispozitive de intrare sau ieşire: modem, placă de reţea
- dispozitive de stocare: disc (hard disk, floppy disk), bandă magnetică

2.4.1. Magistrale – structuri de interconectare

O magistrală este un subsistem prin care se transportă informaţie (date, instrucţiuni, semnale de control) sau energie între diferite componente ale unui SC sau între diferite SC. Spre deosebire de conexiunile punct la punct, o singură magistrală 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 magistralele seriale se transportă informaţia ca şir de biţi (bit după bit). Magistralele paralele transportă simultan informaţie prin mai multe fire, mărindu-se astfel rata de transfer. De exemplu, pe o magistrală de 16 biţi se pot transmite simultan doi octeţi. Acest lucru nu înseamnă că pe o magistrală paralelă viteza de transfer a informaţiei va fi neapărat mai mare decât pe una serială. Dimpotrivă, datorită costurilor mari implicate de transmisia paralelă a datelor, în ultimul timp se remarcă renunţarea la magistralele paralele şi concentrarea pe magistrale seriale care să lucreze la frecvenţe de transfer mari (de exemplu: magistrala serială S-ATA are o frecvenţă de transfer mai mare decât magistrala paralelă IDE/ATA; similar pentru interfaţa serială USB în raport cu oricare interfaţă paralelă aflată în uz).

Un sistem de calcul include magistrale interne (locale) care fac legătura între componente interne ale sistemului (de exemplu: între CPU şi memoria internă) şi magistrale externe, pentru realizarea conexiunilor către echipamente periferice sau către alte maşini.

Sub-sistemul de magistrale al unui SC poartă numele de magistrală sistem (*system bus*). Raportat la tipul informaţiei transportate, pot fi identificate trei tipuri de magistrale într-un SC:

- *magistrale de date* (*data bus*) – transportă informaţie de tip dată sau instrucţiune; performanţa depinde în primul rând de dimensiune (de exemplu: magistrale de 8, 16, 32, 64 biţi)
- *magistrale de adrese* (*address bus*) – informaţia comunicată este adresa locaţiei de memorie pe care componenta solicitantă doreşte să o acceseze (în citire sau scriere); dimensiunea magistralei determină capacitatea maximă de memorie adresabilă din sistem (de exemplu: sistemele de calcul cu regiştri pe 8 biţi deţineau magistrale pe 16 biţi, de

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

- *magistrale de control* (*control bus*) – transmit informaţie de control şi 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 funcţie de tipul componentelor interconectate, magistralele interne pot fi:

- magistrale CPU-memorie: au arhitectură specifică producătorului, asigură o comunicare rapidă directă între procesor şi memorie şi sunt de lungime redusă;
- magistrale de I/O: au în general arhitectura standardizată şi asigură o viteză ridicată în comunicarea informaţiei dintre CPU, memorie şi 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*) şi AGP (*Accelerated Graphics Port*). Aceste magistralele 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 ataşabilă sistemului de calcul. De obicei, fiecare controler prezintă două interfeţe:

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

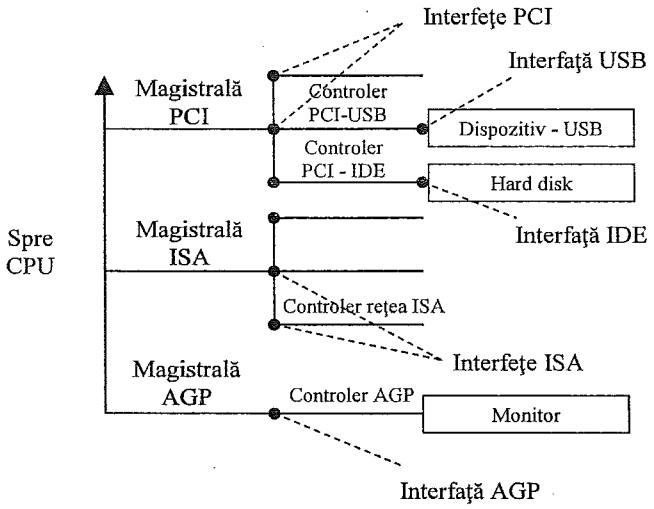


Fig. 2.5. Magistrale de I/O

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

Practic prin interfaţă (fie că este vorba de interfaţa dintre controler şi magistrala I/O internă, fie că este vorba de interfaţa dintre controler şi echipamentul periferic) vom înţelege atât expresia fizică a acesteia (specificaţiile de interconectare fizică, portul, slotul, mufa), cât şi setul de caracteristici funcţionale, protocoale, specificaţii logice necesare comunicării pe magistrala asociată.

2.4.2. Magistrale I/O interne şi interfeţele 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 sfârşitul anilor '90. ISA a fost dezvoltată mai întâi pe 8 biţi, iar ulterior pe 16 biţi, operând la 8 MHz. Aceste valori au fost potrivite pentru dimensiunea magistralei sistem şi frecvenţa procesorului 286, permiţând viteze de până la 16 Megaocţeti/secundă. Odată cu creşterea vitezei procesoarelor şi a foamei de lăţime de bandă de către unele controlere şi periferice, ca de exemplu adaptoarele video, hard disk-urile, controlerele de reţea, viteza oferită de o magistrală ISA a devenit insuficientă. Deşi, teoretic, pe o magistrală ISA se pot obţine viteze de până la 16 Megaocteţi/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 sfârşitul anilor '90 pentru conectarea la calculator mai ales a controlerelor de reţea de până la 10 Mbps (Mbps = megabiţi pe secundă), a plăcilor de sunet şi a modemurilor. Calculatoarele personale de astăzi, păstrează încă o relicvă a acestui tip de magistrală. Portul pentru tastatură şi mouse, iar în unele cazuri porturile seriale şi paralele, precum şi controlerul unităţii de dischetă, sunt conectate la o magistrală ISA care este cascadată la magistrala de I/O a sistemului prin intermediul magistralei PCI.

Magistrala PCI (Peripheral Component Interconnect)

A fost introdusă de Intel la sfârşitul anilor '90 şi se bucură de succes chiar şi astăzi. Este o magistrală pe 32 de biţi şi operează

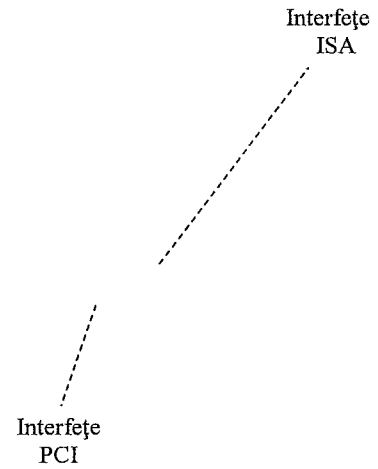


Fig. 2.6. Interfeţele ISA şi PCI din punct de vedere fizic în cadrul unui calculator personal

la frecvenţe începând cu 33 MHz. Aceste valori permit viteze teoretice de până la 132 Megaocteţi pe secundă (33 • 10^6 • 4 octeţi). Majoritatea controlerelor existente în momentul de faţă se conectează la această magistrală: controlere IDE/ATA pentru conectarea hard disk-urilor, controlere de reţea, controlere multimedia de captură, sunet, etc. Este specifică şi altor tipuri de calculatoare, nu numai calculatoarelor personale, cum ar fi Power Macintosh.

Magistrala AGP (Accelerated Graphics Port)

S-a născut din nevoia de lăţime de bandă mai mare spre controlerul video. Aplicaţii precum jocurile cereau o lăţime de bandă pe care magistrala PCI nu o putea oferi. Făcând un calcul simplu, un joc care rulează la rezoluţia 1600 x 1200, cu 32 biţi de culoare per pixel, pentru a putea reda 30 de cadre pe secundă are nevoie de o lăţime de bandă spre controlerul video de 1600 • 1200 • 4 octeţi • 30 ≈ 230 Megaocteţi pe secundă, cu mult peste posibilităţile oferite de magistrala PCI. De fapt numele de magistrala AGP este folosit impropriu, AGP fiind un canal de comunicaţie punct la punct. Magistrala AGP 1x este pe 32 de biţi şi lucrează la frecvenţe de 66 MHz, oferind o viteză de aproximativ 266 Megaocteţi pe secundă – dublu faţă de viteza oferită de magistrala PCI. Această lăţime de bandă este dedicată în întregime controlerului video, în timp ce un cotroler video PCI era nevoit să partajeze această viteză cu alte controlere PCI, mari consumatoare de lăţime de bandă, cum ar fi controlere IDE/ATA. Magistralele AGP 8x actuale, permit atingerea unor viteze de 8 ori mai mare decât specificaţiile AGP iniţiale (2133 Megaocteţi pe secundă).

Atât magistrala PCI cât şi cea AGP vor fi înlocuite de magistrala PCI Express, care promite, cel puţin teoretic, viteze de până la 4 Gigaocteţi, atât dinspre controler înspre procesor, cât şi în sens invers.

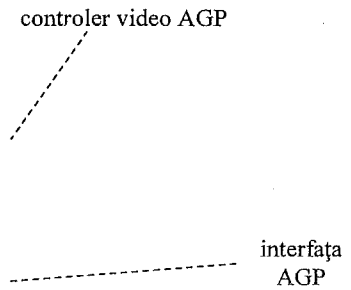


Fig. 2.7. Interfaţa AGP şi un controler video AGP

Ansamblul format din controler, indiferent de tipul de magistrală I/O internă la care este conectat acesta, împreună cu interfaţa dintre acesta şi echipamentul periferic este referit şi sub numele de magistrală externă. Printre cele mai cunoscute magistrale externe amintim: IDE/ATA, SCSI, USB.

2.4.3. Magistrale I/O externe şi interfeţele asociate

IDE/ATA (Integrated Drive Electronics/Advanced Technology Attachment)

Interfaţa IDE sau ATA cum mai este cunoscută, a fost folosită de la mijlocul anilor '80 în vederea conectării perifericelor de stocare cum ar fi hard disk-urile şi unităţile optice. Primele specificaţii ATA, foloseau procesorul (aşa numitul *programmed input/output mode* sau mod PIO) pentru a transfera informaţia octet cu octet de pe dispozitivul periferic de stocare în memoria internă prin intermediul regiştrilor procesorului. Acest mod de lucru, permitea obţinerea de viteze relativ mici de până la 16.7 Megaocteţi pe secundă şi, mai grav, reţineau procesorul ocupat în timpul acestui transfer. Această viteză a crescut însă odată cu folosirea DMA (*Direct Memory Access*), permiţându-se astfel copierea informaţiei în blocuri de pe dispozitivul de stocare direct în memoria internă fără a mai implica procesorul în acest transfer. Vitezele actuale ale interfeţelor ATA 133 se apropie cel puţin teoretic de limitele magistralei PCI de 132 Megaocteţi pe secundă pentru transferul datelor (să nu uităm că informaţia citită de pe dispozitivul de stocare trebuie să treacă prin magistrala PCI via controler).

Specificaţiile ATA descriu şi modul de adresare a dispozitivului de stocare. Ultimele specificaţii prevăd o adresare pe 48 de biţi, lucru care permite folosirea teoretică a dispozitivelor de stocare cu capacităţi de până la 32 de Teraocteţi. Au existat situaţii când proiectanţii de controlere IDE au subestimat viteza de creştere a capacităţii unităţilor de stocare, modele noi de astfel de dispozitive fiind incompatibile cu controlerele mai vechi (astfel de bariere au fost întâlnite la trecerea la capacităţi de stocare de peste 504 Megaocteţi, 8, 32, 137 Gigaocteţi).

Fizic, pe un controler IDE/ATA se pot conecta doar două dispozitive de stocare, unul operând în mod *master* şi celălalt în mod *slave*. Majoritatea calculatoarelor personale, fiind dotate cu două controlere IDE (numite *primary* şi *secondary* IDE), pot folosi la un moment dat doar 4 dispozitive periferice IDE. Dacă se doreşte folosirea unui număr mai mare de periferice IDE/ATA, este necesară instalarea unui controler IDE/ATA suplimentar, de obicei PCI.

SCSI (Small Computer System Interface)

Acest tip de magistrală s-a născut la mijlocul anilor '80, fiind cel mai des folosită pentru conectarea dispozitivelor de stocare cum ar fi hard disk-uri, unităţi optice, unităţi de bandă magnetică. Poate fi însă folosită şi pentru conectarea altor tipuri de periferice cum ar fi scannere sau imprimante. Nu s-a bucurat niciodată de popularitate în calculatoarele personale, fiind întâlnită mai ales în calculatoarele Apple şi în staţiile şi serverele Sun. Acest tip de magistrală, poate fi folosită atât ca magistrală internă, cât şi ca magistrală externă. În calculatoarele personale, a fost folosită doar ca magistrală externă, cumunicând cu procesorul doar prin intermediul unui cotroler ISA sau PCI prin magistrala internă corespunzătoare tipului de controler. Funcţional, o magistrală SCSI operează la frecvenţe cuprinse între 5 MHz (conform primelor specificaţii SCSI) şi 80 de MHz, dimensiunea magistralei fiind de 8 sau 16 biţi. O magistrală SCSI pe 16 biţi operând la 80 MHz poate atinge viteze de până la 320 Megaocteţi pe secundă în transferul datelor.

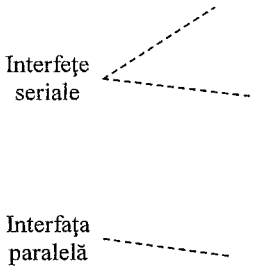
Fiecărui dispozitiv periferic conectat la o magistrală SCSI i se asociază un identificator – *SCSIid*. Numărul de biţi pe care se reprezintă acest identificator implică şi numărul de dispozitive care se pot conecta pe aceeaşi magistrală. Astfel, spre deosebire de IDE care permitea doar două dispozitive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

Interfaţa serială

Comunică 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) şi terminalele virtuale. Controlerul serial la calculatoarele personale de azi este pe cale de dispariţie, majoritatea perifericelor care se conectau la calculator pe această interfaţă conectându-se în prezent prin intermediul unei interfeţe 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 biţi /secundă ≈ 16 kiloocteţi / secundă.

Interfaţa paralelă

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



Înainte de reducerea costurilor controlerelor de reţea şi a răspândirii reţelelor locale, interfeţele seriale şi paralele au fost des folosite la legarea în reţea a două calculatoare. Chiar şi azi, multe legături în reţeaua Internet sunt legături punct la punct, realizate între interfeţele seriale a două calculatoare.

Fig. 2.8. Interfeţe seriale şi paralele

USB (Universal Serial Bus)

Interfaţa USB s-a născut din nevoia unui mecanism universal de conectare a echipamentelor periferice externe la calculator, indiferent de tipul acestora: imprimante, tastaturi, *mouse*-uri, hard disk-uri, unităţi optice, camere video, scannere, telefoane mobile, etc. Viteza prevăzută de specificaţiile actuale, USB 2.0, de 57 Megaocteţi pe secundă este practic suficientă pentru conectarea oricărui dispozitiv extern calculatorului, mai puţin a unui monitor. Principiul de comunicare cu acestea este serial, interfaţa 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 biţi (de fapt un id este consumat de controlerul USB însuşi). Lucru deosebit de important, specificaţiile USB permit adăugarea în timpul funcţionării 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 montează procesorul, memoria, pe ea sunt cablate fizic magistralele interne ale calculatorului, interfeţele PCI, ISA şi AGP.

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

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

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

- eventual interfeţe ISA;
- cel puţin două interfeţe USB dacă placa de bază are un asemenea controler;
- una sau două interfeţe seriale şi o interfaţă paralelă (vezi paragraful 2.4.3.);
- cel puţin trei interfeţe audio (intrare, ieşire şi microfon) dacă placa de bază are un controler (placă) de sunet;
- interfaţă de reţea în cazul prezenţei unui asemenea controler pe placa de bază;
- interfeţe PS/2 pentru ataşarea unei tastaturi şi a unui mouse;
- interfaţă specială pentru ataşarea unui *joystick* (aşa numitul *game port*) sau a altor periferice destinate jocurilor.

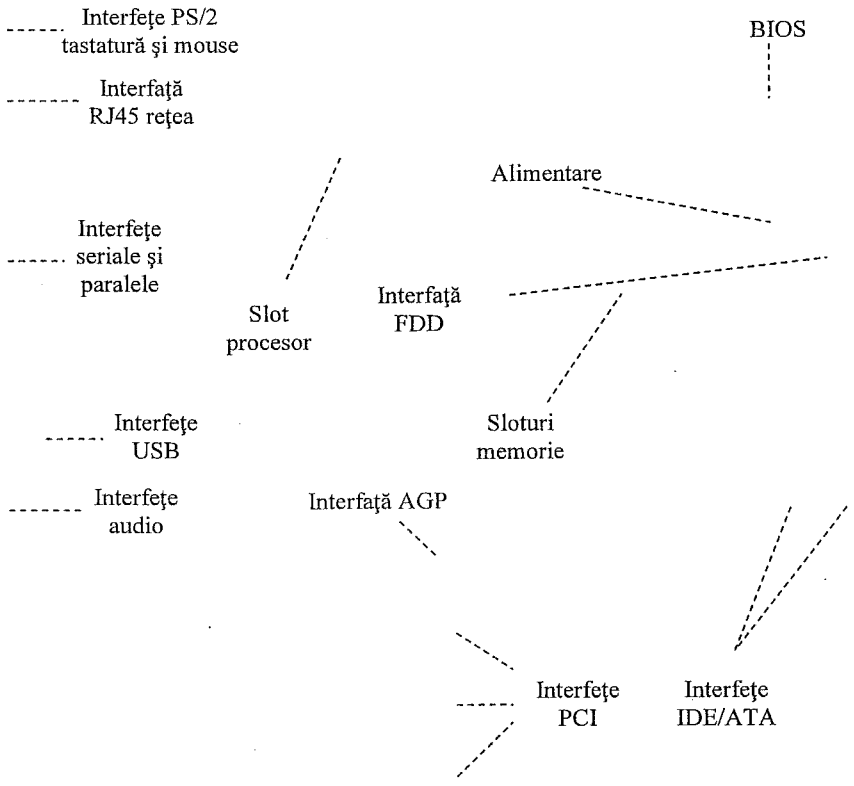


Fig. 2.9. Placă de bază a unui calculator personal

Aceste interfeţe sunt ilustrate în figura 2.9. Există situaţii în care echipamentul periferic nu se ataşează direct interfeţei oferită de controler sau de placa de bază, fiind necesar un cablu (sau "panglică") prelungitor care este parte integrantă a interfeţei fizice de conectare oferită de controlerul respectiv. Spre exemplu, un hard disk nu se conectează direct interfeţei IDE/ATA de pe placa de bază, fiind necesar un cablu (numit eronat uneori controler IDE) prelungitor de la interfaţa IDE a plăcii de bază la hard disk. De asemenea, un mouse serial se poate conecta la controlerul serial prin intermediul unei cablu prelungitor serial.

Prezentăm în continuare lista echipamentelor periferice ce se pot conecta pe fiecare dintre aceste interfeţe fizice:

- interfeţele PCI (vezi figura 2.6.): permit conectarea oricărui controler care comunică cu procesorul prin intermediul magistralei PCI. Majoritatea controlerelor standard sunt pe placa de bază, însă la nevoie se pot adăuga altele noi: adaptoare SCSI, TV-Tunere, controlere IDE sau USB suplimentare, plăci de sunet, modemuri, controlere *wireless* (pentru comunicaţii fără fir);

- interfaţa AGP (vezi figura 2.7.) permite conectarea unui controler video. Un controler video din zilele noastre poate fi considerat un calculator în calculator, fiind dotat cu propriul procesor grafic şi propria memorie. Uneori, procesorul grafic şi memoria controlerului video pot fi comparabile ca putere şi dimensiune cu procesorul şi memoria sistemului (acest lucru se datorează în primul rând cifrelor mari de afaceri din industria jocurilor pe calculator).

- interfaţa IDE/ATA permite conectarea hard disk-urilor şi unităţilor optice (CD-RW, DVD-ROM, etc). Pentru conectarea acestor echipamente la această interfaţă este necesară o "panglică" suplimentară. Este permisă adăugarea pe un singur controler a două astfel de echipamente periferice, unul operând în mod master şi celălalt în mod slave. Acest mod de operare se configurează cu ajutorul unor jumperi prezenţi la fiecare hard disk sau unitate optică IDE.

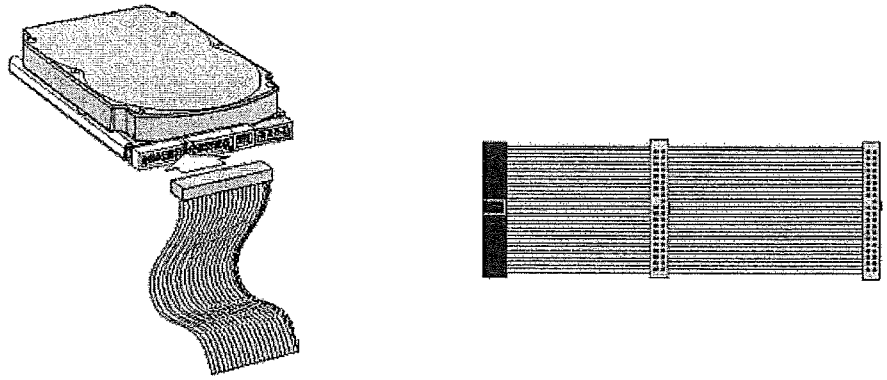


Fig. 2.10. Hard disk şi cablu de conectare IDE/ATA

- interfaţa FDD: permite tot prin intermediul unei "panglici" ataşarea a două unităţi de dischetă.

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

- interfaţa de reţea: permite conectarea calculatorului la o reţea locală şi, prin intermediul acesteia, la Internet;

- interfeţele audio: permit conectarea la calculator a unor boxe, căşti sau a unui microfon;

- interfeţele PS/2 permit conectarea la calculator a unei tastaturi şi a unui mouse.

- interfeţele USB permit conectarea de echipamente periferice externe. Cele mai des întâlnite echipamente periferice USB sunt: tastaturi şi mouse-uri USB, echipamente de stocare (*memory-stickuri*, hard disk-uri, unităţi optice externe, unităţi de dischetă externe), telefoane mobile, imprimante, adaptoare pentru comunicaţii fără fir (*wireless*), *joystick*-uri şi alte echipamente destinate jocurilor pe calculator, camere video şi aparate de fotografiat digitale, scannere, etc. Practic interfaţa USB va duce la dispariţia totală din calculator a unor interfeţe ca cele seriale, paralele, PS/2 şi a *game port*-ului.

Conector RJ45

Conector AUI

Conector BNC

Fig. 2.11. Interfeţe de reţea

Fig.2.12. Interfeţe audio

Interfaţă PS/2

Interfeţe USB

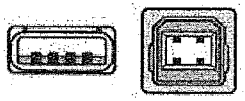


Fig. 2.13.
Interfeţe USB şi PS/2

2.5. PERFORMANŢELE UNUI SISTEM DE CALCUL

În general sunt utilizate două tipuri de criterii de evaluare a performanţelor:
- *timpul de execuţie* (*Execution Time*) = timpul în care este executată o sarcină, timpul de răspuns
- *rata de execuţie* (*Throughput, Bandwidth*) = numărul de sarcini rezolvate într-un anumit interval de timp (zi, oră, secundă, milisecundă...),

de unde rezultă că îmbunătăţirea performanţei poate fi privită sub două aspecte:
- reducerea timpului de rezolvare a unei sarcini
- creşterea ratei de execuţie.

Acordarea unei „note" întregului sistem de calcul este dificil de realizat, motiv pentru care se face referire la diferite aspecte ale performanţelor componentelor sale. De exemplu:
- CPU:
  - timpul de execuţie (perioadă de ceas redusă, execuţie paralelă a instrucţiunilor)
  - rata de execuţie (MIPS, MFLOPS, planificarea eficientă a instrucţiunilor)
  - 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 afişate pe secundă
- ş.a.m.d.

Totuşi, alături de performanţele individuale ale componentelor, trebuie avute în vedere şi alte aspecte precum: compatibilitatea componentelor, tipurile de date gestionate sau de aplicaţii executate, sistemul de operare, disponibilitatea software-ului, costurile de proiectare sau costurile de achiziţionare sau întreţinere.

2.6. ARHITECTURA MICROPROCESORULUI 8086

2.6.1. Structura microprocesorului

Această structură este ilustrată în figura 2.14. Microprocesorul dispune de mai mulţi regiştri generali pe 16 biţi. El este format din două componente mari:

- **EU** (*Executive Unit*) care execută instrucţiunile maşină prin intermediul componentei **ALU** (*Aritmetic and Logic Unit*).
- **BIU** (*Bus Interface Unit*) este componenta care pregăteşte execuţia fiecărei instrucţiuni maşină. În esenţă, această componentă citeşte o instrucţiune din memorie, o decodifică şi calculează adresa din memorie a unui eventual operand. Configuraţia rezultată este depusă într-o zonă tampon cu dimensiunea de 6 octeţi, de unde va fi preluată de **EU**.

Cele două componente lucrează în paralel, în sensul că în timp ce **EU** execută instrucţiunea curentă, **BIU** pregăteşte instrucţiunea următoare. Cele două acţiuni sunt sincronizate, în sensul că cea care termină prima aşteaptă după cealaltă.

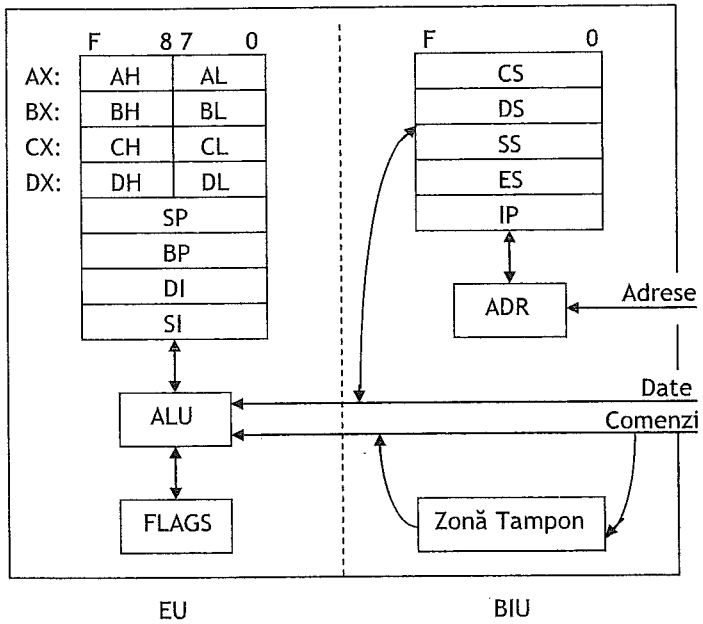


Fig. 2.14. Arhitectura microprocesorului 8086

2.6.2. Regiştrii generali EU

Registrul **AX** este *registrul acumulator*. El este folosit de către majoritatea instrucţiunilor ca unul dintre operanzi.

Registrul **BX** este folosit în principal ca *registru de bază*. Folosirea lui în această accepţiune va fi descrisă în secţiunea 2.6.7.

Registrul **CX** este folosit în principal ca *registru de numărare (registru contor)* pentru instrucţiunile care au nevoie de indicaţii numerice.

Registrul **DX** este un *registru de date*. Împreună cu registrul AX se foloseşte în calculele ale căror rezultate depăşesc dimensiunea unui cuvînt.

Fiecare dintre regiştrii AX, BX, CX, DX au capacitatea de 16 biţi. Fiecare dintre ei poate fi privit în acelaşi timp ca fiind format prin concatenarea (alipirea) a doi (sub)regiştri. Subregistrul superior conţine cei mai semnificativi 8 biţi (partea HIGH) ai registrului de 16 biţi din care face parte. Există astfel regiştrii **AH, BH, CH, DH**. Subregistrul inferior conţine cei mai puţin semnificativi 8 biţi (partea LOW) ai registrului de 16 biţi din care face parte. Există astfel regiştrii **AL, BL, CL, DL**.

Regiştrii **SP** şi **BP** sunt regiştri destinaţi lucrului cu *stiva*. O stivă se defineşte ca fiind o zonă de memorie în care se pot depune succesiv valori, extragerea lor ulterioară făcându-se în ordinea inversă depunerii.

Registrul **SP** (*Stack Pointer*) punctează spre elementul ultim introdus în stivă (elementul din *vârful stivei*).

Registrul **BP** (*Base pointer*) punctează spre primul element introdus în stivă (indică *bazei stivei*). Rolul lui va fi evidenţiat în capitolul 8, secţiunea 8.3.

Regiştrii **DI** şi **SI** sunt *regiştrii de index* utilizaţi de obicei pentru accesarea elementelor din şiruri de octeţi sau de cuvinte. Denumirile lor (*Destination Index* şi *Source Index*) precum şi rolurile lor vor fi clarificate în capitolul 4, secţiunea 4.4.2.

2.6.3. Flagurile

Un *flag* este un indicator reprezentat pe un bit. O configuraţie a *registrului de flaguri* indică un rezumat sintetic a execuţiei fiecărei instrucţiuni. Pentru 8086 acest registru (notat FLAGS în figura 2.14) are 16 biţi dintre care sunt folosiţi numai 9. Structura în detaliu a registrului FLAGS este dată în figura 2.15.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
x	x	x	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

Fig. 2.15. Structura registrului de flaguri 8086

CF (*Carry Flag*) este flagul de transport. Are valoarea 1 în cazul în care în cadrul operaţiei s-a făcut transport în afara domeniului de reprezentare a rezultatului. De exemplu, dacă se efectuează următoarea adunare între doi octeţi:

```
 1001 0011 +
 0111 0011
 ----------
1 0000 0110
```

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

PF (*Parity Flag*) este flagul de paritate. Valoarea lui se stabileşte în aşa fel încât împreună cu numărul de biţi 1 din reprezentarea rezultatului instrucţiunii să rezulte un număr impar de cifre 1.

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

ZF (*Zero Flag*) primeşte valoarea 1 dacă rezultatul instrucţiunii este egal cu zero şi valoarea 0 la rezultat diferit de zero.

SF (*Sign Flag*) primeşte valoarea 1 dacă rezultatul execuţiei instrucţiunii este un număr strict negativ şi valoarea 0 în caz contrar.

TF (*Trap Flag*) este un flag de depanare. Dacă are valoarea 1, atunci maşina se opreşte după fiecare instrucţiune.

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

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

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

Semnificaţiile de mai sus sunt generale. De fapt, fiecare instrucţiune îşi specifică modul propriu de setare şi interpretare a flagurilor.

2.6.4. Regiştrii de adresă şi calculul de adresă

Prin definiţie, *adresa unei locaţii de memorie* este numărul de octeţi consecutivi aflaţi între începutul memoriei RAM şi începutul locaţiei respective.

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

Memoria se adresează deci pe 20 de biţi şi există doar regiştri de 16 biţi. Pentru rezolvarea situaţiei a apărut conceptul de segment de memorie. *Segmentul de memorie* reprezintă o succesiune continuă de octeţi care are următoarele proprietăţi: începe la o adresă multiplu de 16 octeţi, are lungimea multiplu de 16 octeţi şi are lungimea de maximum 64 Ko. Deoarece adresa de început a fiecărui

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

Vom numi *offset* sau *deplasament* adresa unei locaţii faţă de începutul unui segment. Deoarece un segment are maximum 64Ko, sunt suficienţi 16 biţi pentru a reprezenta orice offset.

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

$$s_3s_2s_1s_0 : o_3o_2o_1o_0$$

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

$$a_4a_3a_2a_1a_0 := s_3s_2s_1s_0 0 + o_3o_2o_1o_0$$

unde $a_4a_3a_2a_1a_0$ este adresa calculată (scrisă în hexazecimal). Deci configuraţia de 16 biţi $s_3s_2s_1s_0$ care localizează începutul segmentului este înmulţită cu 16 (i se adaugă o cifră 0 în baza 16, sau 4 cifre 0 în baza 2) şi la rezultat se adună valoarea offsetului $o_3o_2o_1o_0$. 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 uşor de observat că există mai multe specificări pentru aceeaşi adresă. De exemplu, adresa de mai sus poate fi specificată şi prin 810B : 0003. În acest fel este posibilă suprapunerea mai multor segmente în aceeaşi arie de memorie.

Acest mecanism de adresare este tipic pentru 8086 şi 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 8O86*;

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 existenţa a patru tipuri de segmente:

- *segment de cod*, care conţine instrucţiuni maşină;
- *segment de date*, care conţine date asupra cărora se acţionează în conformitate cu instrucţiunile;

- *segment de stivă*;
- *segment suplimentar* (extrasegment).

Fiecare program este compus din unul sau mai multe segmente, de unul sau mai multe dintre tipurile de mai sus. În fiecare moment al execuţiei este declarat activ câte un singur segment din fiecare tip. Regiştrii **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*) şi **ES** (*Extra Segment*) din **BIU** reţin adresele de început ale segmentelor active, corespunzător fiecărui tip. Deci regiştrii CS, DS, SS şi ES conţin (de fapt) adresele de început ale segmentelor active: de cod, de date, de stivă şi suplimentar. Registrul **IP** conţine offsetul instrucţiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către **BIU**

2.6.5. Reprezentarea instrucţiunilor maşină

O instrucţiune maşină 8086 are maximum doi operanzi. Pentru cele mai multe dintre instrucţiuni, cei doi operanzi poartă numele de *sursă*, respectiv *destinaţie*. Dintre cei doi operanzi, maximum unul se poate afla în memoria RAM. Celălalt se află fie într-un registru al **EU**, fie este o constantă întreagă. Astfel, o instrucţiune apare sub forma:

numeinstrucţiune *destinaţie, sursă*

Formatul intern al unei instrucţiuni este variabil, el putând ocupa între 1 şi 6 octeţi. Semnificaţia generală a octeţilor reprezentării este următoarea:

Primul octet, numit *cod* identifică instrucţiunea de executat.
Al doilea octet, numit *octetmod* specifică pentru unele dintre instrucţiuni natura şi locul operanzilor (registru, memorie, constantă întreagă etc.). Unele instrucţiuni folosesc acest octet pentru a reprezenta în el o *constantă scurtă (short)*, adică o constantă care poate lua valori între -128 şi 127.

Majoritatea instrucţiunilor folosesc pentru reprezentare fie numai octetul cod, fie octetul cod urmat de octetmod.

Următorii maximum patru octeţi, dacă apar, identifică fie o adresă de memorie, fie o constantă reprezentată pe mai mult de un octet. De aici se deduce o restricţie suplimentară privind operanzii: nu este permisă folosirea unei adrese de memorie dacă celălalt operand este o constantă reprezentată pe mai mult de un octet.

2.6.6. Adrese FAR şi NEAR

Aşa cum am văzut în secţiunea 2.6.4., pentru a adresa o locaţie 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 alte specificări, adresa segmentului din unul dintre regiştrii de segment CS, DS, SS sau ES. Alegerea implicită a unui registru de segment se face după nişte reguli proprii instrucţiunii folosite.

Prin definiţie, o adresă în care se specifică doar offsetul, urmând ca segmentul să fie preluat implicit dintr-un registru de segment poartă numele de *adresa NEAR* (adresă apropiată). O adresă NEAR se află întotdeauna în interiorul unuia din cele patru segmente active.

O adresă în care programatorul specifică explicit adresa de început a segmentului poartă numele de *adresă FAR* (adresă îndepărtată). O adresă FAR se poate specifica în trei moduri:

- $s_3s_2s_1s_0$: specificare_offset unde $s_3s_2s_1s_0$ este o constantă;
- CS : specificare_offset sau
  DS : specificare_offset sau
  ES : specificare_offset sau
  SS : specificare_offset;
- VARDOUBLE unde prin VARDOUBLE se indică numele (adresa) unei variabile de tip dublu cuvânt, care conţine o adresă FAR.

Formatul intern al unei adrese FAR este: la adresa mai mică se află cuvântul care indică offsetul, iar la adresa mai mare cu 2 (cuvântul care urmează) se află cuvântul care indică segmentul.

După cum se observă, şi reprezentarea adreselor respectă principiul reprezentării little-endian expus în capitolul 1, paragraful 1.3.2.3: partea cea mai puţin semnificativă are adresa cea mai mică, iar partea cea mai semnificativă are adresa cea mai mare.

Faptul că o instrucţiune foloseşte o adresă FAR sau NEAR se reflectă în conţinutul octeţilor care codifică instrucţiunea. În capitolul 4 vom vedea cum poate fi specificată natura adresei în limbaj de asamblare, lăsând astfel pe seama asamblorului codificarea instrucţiunii.

2.6.7. Calculul offsetului unui operand. Moduri de adresare

În cadrul unei instrucţiuni există mai multe moduri de a calcula offsetul unui operand pe care aceasta îl solicită:

- *modul registru*, dacă pe post de operand se află un registru al maşinii;
- *modul imediat*, atunci când în instrucţiune se află chiar valoarea operandului (nu adresa lui şi nici un registru în care să fie conţinut);
- *modul adresare la memorie*, dacă operandul se află efectiv undeva în memorie. În acest caz, adresa offsetului lui se calculează după următoarea formulă:

$$adresaoffset = [\ BX \mid BP\ ] + [\ SI \mid DI\ ] + [\ constanta\ ]$$

Deci *adresaoffset* se obţine adunând următoarele trei elemente, sau numai unele dintre ele:

- conţinutul unuia dintre regiştrii BX sau BP
- conţinutul unuia dintre regiştrii SI sau DI;
- valoarea unei constante.

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 şi indexată etc.

La instrucţiunile de salt mai apar două tipuri de adresări.

Adresa relativă indică poziţia următoarei instrucţiuni de executat, în raport cu poziţia curentă. Poziţia este indicată prin numărul de octeţi de cod peste care se va sări, număr ce ia valori între -128 şi 127. O astfel de adresă mai poartă numele de *adresă scurtă (SHORT Adress)*.

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

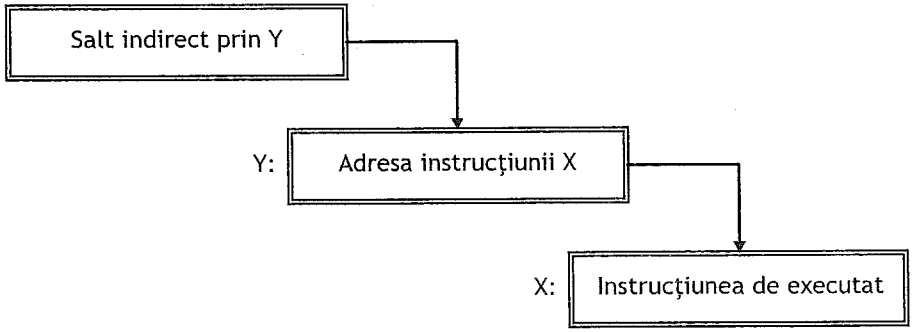


Fig. 2.16. Adresarea indirectă

Adresarea indirectă asigură o mai mare flexibilitate în controlul succesiunii instrucţiunilor. De exemplu, în figura 2.16 conţinutul locaţiei Y poate să difere de la un moment la altul a execuţiei programului, ceea ce permite execuţia la momente diferite a unor instrucţiuni 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 instrucţiunilor coincide cu operaţiile maşinii şi ale cărui structuri de date coincid cu structurile primare de date ale maşinii.

Limbajul maşină al unui sistem de calcul (SC) este format din totalitatea instrucţiunilor maşină puse la dispoziţie de procesorul SC. Acestea se reprezintă sub forma unor şiruri de biţi cu semnificaţie prestabilită. În 2.6.5. am prezentat formatul instrucţiunilor maşină pentru 8086.

În ultimă instanţă deci, orice program apare în calculator ca un şir (mare!) de biţi. A manipula aceste structuri sub un astfel de format este extrem de dificil. Limbajul de asamblare vine să uşureze această sarcină. Astfel, codurile instrucţiunilor maşină se rescriu sub forma unor cuvinte simbol, numite ***mnemonice***, suficient de sugestive pentru a indica semantica instrucţiunii respective. Adresele de memorie frecvent folosite (fie ca destinaţii ale unor salturi în program, fie ca denumiri asociate unor locaţii de memorie) sunt notate şi ele prin nişte cuvinte simbol, numite ***etichete***.

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

Principalele servicii oferite de către un asamblor (şi în consecinţă şi de către asamblorul 8086) constau în:

- detectarea erorilor de sintaxă
- traducerea din scrierea cu mnemonice în cod binar
- generarea de octeţi folosind pseudoinstrucţiuni
- calculul unor adrese de salt (deplasamente) folosind operaţii cu etichete
- evaluarea în timpul translatării a unor expresii aritmetice simple
- posibilitatea asamblării condiţionate
- posibilitatea definirii şi utilizării de macroinstrucţiuni

Elementele cu care lucrează un asamblor sunt:

* **etichete** - nume scrise de utilizator, cu ajutorul cărora acesta se poate referi anumite date sau zone de memorie.

* **instrucţiuni** - scrise sub forma unor mnemonice care sugerează acţiunea, fiecare instrucţiune a limbajului de asamblare corespunzând unei instrucţiuni maşină. Asamblorul generează octeţii care codifică instrucţiunea respectivă.

* **directive** - sunt indicaţii date asamblorului în diverse scopuri cum ar fi: relaţii între modulele obiect, definirea unor segmente, indicaţii de asamblare condiţionată, machete de macrogenerare, controlul listingului, etc. Directivele care indică asamblorului să încarce o zonă de memorie cu un conţinut dorit de programator sau care cer rezervarea unui număr de octeţi în vederea folosirii ulterioare se mai numesc şi *pseudoinstrucţiuni* sau *directive de generare a datelor.*

* **contor de locaţii** - este un număr întreg gestionat de asamblor. În fiecare moment, valoarea contorului coincide cu numărul de octeţi generaţi corespunzător instrucţiunilor şi directivelor deja întâlnite în cadrul segmentului respectiv (deplasamentul curent în cadrul segmentului). Programatorul poate utiliza această valoare (accesare doar în citire!) prin simbolul '$'. Să reţinem deci că acest contor de locaţii nu are regimul unei valori globale la nivelul unui program, ci în mod practic fiecare segment poate face referire la propriul contor de locaţii, această valoare având astfel regim de valoare locală (fiecare segment de memorie are asociat propriul contor de locaţii la care poate face apel prin intermediul simbolului '$'). Spre exemplu, simbolul '$' utilizat în cadrul unui segment de date va desemna numărul de octeţi generaţi până în momentul accesării valorii '$' în cadrul acelui segment de date. Acelaşi simbol '$' utilizat în cadrul unui segment de cod va desemna numărul de octeţi generaţi corespunzător în cadrul segmentului de cod respectiv.

Introducerea noţiunii de contor de locaţii se va dovedi utilă în scopul identificării/utilizării dinamice a unui punct curent de execuţie în cadrul unui program (segment de cod/instrucţiuni) sau a adreselor unor locaţii de memorie care tocmai au fost alocate în cadrul unor segmente de date. Acest concept al contorului de locaţii propriu fiecărui segment de memorie este relevant pentru a înţelege modul în care trebuie să privim execuţia unor programe în limbaj de asamblare: **execuţia unui program în limbaj de asamblare nu este altceva decât o generare de octeţi corespunzătoare instrucţiunilor şi directivelor specificate în codul sursă de către programator!**

Într-adevăr, un program în execuţie va genera în cadrul unui segment de cod octeţi reprezentând codurile instrucţiunilor ce trebuie executate, iar în cadrul unui segment de date se vor genera corespunzător în memorie octeţii ce vor reflecta alocarea datelor declarate de către programator, în funcţie de tipul de date specificat. În acest sens este important de reţinut că în cadrul limbajului de asamblare **noţiunea de *tip de dată* este echivalentă cu *dimensiunea de reprezentare*** a datei respective (viziune low-level asupra noţiunii de tip de dată specifică limbajelor de asamblare). Vom reveni practic asupra tuturor acestor aspecte şi asupra efectelor lor pe tot parcursul volumului

deoarece este foarte important să punctăm semantica asociată instrucţiunilor şi directivelor din exemplele ce se vor prezenta prin referirea la noţiunile şi conceptele definite aici.

3.1. FORMATUL UNEI LINII SURSĂ

Formatul unei linii sursă în limbajul de asamblare 8086 este următorul:

[*etichetă*] [*mnemonică*] [*operanzi*] [;*comentariu*]

Caracterele din care poate fi constituită o *etichetă* sunt următoarele:

A-Z a-z _ @ $? 0-9

Cifrele 0-9 nu pot fi folosite ca prim caracter, iar simbolurile $ şi ? folosite singure au un înţeles special: simbolul '$' desemnează valoarea contorului de locaţii iar simbolul '?' este folosit pentru rezervarea de spaţiu de memorie fără iniţializare. Astfel, ele nu pot fi utilizate drept nume de simbol al utilizatorului.

La nivelul limbajului de asamblare se întâlnesc două categorii de etichete:

1). ***etichete de cod***, care apar în cadrul secvenţelor de instrucţiuni (deci în cadrul unor segmente de cod) cu scopul de a defini destinaţiile de transfer ale controlului în cadrul unui program. O etichetă de cod este de fapt echivalentul noţiunii de *etichetă* ce apare la nivelul unui limbaj de programare de nivel înalt, semantica ei fiind definirea destinaţiei unor instrucţiuni de tip **goto**. La definirea lor în cadrul programelor, etichetele ce prefixează instrucţiuni (etichetele de cod) trebuie să fie urmate de caracterul ':'.

2). ***etichete de date***, care identifică simbolic unele locaţii de memorie, din punct de vedere semantic ele fiind echivalentul noţiunii de *variabilă* din limbajele de nivel înalt.

Fiecare etichetă (de cod sau de date) trebuie să fie definită o singură dată (cu alte cuvinte, etichetele trebuie să fie unice). Ca operand, o aceeaşi etichetă poate să fie accesată însă de oricâte ori în cadrul unui program.

***Valoarea* unei etichete în limbaj de asamblare este un număr întreg reprezentând adresa instrucţiunii sau directivei ce urmează etichetei.**

Pentru un programator obişnuit cu limbajele de nivel înalt acest lucru este evident uşor de acceptat în cazul etichetelor de cod, deoarece este normal ca destinaţia unui salt să fie specificată printr-o adresă. Apare însă în mod natural întrebarea: *de ce s-a luat această decizie şi pentru o etichetă de date?* Adică, de ce s-a hotărât şi în cazul numelui unei variabile ca identificatorul respectiv să fie asociat ca valoare cu adresa acelei variabile? Pe de altă parte, dacă lucrurile stau în acest fel şi de fapt simbolul p asociat unei variabile va desemna în limbajul de asamblare adresa sa, cum vom

putea atunci avea acces la conţinutul acelei locaţii ? Avem oare nevoie de o altă notaţie 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 notaţie sintactică diferită sau specifică ***pentru a diferenţia între accesarea adresei şi respectiv a conţinutului unei locaţii cu nume*** ci acest lucru se va face **în funcţie de contextul utilizării**. Este foarte important să înţelegem aceste mecanisme: uneori, simbolul asociat unei variabile (numele său) va desemna NUMAI adresa sa (conform definiţiei) – spre exemplu cazul în care manipulăm doar adresele unor locaţii fără a fi interesaţi de conţinutul acestora; alteori, când vom dori manipularea conţinutului unei locaţii vom nota acest conţinut tot cu acel nume, însă contextul utilizării acelui simbol (adică alegerea adecvată a unei instrucţiuni care să ştie să extragă acel conţinut de la adresa furnizată prin specificarea numelui variabilei) va provoca interpretarea simbolului respectiv drept conţinutul acelei locaţii (similar utilizării numelor de variabile în cazul limbajelor de nivel înalt).

Decizia în discuţie s-a luat pentru a putea oferi, **în funcţie de contextul utilizării**, acces fie la valoarea unei variabile fie la adresa sa, fără a utiliza operatori suplimentari în acest scop. Să reţinem 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 instrucţiunilor:

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

deşi se utilizează acelaşi simbol *v* ca operand sursă, interpretarea sa va veni din contextul utilizării adică din tipul de instrucţiune folosit: instrucţiunea *lea* va considera *v* drept adresă, în timp ce *mov* îl va interpreta drept conţinut (de fapt mai exact şi instrucţiunea *mov* va considera într-o primă fază faptul că v desemnează o adresă numai că în plus, *mov* fiind o instrucţiune de transfer a valorilor, va provoca şi extragerea valorii de la acea adresă!). Cu alte cuvinte, operaţia de ***dereferenţiere*** (extragerea valorii de la o anumită adresă specificată) are loc implicit, în funcţie de contextul utilizării acelui simbol.

Acceptarea în cadrul instrucţiunilor şi expresiilor limbajului de asamblare a numelor de variabile drept adrese oferă posibilitatea efectuării ***aritmeticii de adrese*** (*pointer arithmetic* – adunare, scădere de constante la pointer sau scădere de doi pointeri). De exemplu, expresia b-a (cu a şi b nume de variabile de memorie şi "–" desemnând un operator şi nu o instrucţiune) va desemna o aritmetică de adrese în limbajul de asamblare şi nu o scădere a două "conţinuturi" de variabile. Se remarcă astfel aici influenţa pe care limbajul de asamblare a avut-o şi în deciziile de proiectare ale unor limbaje de nivel înalt, cum ar fi limbajul C spre exemplu, despre care ştim de asemenea că acceptă aritmetica de adrese şi că "numele unui tablou în C este de fapt adresa sa de început"! Această situaţie nu face decât să confirme că şi în limbajul C numele de variabile (tablouri cel puţin) reprezintă şi ele adrese de memorie la fel ca şi 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 reţinem că în limbajul de asamblare, valoarea asociată simbolului ce desemnează o variabilă (etichetă de date) este prin definiţie adresa sa. Totuşi, acest lucru nu înseamnă că referirea la numele variabilei nu va putea însemna niciodată accesul la conţinutul său! După cum am precizat şi exemplificat mai sus, această distincţie se face în funcţie de contextul utilizării numelui respectiv. În cadrul anumitor expresii şi instrucţiuni un acelaşi simbol va reprezenta ***adresa*** iar în altele va reprezenta ***conţinutul de la acea adresă***. **Înţelegerea deplină şi exactă a acestor diferenţe de interpretare reprezintă una dintre cheile aprofundării mecanismelor de execuţie la nivelul limbajului de asamblare.** Studiul utilizării instrucţiunilor (capitolul 4) şi experienţa de programare care va fi câştigată de programator cu această ocazie vor conduce cu siguranţă la o utilizare adecvată şi la interpretări corecte din partea programatorului relativ la problematica de mai sus.

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

Operanzii sunt parametri care definesc valorile ce vor fi prelucrate de instrucţiuni sau de directive. Ei pot fi regiştri, constante, etichete, expresii, cuvinte cheie sau alte simboluri. Semnificaţia operanzilor depinde de mnemonica instrucţiunii sau directivei asociate.

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

Parantezele drepte precizează că prezenţa elementului respectiv este opţională. Aşadar, putem avea linii cu toate cele patru elemente prezente, după cum putem avea şi linii fără nici un element (linii vide) sau linii formate cu doar unul dintre cele patru elemente.

3.2. EXPRESII

O *expresie* constă din mai mulţi operanzi care sunt combinaţi pentru a descrie o valoare sau o locaţie 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 excepţia acelor părţi care desemnează conţinuturi de regiştri şi care vor fi determinate la execuţie).

3.2.1. Moduri de adresare

Operanzii instrucţiunilor pot fi specificaţi în diferite forme, numite ***moduri de adresare***. Modurile de adresare indică procesorului modalitatea de a obţine valoarea reală a unui operand în momentul execuţiei.

Relativ la modurile de adresare, cele trei tipuri de operanzi sunt ***operanzi imediaţi, operanzi registru*** şi ***operanzi în memorie***. Valoarea operanzilor este calculată în momentul asamblării pentru operanzii imediaţi, în momentul încărcării programului pentru adresarea directă şi în momentul execuţiei pentru operanzii registru şi cei adresaţi indirect (vezi 2.6.7.).

Instrucţiunile care au doi sau mai mulţi operanzi operează întotdeauna de la dreapta spre stânga. Operandul din dreapta este operandul *sursă*. El specifică datele care vor fi folosite în operaţie, dar nu şi modificate. Operandul din stânga este operandul *destinaţie*. El specifică datele care vor fi folosite şi, probabil, modificate de către instrucţiune.

3.2.1.1. Utilizarea operanzilor imediaţi

Operanzii imediaţi sunt formaţi din date numerice constante cunoscute sau calculabile la momentul asamblării.

Constantele sunt utilizate ca operanzi în expresii. Limbajul de asamblare recunoaşte patru tipuri de valori constante: ***întregi, şiruri, numere reale şi constante împachetate codificate binar zecimal.***

Constantele întregi reprezintă valori întregi. Ele pot fi utilizate în declaraţii sau ca operanzi imediaţi.

Constantele întregi se specifică prin valori binare, octale, zecimale sau hexazecimale. Baza de numeraţie se dă printr-un specificator al bazei de numeraţie după ultima cifră a numărului, astfel: pentru numere binare specificatorul B, pentru cele octale -Q sau O, pentru numere zecimale - D şi pentru hexazecimale specificatorul H.

Pentru a nu fi tratate la asamblare drept simboluri, numerele hexazecimale trebuie să înceapă întotdeauna cu o cifră hexazecimală cuprinsă între 0 şi 9. De exemplu, OABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol. Cifrele hexazecimale de la A la F precum şi specificatorii bazei de numeraţie pot fi atât litere mari cât şi mici. Exemple: 123d (echivalent cu 123), 94fah, 63701o, 01101011b, 0FAB2h, etc.

Dacă nu se foloseşte nici un specificator asamblorul interpretează
întregul folosind baza de numeraţie implicită. Iniţial, baza de numeraţie implicită este cea zecimală. Ea poate fi modificată cu ajutorul directivei RADIX, care are următoarea sintaxă:

.RADIX *expresie*

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

O *constantă de tip şir* este formată din unul sau mai multe caractere ASCII delimitate de ghilimele sau de apostrofuri. Dacă printre aceste caractere ASCII trebuie să apară caracterul delimitator, acesta trebuie dublat. Exemple: 'a', "a", "Acesta este un mesaj", 'Ia vino' 'ncoa', "Ia vino 'ncoa", "Acest ""maestru"" este un farseur", 'Acest "maestru" este un farseur' etc.

Pentru unele instrucţiuni există o limită maximă în ceea ce priveşte dimensiunea de reprezentare a valorilor imediate (de obicei 8, 16 sau 32 biţi). Constantele şir mai lungi de două caractere (patru caractere la procesoarele 80386) nu pot fi date imediate. Ele trebuie să fie stocate în memorie înainte de a fi prelucrate de instrucţiuni.

Datele imediate nu sunt admise ca operand destinaţie (aşa cum nici 2:=N nu este o instrucţiune validă în nici un limbaj de programare de nivel înalt!).

Constantele întregi împachetate zecimal constituie un tip special de constante ce pot fi utilizate numai pentru iniţializarea variabilelor codificate binar zecimal (BCD). Cu *numere reale* se poate opera numai în prezenţa unui coprocesor matematic.

Deplasamentul unei etichete – valoare constantă determinabilă la momentul asamblării.

Este foarte important de semnalat faptul că pe lângă constantele numerice "clasice" echivalente ca semnificaţie celor din limbajele de nivel înalt, la momentul asamblării, asamblorul poate determina şi o altă categorie de valori care rămân constante pe tot parcursul execuţiei programului şi anume **deplasamentele etichetelor de date şi de cod**. De aceea, de exemplu, o instrucţiune de genul celei amintite mai sus:

```
lea ax, v        ; transfer în registrul ax a deplasamentului variabilei v
```

va putea fi evaluată la momentul asamblării drept de exemplu

```
lea ax, 0008 ; distanţă de 8 octeţi faţă de începutul segmentului de date
```

pe baza ordinii de declarare a variabilelor şi a tipului de dată (dimensiunea de reprezentare) specificat la declarare pentru variabila *v* – lucruri ce rămân constante relativ la execuţia programului şi astfel calculabile la momentul asamblării . În mod similar, o instrucţiune de tipul

```
jmp  et          ; salt în cadrul programului la o etichetă de cod et
```

va fi evaluată la momentul asamblării de exemplu sub forma

```
jmp [0004]↓
```

aceasta însemnând "salt cu 4 octeţi mai jos faţă de poziţia curentă".

Chiar dacă adresele fizice de început de segment pot fi cunoscute în mod logic numai *după* încărcarea programului în memorie, deplasamentele în cadrul acelor segmente ale etichetelor declarate acolo (de cod sau de date) sunt şi vor rămâne valori constante pe timpul execuţiei şi ce este şi mai important aceste valori sunt determinabile la momentul asamblării! Aceasta se întâmplă deoarece octeţii corespunzători segmentelor sunt generaţi la momentul asamblării (se poate din nou

sublinia în acest sens aici importanţa introducerii conceptului de *contor de locaţii* la nivelul limbajului de asamblare!). Semantic, "constanţa" acestor valori derivă din regulile de alocare adoptate de limbajele de programare în general şi care statuează că ordinea de alocare în memorie a variabilelor declarate (mai precis distanţa faţă de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanţele salturilor destinaţie în cazul unor instrucţiuni de tip goto sunt valori constante pe parcursul execuţiei unui program.

Adică: o variabilă odată alocată în cadrul unui segment de memorie nu îşi va schimba niciodată locul alocării (adică poziţia sa faţă de începutul acelui segment) iar această informaţie determinabilă la momentul asamblării derivă din ordinea specificării variabilelor la declarare în cadrul textului sursă şi din dimensiunea de reprezentare dedusă pe bază informaţiei 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 declaraţie de tipul **var a:real**; va provoca alocarea unei zone de 6 octeţi în cadrul segmentului în care apare o astfel de declaraţie, deoarece sizeof(real) = 6, iar o declaraţie 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 noţiunii 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 declaraţii 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 informaţiei de tip.

3.2.1.2. Utilizarea operanzilor registru

Regiştrii sunt probabil cei mai des folosiţi operanzi în cadrul instrucţiunilor limbajului de asamblare. Ei pot servi ca operanzi sursă sau ca operanzi destinaţie, putând conţine şi adrese de salt pentru instrucţiunile rezervate acestui scop. În plus, există unele instrucţiuni care pot fi folosite numai cu operanzi regiştri şi instrucţiuni care pot fi folosite numai cu anumiţi regiştri. Deseori, instrucţiunile au coduri mai scurte (şi operaţiile sunt mai rapide) dacă este specificat registrul acumulator (AX sau AL). Regiştrii microprocesorului 8086 au fost prezentaţi capitolul 2, iar instrucţiunile care îi vor utiliza ca operanzi vor fi prezentate în capitolul 4.

Operanzii-registru sunt formaţi din datele memorate în regiştri. Modul de *adresare directă* în cazul regiştrilor înseamnă folosirea valorii reale din interiorul registrului în momentul execuţiei instrucţiunii (de exemplu **mov ax,bx** – provoacă transferul valorii din registrul *bx* în registrul *ax*). De asemenea, regiştrii pot fi folosiţi *indirect* pentru a indica locaţiile de memorie (de exemplu instrucţiunea **mov ax,[bx]** va provoca transferul cuvântului de memorie de la adresa desemnată de *bx* spre registrul *ax*), aşa cum va fi prezentat în 3.2.1.4.

3.2.1.3. Utilizarea operanzilor din memorie

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

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

Observaţie. După cum rezultă şi din 2.6.5., nu sunt admise operaţiile pentru care atât sursa cât şi destinaţia sunt operanzi din memorie.

Operandul cu *adresare directă* este o constantă sau un simbol care reprezintă adresa (segment şi deplasament) unei instrucţiuni sau a unor date. Aceşti operanzi pot fi *etichete* (de ex: jmp et), *nume de proceduri* (de ex: call proc1) sau *valoarea contorului de locaţii* (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 execuţie (*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 şi 3.3.1.). Segmentul implicit poate fi înlocuit cu ajutorul operatorului de prefixare segment (notat ":" şi care se mai numeşte, 'operatorul de specificare a segmentului' - vom reveni în 3.2.2.6.).

Observaţie.
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 aceeaşi 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 şi nu valoarea pe care o indică. De exemplu, [100h] desemnează chiar valoarea 100h, şi nu valoarea de la adresa 100h.

3.2.1.4. Operanzi cu adresare indirectă

Operanzii cu *adresare indirectă* (vezi 2.6.7.) utilizează regiştri pentru a indica adrese din memorie. Deoarece valorile din regiştri se pot modifica la momentul execuţiei, adresarea indirectă este indicată pentru a opera în mod dinamic asupra datelor.

În cazul microprocesoarelor 8086 numai patru regiştri pot fi folosiţi în adresarea indirectă: BX, BP (*regiştri de bază*), DI şi SI (*regiştri index*). Orice tentativă de a folosi alţi regiştri, diferiţi de cei patru de mai sus, într-o instrucţiune care accesează memoria indirect, va produce o eroare.

Regiştrii de bază sau index pot fi folosiţi separat sau împreună, cu sau fără specificarea unui deplasament. Forma generală pentru accesarea indirectă a unui operand de memorie este dată de formula de calcul a offset-ului unui operand prezentată în 2.6.7., şi anume:

(1) [*registru_de_bază* + *registru_index* + *deplasament*]

unde toate cele trei componente sunt opţionale, însă trebuie să existe întotdeauna măcar o componentă.

Deplasament este o expresie a cărei valoare este determinabilă la momentul asamblării. De exemplu, [bx + di + table + 6] desemnează un operand prin adresare indirectă, unde atât *table* cât şi 6 sunt deplasamente. Asamblorul calculează deplasamentul real adunând *table* şi 6, pentru a obţine deplasamentul total.

Operanzii *registru_de_bază* şi *registru_index* sunt folosiţi de obicei pentru a indica o adresă de memorie referitoare la un tablou.

În ceea ce priveşte *regulile implicite de determinare a adresei de segment corespunzătoare unui deplasament specificat*, acestea sunt: dacă în expresia de calcul a deplasamentului este folosit ca registru de bază BX sau dacă nu este specificat nici un registru de bază, la calculul adresei efective a unui operand cu adresare indirectă, procesorul utilizează **DS** ca registru de segment implicit. Dacă BP este folosit oriunde în operand, registrul de segment implicit este SS. Segmentul implicit poate fi înlocuit prin operatorul de prefixare segment (:), aşa cum se prezintă în 3.2.2.6.

Pe lângă forma standard dată mai sus, diverse asambloare sau moduri de asamblare permit şi alte modalităţi de a specifica operanzi cu adresare indirectă. Orice operator care indică adunarea (+,[],.) poate fi folosit pentru a combina deplasamentul cu regiştri de bază sau index. De exemplu, următoarele moduri de specificare sunt toate echivalente:

table [bx] [di] + 6	[bx+6][di] + table
6 + table [bx+di]	table [di] [bx] + 6
[table+bx+di] + 6	bx + di + table[6]
[bx][di].table + 6	di + table + bx[6]

Când se utilizează modurile de adresare bază-index, unul dintre regiştri trebuie să fie registru de bază, iar celălalt trebuie să fie registru index. Următoarele instrucţiuni sunt incorecte:

```
mov ax, table [bx] [bp]        ;ilegal - doi regiştri de bază!
mov ax, table [di] [si]        ;ilegal - doi regiştri index!
```

Deci, să reţinem că pentru adresarea indirectă, esenţială este specificarea între paranteze drepte a cel puţin unuia dintre elementele componente ale formei (1).

3.2.2. Utilizarea operatorilor

Limbajul de asamblare oferă o gamă variată de operatori pentru combinarea, compararea, modificarea şi analiza operanzilor. Unii operatori lucrează cu constante întregi, alţii cu valori întregi memorate, iar alţii cu ambele tipuri de operanzi.

Este importantă înţelegerea diferenţei dintre operatori şi instrucţiuni. Operatorii efectuează calcule cu valori constante determinabile la momentul asamblării. Instrucţiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuţiei. De exemplu, operatorul de adunare (+) efectuează adunarea în momentul asamblării, în timp ce instrucţiunea ADD efectuează adunarea în timpul execuţiei.

Expresiile sunt evaluate conform următoarelor reguli:

- Operaţiile cu prioritatea cea mai mare sunt efectuate primele.
- Operaţiile cu aceeaşi prioritate se execută de la stânga la dreapta.
- Ordinea de prioritate poate fi modificată prin folosirea parantezelor. Operaţiile din paranteze se efectuează întotdeauna înaintea oricăror operaţii adiacente.

În tabelul de mai jos sunt prezentaţi în ordinea priorităţii operatorii ce pot fi folosiţi în cadrul expresiilor limbajului de asamblare 8086. Operatorii de pe aceeaşi linie au prioritate egală.

maximă	
1	**(), [], <>, LENGTH, SIZE, WIDTH, MASK**
2	**.(selector pentru membru al unei structuri)**
3	**HIGH,LOW**
4	**+,- (unar)**
5	**: (precizarea explicită a segmentului)**
6	**PTR, OFFSET, SEG, TYPE, THIS**
7	***, /, MOD, SHL, SHR**
8	**+,- (binar)**
9	**EQ, NE, LT, LE, GT, GE**
10	**NOT**

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

În continuare vom descrie unii dintre cei mai utilizaţi operatori în cadrul instrucţiunilor limbajului de asamblare 8086 şi vom da exemple de expresii formate cu aceşti operatori.

3.2.2.1. Operatori aritmetici

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

Pentru toţi operatorii aritmetici, cu excepţia operatorului de adunare (+) şi a celui de scădere (-), expresiile asupra cărora se efectuează operaţiile trebuie să fie constante întregi. Operatorii de adunare si scădere pot fi folosiţi pentru efectuarea operaţiilor de adunare şi scădere între o constantă întreagă şi 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 locaţii din interiorul aceluiaşi segment. În acest caz rezultatul va fi o constantă, reprezentând numărul de octeţi dintre cele două locaţii desemnate.

OPERATOR	SINTAXA	SEMNIFICAŢIE
+	+ expresie	pozitiv (unar)
-	- expresie	negativ (unar)
*	expresie1 * expresie2	înmulţire
/	expresie1 / expresie2	împărţire întreagă
MOD	expr1 MOD expr2	rest (modulo)
+	expresie1 + expresie2	adunare
-	expresie1 - expresie2	scădere

De exemplu, fie A şi B două etichete definite într-un acelaşi 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 şi A-7 pot fi folosiţi ca operanzi în memorie. Expresia B-A are valoarea 50h şi poate fi folosită ca şi 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. Restricţiile privind adunarea operanzilor păstraţi în memorie ce se aplică la operatorul de adunare sunt valabile şi pentru operatorul de indexare. De exemplu, nu se pot aduna doi operanzi în memorie adresaţi în mod direct. Expresia *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 adresaţi indirect. Paragraful 3.2.1 a clarificat rolul operatorului [] în adresarea indirectă.

3.2.2.3. Operatori de deplasare de biţi

Operatorii **SHR** (***SHift Right***) şi **SHL** (***SHift Left***) realizează deplasări ale expresiei operand (la dreapta pentru SHR şi respectiv la stânga pentru SHL) cu un număr de biţi egal cu valoarea celui de-al doilea operand. Biţii din dreapta (pentru SHL) şi cei din stânga (pentru SHR) sunt completaţi cu zerouri când conţinuturile lor sunt deplasate în afara poziţiilor limitrofe. Sintaxa instrucţiunilor este:

expresie **SHR** *cu_cât* şi *expresie* **SHL** *cu_cât*

expresie este deplasată la dreapta sau la stânga cu un număr *cu_cât* de biţi. Biţii deplasaţi dincolo de un capăt sau de celălalt al reprezentării expresiei sunt pierduţi. 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 direcţie opusă. Exemple (presupunem că expresia se reprezintă pe un octet):

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

Operatorii **SHR** şi **SHL** nu trebuie confundaţi cu instrucţiunile omonime ale procesorului (care vor fi tratate în capitolul următor). Operatorii acţionează asupra constantelor întregi numai la momentul asamblării. Instrucţiunile procesorului acţionează asupra valorilor păstrate în regiştri sau în memorie la momentul execuţiei. Asamblorul deosebeşte din context instrucţiunile SHR şi SHL de operatorii SHR şi respectiv SHL.

3.2.2.4. Operatori logici pe biţi

Operatorii pe biţi efectuează operaţii logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante. Tabelul următor conţine lista operatorilor logici şi semnificaţia acestora.

OPERATOR	SINTAXA	SEMNIFICAŢIE
NOT	NOT expresie	complementare biţi
AND	expr1 AND expr2	ŞI bit cu bit
OR	expr1 OR expr2	SAU bit cu bit
XOR	expr1 XOR expr2	SAU exclusiv bit cu bit

Exemple (presupunem că expresia se reprezintă pe un octet):

```
NOT  11110000b              ; desemnează valoarea 00001111b
01010101b AND 11110000b     ; are ca rezultat valoarea 01010000b
01010101b OR 11110000b      ; are ca rezultat valoarea 11110101b
01010101b XOR 11110000b     ; are ca rezultat valoarea 10100101b
```

3.2.2.5. Operatori relaţionali

Un operator relaţional compară (cu semn!) două expresii şi întoarce valoarea adevărat (-1) când condiţia specificată de operator este îndeplinită, sau valoarea fals (0) când nu este îndeplinită. Indiferent de dimensiunea de reprezentare, valoarea -1 se reprezintă în cod complementar ca un şir de biţi, având toţi valoarea 1. Expresiile evaluate au ca rezultat valori constante. Numele şi semantica acestor operatori sunt similare celor din FORTRAN. Ei sunt: **EQ** (*EQual*), **NE** (*Not Equal*), **LT** (*Less Than*), **LE** (*Less or Equal*), **GT** (*Greater Than*) şi **GE** (*Greater or Equal*). Exemple:

```
4 EQ 3  ; fals (0)          - 4 LT 3  ; adevărat (-1)
4 NE 3  ; adevărat (-1)     - 4 GT 3  ; fals (0)
```

3.2.2.6. Operatorul de specificare a segmentului

Operatorul de specificare a segmentului (:) comandă calcularea adresei FAR a unei variabile sau etichete în funcţie de un anumit segment. Sintaxa este:

segment:*expresie*

unde *segment* poate fi specificat în mai multe moduri. El poate fi unul dintre regiştri de segment: CS, DS, SS, sau ES (sau FS sau GS pe 80386). El poate fi de asemenea un nume de segment. În acest caz, numele trebuie să fie definit în prealabil cu o directivă **SEGMENT** (pe care o vom prezenta în 3.3.1.) şi eventual asociat unui registru de segment cu o directivă **ASSUME** (o vom prezenta tot în 3.3.1.). *Expresie* poate fi o constantă, o expresie sau o expresie **SEG** (vezi în continuare 3.2.2.7.). Exemple:

```
ss:[bx+4]    ; deplasamentul e relativ la SS
es:082h      ; deplasamentul e relativ la ES
```



```
date:var     ; adresa de segment este adresa de început a segmentului cu numele date, iar
             offsetul este valoarea etichetei var.
```

3.2.2.7. Operatori de tip

În acest paragraf sunt descrişi operatorii limbajului de asamblare care specifică sau analizează tipurile unor expresii şi a unor operanzi păstraţi în memorie.

Operatorul PTR

Operatorul **PTR** specifică tipul (înţeles în sensul dimensiunii de reprezentare) pentru o variabilă sau o etichetă de cod. Sintaxa este

tip **PTR** *expresie*

Operatorul forţează ca *expresie* să fie tratată ca având dimensiunea de reprezentare *tip*, fără însă a-i modifica definitiv (distructiv) valoarea în sensul precizat de conversia dorită. De aceea, operatorul PTR este considerat un *operator de conversie (temporară) nedistructivă*. Pentru operanzii păstraţi în memorie, *tip* poate fi **BYTE**, **WORD**, **DWORD**, **QWORD** sau **TBYTE**, având dimensiunile de reprezentare 1, 2, 4, 8 şi respectiv 10 biţi. Pentru etichetele de cod el poate fi **NEAR** (adresă pe 2 octeţi) sau **FAR** (adresă pe 4 octeţi).

Operatorul **PTR** este folosit pentru a da posibilitatea instrucţiunilor să-şi "vadă" operanzii ca fiind de tipul indicat în cazul în care aceştia au alt tip. De exemplu, operatorul PTR poate fi folosit pentru a accesa octetul cel mai semnificativ al unei variabile de dimensiune WORD. De asemenea, indiferent de semnificaţia lui A, expresia **byte ptr** A va indica doar primul octet de la adresa indicată de A. Analog, **dword ptr** A indică dublucuvântul ce începe la adresa A.

Operatorul THIS

Operatorul **THIS** creează un operand ale cărui valori de deplasament şi segment sunt egale cu valoarea curentă a contorului de locaţii şi al cărui tip este specificat de operator. Sintaxa lui este:

THIS *tip*

Pentru operanzii păstraţi în memorie, *tip* poate fi **BYTE, WORD, DWORD, QWORD** sau **TBYTE**. Pentru etichete, *tip* poate fi **NEAR** sau **FAR**. Operatorul **THIS** se foloseşte de obicei cu directivele **EQU** sau = pentru crearea unor etichete sau variabile. Rezultatul este similar cu cel obţinut prin utilizarea directivei **LABEL** (despre directive vom vorbi în 3.3).

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

Astfel de forme se utilizează de obicei pentru iniţializarea unor simboluri cu lungimea (dimensiunea) unui tablou. De exemplu:

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

Operatorii HIGH si LOW

Operatorii **HIGH** şi **LOW** întorc octetul cel mai semnificativ, respectiv cel mai puţin semnificativ, al unei expresii constante reprezentată pe cuvânt. Sintaxa lor este:

HIGH *expresie* şi **LOW** *expresie*

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

Operatorii SEG şi OFFSET

Sintaxele acestor doi operatori sunt:

SEG *expresie* şi **OFFSET** *expresie*

unde *expresie* adresează direct o locaţie de memorie.

Operatorul **SEG** întoarce adresa de segment a locaţiei de memorie referite. Valoarea întoarsă de operatorul **OFFSET** este o constantă reprezentând numărul de octeţi dintre începutul segmentului şi locaţia de memorie referită. Valorile întoarse de aceşti doi operatori sunt determinate la momentul încărcării programului, ele rămânând neschimbate pe parcursul execuţiei. 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 regiştrilor înseamnă folosirea valorii din regiştri, ca şi caz particular operatorii SEG şi OFFSET acceptă şi regiştri 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 reţinem 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 şi datele în momentul asamblării. Majoritatea directivelor au ca operanzi constante şir sau numerice şi 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 instrucţiuni, ai căror operanzi pot fi necunoscuţi în momentul asamblării şi pot varia în timpul execuţiei.

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 şi segmentul de date suplimentar. Aceste segmente logice pot să corespundă la patru segmente fizice distincte, dar pot să existe şi suprapuneri.

Există două tipuri de directive segment: directive segment *standard* (**SEGMENT, ENDS, ASSUME** şi **GROUP** - pe care le vom discuta în cele ce urmează) şi 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 sfârşitul segmentului este definit cu directiva **ENDS**. Sintaxa unei definiri de segment este următoarea:

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

[*instrucţiuni*]

nume **ENDS**

Numele segmentului este definit de eticheta *nume*. Acestui nume i se asociază ca valoare adresa de segment (16 biţi) corespunzătoare poziţiei segmentului în memorie în faza de execuţie (conform 3.1 – vezi şi discuţia de mai sus referitoare la operatorii SEG şi 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 fişier sursă numai dacă fiecare definiţie de segment ce utilizează acel nume va avea fie exact aceleaşi atribute, fie atribute care concordă.

Argumentele opţionale *aliniere, combinare, utilizare* şi *'clasa'* dau editorului de legături şi asamblorului indicaţii referitoare la modul de încărcare şi combinare a segmentelor. În absenţa unor argumente vor fi utilizate valori implicite.

Argumentul opţional *aliniere* specifică multiplul numărului de octeţi la care trebuie să înceapă segmentul respectiv. Alinierile posibile sunt următoarele:

- **BYTE** - multiplu de 1 (adresă de octet)
- **WORD** - multiplu de 2 (adresă de cuvânt)
- **DWORD** - multiplu de 4 (adresă de dublu cuvânt)
- **PARA** - multiplu de 16 (adresă de paragraf)
- **PAGE** - multiplu de 256 (adresă de pagină)

Dacă argumentul *aliniere* lipseşte, atunci se consideră implicit că este vorba despre o aliniere de tip PARA.

Argumentul opţional *combinare* controlează modul în care segmente cu acelaşi nume din cadrul altor module vor fi combinate cu segmentul în cauză la momentul editării de legături. Valorile posibile sunt:

- **PUBLIC** - indică editorului de legături să concateneze acest segment cu alte eventuale segmente cu acelaşi nume, obţinându-se un unic segment a cărei lungime este suma lungimilor segmentelor componente.

- **COMMON** - specifică faptul că începutul acestui segment trebuie să se suprapună peste începutul tuturor segmentelor ce au acelaşi nume. Se obţine un segment având dimensiunea egală cu cea a celui mai mare segment având acelaşi nume.

- **AT** *<expresie>* - impune ca segmentul să fie încărcat în memorie la adresa reprezentată de valoarea expresiei.

- **STACK** - segmentele cu acelaşi nume vor fi concatenate. În faza de execuţie segmentul rezultat va fi segmentul stivă.

- **MEMORY** - segmentele cu acest atribut vor fi aşezate în memorie în spaţiul disponibil rămas după încărcarea în memorie a segmentelor de argument combinare diferit de MEMORY.

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

Observaţii.

1. Pentru programele care se intenţionează a deveni programe **.COM** (vezi 5.5.3.) nu trebuie specificată alinierea.

2. În mod normal, într-un program se va furniza minimum un segment de stivă (cu ajutorul tipului de combinare STACK). Dacă nu este declarat nici un segment de stivă, atunci editorul de legături va considera spaţiu pentru stivă în continuarea unuia dintre segmentele existente.

Argumentul *utilizare* se foloseşte numai în cazul microprocesoarelor 80386 şi celor ulterioare pentru specificarea mărimii cuvântului.

Argumentul *'clasa'* are rolul de a permite stabilirea ordinii în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeaşi clasă vor fi plasate într-un bloc contiguu de memorie indiferent de ordinea lor în cadrul codului sursă. Numele claselor se vor da cu litere mari. Numele 'STACK' este rezervat pentru segment de stivă.

Directiva **GROUP** este utilizată pentru a combina două sau mai multe segmente într-o singură entitate logică, astfel încât toate segmentele componente să poată fi adresate relativ la un singur registru segment.

3.3.1.2. Directiva ASSUME şi gestiunea segmentelor

Directiva **ASSUME** stabileşte care sunt segmentele active la un moment dat. Ea are sintaxa generală:

ASSUME CS:*nume1*, SS:*nume2*, DS:*nume3*, ES:*nume4*

unde ordinea specificărilor de după ASSUME nu este importantă, oricare şi oricâte dintre acestea putând să lipsească. Fiecare dintre *nume1, nume2, nume3* şi *nume4* este un nume de segment sau cuvântul rezervat NOTHING.

Rolul directivei ASSUME este de a preciza asamblorului registrii de segment ce trebuie utilizaţi pentru calculul adreselor efective ale etichetelor şi ale variabilelor folosite în program. Existenţa acestei directive este necesară, deoarece un program poate fi alcătuit din mai multe segmente şi este necesar ca asamblorul să cunoască în fiecare moment care sunt segmentele active.

Prefixarea explicită a etichetelor şi variabilelor cu numele registrului de segment corespunzător (deci utilizarea operatorului de specificare a segmentului) furnizează în mod imediat această informaţie, având prioritate în cazul respectiv faţă de asocierea declarată prin directiva ASSUME.

Facem de la început precizarea că prezenţa acestei directive nu este necesară (având doar caracter de documentare) în cazul în care programul nu accesează etichete şi variabile (astfel de programe sunt însă extrem de rare). De asemenea, dacă operanzii în cadrul accesărilor indirecte nu fac referiri la etichete (de exemplu [bx+2] sau [bp+di]) atunci se foloseşte ca registru de segment SS dacă apare BP şi respectiv DS în celelalte cazuri. Aceste asocieri se fac automat, independent de ASSUME.

Este foarte important de reţinut faptul că rolul acestei directive nu este şi de a încărca registrii segment cu adresele corespunzătoare !

Unde şi cum intervine însă exact informaţia furnizată de către specificarea directivei ASSUME ?

Valoarea unei etichete este deplasamentul ei. Pentru completarea implicită de către asamblor a adresei sale de segment este nevoie de informaţia 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 definiţia etichetei, va identifica numele segmentului în care apare această definiţie de etichetă după care va recurge imediat la ASSUME pentru a vedea cu ce fel de registru de segment a fost asociat numele de segment identificat şi 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 înţelege pe deplin necesitatea furnizării de asocieri 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 şi astfel directiva ASSUME nu va mai fi aplicată şi 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:v*). Celelalte adrese (adică cele NEAR) trebuie să facă obiectul unor asocieri implicite cu adrese de segment (şi de aceea este nevoie de ASSUME) iar limbajul de asamblare specifică faptul că obiectul unor astfel de asocieri implicite pot fi numai regiştri 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 regiştri de segment, nu şi nume de alte segmente sau constante (chiar dacă acestea din urmă reprezintă modalităţi valide de specificare ale unor adrese FAR) !

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

```
ASSUME CS:c                 ;asociază registrul segment CS cu segmentul de cod c

c   segment
start: jmp far ptr etd      ;salt far necondiţionat la eticheta etd din cadrul segmentului de cod
                            ;d
       . . .                ;(aici nu e nevoie de ASSUME deoarece etd este forţată a fi
                            ;considerată FAR)
       etc:    jmp  x       ;salt near necondiţionat 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
;sfârşitul textului sursă.

d segment
       etd:    jmp y        ;salt near necondiţionat 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 necondiţionat la eticheta etc din cadrul segmentului de
                            ;cod c
d ends
end start
```

Observaţie. Dacă segmentul de cod nu conţine referiri la etichete locale, prezenţa unei directive ASSUME nu mai este obligatorie. Astfel, exemplul de mai jos conţine două segmente de cod, fiecare conţinând doar o instrucţiune: salt far necondiţionat 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 necondiţionat la eticheta etd din cadrul segmentului
                            ;de cod d
       etc:    . . .        ;nu se impune aici prezenţa vreunei directive ASSUME deoarece
                            ;eticheta locală etc este referită doar ca etichetă FAR din segmentul
                            ;d
c   ends

d   segment
       etd:    jmp far ptr etc  ;salt far necondiţionat la eticheta etc din cadrul segmentului de
                            ;cod c
               . . .        ;(aceeaşi observaţie ca mai sus şi pentru eticheta etd)
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, operaţiile la nivel de stivă se rezumă la introducerea sau scoaterea de cuvinte în şi respectiv din vârful stivei.

Directiva ASSUME pentru SS este necesară numai în cazul în care se defineşte un segment de stivă şi numai dacă există accesări de elemente ale stivei prin intermediul unor etichete definite în acest segment. De exemplu:

```
s segment stack 'STACK'
        . . .
        v db . . .
        . . .
s ends

ASSUME SS:s
c segment
        start:
        .       .       .       ; se poate accesa v din stivă
c ends
end start
```

În ceea ce priveşte 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 definită trebuie să facă obiectul unei asocieri prin directiva ASSUME cu registrul segment DS sau ES. Dacă se doreşte accesarea simultană a datelor din două segmente, unul dintre ele va fi asociat registrului DS (va fi segment principal de date) iar celălalt se va asocia cu ES (segment suplimentar de date). Dacă *nume3* sau *nume4* este NOTHING atunci asocierea respectivă este anulată. Se întâmplă frecvent ca DS şi ES să fie asociaţi la acelaşi segment de date.

În ceea ce priveşte încărcarea regiştrilor de segment, precizăm următoarele:

- registrul CS este încărcat automat
- registrul SS este deasemenea încărcat automat. Dacă programatorul doreşte să schimbe segmentul de stivă, atunci el trebuie să încarce noua valoare în SS (dăm mai jos un exemplu)
- regiştrii DS şi ES, în cadrul programelor **.EXE** (vezi 5.5.2.) trebuie încărcaţi de către programator.

Anticipând instrucţiunea MOV (care va fi prezentată în 4.1.1) dăm mai jos un exemplu de accesare a datelor din două segmente de date. Să facem precizarea că regiştrii de segment nu pot fi încărcaţi direct, ci numai printr-un intermediar (registru sau stivă). În cazul nostru, intermediar este registrul AX:

```
d1      segment
        a1      db      ...     ;definire a1
d1      ends
```



```
d2      segment
        a2      db      ...     ;definire a2
d2      ends

ASSUME      CS:c, DS:d1, ES:d2

c       segment

start:  mov ax,d1       ;încărcarea registrului segmentului
        mov ds,ax       ;principal de date
        mov ax,d2       ;încărcarea registrului segmentului
        mov es,ax       ;de date suplimentar

        mov al,a1       ;accesare a1 relativ la DS (adică mov al, DS:a1, deoarece segmentul d1 în
                        ;cadrul căruia apare eticheta a1 a fost asociat prin ASSUME cu reg. DS)
        mov ah,a2       ;accesare a2 relativ la ES (adică mov ah, ES:a2, deoarece segmentul d2 în
                        ;cadrul căruia apare eticheta a2 a fost asociat prin ASSUME cu registrul ES)

;dacă nu se specifică directiva ASSUME, la întâlnirea celor două referiri de etichete (deplasamente)
;de mai sus, asamblorul va furniza mesajul de eroare sintactică:
;                       "Can't address with currently ASSUMEd registers".
c       ends

        end start
```

Pe de altă parte, chiar dacă programatorul înţelege necesitatea utilizării unei directive ASSUME pentru definitivarea adreselor de accesare a etichetelor definite în program şi evită astfel o eroare de sintaxă de genul celei de mai sus, utilizarea directivei ASSUME se poate constitui şi într-o potenţială sursă de erori logice! Acest lucru se întâmplă atunci când deşi este prezentă o directivă ASSUME, specificările asocierilor între regiştri de segment şi numele de segmente din program nu sunt urmate şi de o încărcare corespunzătoare cu adresele acestor segmente a regiştrilor de segment asociaţi prin ASSUME. Să luăm ca exemplu o variantă a codului sursă anterior, în care vom inversa asocierile specificate prin ASSUME, însă nu vom şi actualiza corespunzător acţiunile de încărcare ale regiştrilor DS şi ES:

```
d1      segment
        ... a1  db      ...     ;definire a1
d1      ends

d2      segment
        ... a2  db      ...     ;definire a2
d2      ends
```

```
ASSUME    CS:c, DS:d2, ES:d1  ;aici am inversat asocierile iniţiale pentru DS şi ES!

c         segment

          start: mov ax,d1    ;încărcarea registrului DS cu adresa segmentului de date d1
                 mov ds,ax    ;deşi ASSUME a fost modificată, acţiunile de încărcare ale
                              ;regiştrilor DS şi 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 şi nu cu adresa
;de început a segmentului d1 unde se află a1, deci în AL se va încărca conţinutul locaţiei de
;memorie de deplasament a1 din cadrul segmentului d2 ceea ce va reprezenta o eroare
;logică, deoarece a1 trebuie raportat la segmentul d1 şi 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 şi nu cu adresa
;de început a segmentului d2 unde se află a2, deci în AH se va încărca conţinutul locaţiei de
;memorie de deplasament a2 din cadrul segmentului d1 ceea ce va reprezenta o eroare
;logică, deoarece a2 trebuie raportat la segmentul d2 şi 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 şi acţiunile de încărcare cu adresele corespunzătoare de început de segment ale regiştrilor DS şi 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 şi aici adaptarea la asocierea furnizată prin ASSUME !
```

În final dăm un exemplu în care se lucrează alternativ cu două stive. Stiva iniţială 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 spaţiu stivă
ns     ends
vs     segment stack 'STACK'
       .     .    .    ;rezervare spaţiu 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 mulţi regiştri segment nu pointează spre nici un segment se foloseşte cuvântul rezervat **NOTHING**.

Orice program scris în limbaj de asamblare trebuie să conţină directiva **END** pentru marcarea sfârşitului 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 opţional indicând adresa din program de unde va începe execuţia. Într-un program care constă dintr-un singur modul (fişier sursă) specificarea adresei de start în cadrul directivei END este obligatorie. Şi pentru un program constituit din mai

multe module fiecare modul are în final o directivă END. Însă numai directiva END a modulului ce conţine instrucţiunea de la care programul trebuie să-şi înceapă execuţia va conţine specificarea adresei de start. Semnificaţia acestor reguli este clară şi naturală - fiecare program trebuie să aibă neapărat un început, şi numai unul, indiferent de numărul modulelor ce îl compun. Eventualele excepţii de la aceste reguli pot apare la nivelul legării cu module scrise în alte limbaje de programare. Precizările necesare în acest sens se vor aduce în capitolul 8.

3.3.2. Directive pentru definirea datelor

În contextul general al limbajelor de programare *definirea* unei date înseamnă specificarea atributelor acesteia şi alocarea spaţiului de memorie necesar. Pe scurt deci, *definire* date = *declarare* (specificarea atributelor) + *alocare* (rezervarea spaţiului de memorie necesar).

În cadrul limbajului de asamblare, definirea unei date se realizează prin utilizarea directivelor de definire a datelor. Acestea au ca sarcină declararea datelor (principalul atribut specificat cu această ocazie este *tipul de dată* = *dimensiunea de reprezentare* – octet, cuvânt sau dublucuvânt) precum şi alocarea acestora în cadrul segmentului în care datele au fost declarate (datorită acestei alocări deplasamentul unei date în cadrul segmentului este o constantă determinabilă de la momentul asamblării). Să nu uităm că asamblorul are ca sarcină generarea de octeţi corespunzătoare directivelor şi instrucţiunilor întâlnite (reamintim în acest context rolul *contorului de locaţii* - $). Ca urmare, asamblorul va genera şi octeţii ce corespund directivelor de definire a datelor întâlnite realizând în acest fel de fapt acţiunea de alocare a datelor în cadrul segmentului respectiv.

În acelaşi timp, pentru datele alocate se pot specifica valorile iniţiale. Datele pot fi specificate ca numere, şiruri de caractere sau expresii evaluate ca fiind constante. Asamblorul transformă aceste valori constante în octeţi, cuvinte sau alte unităţi de date. Datele codificate sunt scrise în fişierul obiect în momentul asamblării.

Forma generală a unei linii sursă în cazul unei declaraţii de date este:

[*nume*] *tip_data lista_expresii* [;*comentariu*]

sau

[*nume*] *tip_data factor* **DUP** (*lista_expresii*) [;*comentariu*]

unde *nume* este o etichetă prin care va fi referită data. Acest nume are asociat un tip şi o valoare. Tipul rezultă din tipul datei (dimensiunea de reprezentare) iar valoarea este adresa la care se va găsi în memorie primul octet rezervat pentru data etichetată cu numele respectiv.

Elementul *lista_expresii* reprezintă lista unor expresii cu ale căror valori se vor iniţializa zonele de date rezervate prin declaraţia respectivă. Dacă apare numai caracterul '?' atunci zona de date corespunzătoare va fi numai rezervată, nu şi iniţializată.

factor este un număr care indică de câte ori se repetă lista de expresii care urmează în paranteză.

Tip_data este o directivă de definire a datelor, una din următoarele:

DB - date de tip octet (BYTE)
DW - date de tip cuvânt (WORD)
DD - date de tip dublucuvânt (pointer - DWORD)
DQ - date de tip 8 octeţi (QWORD - utilizate pentru memorarea constantelor reale)
DT - date de tip 10 octeţi (TWORD - utilizate pentru memorarea constantelor BCD)

(de la microprocesorul 80386 începând există în plus două directive - **DF** şi **DP** - ce definesc pointeri far pe 6 octeţi).

De exemplu, secvenţa de mai jos defineşte şi iniţializează cinci variabile de memorie:

```
data  segment
      varb  DB   'd'   ;1 octet
      varw  DW   101b  ;2 octeţi
      vard  DD   2bfh  ;4 octeţi
      varq  DQ   307o  ;8 octeţi (1 quadword)
      vart  DT   100   ;10 octeţi
data ends
```

După o directivă de definire a datelor pot să apară mai multe valori, permiţându-se astfel declararea şi iniţializarea de tablouri. De exemplu, declaraţia

```
Tablou      DW    1,2,3,4,5
```

crează un tablou de 5 întregi reprezentaţi pe cuvinte având valorile respectiv 1,2,3,4,5. Dacă valorile de după directivă nu încap pe o singură linie se pot adăuga oricâte linii este necesar, linii ce vor conţine numai directiva şi valorile dorite. Exemplu:

```
Tabpatrate  DD    0, 1, 4, 9, 16, 25, 36
            DD    49, 64, 81
            DD    100, 121, 144, 169
```

Operatorul **DUP** se foloseşte pentru definirea unor blocuri de memorie iniţializate repetitiv cu o anumită valoare. De exemplu

```
Tabzero     DW    100h   DUP (0)
```

rezervă 256 de cuvinte pentru tabloul *Tabzero* iniţializându-le cu 0, iar

```
Tabchar     DB    80 DUP ('a')
```

crează un tablou de 80 de octeţi iniţializaţi fiecare cu codul ASCII al caracterului 'a'.

Dacă se doreşte doar rezervarea de spaţiu de memorie pentru tablouri, fără iniţializarea acestora cu anumite valori se va folosi caracterul '?'. Într-o astfel de situaţie declaraţiile de mai sus ar deveni

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

Dacă dorim definirea de şiruri de caractere este suficient să ţinem cont că ele sunt de fapt ca reprezentare internă şiruri de octeţi, asamblorul considerând echivalente declaraţii ca:

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

Valoarea de iniţializare poate fi şi o expresie, ca de exemplu: `vartest  DW  (1002/4+1)`

Valoarea curentă a contorului de locaţii poate să fie referită cu ajutorul simbolului **$** sau al expresiei **this** <*tip*>. Ca urmare, putem avea următoarele secvenţe 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 şi 3.3.3) faţă de primele 3 nu va şi genera spaţiu 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 conţine declaraţii şi iniţializări de date şi care este edificator în ceea ce priveşte posibilităţile de combinare a facilităţilor 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   0ffddh, 0800h SHR 2
      a5   DW   4 DUP ('13')
      a6   DW   a4
      a7   DD   a4
      .    .    .
data  ends
```


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

3.3.3. Directivele LABEL, EQU, =

Directiva **LABEL** permite numirea unei locaţii fără alocarea de spaţiu de memorie sau generare de octeţi, precum şi 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 şi dacă acesta se va referi la cod sau la date.

Numele primeşte ca valoare contorul de locaţii. *Tip* poate să fie una din următoarele:

BYTE	**NEAR**	**FAR**
WORD	**QWORD**	**PROC**
DWORD	**TBYTE**	**UNKNOWN**

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

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

Tipul **PROC** (existent numai în varianta TASM) va furniza un tip **NEAR** sau **FAR** în funcţie de modelul de memorie folosit în cazul utilizării directivelor segment simplificate.

Tipul **UNKNOWN** declară un tip necunoscut şi este folosit atunci când se doreşte să existe posibilitatea accesării unei variabile de memorie în mai multe moduri (se aseamănă cu tipul **void** din limbajul C). De exemplu, accesarea variabilei *tempvar* din secvenţa de mai jos uneori ca octet şi alteori ca şi cuvânt poate fi realizată prin declararea ei ca etichetă de tip UNKNOWN:

```
data segment                    code segment
      tempvar LABEL UNKNOWN     .     .     .
              DB ?,?            mov  tempvar,ax    ;utilizare ca şi cuvânt
      .     .     .             .     .     .
data ends                       add  dl,tempvar    ;utilizare ca şi octet
                                .     .     .
```

Reamintim că o altă soluţie pentru adresarea unei date cu un alt tip decât cel cu care a fost declarată este utilizarea operatorului de conversie PTR, prezentat în 3.2.2.7.

Directiva **EQU** permite atribuirea, în faza de asamblare, unei valori numerice sau şir de caractere unei etichete fără alocarea de spaţiu de memorie sau generare de octeţi. Sintaxa directivei EQU este

nume **EQU** *expresie*

unde numelui îi este atribuită valoarea numerică a expresiei. Simbolul *nume* trebuie să fie un simbol neutilizat anterior sau utilizat anterior tot într-o directivă EQU. Dacă simbolul a fost utilizat anterior, el trebuie să fi avut ca rezultat al evaluării expresiei un şir de caractere. Cu alte cuvinte nu se admit redefiniri de etichete decât în cazul echivalării lor iniţiale cu şiruri de caractere. Rolul directivei EQU este similar definirii de constante simbolice din cadrul unui limbaj de nivel înalt.

Exemple:

```
END_OF_DATA     EQU  '!'
BUFFER_SIZE     EQU  1000h
INDEX_START          EQU  (1000/4 + 2)
VAR_CICLARE          EQU  i
```

Prin utilizarea de astfel de echivalări textul sursă poate deveni mai lizibil.

Se observă asemănarea etichetelor echivalate prin directiva EQU cu constantele din limbajele de programare de nivel înalt.

Expresia pentru echivalarea unei etichete definite prin directiva EQU poate conţine la rândul ei etichete definite prin EQU:


```
TABLE_OFFSET     EQU  1000h
INDEX_START      EQU  (TABLE_OFFSET + 2)
DICTIONAR_STAR   EQU  (TABLE_OFFSET + 100h)
```

Directiva = este echivalentă directivei **EQU** cu excepţia următoarelor aspecte:

- etichetele definite cu EQU nu pot fi redefinite în timp ce acelea definite cu = pot.
- operanzii expresiei directivei = trebuie să furnizeze în final o valoare numerică, nefiind permisă asignarea de şiruri de caractere etichetelor.

3.3.4. Directiva PROC

Noţiunea de subrutină este cunoscută din studiul limbajelor de programare de nivel înalt. Ea oferă posibilitatea dezvoltării modulare a programelor, permiţând concentrarea atenţiei asupra punctelor esenţiale ale unui program, ignorându-se în acele locuri detaliile de implementare. De asemenea, prezenţa subrutinelor aduce avantajul reutilizării codului (deci obţinerea în final a unui cod cât mai compact), o subrutină fiind scrisă o singură dată, putând fi însă apelată de oricâte ori şi în orice punct al programului cu valori diferite ale eventualilor parametri.

În limbajul de asamblare 8086 definirea unei subrutine începe cu o directivă **PROC**:

<*nume_procedură*> **PROC** [*tip_apel*]

unde *nume_procedură* este o etichetă reprezentând numele procedurii, iar *tip_apel* este **NEAR** sau **FAR**. Dacă lipseşte, atunci se consideră implicit NEAR. Procedura va fi NEAR dacă va fi apelată numai în cadrul segmentului de cod în care este definită. Procedura va fi FAR dacă va fi apelată şi din alte segmente de cod.

Directiva **ENDP** marchează sfârşitul unei subrutine ce începe cu **PROC**. Sintaxa ei este

<*nume_procedură*> **ENDP**

Între cele două directive se vor scrie instrucţiunile şi directivele corpului procedurii. Problematica subrutinelor este reluată în capitolul 4 în cadrul prezentării instrucţiunilor de apel (CALL) şi respectiv de revenire (RET) dintr-o subrutină (4.3.4).

3.3.5. Blocuri repetitive

Un *bloc repetitiv* este o construcţie prin care i se cere asamblorului să genereze în mod repetat o configuraţie de octeţi. Prezentăm în continuare trei tipuri de blocuri repetitive: **REPT**, **IRP** şi **IRPC**.

Un bloc repetitiv delimitat de directivele **REPT** şi **ENDM** are următoarea sintaxă de definire:

REPT *contor*
secvenţă
ENDM

cu semnificaţia că *secvenţa* va fi asamblată de *contor* ori. De exemplu secvenţele

```
REPT 5
     dw   0
ENDM
```

şi

```
dw 0
dw 0
dw 0
dw 0
dw 0
```

generează acelaşi cod, lucru ce se poate realiza bineînţeles şi cu: `dw 5 DUP (0)`

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

```
Intval = 0
REPT  10
      dw     Intval
      Intval = Intval + 1
ENDM
```

care va genera

```
dw 0
dw 1
dw 2
dw 3
dw 4
```

De asemenea, blocurile repetitive pot fi imbricate. Secvenţa

```
REPT 5
REPT 2
   secvenţă
ENDM
ENDM
```

generează de 10 ori secvenţa specificată.

Directiva **IRP** are sintaxa

IRP *parametru*, *<arg1 [,arg2]...>*
secvenţă
ENDM

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


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

generează secvenţa

```
db   0
db   1
db   4
db   9
db   16
db   25
```

iar

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

generează secvenţa

```
mov  ax,di
mov  bx,di
mov  cx,di
mov  dx,di
```

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

IRPC *parametru,string*
secvenţă
ENDM

De exemplu

```
IRPC   nr,1375
       db nr
ENDM
```

crează 4 octeţi având respectiv valorile 1, 3, 7 şi 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 fişierul *numefisier* în textul sursă curent. Inserarea se face în locul în care apare directiva INCLUDE respectivă. *numefisier* este specificarea unui nume de fişier DOS, putând aşadar să conţină specificări de unitate de disc, directoare, nume de fişier şi tip. În lipsa specificării unui tip se consideră implicit .ASM. De exemplu, dacă fişierul *prog.asm* conţine codul

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

iar fişierul *instr2.asm* conţine codul

```
        mov   bx,3
        add   ax,bx
        dec   bx
```

rezultatul asamblării fişierului *prog.asm* va fi echivalent cu cel al asamblării codului

```
cod     segment
        mov   ax,1
        mov   bx,3
        add   ax,bx
        dec   bx
        push  ax
        .     .     .
```

Fişierele incluse pot conţine la rândul lor alte directive INCLUDE, ş.a.m.d. până la orice nivel, instrucţiunile respective fiind inserate corespunzător pentru crearea în final a unui singur cod sursă.

3.3.7. Macrouri

Un ***macro*** este un text parametrizat căruia i se atribuie un nume. La fiecare întâlnire a numelui, asamblorul pune în codul sursă textul cu parametrii actualizaţi. Operaţia este cunoscută şi sub numele de *expandarea* macroului. Se poate face o analogie cu directiva INCLUDE prezentată anterior. Faţă de fişierele incluse, macrourile prezintă un grad sporit de flexibilitate permiţând transmiterea de parametri şi existenţa de etichete locale.

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

nume **MACRO** [*parametru* [,*parametru*]...]
    corp instrucţiuni
ENDM

De exemplu, pentru interschimbarea valorilor a două variabile cuvânt putem defini următorul macro:

```
swap  MACRO  a,b
      mov    ax,a
      mov    a,b
      mov    b,ax
ENDM
```

Pentru a creşte consistenţa exemplelor de mai jos vom utiliza şi instrucţiuni care vor fi prezentate abia în următorul capitol. Oricum, în contextul acestei secţiuni nu este esenţială semnificaţia lor precisă. Dorim doar promovarea unei baze de discuţie asupra structurii programelor pentru a putea scoate în evidenţă semnificaţia elementelor prezentate aici.

Pentru înmulţirea cu 4 a valorii unei variabile (rezultatul depunându-se în DX:AX) se poate scrie următorul macro:

```
inmcu4 MACRO  a
       mov    ax,a
       sub    dx,dx
       shl    ax,1
       rcl    dx,1
       shl    ax,1
       rcl    dx,1
ENDM
```

O utilizare a sa sub forma **inmcu4 varm** va genera secvenţa

```
       mov    ax,varm
       sub    dx,dx
       shl    ax,1
       rcl    dx,1
       shl    ax,1
       rcl    dx,1
```

Macrourile pot conţine blocuri repetitive. În acest sens putem lua ca exemplu chiar macroul **inmcu4** de mai sus, care se poate rescrie astfel:

```
inmcu4 MACRO  a
       mov    ax,a
       sub    dx,dx
       REPT   2
       shl    ax,1
       rcl    dx,1
       ENDM
ENDM
```

O posibilă problemă ce apare se referă la definirea unei etichete într-un macro. Să presupunem că într-un macro se defineşte o etichetă şi în program există mai mult de un apel al acelui macro. Eticheta definită va apărea la fiecare expandare a macroului în codul programului, cauzând o eroare de "redefinire de etichetă". Exemplu:

```
scade  MACRO
       jcxz   Etich
       dec    cx
Etich:
ENDM
.      .      .
```

```
scade           ;apare eticheta Etich
.     .     .
scade           ;şi aici apare Etich!
.     .     .
```

Soluţia unei astfel de probleme este oferită de către *directiva* **LOCAL**, care, la apariţia ei în cadrul unui macro forţează ca domeniul de vizibilitate al etichetelor specificate ca argumente să fie numai acel macro. Soluţia 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 referinţele 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 amestecaţi cu alt text. De exemplu, să presupunem că dorim să scriem un macro care să realizeze depunerea în stivă a conţinutului unuia din cei patru regiştri generali (AX, BX, CX sau DX) în funcţie 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 şirul operand al lui push este de fapt parametrul formal al macroului şi va considera că este vorba de operandul *rliterax*.

Soluţia este oferită de asamblor, care permite încadrarea în cadrul corpului macroului a numelui parametrului formal într-o pereche de caractere **&** (ampersand, numit operatorul de substituţie). 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 substituţie **&** poate fi utilizat şi în cadrul directivelor IRP sau IRPC. De exemplu:

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

După cum am văzut, macrourile pot conţine 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** conţine un bloc repetitiv, care la rândul lui conţine o invocare a macroului **push_reg**.

CAPITOLUL 4

INSTRUCŢIUNI ALE LIMBAJULUI DE ASAMBLARE

Pentru ca cititorul să poată experimenta instrucţiunile prezentate în acest capitol, prezentăm pentru început forma generală (sau cel puţin cea mai des uzitată) a unui program ASM:

```
assume  cs:code, ds:data  ;rolul directivei ASSUME este detaliat în 3.3.1.2.

data segment              ;data este aici numele segmentului de date – numele poate fi modificat
        ...       ;definiţii de date utilizând directivele de definire a datelor (3.3.2.) – DB, DW, DD.
data ends

code segment              ;code este aici numele segmentului de cod – numele poate fi modificat
start:  mov ax,data
        mov ds,ax         ;încărcarea registrului segment DS cu adresa segmentului de date
        ...
        mov ah,4ch
        int 21h           ;apelarea funcţiei 4ch a întreruperii 21h pentru a provoca încheierea
                          ;execuţiei programului curent
code ends
end start                 ;precizarea sfârşitului textului sursă (directiva END) împreună cu
                          ;definirea punctului de intrare în program – eticheta de cod "start"
```

Pentru testarea unor programe foarte simple, există posibilitatea de a elabora programe care conţin numai câte un segment de date şi unul de cod, având nume predefinite (neapărat *data* şi *code*). În acest scop se pot folosi directivele simplificate (caracteristică TASM). Un astfel de program are structura:

```
.model    small
.data
          .    .    .    ;conţinutul segmentului de date
.code
Start:
          mov  ax,@Data
          mov  ds,ax
          .    .    .    ;conţinutul segmentului de cod
          mov ah,4ch
          int 21h         ;apelarea funcţiei 4ch a întreruperii 21h
end       Start
```

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

nume.ASM

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

...>TASM *nume*
...>TLINK *nume*
...>TD *nume*

realizează asamblarea, editarea de legături şi execuţia asistată a programului respectiv.

4.1. MANIPULAREA DATELOR

Pentru o mai uşoară referire ulterioară de către cititor a instrucţiunilor şi operaţiilor prezentate în acest capitol vom prezenta la începutul fiecărei secţiuni câte un tabel cu trei rubrici ele reprezentând în ordine *forma generală (sintaxa), efectul* şi respectiv *flagurile afectate*. Notaţia <*op*> desemnează conţinutul operandului *op*. Simbolurile *d* şi *s* vor desemna operandul destinaţie şi respectiv operandul sursă al instrucţiunii respective.

4.1.1. Instrucţiuni de transfer al informaţiei

4.1.1.1. Instrucţiuni de transfer de uz general

MOV *d,s*	<d> <-- <s>	-
PUSH *s*	depune <s> în stivă	-
POP *d*	extrage elementul curent din stivă şi îl depune în d	-
XCHG *d,s*	<d> <--> <s>	-
XLAT [*tabelă_de_translatare*]	AL <-- < seg:[BX+<AL>] >	-

Cea mai importantă instrucţiune de transfer de informaţie este **MOV**. Forma generală este

MOV *destinaţie, sursă*

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

Efectul instrucţiunii este copierea valorii operandului sursă în operandul destinaţie cu păstrarea valorii din operandul sursă. De exemplu, secvenţa

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

depune întâi în registrul AX constanta 0, apoi reţine constanta 7 în BX, după care copiază conţinutul registrului BX în AX. În final, regiştrii AX şi BX vor conţine amândoi aceeaşi valoare 7.

Dacă destinaţia este unul dintre cei patru regiştri segment atunci sursa trebuie să fie unul dintre cei opt regiştri sau o variabilă de memorie. Cum numele segmentelor sunt valori constante (adresele de început ale acelor segmente), ele trebuie încărcate în regiştrii segment corespunzători prin intermediul unui registru general sau unei locaţii de memorie. De exemplu, cum am mai arătat şi în 3.3.1.2., secvenţa 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 doreşte de fapt dar nu este permis în mod direct este: *mov es, data.*

Instrucţiunea **MOV** poate fi folosită şi pentru accesarea informaţiei din stivă, prin intermediul modului de adresare ce utilizează BP ca registru de bază. De exemplu, instrucţiunea

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

încarcă AX cu conţinutul cuvântului aflat la deplasamentul BP+4 în cadrul stivei. Pentru accesarea conţinutului stivei se folosesc de regulă alte două instrucţiuni şi anume PUSH şi POP.

Instrucţiunile **PUSH** şi **POP** au sintaxa

PUSH *s* şi **POP** *d*

Operanzii trebuie să fie reprezentaţi pe cuvânt, deoarece stiva este organizată pe cuvinte. Stiva creşte de la adrese mari spre adrese mici, din doi în doi octeţi, SP punctând întotdeauna spre cuvântul din vârful stivei. Instrucţiunea **PUSH** depune operandul (sursă) *s* în vârful stivei (implicit operandul destinaţie), decrementând întâi valoarea din registrul SP, iar instrucţiunea **POP** extrage valoarea din vârful stivei (implicit operandul sursă aici) şi o depune în operandul (destinaţie) *d*, incrementând ulterior valoarea din registrul SP.

De exemplu, să presupunem că SP conţine iniţial valoarea 1000h şi să urmărim evoluţia conţinutului stivei precum şi a regiştrilor AX, BX şi SP pe parcursul execuţiei secvenţei

```
mov ax,1
push ax
mov bx,2
push bx
pop ax
pop bx
```

La început avem situaţia

Regiştri		Stiva
AX	?	996: ?
BX	?	998: ?
SP	1000	1000: ?

După mov ax,1 şi push ax avem

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

După mov bx,2 şi push bx vom avea

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

După execuţia instrucţiunii pop ax vom avea

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

iar după execuţia instrucţiunii pop bx vom obţine configuraţia

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

Încărcarea unui registru de segment se poate face şi prin intermediul instrucţiunilor PUSH şi POP. De exemplu, secvenţa de mai sus de încărcare a registrului ES poate fi înlocuită cu


```
push data
pop es
```

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

Instrucţiunea **XCHG** permite interschimbarea conţinutului a doi operanzi de aceeaşi dimensiune (octet sau cuvânt), cel puţin unul dintre ei trebuind să fie registru. Sintaxa ei este

XCHG *operand1, operand2*

Această instrucţiune oferă o modalitate directă de a efectua o acţiune pentru care altfel s-ar impune minim trei instrucţiuni. De exemplu

```
xchg ax,bx
```

schimbă conţinutul regiştrilor AX şi BX, operaţie echivalentă cu

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

iar

```
xchg al,varmem (sau xchg varmem,al)
```

schimbă conţinutul registrului AL cu cel al variabilei octet *varmem*.

Instrucţiunea **XLAT** "traduce" octetul din AL într-un alt octet, utilizând în acest scop o tabelă de corespondenţă creată de utilizator, numită *tabelă de translatare*. Instrucţiunea are sintaxa

XLAT [*tabelă_de_translatare*]

tabelă_de_translatare este adresa directă a unui şir de octeţi. Instrucţiunea pretinde la intrare adresa far a tabelei de translatare furnizată sub unul din următoarele două moduri:

- DS:BX (implicit, dacă lipseşte operandul)
- registru_segment:BX, dacă operandul instrucţiunii XLAT este prezent, registrul segment fiind determinat pe baza directivei ASSUME corespunzătoare operandului.

Efectul instrucţiunii **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, secvenţa

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

depune conţinutul celei de-a 7-a locaţii de memorie (de index 6) din *Tabela* în AL.

Dăm un exemplu de secvenţă care translatează o valoare zecimală 'numar' cuprinsă între 0 şi 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 şi se dovedeşte 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. Instrucţiuni de transfer de intrare-ieşire

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

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

Instrucţiunea **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.

Instrucţiunea **OUT** este complementara instrucţiunii **IN**, realizându-se un transfer de informaţie dinspre acumulator spre un port de ieşire. De exemplu, secvenţa

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

scrie valoarea 63 la portul I/O 500.

4.1.1.3. Instrucţiuni de transfer al adreselor

Aceste instrucţiuni sunt deosebit de utile la operaţiile cu şiruri, la transmiterea de parametri către proceduri etc. În tabelul de mai jos *reg* desemnează un registru general (deci orice registru diferit de regiştri segment).

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

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

```
lea     ax,v
```

încarcă în AX offsetul variabilei v, instrucţiune echivalentă (cu excepţia situaţiilor prezentate mai jos) cu

```
mov     ax, offset v
```

Faţă de această ultimă variantă, instrucţiunea **LEA** are avantajul că operandul sursă poate fi indexat. De exemplu, instrucţiunea

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

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

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

De exemplu, presupunând că variabila *varp* conţine valoarea 1234h:5678h (posibil a fi interpretată ulterior ca adresa far a unei zone de memorie), putem obţine primul octet al acestei zone în AL prin secvenţa

```
.       .       .
varp    dd      12345678h
.       .       .
lds     bx,varp         ;se transferă conţinutul variabilei varp în DS:BX în
                        ;DS vom avea 1234h iar în BX vom avea 5678h
mov     al,[bx]         ;se transferă în AL octetul de la adresa DS:[BX],
                        ;adică octetul de la adresa 1234h:5678h
```

Să reţinem că instrucţiunea LDS a asigurat doar transferul valorii de tip dublucuvânt din *varp* în DS:BX, acţionând în acest sens ca un fel de "instrucţiune *mov* pe 4 octeţi" (instrucţiunea *mov* poate avea doar operanzi octet sau cuvânt!). Utilizarea valorii dublucuvânt transferate pe post de adresă s-a făcut numai în cadrul instrucţiunii *mov al,[bx],* care datorită adresării indirecte specificate a provocat interpretarea operandului sursă [bx] drept "conţinutul de la adresa DS:BX" şi nu "conţinutul registrului bx", ca şi în cazul specificării *mov ax,bx.*

Iată în continuare un exemplu care demonstrează utilitatea instrucţiunilor de transfer al adreselor:

```
data1 segment
      db 'primul segment de date'
aa    db 31
data1 ends

data2 segment
      db 'al doilea segment de date'
      db '01234567'
bb    dd aa                 ;bb va conţine adresa far a etichetei aa
data2 ends

cod   segment
      assume cs:cod, ds:data2, es:data1

start: .      .      .
      mov   ax,data2
      mov   ds,ax
      mov   ax,data1
      mov   es,ax
      lea   bx,aa         ;echivalent cu mov bx,offset aa
      lds   di,bb         ;ds:di := es:offset aa
                          ;(practic variabila bb este utilizată aici ca
                          variabilă pointer ce conţine adresa far a
                          variabilei aa)
```



```
      mov   al, ds:[di]   ;al := 31 (valoarea de la adresa etichetei de
      .     .     .       ;date aa)
cod ends
      end start
```

Dintre cele trei instrucţiuni prezentate în cadrul acestei secţiuni, instrucţiunea LEA este de fapt singura ce corespunde cu adevărat titlului secţiunii, fiind într-adevăr o instrucţiune ce transferă o adresă. Este vorba despre adresa near (offsetul) a operandului specificat. Următoarele două instrucţiuni (LDS şi LES) sunt însă net diferite de LEA, ele transferând de fapt nu adrese ale unor entităţi din memorie, ci conţinuturi ale unor locaţii de memorie ce au posibilitatea de a fi interpretate ulterior drept adrese (similare deci ca semantică cu noţiunea de variabile pointer din limbajele de programare de nivel înalt). Ca urmare, în momentul aplicării instrucţiunilor LDS şi LES asamblorul va face doar un transfer de dublucuvinte, nefiind sigură interpretarea ulterioară a acestor valori drept adrese! Deşi este adevărat că în majoritatea cazurilor se întâmplă ca manipularea unor dublucuvinte să fie chiar în sensul de adrese fizice complete (adrese far, adică de forma *segment:deplasament*) se observă că sintaxa celor două instrucţiuni (LDS şi LES) nu impune în nici un fel ca operandul sursă să fie neapărat o adresă.

În acest context, este mai natural să interpretăm intuitiv aceste instrucţiuni drept nişte "superinstrucţiuni MOV" capabile să opereze direct cu dublucuvinte (instrucţiunea MOV poate lucra doar cu octeţi sau cuvinte). De aceea poziţia noastră este că LDS şi LES ar trebui tratate mai degrabă alături de instrucţiunea MOV în cadrul secţiunii "Instrucţiuni de transfer de uz general". Pe de altă parte, este de înţeles de ce în literatura de specialitate ele sunt tratate în cadrul secţiunii "instrucţiuni de transfer al adreselor": ele sunt într-adevăr instrucţiuni de transfer, permiţând interpretarea şi utilizarea ulterioară a valorilor transferate drept adrese! Sau altfel spus: dacă se întâmplă să avem de manipulat global adrese far, atunci trebuie neapărat ca acest lucru să îl facem cu ajutorul instrucţiunilor LDS sau/şi LES.

Să reţinem deasemenea că instrucţiunile LDS şi LES sunt singurele instrucţiuni ale limbajului de asamblare ce acceptă ca ambii operanzi sa fie dublucuvinte şi această chiar de o manieră explicită ! (conţinutul <mem> dublucuvânt se transferă în dublucuvântul DS:reg sau respectiv ES:reg).

O să vedem în cadrul secţiunii dedicată operaţiilor aritmetice că şi instrucţiunile mul, imul, div şi idiv acceptă ca un operand să fie dublucuvânt, însă acesta e predefinit (DX:AX) şi ca urmare din punct de vedere al programatorului acesta este limitat tot la specificarea unui operand octet sau cuvânt pe post de înmulţitor sau împărţitor!

Am ţinut să facem aceste observaţii deoarece din experienţa noastră didactică se remarcă printre studenţi o rată foarte înaltă de confuzie asupra mecanismelor implicate de LEA în raport cu LDS şi LES. Din cauza tratării lor sub aceeaşi titulatură de "instrucţiuni de transfer al adreselor" şi datorită relativei uşurinţe de a înţelege efectul instrucţiunii LEA, există tentaţia de a deduce că LDS şi LES sunt similare lui LEA, în sensul că ele nu vor încărca numai deplasamentul operandului sursă (ca LEA) ci (se interpretează greşit!) vor transfera ***întreaga adresă (segment:deplasament) a***

operandului sursă! Aici apare greşeala de interpretare comisă de cei în cauză: ***LDS şi LES nu transferă adresa far a operandului sursă, ci conţinutul operandului sursă!***

Revenind şi reanalizând exemplul anterior în care am avut

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

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

```
; secvenţa 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. Instrucţiuni asupra flagurilor

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

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

Instrucţiunea **LAHF** (*Load register AH from Flags*) copiază indicatorii SF, ZF, AF, PF şi CF din registrul de flag-uri în biţii 7, 6, 4, 2 şi respectiv 0 ai registrului AH. Conţinutul biţilor 5,3 şi 1 este nedefinit. Indicatorii nu sunt afectaţi în urma acestei operaţii de transfer (în sensul că instrucţiunea LAHF nu este ea însăşi generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor şi atât).

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

Instrucţiunea **PUSHF** transferă toţi indicatorii în vârful stivei (conţiunutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectaţi în urma acestei operaţii.

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

Acţiunea unor instrucţiuni ale limbajului de asamblare este determinată de valoarea unora dintre indicatorii de condiţie (de exemplu, după cum se va vedea, instrucţiunile pe şiruri acţionează "crescător" sau "descrescător" în funcţie de valoarea flag-ului DF). Limbajul de asamblare pune la

dispoziţia programatorului nişte *instrucţiuni de setare* a valorii indicatorilor de condiţie, pentru ca programatorul să poată influenţa după dorinţă modul de acţiune a instrucţiunilor care exploatează flaguri.

Operanzii instrucţiunilor de setare a flag-urilor sunt impliciţi (fiecare instrucţiune setează numai un anumit flag), apelul acestor instrucţiuni făcându-se în consecinţă doar prin specificarea mnemonicii. Nici una dintre aceste instrucţiuni 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

Instrucţiunea **CLC** (*CLear Carry flag*) poziţionează la 0 indicatorul de transport CF. Instrucţiunea **CMC** (*CoMplement Carry flag*) setează valoarea flag-ului CF în valoarea sa complementară, iar instrucţiunea **STC** (*SeT Carry flag*) îl poziţionează la 1.

Instrucţiunea **CLD** (*CLear Direction flag*) poziţionează la 0 indicatorul de direcţie DF. Instrucţiunea **STD** (*SeT Direction flag*) poziţionează DF la valoarea 1.

Instrucţiunea **CLI** (*CLear Interrupt-enable flag*) poziţionează la 0 indicatorul de validare a întreruperii (IF), ceea ce va produce dezactivarea întreruperilor mascabile. Activarea acestor întreruperi se face cu ajutorul instrucţiunii **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. Instrucţiuni de conversie

Scopul unor tehnici sau instrucţiuni 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.

Instrucţiunile de conversie prezentate în această secţiune sunt considerate ca fiind de tip distructiv, deoarece efectul lor este modificarea conţinutului unor regiştri (mai precis a regiştrilor AH şi DX). Aceste instrucţiuni sunt **CBW** (Convert **B**yte to **W**ord) şi respectiv **CWD** (Convert **W**ord to **D**oubleword).

CBW	conversie octet conţinut în AL la cuvânt în AX (extensie de semn)	-
CWD	conversie cuvânt conţinut în AX la dublu cuvânt în DX:AX (extensie de semn)	-

Instrucţiunea **CBW** converteşte octetul cu semn din AL în cuvântul cu semn AX (mai precis, extinde bitul de semn al octetului din AL la nivelul cuvântului din AX, modificând distructiv conţinutul registrului AH). De exemplu,

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

extinde valoarea octet -1 din AL în valoarea cuvânt -1 din AX.

Analog, pentru conversia cu semn cuvânt - dublu cuvânt, instrucţiunea **CWD** extinde cuvântul cu semn din AX în dublucuvântul cu semn DX:AX. Exemplu:

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

obţine valoarea -10000 în DX:AX.

Este evident deci că aceste două instrucţiuni (CBW, CWD) sunt instrucţiuni de conversie cu semn. Care sunt atunci instrucţiunile (echivalente lor) de conversie fără semn ? Răspuns: nu există în cadrul limbajului de asamblare instrucţiuni speciale de conversie fără semn, deoarece extinderea dimensiunii de reprezentare în cazul fără semn înseamnă **întotdeauna** simpla completare cu zerouri nesemnificative (de asemenea conversie distructivă!). În acest scop, o simplă instrucţiune MOV realizează scopul dorit! Exemple:

```
mov al, 255
mov ah,0      ;conversie fără semn a valorii din AL în AX (byte – word)

mov ax, 62784
mov dx,0      ;conversie fără semn a valorii din AX în DX:AX (word –
              ;doubleword)
```

În cazul conversiei cu semn însă a fost necesară introducerea de instrucţiuni speciale deoarece programatorul nu poate decide simplu într-un caz anume dacă acea conversie cu semn va necesita

completarea cu 8 sau 16 zerouri sau cu 8 sau 16 de cifre binare 1 (acest lucru depinde tocmai de semnul numărului!) şi ca urmare nu se poate folosi un simplu MOV în acest caz.

Ca urmare, să reţinem: conversia **fără semn** se realizează prin zerorizarea octetului sau cuvântului superior al valorii de la care s-a plecat. Conversia **cu semn** este realizată prin utilizarea instrucţiunilor special prevăzute în acest sens: CBW şi CWD.

4.1.3. Impactul reprezentării little-endian asupra accesării datelor.

După cum am văzut în capitolul 1 (secţiunea 1.3.2.3.) arhitecturile de tip 80x86 utilizează modalitatea *little-endian* de reprezentare a datelor. Întrebarea care se pune este cât de conştient trebuie să fie programatorul despre particularităţile de reprezentare ale unor date atunci când elaborează un cod sursă, când şi în ce fel trebuie să ţină cont acesta de detaliile de reprezentare little endian ale datelor pe care le accesează?

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octeţilor drept octeţi şi nu drept secvenţe de octeţi interpretate ca şi cuvinte sau dublucuvinte, accesarea de cuvinte ca şi cuvinte şi nu ca perechi de octeţi, accesarea de dublucuvinte ca şi dublucuvinte şi nu ca secvenţe de octeţi sau de cuvinte) atunci instrucţiunile limbajului de asamblare vor ţine cont în mod automat în cadrul manipulării datelor de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiţie programatorul nu trebuie să intervină suplimentar în nici un fel pentru a asigura corectitudinea accesării şi manipulării datelor utilizate. Exemplu:

```
a  db  'd', -25, 120
b  dw -15642, 2ba5h
c  dd  12345678h
...
mov al, a     ;se încarcă în AL codul ASCII al caracterului 'd'
mov bx, b     ;se încarcă în BX valoarea -15642; ordinea octeţilor în BX va fi însă
              ;inversată faţă de reprezentarea în memorie, deoarece numai reprezentarea în
              ;memorie foloseşte reprezentarea little-endian! În regiştri datele sunt
              ;memorate conform reprezentării structurale normale, echivalente unei
              ;reprezentări big endian. Instrucţiunea mov realizează automat în mod
              ;corect acest transfer -- cu inversarea corespunzătoare a ordinii octeţilor în
              ;cadrul reprezentării la nivel de registru - tocmai datorită utilizării valorii b
              ;consistent cu dimensiunea de reprezentare definită (utilizarea sa drept
              ;valoare de tip cuvânt).

les dx, c     ;se încarcă în combinaţia de regiştri ES:DX valoarea dublucuvânt
              ;1234:5678h, mai precis, în ES se încarcă 1234h iar în DX se încarcă 5678h.
              ;Aceeaşi observaţie şi aici: faţă de ordinea octeţilor din memorie care este
```

;78h 56h 34h 12h, după acţiunea instrucţiunii **les** ordinea octeţilor în cadrul
;regiştrilor va fi cea structurală normală, adică 12h 34h 56h 78h.

Dacă însă se doreşte accesarea sau interpretarea datelor sub o formă diferită faţă de modalitatea de definire (ex: octeţi ai cuvintelor sau dublucuvintelor, cuvinte ale dublucuvintelor, secvenţe de octeţi interpretate drept cuvinte sau dublucuvinte) atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să îşi asume însă întreaga responsabilitate a interpretării şi accesării corecte a datelor. Cu alte cuvinte, în astfel de situaţii programatorul este obligat să conştientizeze particularităţile de reprezentare little-endian (este vorba despre ordinea de plasare a octeţilor în memorie) şi să utilizeze modalităţi 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 octeţii sunt
                              plasaţi ;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 semnificaţie 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 instrucţiunea

```
mov al, a          ;syntax error! a este cuvânt, iar AL este octet;
```

vom obţine o eroare de sintaxă datorită dimensiunii diferite de reprezentare dintre operanzii instrucţiunii **mov**. Ca urmare, va trebui să utilizăm o instrucţiune 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 şi
                    ;transferul acelui octet în registrul AL
```


Dacă facem însă aşa 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 găseşte la adresa a+1. Ca urmare, instrucţiunea 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) şi
                     ;transferul său în registrul AL.
```

Simbolul '+' care apare în cadrul expresiei a+1 reprezintă un **operator** şi nu o **instrucţiune**. Operatorii (vezi capitolul 3) efectuează calcule cu valori constante determinabile la momentul asamblării. Semantica sa nu reprezintă aici 12h+1 = 13h, adică nu înseamnă "conţinutul lui a" +1, deoarece "conţinutul unei zone de memorie" nu poate fi o valoare determinabilă la momentul asamblării (când zona destinată execuţiei programului nici măcar nu există!). În schimb, ştim din capitolul 3 (secţiunea 3.1) că în limbajul de asamblare, valoarea asociată simbolului ce desemnează o variabilă (etichetă de date) este prin definiţie adresa sa (deplasament), valoare determinabilă la momentul asamblării. Ca urmare, expresiei a+1 i se asociază de fapt ca înţeles "valoarea etichetei de date a" +1, reprezentând astfel o aritmetică de adrese.

Dacă s-ar fi dorit transferul în AL a "conţinutului lui a" + 1 acest lucru nu putea fi exprimat prin utilizarea unui operator, ci trebuie făcut prin utilizarea instrucţiunii **add** corespunzătoare:

```
mov al, a    ;"conţinutul lui a" se transferă în registrul AL
add al, 1    ;aici se adună 1 la "conţinutul lui a"
```

Pe baza celor discutate până acum şi ţinând cont de ordinea de mai sus a plasării în memorie a octeţilor se pot uşor 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 aceeaşi adresă şi anume
                        ;adresa octetului 22h.

mov dx, a+4             ;deoarece a este de tip cuvânt această instrucţiune 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 octeţii 34h 12h 44h 33h care
```

```
                        ;(datorită reprezentării little-endian) înseamnă de fapt
                        ;dublucuvântul 33441234h.

lds bx, dword ptr b     ;ds:bx := 1122h:3344h
mov ax, word ptr a+1    ;ax := 4412h

mov bh, byte ptr b      ;bh := 44h
mov ch, byte ptr b-1    ;bh := 12h
```

4.2. OPERAŢII

După cum s-a văzut, transferul de informaţie joacă un rol esenţial în funcţionarea microprocesorului. La fel de important însă este să dispunem de mijloace de transformare a acestei informaţii. Operaţiile aritmetice şi logice reprezintă tocmai astfel de mijloace.

4.2.1. Operaţii aritmetice

Facilităţile de calcul aritmetic oferite de instrucţiunile maşină se dovedesc a fi surprinzător de rudimentare. De exemplu, nu se permite lucrul simplu în aritmetica numerelor reale ! Putem totuşi realiza aceasta, dar numai prin intermediul scrierii unor programe specializate sau prin integrarea unui coprocesor matematic specializat în operaţii aritmetice de mare precizie. De asemenea, nu există instrucţiuni aritmetice sau logice care să poată manipula direct operanzi specificaţi explicit reprezentaţi pe mai mult de 16 biţi (sau 32 de biţi în cazul microprocesoarelor >= 80386). Operaţiile primitive de care dispune un microprocesor 80x86 sunt adunarea, scăderea, înmulţirea şi împărţirea valorilor întregi reprezentate pe 8, 16 sau eventual 32 de biţi. Tabelul de pe pagina următoare conţine instrucţiunile aritmetice, semnificaţia lor şi efectul asupra flagurilor.

4.2.1.1. Adunarea şi scăderea

Aceste operaţii coincid cu operaţiile cunoscute din mulţimea numerelor întregi. Operanzii sunt reprezentaţi în cod complementar (vezi 1.5.2.). Datorită proprietăţilor reprezentării în cod complementar, microprocesorul realizează adunările şi scăderile "văzând" doar configuraţii de biţi şi nu numere cu semn sau fără. Regulile de efectuare a adunării şi scăderii presupun adunarea de configuraţii binare, fără a fi nevoie de a interpreta operanzii drept cu semn sau fără semn anterior efectuării operaţiei! Deci, la nivelul acestor instrucţiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucţiuni separate pentru adunarea/scăderea cu semn faţă de adunarea/scăderea fără semn. Adunarea şi scăderea se efectuează întotdeauna la fel (adunând sau scăzând configuraţii binare) indiferent de semnul (interpretarea) acestor configuraţii! După cum vom vedea acest lucru nu este valabil şi pentru înmulţire şi împărţire. În cazul acestor operaţii trebuie să ştim apriori dacă operanzii vor fi interpretaţi drept cu semn sau fără semn. De exemplu, fie doi operanzi A şi B reprezentaţi fiecare pe câte un octet:

INSTRUCŢIUNI ARITMETICE

ADD *d,s*	<d> <-- <d>+<s>	AF,CF,OF,PF,SF,ZF
ADC *d,s*	<d> <-- <d>+<s>+<CF>	AF,CF,OF,PF,SF,ZF
INC *d*	<d> <-- <d>+1	AF,OF,PF,SF,ZF
SUB *d,s*	<d> <-- <d>-<s>	AF,CF,OF,PF,SF,ZF
SBB *d,s*	<d> <-- <d>-<s>-<CF>	AF,CF,OF,PF,SF,ZF
DEC *d*	<d> <-- <d>-1	AF,OF,PF,SF,ZF
NEG *d*	<d> <-- (0-<d>)	AF,CF,OF,PF,SF,ZF
MUL *s*	dacă s este de tip octet atunci AX <-- <AL>*<s> dacă s e de tip cuvânt atunci DX:AX <-- <AX>*<s> operanzii sunt trataţi ca întregi fără semn	CF,OF modificaţi AF,PF,SF,ZF nedefiniţi; dacă CF şi OF sunt 1, atunci AH (DX) conţine valori ≠ 0
IMUL *s*	dacă s este de tip octet atunci AX <-- <AL>*<s> dacă s e de tip cuvânt atunci DX:AX <-- <AX>*<s> operanzii sunt trataţi ca întregi cu semn	CF,OF modificaţi AF,PF,SF,ZF nedefiniţi; dacă CF şi OF sunt 1, atunci AH (DX) conţine valori ≠ 0
DIV *s*	dacă s este de tip octet atunci AL <-- (<AX> DIV <s>) AH <-- (<AX> MOD <s>) dacă s e de tip cuvânt atunci AX <-- (<DX:AX> DIV <s>) DX <-- (<DX:AX> MOD <s>) operanzi fără semn	AF,CF,OF,PF,SF,ZF nedefiniţi La obţinerea unui cât ce nu încape în AL (AX) se semnalează "depăşire"
IDIV *s*	dacă s este de tip octet atunci AL <-- (<AX> DIV <s>) AH <-- (<AX> MOD <s>) dacă s e de tip cuvânt AX <-- (<DX:AX> DIV <s>) DX <-- (<DX:AX> MOD <s>) operanzi cu semn	AF,CF,OF,PF,SF,ZF nedefiniţi La obţinerea unui cât ce nu încape în AL (AX) se semnalează "depăşire"

A = 9Ch = 10011100b (= 156 în interpretarea fără semn şi -100 în interpretarea cu semn)
B = 4Ah = 01001010b (= 74 atât în interpretarea fără semn cât şi în interpretarea cu semn)

Microprocesorul realizează adunarea C = A + B şi obţine

C = E6h = 11100110b (= 230 în interpretarea fără semn şi -26 în interpretarea cu semn)

Se observă deci că simpla adunare a configuraţiilor de biţi (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obţinut, atât în interpretarea cu semn cât şi în cea fără semn.

Instrucţiunea **ADD** are forma

ADD *destinaţie, sursă*

Ea adună cei doi operanzi, rezultatul fiind depus în operandul destinaţie, cu actualizarea corespunzătoare a flagurilor. Cei doi operanzi trebuie să fie reprezentaţi ambii pe octet sau ambii pe cuvânt.

Instrucţiunea **SUB** are forma

SUB *destinaţie, sursă*

Aceasta scade operandul sursă din operandul destinaţie, păstrând rezultatul în operandul destinaţie. De exemplu, atribuirea E:=A+B-C (unde A, B, C şi E sunt valori reprezentate pe cuvânt) se realizează prin secvenţa

```
mov  ax, A
add  ax, B
sub  ax, C
mov  E, ax
```

În urma efectuării unei operaţii de adunare sau scădere, 80x86 setează flag-ul C (*carry flag*) cu valoarea 0 (marcând desfăşurarea normală a operaţiei) sau 1 (marcând depăşirea spaţiului de reprezentare a operandului destinaţie de către valoarea rezultată în cazul unei adunări, respectiv necesitatea "împrumutului" unei unităţi în cazul scăderii).

Este evident aşadar că pentru programarea unei adunări sau scăderi pe mai mult de 16 biţi trebuie să ţinem cont de valoarea din CF. Este ceea ce fac instrucţiunile **ADC** şi **SBB**. **ADC** are acelaşi efect ca şi ADD numai că adună în plus şi valoarea din CF. **SBB** are acelaşi efect ca şi **SUB**, scăzând însă şi "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 şi utilitatea instrucţiunii ADC. Fie atribuirea x := x + y, unde x şi y sunt numere fără semn reprezentate în baza 2 pe câte 48 de biţi. 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. Secvenţa de instrucţiuni 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 rămase!
```

Scăderea valorii dublu cuvânt CX:BX din valoarea dublu cuvânt din DX:AX (ceva de genul "DX:AX := DX:AX – CX:BX") trebuie făcută ţinând cont de eventuala cifră binară "împrumutată" la prima operaţie de scădere :

```
sub  ax,bx
sbb  dx,cx
```

Atragem atenţia că pentru corecta funcţionare a instrucţiunilor **ADC** şi **SBB** trebuie să se aibă în vedere păstrarea valorii din CF între respectivele operaţii compuse. De exemplu, secvenţa

```
add  ax,bx
sub  si,si          ;setează CF pe 0
adc  dx,cx
```

nu va aduna corect CX:BX la DX:AX deoarece instrucţiunea SUB modifică potenţial valoarea din CF între cele două operaţii ADD şi ADC.

Datorită frecvenţei ridicate a apariţiei în cadrul programelor a adunărilor cu 1 (incrementare) precum şi a scăderilor cu 1 (decrementare) limbajul de asamblare conţine instrucţiunile **INC** şi respectiv **DEC** cu sintaxa

INC	operand	şi	**DEC**	operand	echivalente ca efect deci cu
ADD	operand,1	şi	**SUB**	operand,1	

Avantajele introducerii acestor instrucţiuni rezidă atât în codul compact rezultat (se reprezintă doar pe 1 octet comparativ cu variantele ADD sau SUB care necesită 3) cât şi în viteza mult mai mare de execuţie.

Instrucţiunea **NEG** are sintaxa

NEG *destinaţie*

şi are ca efect schimbarea semnului valorii operandului (registru sau variabilă de memorie). De exemplu, în urma execuţiei secvenţei

```
mov  ax,1
neg  ax      ;conţinutul din ax devine -1 (16 cifre 1)
mov  bx,ax
neg  bx      ;conţinutul registrului bx devine 1
```

o să avem -1 în AX (în interpretarea cu semn) şi 1 în BX. Facem precizarea că valoarea din AX se poate interpreta şi ca valoare fără semn, aceasta fiind în acest caz 65535.

4.2.1.2. Înmulţirea şi împărţirea

Limbajul de asamblare pune la dispoziţia utilizatorilor instrucţiuni de înmulţire şi împărţire cu semn şi fără semn.

Forma generală a instrucţiunii de înmulţire fără semn este

MUL *operand*

unde *operand* este un registru general sau o variabilă de memorie. Instrucţiunea MUL realizează înmulţirea fără semn a valorii operandului cu valoarea din AL sau AX, acţiunea instrucţiunii depinzând de dimensiunea operandului:

* dacă *operand* este octet, el este multiplicat cu valoarea din AL, rezultatul memorându-se în AX.
* dacă *operand* este cuvânt, el este multiplicat cu valoarea din AX, rezultatul memorându-se în DX:AX, DX conţinând cuvântul superior al produsului.

Deşi înmulţirea furnizează rezultatul pe un spaţiu dublu decât operanzii, ea semnalează dacă produsul nu se poate reprezenta pe un spaţiu de aceeaşi dimensiune cu operanzii, punând CF=1 şi OF=1. În caz contrar se pune CF=0 şi OF=0.

De exemplu, calculul valorii E := A*B, cu operanzii reprezentaţi pe octet, se poate efectua prin secvenţa:

```
mov  al, A
mul  B
mov  E, al
```

Instrucţiunea **mul ax** depune în DX:AX pătratul vechii valori din AX.

Pentru cazul în care se doreşte realizarea de înmulţiri cu semn se utilizează instrucţiunea

IMUL *operand*

care are aceleaşi reguli de acţiune ca şi MUL, cu precizarea că respectă regula semnelor cunoscută din aritmetică. Astfel, secvenţa

```
mov  al,-2
mov  ah,10
imul ah
```

plasează valoarea -20 (0FFECh) în AX. Dacă în loc de **IMUL** s-ar folosi **MUL** rezultatul depus în AX ar fi dictat de reprezentarea fără semn a conţinutului iniţial din AL (0FEh = 254), care, înmulţit cu 10 ar produce în AX valoarea 254 * 10 = 2540 (09ECh).

Vom urmări în secvenţa de mai jos un mod de calcul al valorii Z := X * Y, unde X şi Y sunt valori fără semn reprezentate respectiv pe 48 şi 16 biţi. Rezultă în mod natural că pentru reprezentarea lui Z sunt necesari 64 biţi. Iată exemplul:

```
mov ax, X    ;depune cuvântul cel mai puţin semnificativ al valorii X în AX
mul Y        ;înmulţire cu valoarea din Y, rezultatul în DX:AX
mov Z, ax    ;pune primele 2 cifre hexa obţinute (cele mai puţin semnificative) în locul
             ;corespunzător din Z
mov bx,dx    ;pune în BX cuvântul superior al rezultatului pentru a servi ca transport

mov ax,X+2   ;preia următorul cuvânt al sursei pentru a-l înmulţi
mul Y        ;înmulţire cu valoarea din Y
add ax,bx    ;se adună transportul anterior de la înmulţire
adc dx,0     ;şi se ţine cont de eventualul transport în urma adunării
mov Z+2,ax   ;se actualizează rezultatul cu următoarele cifre
mov bx,dx    ;reţine transportul în BX

mov ax, X+4  ;preia cuvântul cel mai semnificativ al sursei
mul Y        ;înmulţeşte corespunzător
add ax,bx    ;adună transportul înmulţirii
adc dx,0     ;şi ţine cont de transport adunare
mov Z+4,ax   ;depune valoarea obţinută în AX ca cel de-al treilea cuvânt al rezultatului
mov Z+6,dx   ;şi pune cuvântul superior al ultimului rezultat obţinut pe poziţiile
             ;corespunzătoare celor mai semnificative 2 cifre hexa ale rezultatului final
```

Împărţirea fără semn a două numere întregi se face cu ajutorul instrucţiunii **DIV**, care are forma

DIV *operand*

unde *operand* poate fi un registru de memorie sau o variabilă de memorie, el reprezentând împărţitorul. Deîmpărţitul este determinat în funcţie de dimensiunea de reprezentare a operandului împărţitor, astfel:

* dacă împărţitorul este octet, atunci deîmpărţitul este conţinutul registrului AX. Câtul va fi depus în AL, iar restul în AH.
* dacă împărţitorul este reprezentat pe cuvânt, atunci deîmpărţitul este DX:AX. Câtul va fi depus în AX, iar restul în DX.

De exemplu,

```
mov  ax,51
mov  dl,10
div  dl
```

realizează împărţirea conţinutului lui AX la conţinutul lui DL, memorându-se câtul 5 în AL şi 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 câtul 1000h în AX şi restul 2 în DX.

Să observăm că, dacă n este lungimea de reprezentare a deîmpărţitului, câtul se impune a fi reprezentat pe n/2 biţi. Dacă rezultatul obţinut va necesita pentru reprezentare o lungime mai mare, atunci microprocesorul va semnala depăşire prin generarea unei întreruperi 0 (împărţire prin 0). De exemplu, următoarea secvenţă generează o întrerupere 0 (vezi capitolul 5):

```
mov  ax,0ffffh
mov  bl,1
div  bl
```

Pentru împărţirea cu semn dispunem de instrucţiunea **IDIV** *operand*

cu aceleaşi observaţii şi reguli de funcţionare ca şi DIV, ţinând cont în plus de semnul operanzilor. Spre deosebire de teorema fundamentală a împărţirii 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 execuţiei secvenţei

```
data segment
        Divizor      DW    100
.       .       .
code segment
.       .       .
```



```
mov   ax,-667
cwd                    ;extinde numărul în DX:AX
idiv  Divizor
```

se va depune -6 în AX şi -67 în DX.

4.2.1.3. Exemple şi exerciţii propuse.

a). .....

```
mov ah,0
mov al, -5   ;echivalent cu   mov al, 11111011b   deci echivalent şi cu mov al, 251
             ;echivalent şi cu mov al, 0fbh în hexazecimal; f = 1111b şi b=1011b

;ansamblul celor 2 instrucţiuni de mai sus este echivalent cu   mov ax,251 (echivalent în
;binar cu mov ax, 0000000011111011b şi în hexazecimal cu mov ax, 00fbh) însă nu şi
;cu mov ax, -5 (echivalent în binar cu mov ax, 1111111111111011b şi în hexazecimal cu
;mov ax, 0fffbh) - diferenţa constă în completarea conţinutului lui AH cu 8 cifre binare 0
;în cazul lui 251 – interpretare fără semn - şi 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) şi DX:=0

(deşi aici prin imul e forţată 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 observaţiei de mai sus

mov cl, -100 (= 9ch = 10011100b = 156 în interpretarea fără semn!)

idiv cl      ;AX=2510; AL (câtul):= AX idiv (-100) = -25 (=e7);AH (restul):=10 (0ah)
             ;conţinutul 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 iniţiale din AX:=2510;

div  cl      ;AX=2510; AL (câtul) := AX div 156 = 16 (=10h); AH (restul) := 14 (0eh)
             ;conţinutul lui AX este acum AX=0e10h
```

Reluaţi discuţia, analizaţi şi justificaţi rezultatele furnizate în situaţia în care ultimele 5 instrucţiuni de mai sus devin (CL se înlocuieşte cu CX):

```
mov cx, -100
idiv cx
imul cx
add ax,10
div cx
```

b). Ce se întâmplă cu exemplul de mai sus dacă acesta devine:

```
mov ax, -5    ;echivalent cu      mov al, -5
              ;                   cbw
mov bx,10
imul bx
```

Ce valoare se obţine acum ca rezultat ? Comparaţi rezultatul furnizat cu cel obţinut prin aplicarea operaţiei **mul bx** în loc de **imul bx**.

c). .....

```
mov al, 251        ; 251 = 0fbh = -5 - deci instrucţiune echivalentă cu   mov al, -5
```

(suma valorilor absolute ale reprezentărilor cu semn şi fără semn la nivelul unui **octet** de memorie este **256** - regulă practică !) – aici se verifică prin 256 = 5 + 251

```
mov cl,255
mov bl, 100
imul bl            ; ax := al * bl = -5 * 100 = -500 = 0fe0ch = 65036
```

(suma valorilor absolute ale reprezentărilor cu semn şi fără semn la nivelul unui **cuvânt** de memorie este **65536** - regulă practică !) – aici se verifică prin 65536 = 500 + 65036

```
div cl        ;div impune ca valoarea din ax să fie interpretată acum fără semn!
              ;al := ax div bl = 65036 div 255 = 255 = ffh ; restul în AH = 11 = 0bh
              ;deci conţinutul lui AX este acum AX = 0bffh
```

Ce se întâmplă dacă **"imul bl"** este înlocuită cu instrucţiunea **"mul bl"** ? Analizaţi şi explicaţi comparativ rezultatele furnizate.

Ce observaţi că se întâmplă dacă **"div cl"** este înlocuită cu instrucţiunea **"idiv cl"** ?

Încercaţi să (vă) explicaţi potenţiala cauză a unui astfel de comportament (vezi şi 5.2.2 – INT 0) prin efectuarea "pe hârtie" a operaţiei **"idiv cl"** şi analizând apoi încadrarea valorilor astfel obţinute (cât şi rest) în dimensiunile de reprezentare puse la dispoziţie de o operaţie div sau idiv.

Dar dacă apoi **"div cl"** este înlocuită întâi cu **"div cx"** iar apoi cu **"idiv cx"** ? Justificaţi şi explicaţi rezultatele obţinute. De ce în acest caz nu se mai manifestă situaţia apărută în cazul **"idiv cl"** ?

4.2.2. Operaţii logice pe biţi

Limbajul de asamblare dispune de un set de patru instrucţiuni pentru realizarea de operaţii logice la nivel de bit: **AND, OR, XOR** şi **NOT**.

AND *d,s*	<d> <-- <d> **şi** <s>	CF,OF,PF,SF,ZF modificaţi; AF - nedefinit
OR *d,s*	<d> <-- <d> **sau** <s>	CF,OF,PF,SF,ZF modificaţi; AF - nedefinit
XOR *d,s*	<d> <-- <d> **sau exclusiv** <s>	CF,OF,PF,SF,ZF modificaţi; AF - nedefinit
NOT *s*	<s> <-- <s> negat complement faţă de 1	

Pentru doi biţi, unul din sursă, altul din destinaţie, instrucţiunile logice produc noul bit destinaţie având valoarea conform următoarelor reguli:

bitul d	bitul s	d AND s	d OR s	d XOR s
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Pentru doi operanzi instrucţiunile logice realizează aceste operaţii asupra perechilor corespunzătoare de biţi. Operanzii sursă şi destinaţie trebuie să aibă ambii aceeaşi dimensiune (octet sau cuvânt). Fie

```
A    DB    10010101b
B    DB    01011010b
```

Atunci avem

```
mov  al, A
and  al, B   ;rezultă 00010000b în AL

mov  al, A
or   al, B   ;rezultă 11011111b în AL

mov  al, A
xor  al, B   ;rezultă 11001111b în AL
```

Instrucţiunea AND este indicată pentru izolarea unui anumit bit sau pentru forţarea anumitor biţi la valoarea 0. Astfel, dacă dintr-o configuraţie pe 8 biţi

a = x x x x x x x x b

dorim izolarea bitului i (0 <= i <= 7), vom crea expresia AND a,2^i

iar dacă dorim forţarea anumitor biţi din a la valoarea 0, să zicem de exemplu biţii 0, 2 şi 3 (restul rămânând neschimbaţi) vom crea expresia

```
AND a,11110010b  (= 242)
```

(dacă a conţine iniţial valoarea 01010110b de exemplu, atunci după execuţia instrucţiunii de mai sus, în a se va afla valoarea 01010010b).

Instrucţiunea OR poate fi folosită printre altele pentru forţarea anumitor biţi la valoarea 1. De exemplu, presupunând că variabila a conţine iniţial aceeaşi valoare de mai sus (01010110b), dacă dorim forţarea biţilor 0, 2 şi 3 ai acestei valori la 1 vom folosi instrucţiunea OR astfel:

```
OR a, 00001101b  (=13)
```

Valoarea iniţială din a este distrusă, în locul ei depunându-se noua valoare rezultată, adică 01011111b.

Instrucţiunea XOR este indicată pentru schimbarea valorii unor biţi din 0 în 1 sau din 1 în 0. De exemplu, dacă dorim ca biţii 4-7 din AL să-şi schimbe valorile iar ceilalţi să rămână neschimbaţi, atunci aceasta se poate realiza prin

```
XOR  al, 11110000b
```

Dacă presupunem că iniţial în AL am avut valoarea 01010101b, atunci după execuţia instrucţiunii de mai sus în AL se va afla valoarea 10100101b (= A5h). Se observă că biţii 1 comută valorile iniţiale iar biţii 0 le lasă neschimbate. De asemenea, instrucţiunea XOR poate fi utilizată pentru zerorizarea unui registru:

```
XOR  ax, ax
```

Instrucţiunea **NOT** modifică biţii operandului în valoarea lor complementară (0 în 1 şi 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 şi rotiri de biţi

Microprocesoarele 80x86 dispun de o serie de instrucţiuni cu ajutorul cărora pot deplasa sau roti biţii din cadrul unui operand în memorie sau registru.
În cadrul octeţilor sau cuvintelor biţii pot fi deplasaţi aritmetic sau logic sau rotiţi la dreapta sau la stânga cu una sau mai multe poziţii. Numărul de deplasări este fie 1, fie o valoare cuprinsă între 1 şi 255, valoare specificată în registrul CL.

Instrucţiunile de *deplasare* de biţi se clasifică în:

- Instrucţiuni de deplasare logică
  - stânga - **SHL**
  - dreapta - **SHR**
- Instrucţiuni de deplasare aritmetică
  - stânga - **SAL**
  - dreapta - **SAR**

Instrucţiunile de *rotire* a biţilor în cadrul unui operand se clasifică în:

- Instrucţiuni de rotire fără carry
  - stânga - **ROL**
  - dreapta - **ROR**
- Instrucţiuni de rotire cu carry
  - stânga - **RCL**
  - dreapta - **RCR**

Pentru a defini deplasările şi rotirile să considerăm ca şi configuraţie iniţială 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 şi CF = a
SHR X,1  ;rezultă X = 0abcdefg şi CF = h
SAL X,1  ;     identic cu SHL
SAR X,1  ;rezultă X = aabcdefg şi CF = h
ROL X,1  ;rezultă X = bcdefgha şi CF = a
ROR X,1  ;rezultă X = habcdefg şi CF = h
RCL X,1  ;rezultă X = bcdefghk şi CF = a
RCR X,1  ;rezultă X = kabcdefg şi CF = h
```

Deplasările şi rotirile pe 16 biţi se fac analog. Deplasările şi 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 părăseşte configuraţia trece în CF.

INSTRUCŢIUNILE DE DEPLASARE ŞI ROTIRE DE BIŢI

SHL *s,1* **SAL** *s,1*	deplasare logică (aritmetică) stânga cu o poziţie. CF conţi-ne bitul cel mai semnificativ deplasat. La dreapta se introduc zerouri.	OF,SF,ZF,PF,CF modificaţi AF - nedefinit
SHL *s,cl* **SAL** *s,cl*	deplasare logică (aritmetică) stânga cu cl poziţii. CF con-ţine ultimul bit deplasat. La dreapta se introduc zerouri.	OF,SF,ZF,PF,CF modificaţi; AF - nedefinit Dacă CL≠1 , OF este nedefinit
SHR *s,1*	deplasare logică dreapta cu o poziţie. La stânga se introduc zerouri. Bitul cel mai puţin semnificativ este deplasat în CF.	OF,SF,ZF,PF,CF modificaţi AF - nedefinit
SHR *s,cl*	deplasare logică dreapta cu cl poziţii. La stânga se introduc zerouri. CF va conţine ultimul bit deplasat.	OF,SF,ZF,PF,CF modificaţi AF – nedefinit; Dacă CL≠1, OF este nedefinit
SAR *s,1*	deplasare aritmetică dreapta cu o poziţie. La stânga se extinde bitul de semn. Bitul cel mai puţin semnificativ e deplasat în CF.	OF,SF,ZF,PF,CF modificaţi AF - nedefinit
SAR *s,cl*	deplasare aritmetică dreapta cu cl poziţii. La stânga se extinde bitul de semn. CF va conţine ultimul bit deplasat.	OF,SF,ZF,PF,CF modificaţi AF – nedefinit ; Dacă CL≠1, OF este nedefinit
ROL *s,1*	rotire stânga cu o poziţie. CF va conţine bitul (cel mai semnificativ) rotit.	OF,CF
ROL *s,cl*	rotire stânga cu cl poziţii. CF va conţine ultimul bit rotit.	OF,CF ; Dacă CL≠1, OFeste nedefinit
ROR *s,1*	rotire dreapta cu o poziţie. CF va conţine bitul cel mai puţin semnificativ al lui s.	OF,CF
ROR *s,cl*	rotire dreapta cu cl poziţii. CF va conţine ultimul bit rotit.	OF,CF ; Dacă CL≠1, OF este nedefinit
RCL *s,1*	rotire stânga cu carry cu o poziţie.	OF,CF
RCL *s,cl*	rotire stânga cu carry cu cl poziţii.	OF,CF ; Dacă CL≠1, OF este nedefinit
RCR *s,1*	rotire dreapta cu carry cu o poziţie.	OF,CF
RCR *s,cl*	rotire dreapta cu carry cu cl poziţii.	OF,CF ; Dacă CL≠1, OF este nedefinit

Deplasările logice pot fi folosite pentru izolarea anumitor biţi în interiorul octeţilor (cuvintelor), iar deplasările aritmetice se pot utiliza pentru înmulţirea sau împărţirea numerelor binare cu puteri ale lui 2.

Instrucţiunile de deplasare afectează flagurile astfel:

- Flag-ul AF este întotdeauna nedefinit în urma unei operaţii de deplasare.

- Indicatorii PF, SF şi ZF sunt actualizaţi ca şi în cazul instrucţiunilor logice pe biţi.

- Indicatorul CF conţine întotdeauna valoarea ultimului bit deplasat din cadrul operandului.

- Conţinutul indicatorului OF este întotdeauna nedefinit în cazul unor operaţii de deplasare a mai multor biţi. În cazul în care se deplasează un singur bit, OF este setat la valoarea 1 dacă valoarea bitului cel mai semnificativ (semnul) a fost schimbată de operaţia de deplasare.

Instrucţiunile de rotire afectează numai flagurile CF şi OF. CF va conţine întotdeauna valoarea ultimului bit rotit. Pentru operaţiile de rotire cu mai mulţi biţi valoarea flagului OF este nedefinită. La rotirea unui singur bit OF este setat la valoarea 1 dacă bitul cel mai semnificativ îşi schimbă valoarea (schimbare de semn).

Să exemplificăm modul de acţiune al acestor instrucţiuni în câteva situaţii concrete.

De exemplu, dacă AL conţine valoarea 10010110b (=96h =150 în interpretarea fără semn, sau -6Ah = -106 în interpretarea cu semn), atunci după execuţia instrucţiunii **shl al,1** în AL se va găsi valoarea 00101100b (= 2Ch = 44). CF este setat cu 1 (se depune bitul cel mai semnificativ), iar în poziţia rămasă liberă în dreapta se introduce 0. Acţiunea instrucţiunii SHL în acest caz poate fi reprezentată schematic conform desenului din figura 4.1.

O utilizare frecventă a acestei instrucţiuni apare atunci când se doreşte multiplicarea rapidă a operandului cu puteri ale lui 2. Deplasarea spre stânga obţinută ca efect al instrucţiunii SHL este de fapt o înmulţire cu 2 (presupunând desigur că bitul cel mai semnificativ a fost 0, în caz contrar fiind vorba de trunchiere - pierderea celei mai semnificative cifre binare).

De exemplu, dacă dorim multiplicarea conţinutului registrului DX cu 16 putem efectua

```
shl   dx,1      ;DX*2
shl   dx,1      ;DX*4
shl   dx,1      ;DX*8
shl   dx,1      ;DX*16
```

Din păcate, pentru cazul general al microprocesoarelor 8086 nu este permis să scriem aşa cum am dori, adică **shl dx,4** (deşi ultimele versiuni de TASM de exemplu permit acum acest lucru!).

În schimb, se permite utilizarea registrului CL pentru a indica numărul de poziţii cu care se doreşte deplasarea, ceea ce face ca secvenţa de mai sus să fie echivalentă cu

```
mov   cl,4
shl   dx,cl
```

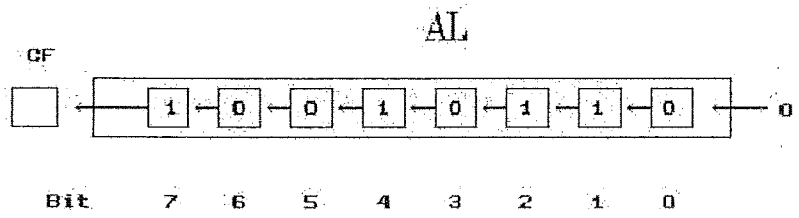


Fig. 4.1. Instrucţiunea **SHL**

Subliniem că variantele de multiplicare cu puteri ale lui 2 folosind SHL sunt mult mai rapide decât variantele echivalente folosind instrucţiunea MUL.

În acest sens, să urmărim secvenţa următoare care realizează înmulţirea cu 10 a valorii octetului *var*, fără a folosi instrucţiuni MUL sau IMUL:

```
xor   ah,ah
mov   cl,3
mov   al,var
shl   ax,cl        ;înmulţire cu opt
add   al,var
adc   ah,0         ;al = var * 9
add   al,var
adc   ah,0         ;al = var * 10
```

În ceea ce priveşte instrucţiunile de deplasare spre dreapta, presupunând că AL conţine 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 instrucţiunea SAR să fie indicată pentru efectuarea împărţirilor 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 acţiunea instrucţiunii ROR AL,1 asupra valorii 10010110b. Rezultatul este valoarea 01001011b, în CF depunându-se valoarea iniţială a ultimului bit, adică 0.

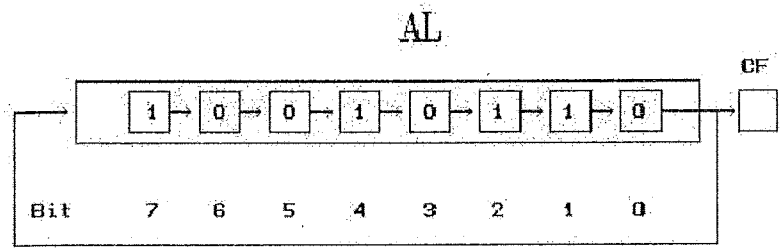


Fig. 4.2. Instrucţiunea **ROR**

Instrucţiunile **ROR** şi **ROL** devin utile în probleme privind realinierea biţilor în cadrul unui octet sau cuvânt. De exemplu, secvenţa

```
mov   si,49f1h
mov   cl,4
ror   si,cl
```

face ca în SI să se afle în final 149Fh, mutând biţii 0-3 în biţii 12-15, biţii 4-7 în biţii 0-3, ş.a.m.d.

Figura 4.3 arată rotirea spre dreapta a valorii 10010110b (= 96h = 150) din AL, participând şi CF (care iniţial conţine valoarea 1). Instrucţiunea **rcr al,1** produce rezultatul 11001011b (= 0CBh = 203) care este depus în AL. CF va conţine valoarea de dinainte de rotire a bitului cel mai puţin semnificativ, adică 0.

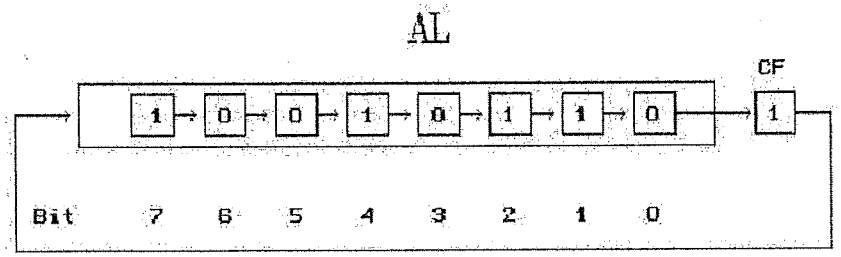


Fig. 4.3. Instrucţiunea **RCR**

Instrucţiunile RCR şi RCL sunt utile pentru realizarea de deplasări de biţi implicând operanzi reprezentaţi pe mai multe cuvinte. De exemplu, secvenţa următoare multiplică cu 4 valoarea din DX:AX :

```
shl   ax,1    ;bitul 15 din AX este depus în CF
rcl   dx,1    ;valoarea din CF se depune în bitul 0 din DX
shl   ax,1    ;bitul 15 din AX se depune în CF
rcl   dx,1    ;valoarea din CF se depune în bitul 0 din DX
```

În general, un set de n instrucţiuni de rotire (cu sau fără carry) cu câte o poziţie este mai rapid şi necesită mai puţină memorie decât o singură instrucţiune de rotire ce are *cu_cât* = n.

Precizare. În cazul utilizării de directive ce permit lucrul cu instrucţiunile specifice adăugate, microprocesoarele 80x86 cu x≥2 admit în TASM pentru *cu_cât* şi o valoare constantă diferită de 1 (pentru orice instrucţiune de deplasare sau rotire). Totuşi, există restricţia ca această valoare să nu depăşească 31.

4.3. RAMIFICĂRI, SALTURI, CICLURI

Un program scris într-un limbaj de programare se execută secvenţial, adică instrucţiunile programului se execută una după alta în ordinea în care ele sunt scrise. Apare în mod natural necesitatea "ruperii" acestei ordini, pentru a putea soluţiona anumite cerinţe specificate de problema de rezolvat. În limbajele de programare de nivel înalt cunoscute (Pascal, C, FORTRAN etc.) există în acest scop aşa numitele *structuri de control* (if, goto, for, while etc.) care determină ordinea de execuţie a instrucţiunilor unui program.

O proprietate de bază pe care trebuie s-o aibă în acest context orice calculator (microprocesor dacă discutăm la nivel scăzut, sau limbaj dacă discutăm la nivel înalt) este posibilitatea de a transfera controlul (condiţionat sau nu) la o instrucţiune, alta decât cea care urmează secvenţial celei curente. Microprocesoarele 80x86 dispun şi ele de instrucţiuni specializate care realizează transferuri ale controlului (salturi), prevăzând şi instrucţiuni care facilitează prelucrarea repetată a unui bloc de instrucţiuni (ciclu).

În cursul execuţiei unui program, adresa următoarei instrucţiuni de executat este indicată întotdeauna de CS:IP. Deci, instrucţiunile care modifică această adresă sunt de fapt instrucţiuni de salt. La o execuţie secvenţială CS şi IP sunt modificate automat de componenta BIU a microprocesorului.

4.3.1. Saltul necondiţionat

În această categorie intră instrucţiunile JMP (echivalentul instrucţiunii GOTO din alte limbaje), CALL (apelul de procedură înseamnă transferul controlului din punctul apelului la prima instrucţiune din procedura apelată) şi RET (transfer control la prima instrucţiune executabilă de după CALL).

JMP *operand*	Salt necondiţionat la adresa determinată de operand	-
CALL *operand*	Transferă controlul procedurii determinată de operand	-
RET [*n*]	Transferă controlul instrucţiunii de după CALL	-

4.3.1.1. Instrucţiunea JMP

Instrucţiunea de salt necondiţionat **JMP** are sintaxa

JMP *operand*

unde *operand* este o etichetă, un registru sau o variabilă de memorie ce conţine o adresă. Efectul ei este transferul necondiţionat al controlului la instrucţiunea ce urmează etichetei, la adresa conţinută în registru, respectiv la adresa conţinută în variabila de memorie. De exemplu, după execuţia secvenţei

```
            mov   ax,1
            jmp   AdunaDoi
AdunaUnu:   inc   ax
            jmp   urmare
AdunaDoi:   add   ax,2
urmare:     .     .     .
```

registrul AX va conţine valoarea 3. Instrucţiunile **inc** şi **jmp** dintre etichetele *AdunaUnu* şi *AdunaDoi* nu se vor executa, decât dacă se va face salt la *AdunaUnu* de altundeva din program.

În mod normal instrucţiunea JMP efectuează un salt NEAR (salt în cadrul aceluiaşi segment). Pentru a reduce codul generat, în cazul în care destinaţia saltului nu este mai departe de 127 octeţi, există posibilitatea aşa numitului "salt scurt". Un salt scurt se specifică sub forma

jmp **SHORT PTR** *eticheta*

Operatorul SHORT impune folosirea unei adrese pe 8 biţi, adresă "scurtă", economisindu-se astfel 1 octet în reprezentarea instrucţiunii JMP. Acest operator îşi justifică utilizarea de către programator numai în cazul salturilor "spre înainte", deoarece asamblorul translatează automat toate salturile "spre înapoi" ca salturi "scurte" dacă este posibil (din cauză că a trecut deja de locul unde se află "ţinta").

După cum am menţionat, saltul poate fi făcut şi la o adresă memorată într-un registru sau într-o variabilă de memorie. Exemple:

```
(1)   mov ax, OFFSET etich        (2)   data  segment
                                        Salt   DW   Dest   ;Salt := offset Dest
      jmp ax      ;operand registru     .      .    .
                                        cod    segment
      etich: . . .                      .      .    .
                                        jmp    Salt        ;salt NEAR
                                        .      .;operand variabilă de memorie
                                        Dest : . . .
```

Să remarcăm faptul că esenţial pentru exemplele date este prezentarea modalităţii în care offset-ul (deplasamentul) adresei destinaţie este permis a fi încărcat în operandul instrucţiunii JMP: în cazul (1) e vorba de o <u>încărcare explicită prin *mov* a unui registru</u>, în cazul (2) este vorba despre <u>declararea cu iniţializare a variabilei de tip cuvânt *Salt*</u>, în cadrul căreia se realizează iniţializarea variabilei *Salt* cu offset-ul etichetei *Dest* (1 cuvânt de memorie = 16 biţi = 2 octeţi = dimensiunea de reprezentare a deplasamentului). Dacă în cazul (1) dorim înlocuirea operandului destinaţie registru cu un operand destinaţie variabilă de memorie, o soluţie posibilă este:

```
        b  dw  ?
(1')          . . .
        mov  b, offset etich
        jmp b          ; salt NEAR – operand variabilă de memorie
```

Instrucţiunea 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 destinaţie ca <u>etichetă far</u> prin directiva LABEL (vezi 3.3.3). De exemplu, instrucţiunea 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
```


b). specificând direct în instrucţiunea JMP <u>tipul FAR PTR pentru eticheta destinaţie</u>, sub forma: JMP FAR PTR *eticheta*. Acest mod de specificare al operandului destinaţie a fost utilizat în exemplele din 3.3.1.2 - directiva ASSUME şi gestiunea segmentelor. Operatorul PTR a fost prezentat în 3.2.2.7.

c). <u>prefixând explicit cu un registru de segment</u> o adresă NEAR specificată indirect (vezi 3.2.1.4 – operanzi cu adresare indirectă), adică furnizarea ca operand al instrucţiunii JMP a unei expresii de forma: *reg_segment: specificare_offset*. Exemplu: jmp es:[bx+di].

d). specificând ca operand <u>o variabilă dublucuvânt</u> care conţine adresa far a destinaţiei:

```
data  segment
      Salt   DD   Dest   ;Salt:= segm:offset (Dest)
.     .      .
cod   segment
      .      .      .
      jmp    Salt        ;salt FAR la eticheta Dest
      .      .      .
code1 segment

Dest : . . .
```

secvenţă echivalentă ca efect cu

```
data segment
      Salt DD ?
      .    .    .
cod  segment
      .    .    .
      lea  ax, Dest        ;ax:=offset(Dest)

      mov  word ptr Salt, ax    ;iniţializare cuvânt inferior al
                                ;variabilei dublucuvânt Salt cu offset(Dest)
                                ;– se ţine cont de reprezentarea little-endian
      mov  ax, cod
      mov  word ptr Salt+2, ax  ;iniţializare cuvânt superior al
                                ;variabilei dublucuvânt Salt (Salt+2 –
                                ;aritmetică de pointeri) cu seg(Dest)
      jmp  Salt            ;salt FAR la adresa Dest
      .    .    .
code1 segment
Dest : . . .
```

Exemplul 4.3.1.2. Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidenţă deosebirile dintre un transfer *direct* şi unul *indirect*.

```
data segment
      aici  DW here        ;echivalent cu aici := offsetul etichetei here din segmentul de cod
data ends

code segment
      .     .     .
      mov  ax,aici         ;se încarcă în AX conţinutul variabilei aici (adică deplasamentul lui
                           ;here în cadrul segmentului code – echivalent deci ca efect cu:
                           ;mov ax, offset here)
      mov  bx,OFFSET aici  ;se încarcă în BX deplasamentul lui aici în cadrul segm. data
```

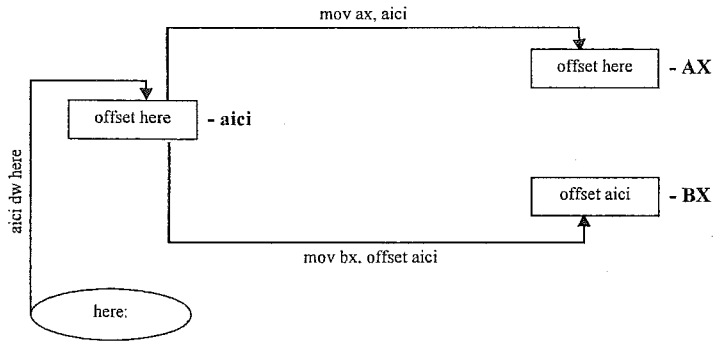


Fig. 4.4. Iniţializarea variabilei aici şi a regiştrilor AX şi BX.

```
jmp  aici     ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui
              ;here), deci instrucţiune echivalentă cu jmp here

jmp  here     ;salt la adresa lui here (sau, echivalent, salt la eticheta here)

jmp  ax       ;salt la adresa conţinută în AX (adresare registru în mod direct), adică la here

jmp  [bx]     ;salt la adresa conţinută în locaţia de memorie a cărei adresă este conţinută în
              ;BX (adresare registru în mod indirect - singura situaţie de apel indirect din
              ;acest exemplu)
```

;în BX se află adresa variabilei aici, deci se accesează conţinutul acestei variabile. În această locaţie ;de memorie se găseşte offset-ul etichetei here, deci se va efectua saltul la adresa here - ca ;urmare, ultimele 4 instrucţiuni sunt toate echivalente cu jmp here

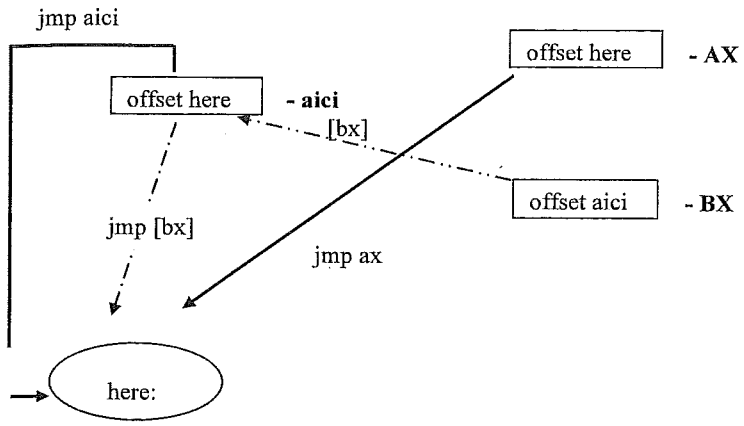


Fig. 4.5. Modalităţi alternative de efectuare a salturilor la eticheta *here*.

```
jmp [ax]      ;eroare de sintaxă: "Illegal indexing mode"
```

;justificare: orice referire de tipul "[reg]" înseamnă interpretarea conţinutului registrului specificat ;drept adresă de memorie al cărei conţinut va fi accesat. Însă conform 2.6.7. şi 3.2.1.4. în formula de ;specificare a unui deplasament (offset) au voie să apară doar regiştri BX sau BP (ca regiştri de ;bază) şi SI sau DI (ca regiştri de index). Deci AX nu poate fi parte a unei astfel de formule de ;calcul şi ca urmare acest fapt este semnalat ca eroare de sintaxă!

```
jmp bx        ;salt (near) la adresa conţinută în BX (adresare registru în mod direct)
```

;în BX avem ca valoare offset-ul lui aici în cadrul segmentului data (vezi mai sus cum a fost ;încărcat BX) – deşi corectă sintactic, o astfel de instrucţiune e o posibilă eroare logică în contextul ;acestui exemplu, deoarece se efectuează un salt în interiorul segmentului code la un offset ;determinat însă relativ la segmentul data!

```
                .     .     .
here:           .     .     .;eticheta destinaţie a salturilor
```

Acest exemplu ilustrează diferenţa dintre operanzii din memorie care reprezintă adrese şi operanzii din memorie care reprezintă valoarea de la o adresă (conţinutul de la acea adresă). Numele de variabile din segmentul de date reprezintă cel mai frecvent valoarea de la acea adresă. În cazul nostru variabila aici a fost utilizată de aşa manieră încât în toate situaţiile în care a apărut, acest nume a desemnat conţinutul de la acea locaţie de memorie (eticheta de date aici ar fi fost interpretată adresă, dacă s-ar fi utilizat de exemplu sub forma mov dx, aici+2). Etichetele codului (în exemplul nostru este vorba de here) reprezintă adresa însăşi.

4.3.2. Instrucţiuni de salt condiţionat

O caracteristică necesară pe care trebuie s-o prezinte orice limbaj de programare este posibilitatea de a lua decizii, ceea ce în plan concret revine la existenţa unor instrucţiuni de salt condiţionat. Elementele care condiţionează acest tip de salt sunt valorile flagurilor şi conţinutul registrului CX.

4.3.2.1. Comparaţii între operanzi

CMP *d,s*	comparaţie valori operanzi (nu modifică operanzii) (execuţie fictivă *d - s*)	OF,SF,ZF,AF,PF şi CF
TEST *d,s*	execuţie fictivă *d* **AND** *s*	OF = 0, CF = 0 SF,ZF,PF - modificaţi AF - nedefinit

Operanzii instrucţiunii **CMP** pot fi de dimensiune octet sau cuvânt, fiind supuşi aceloraşi restricţii de asociere ca şi în cazul instrucţiunii **MOV**.

CMP scade valoarea operandului sursă din operandul destinaţie, dar spre deosebire de instrucţiunea **SUB**, rezultatul nu este reţinut, el neafectând nici una din valorile iniţiale ale operanzilor. Efectul acestei instrucţiuni constă numai în modificarea valorii unor flaguri. Instrucţiunea **CMP** este cel mai des folosită în combinaţie cu instrucţiuni de salt condiţionat.

Efectele execuţiei instrucţiunii **CMP** pot fi utilizate de exemplu atunci când dorim să facem comparaţii matematice. Acestea se pot face *cu semn* sau *fără semn*, în funcţie de configuraţiile de biţi furnizate de către programator ca operanzi.

Instrucţiunile de salt condiţionat se folosesc de obicei în combinaţie cu instrucţiuni de comparare. De aceea, semnificaţiile instrucţiunilor de salt rezultă din semnificaţia operanzilor unei instrucţiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucţiune CMP este de multe ori necesară determinarea relaţiei 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 şi da şi nu! Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 şi este evident mai mare decât 0. Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 şi este mai mic decât 0.

Este valabil şi aici ceea ce am subliniat la prezentarea instrucţiunilor de adunare şi scădere: interpretarea operanzilor şi a rezultatului comparaţiei cu sau fără semn este la latitudinea programatorului! Cum se poate face concret însă această distincţie şi cum se poate ea provoca la nivelul unui program?

Instrucţiunea CMP nu face distincţie între cele două situaţii, deoarece aşa după cum am precizat şi în 4.2.1.1. adunarea şi scăderea se efectuează întotdeauna la fel (adunând sau scăzând configuraţii

binare) indiferent de semnul (interpretarea) acestor configuraţii. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn ***operanzii*** scăderii fictive *d-s*, ci ***rezultatul*** final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparaţiei revine diverselor instrucţiuni de salt condiţionat prezentate în secţiunea următoare (4.3.2.2) care vor prezenta categorii diferite de instrucţiuni pentru cele două tipuri posibile de interpretări.

Din semnificaţia flagurilor rezultă că pentru determinarea relaţiei de ordine între operanzi, după execuţia instrucţiunii CMP sunt semnificative doar OF, SF, ZF şi CF. Această problemă este detaliată în secţiunea următoare, unde vom vedea că instrucţiunile de salt condiţionat diferă tocmai prin condiţia asupra flag-urilor care este verificată de către fiecare instrucţiune în parte.

Instrucţiunea **TEST** este utilă pentru examinarea stării anumitor biţi. În exemplul de mai jos se transferă controlul la eticheta Acolo dacă biţii 1 şi 5 ai registrului AL sunt 0. Starea celorlalţi biţi este ignorată:

```
            test  al, 00100010b     ;mascare prin AND
            jz    Acolo             ;salt dacă s-a obţinut 0
Nuezero: .  .     .
            .     .     .
Acolo: .    .     .
```

4.3.2.2. Salturi condiţionate de flaguri

În tabelul 4.1. prezentăm instrucţiunile de salt condiţionat împreună cu semnificaţia lor şi cu precizarea valorilor flagurilor în urma cărora se execută salturile respective. Precizăm că pentru toate instrucţiunile de salt sintaxa este aceeaşi, şi anume

<instrucţiune_de_salt> etichetă

Semnificaţia instrucţiunilor de salt condiţionat este dată sub forma ***"salt dacă operand1 <<relaţie>> faţă de operand2"*** (unde cei doi operanzi sunt obiectul unei instrucţiuni anterioare CMP sau SUB) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă şi din condiţiile ce trebuie verificate, instrucţiunile ce se află într-o aceeaşi linie a tabelului sunt echivalente.

Când se compară două numere cu semn se folosesc termenii **"less than"** (mai mic decât) şi **"greater than"** (mai mare decât), iar când se compară două numere fără semn se folosesc termenii *"below"* (inferior, sub) şi respectiv *"above"* (superior, deasupra, peste).

Instrucţiunile de salt condiţionat efectuează întotdeauna salturi "scurte", adică adresa de destinaţie nu poate fi la distanţă mai mare de 127 octeţi faţă de instrucţiunea curentă. De exemplu, în urma asamblării secvenţei

MNEMONICA	SEMNIFICAŢIE (salt dacă..<<relaţie>>)	Condiţia verificată
JB JNAE JC	este inferior nu este superior sau egal există transport	CF=1
JAE JNB JNC	este superior sau egal nu este inferior nu există transport	CF=0
JBE JNA	este inferior sau egal nu este superior	CF=1 sau ZF=1
JA JNBE	este superior nu este inferior sau egal	CF=0 şi ZF=0
JE JZ	este egal este zero	ZF=1
JNE JNZ	nu este egal nu este zero	ZF=0
JL JNGE	este mai mic decât nu este mai mare sau egal	SF≠OF
JGE JNL	este mai mare sau egal nu este mai mic decât	SF=OF
JLE JNG	este mai mic sau egal nu este mai mare decât	ZF=1 sau SF≠OF
JG JNLE	este mai mare decât nu este mai mic sau egal	ZF=0 şi SF=OF
JP JPE	are paritate paritatea este pară	PF=1
JNP JPO	nu are paritate paritatea este impară	PF=0
JS	are semn negativ	SF=1
JNS	nu are semn negativ	SF=0
JO	există depăşire	OF=1
JNO	nu există depăşire	OF=0

Tabelul 4.1. Instrucţiunile de salt condiţionat


```
Aici: . .     .
        DB    1000  DUP (?)
        .     .     .
        dec   ax
        jnz   Aici
```

se va semnala eroare, deoarece eticheta Aici se află la peste 1000 de octeţi distanţă faţă de **jnz**. Într-un astfel de caz trebuie procedat în felul următor:

```
Aici: . .     .
        DB    1000  DUP (?)
        .     .     .
              dec   ax
              jz    Acolo
              jmp   Aici
Acolo:.       .     .
```

folosindu-se deci o instrucţiune de salt condiţionat pentru a lua decizia dacă trebuie sau nu făcut un salt necondiţionat la eticheta Aici (un salt necondiţionat nu este supus nici unei restricţii referitoare la distanţa de salt).

Pentru a facilita alegerea corectă de către programator a variantelor de salt condiţionat în raport cu rezultatul unei comparaţii (adică, dacă programatorul doreşte interpretarea rezultatului comparaţiei cu semn sau fără semn) dăm următorul tabel:

Relaţia între operanzi ce se doreşte a fi testată	Comparaţie cu semn	Comparaţie fără semn
d = s	JE	JE
d ≠ s	JNE	JNE
d > s	JG	JA
d < s	JL	JB
d ≥ s	JGE	JAE
d ≤ s	JLE	JBE

Tabelul 4.2. Alegerea corectă a instrucţiunilor de comparare în funcţie de relaţia testată.

Comparând acest tabel cu tabelul 4.1 se poate deduce modul de poziţionare a flagurilor de către instrucţiunea CMP. Studiul acestui tabel reiterează afirmaţia noastră anterioară: nu instrucţiunea CMP este cea care face distincţie între o comparaţie cu semn şi una fără semn! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparaţiei revine NUMAI instrucţiunilor de salt condiţionat specificate ULTERIOR comparaţiei efectuate.

Tabelul de mai sus îl considerăm foarte util pentru interpretarea rezultatelor instrucţiunilor aritmetice în general. Fără a mai efectua o instrucţiune CMP, considerând *d* rezultatul ultimei instrucţiuni aritmetice executate şi punând *s=0*, tabelul rămâne valabil.

4.3.2.3. Exemple comentate

Modul de acţiune a instrucţiunilor de salt condiţionat şi instrucţiunea **cmp**

```
        mov al,80h   ;al := 128 = 10000000b = -128 ! (Interesant! – remarcăm faptul că datorită
                     ;regulilor de reprezentare în cod complementar 128 şi -128 au aceeaşi
                     ;reprezentare binară şi anume 10000000b!)

(*)     cmp al,0     ;instrucţiunea cmp nu interpretează în nici un fel valoarea din AL (ca fiind
                     ;cu semn sau fără semn) ci doar realizează scăderea fictivă al-0 şi afectează
                     ;corespunzător flagurile: SF=1 , CF=ZF=OF=PF=AF=0.

        jl et        ;utilizarea instrucţiunii JL (Jump if Less than) provoacă interpretarea
                     ;comparaţiei al<0 cu semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se
                     ;testează dacă SF≠OF şi cum SF=1 iar OF=0 se decide îndeplinirea condiţiei
                     ;şi saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a
                     ;stat la latitudinea programatorului care prin utilizarea instrucţiunii JL a
                     ;decis că doreşte să compare -128 cu 0 şi cum -128 este "less than" 0
                     ;condiţia a fost îndeplinită (echiv. cu jnge et). În contrast, jnl et sau jge
                     et ;(care vor testa dacă SF=OF) NU vor fi îndeplinite şi NU vor provoca
                     saltul ;la eticheta specificată.

        jb et        ;utilizarea instrucţiunii JB (Jump if Below) provoacă interpretarea
                     ;comparaţiei al<0 fără semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se
                     ;testează dacă CF=1 şi cum CF=0 se decide neîndeplinirea condiţiei deci nu
                     ;se va face saltul la eticheta et. Deducem deci că interpretarea valorilor
                     comparate a stat la latitudinea programatorului care prin utilizarea
                     instrucţiunii JB a decis că doreşte să compare 128 cu 0 şi cum 128 NU este
                     "below" 0 condiţia NU a fost îndeplinită (echivalent cu jnae et sau jc et).

        jae et1      ;se testează fără semn dacă al ≥ 0 (128 ≥ 0?) - CF=0 deci condiţie
                     ;îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la
                     ;eticheta et1

        jbe et2      ;se testează fără semn dacă al ≤ 0 (128 ≤ 0?) – CF = ZF = 0 deci condiţia
                     ;(CF=1 sau ZF=1) NU este îndeplinită şi 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 fără semn dacă al > 0 (128 > 0?) – CF = ZF = 0 deci condiţia
                     ;(CF=0 şi ZF=0) este îndeplinită şi ca urmare se va face saltul la eticheta et3
                     ;(echivalent cu jnbe et3) şi 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, condiţia 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 şi vor provoca
                     ;saltul la eticheta specificată.

        jle et5      ;se testează cu semn dacă al ≤ 0 (-128 ≤ 0?) – OF = ZF = 0 şi SF=1
                     deci ;condiţia (ZF=1 sau SF≠OF) este îndeplinită şi ca urmare se va face
                     saltul la ;eticheta et5 (echivalent cu jng et5) şi rezultat consistent cu jl et,
                     deoarece ;jle implică jl.

        jg et6       ;se testează cu semn dacă al > 0 (-128 > 0?) – OF = ZF = 0 şi SF=1 deci
                     ;condiţia (ZF=0 şi SF=OF) NU este îndeplinită şi ca urmare NU se va face
                     ;saltul la eticheta et6 (echivalent cu jnle et6) şi 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 condiţie 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 şi saltul se va efectua.

        jo et8       ;se testează dacă OF=1 - OF=0 deci condiţie neîndeplinită – nu se efectuează
                     ;saltul (nu există depăşire). În contrast, jno et8 (care testează dacă OF=0)
                     va ;fi îndeplinită şi saltul se va efectua.

        js et9       ;se testează dacă în interpretarea cu semn rezultatul comparaţiei are semn
                     ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucţiunii
                     ;CMP, nu este vorba de a interpreta cu semn sau fără semn operanzii
                     ;scăderii fictive d-s, ci rezultatul final al acesteia !) adică testăm dacă SF=1 -
                     ;condiţie îndeplinită în cazul nostru şi ca urmare saltul se va efectua !. În
                     ;contrast, jns et9 (care testează dacă SF=0) NU va fi îndeplinită şi saltul
                     ;NU se va efectua.

        cmp 0,al     ;eroare de sintaxă : "Illegal immediate" deoarece sintaxa instrucţiunii cmp
                     ;interzice specificarea ca prim operand a unei valori imediate (constante).
                     ;dacă totuşi dorim forţarea unei comparaţii de acest tip (0-al) putem utiliza
                     ;pe post de prim operand un registru iniţializat cu valoarea 0.
        mov bl,0
```

cmp bl, al ;realizează scăderea fictivă bl-al (0-al = 0-80h = 0-10000000b = ;10000000b) şi afectează corespunzător flagurile: CF=SF=OF=1, ;ZF=PF=AF=0.

Exerciţiu propus: Reluaţi discuţia efectului tuturor instrucţiunilor de salt condiţionat de mai sus (analizate pentru cazul comparaţiei (*) cmp al,0) în condiţiile în care această comparaţie e înlocuită de ultimele două instrucţiuni prezentate, adică în cazul în care se efectuează cmp bl,al cu bl=0.

Care ar fi însă justificarea faptului că în cazul cmp bl,al avem CF = OF = SF = 1 iar în cazul *cmp al,0* doar SF=1 iar CF = OF = 0 ?

Pentru a justifica modurile de setare diferite ale flag-urilor trebuie să luăm în discuţie regulile practice de setare a acestor flag-uri. Aceste reguli generale sunt:

- SF ia valoarea bitului de semn al rezultatului obţinut;
- CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obţinut a provocat (CF=1) sau nu (CF=0) un transport în afara spaţiului de reprezentare; dacă e vorba despre o scădere *d-s*, avem: dacă $|d| \geq |s|$ atunci CF=0 (nu e nevoie de cifră de împrumut pentru efectuarea scăderii) iar dacă $|d| < |s|$ atunci CF=1 (este nevoie de cifră de împrumut pentru efectuarea scăderii şi acest lucru se reflectă în CF)
- OF este setat la valoarea 1 dacă există depăşire în interpretarea cu semn a rezultatului ("*OF is set if there exists a signed overflow*"), adică dacă rezultatul obţinut nu se încadrează în intervalul de interpretare admis (acesta fiind [-128..+127] dacă este vorba despre octeţi şi respectiv [-32768..+32767] pentru cuvinte interpretate cu semn).

Ultimele două reguli derivă de fapt din modul de implementare a conceptului de **depăşire** (*overflow*) la nivelul procesorului 80x86.

În cazul operaţiilor/operanzilor fără semn depăşirea va fi semnalată prin setarea indicatorului CF (*carry flag*). În cazul operaţiilor/operanzilor cu semn depăşirea va fi semnalată prin setarea indicatorului OF (*overflow flag*).

Cum să detectăm însă situaţiile de depăşire în cazul operaţiilor de adunare şi scădere ? Care sunt regulile practice de aplicat pentru a înţelege şi a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ? În discuţiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (*overflow flag*) deoarece şi datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situaţii din partea programatorilor ca fiind depăşire sau nu.

Atragem însă atenţia asupra a ceea ce se ignoră de multe ori în acest context şi anume faptul că o situaţie de tipul CF=1 (cu OF=0) semnalează la rândul ei o depăşire, însă pentru cazul numerelor interpretate fără semn.

Pentru ADUNARE: dacă se adună două numere de acelaşi semn şi rezultatul este de semn diferit atunci se semnalează depăşire (OF=1), **în caz contrar nu** (OF=0). Aceasta este deci ceea ce am putea numi *regula depăşirii la adunare* (RDA) în cazul interpretării cu semn.

De exemplu, la nivel de octet, dacă vom considera adunarea 100 + 50 = 150 vom obţine depăşire (!) cu semn (pare surprinzător, nu-i aşa ?). Justificare: 100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b). Operanzii au acelaşi semn dar rezultatul este de semn diferit, deci conform RDA vom avea OF=1. Intuitiv, depăşirea se poate justifica prin faptul că 150 $\notin$ [-128..127] deci se obţine o eroare de tip *"out of range"*. Deşi s-ar putea replica faptul că 150 = 10010110b = -106 (în interpretarea cu semn), iar -106 $\in$ [-128..127], această ultimă interpretare nu poate fi acceptată deoarece operanzii (100 şi 50) au valori pozitive în ambele interpretări (bitul de semn fiind 0). Ca urmare, suma a două numere pozitive nu poate da un număr negativ şi astfel singura interpretare ce poate fi acceptată în acest context pentru 10010110b este 150 $\notin$ [-128..127] deci se setează OF=1.

Pe de altă parte, CF = 0 (nu există cifră de transport în afara spaţiului de reprezentare) deci nu avem depăşire în interpretarea fără semn: rezultatul adunării 100 + 50 = 150 $\in$ [0, 255] (intervalul de interpretare admis pentru numere fără semn).

Analog, în interpretarea cu semn, suma a două numere negative nu poate furniza un număr pozitiv. Luăm exemplul:

```
  10010110 +
  10000010
 1 00011000
```

Se observă din reprezentarea binară că există un transport de cifră 1 în afara spaţiului de reprezentare admis al celor 8 biţi, deci intuitiv este suficient de justificat depăşirea. Din punct de vedere al aplicării RDA obţinem pe 8 biţi în interpretarea cu semn că suma a două numere negative (ele sunt negative deoarece bitul de semn este 1 pentru ambele numere) ar trebui să furnizeze un număr pozitiv: 00011000b = 18h = 24. Această valoare este de fapt o trunchiere a valorii binare corecte (pe 9 biţi!) ce ar fi trebuit obţinută (100011000b = 118h), iar trunchierea are loc tocmai datorită depăşirii. Ca urmare, nu se poate obţine un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depăşire!). Se observă că o astfel de trunchiere înseamnă întotdeauna şi apariţia unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat şi CF=1.

10010110b = 96h = -106 (în interpretarea cu semn) = +150 (în interpretarea fără semn)
10000010b = 82h = -126 (în interpretarea cu semn) = +130 (în interpretarea fără semn)

În interpretarea fără semn avem 150 + 130 = 280 $\notin$ [0..255] (justificarea intuitivă a depăşirii). Tehnic, am văzut deja că CF = 1 şi rezultă astfel clar că avem depăşire în interpretarea fără semn.

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 depăşirii în acest tip de situaţie este:

În interpretarea cu semn avem -106 + (-126) = -232 ∉ [-128..127] deci OF=1.

Această ultimă motivaţie este mai intuitivă pentru justificarea depăşirii 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 aşa... (şi iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depăşire. De asemenea, dacă adunăm două numere de acelaşi semn dar rezultatul are acelaşi semn cu operanzii nu se va semnala nici în acest caz depăşire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeaşi dimensiune ca şi cea a operanzilor). Se poate verifica uşor din punct de vedere matematic că în nici unul din aceste cazuri nu ieşim din intervalul de interpretare admis.

Pentru SCĂDERE: se interpretează **operanzii** respectivi cu semn, se efectuează scăderea solicitată asupra configuraţiilor corespunzătoare de biţi şi dacă rezultatul obţinut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul [-128..127] pentru octeţii cu semn şi respectiv [-32768..32767] pentru cuvinte interpretate cu semn) atunci se semnalează depăşire (*overflow*) şi astfel OF=1. Această formulare o putem numi *regula depăşirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depăşirii 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 semnatic drept "depăşire la scădere în interpretarea fără semn".

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum şi impactul lor asupra modului de setare al flag-urilor.

Exemple:

i).
```
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 şi 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 şi 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 depăşire 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 şi cum -168 ∉ [-128..127] se semnalează *signed overflow* şi 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 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn, înţeleasă 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 şi cum +168 ∉ [-128..127] se semnalează *signed overflow* şi 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 conţinutul lui AH este
            ;126 atât în interpretarea cu semn cât şi în cea fără semn)
mov bh,2ah  ;2ah = 42 (atât în interpretarea cu semn cât şi 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 şi 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 depăşire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică ah-bh = 126 – 42 = 84 şi cum 84 ∈ [-128..127] NU se semnalează *signed overflow* şi 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

scăderi de tipul (256 + 42) – 126 = 172 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn prin setarea flagului carry)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică bh-ah = 42-126 = -84 şi cum -84 ∈ [-128..127] NU se semnalează *signed overflow* şi ca urmare OF=0)

Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii a-b ∈ [-127..127] atunci şi b-a ∈ [-127..127] (situaţia particulară în care a-b = -128 o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului [-32767..32767] cu discuţie asupra cazului particular -32768. Ca urmare se poate concluziona faptul că instrucţiunile **cmp a,b** şi **cmp b,a** vor furniza întotdeauna aceeaşi valoare pentru OF.

iii). - discuţie asupra cazurilor cmp 80h,0 şi cmp 0,80h

```
mov ah,80h  ;80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)
            ; = 10000000b (bitul de semn fiind 1 cele două interpretări diferă)
mov bh,0    ;bh:=0

cmp ah,bh   ;se realizează scăderea fictivă ah-bh= 10000000b-00000000b = 10000000b
            ; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)
```

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul 80h = 10000000b este 1)
CF = 0 (deoarece |80h| > |0| nu se pune problema unei scăderi cu împrumut de cifră, deci nu poate fi vorba despre depăşire în interpretarea fără semn)
OF = 0 (se efectuează scăderea în interpretarea cu semn, adică ah-bh = -128 – 0 = -128 şi cum -128 ∈ [-128..127] NU se semnalează *signed overflow* şi ca urmare OF=0)

```
cmp bh,ah   ;se realizează scăderea fictivă bh-ah= 00000000b-10000000b = 10000000b
            ; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)
```

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul 80h = 10000000b este 1)
CF = 1 (deoarece |0h| < |80h| se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine 0 – 128 = 128 (!) provenită de fapt din necesitatea unei scăderi de tipul (256 + 0) – 128 = 128 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn prin setarea flagului carry)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică bh-ah = 0 – (-128) = +128 şi cum +128 ∉ [-128..127] se semnalează *signed overflow* şi ca urmare OF=1)

CF = 1 în cazul **cmp 0,80h** deoarece se efectuează o scădere cu împrumut de tipul :


```
0 - 10000000b  =   1 00000000 –
                   10000000
                   010000000
```

şi cifra de împrumut se transferă în CF.

Să analizăm în acest context ce înseamnă şi cum s-a ajuns la domeniul "numerelor cu semn posibil a fi reprezentate pe 1 octet" respectiv domeniul "numerelor cu semn posibil a fi reprezentate pe 1 cuvânt".

Pe 1 octet se pot reprezenta 256 de valori, indiferent că vorbim despre interpretarea cu semn sau interpretarea fără semn. În interpretarea fără semn aceste valori sunt cele din intervalul [0..255]. Care sunt însă cele 256 de valori reprezentabile în interpretarea cu semn ? Este vorba despre intervalul [-128..127] sau despre intervalul [-127..128] ? Pentru că nu poate fi vorba despre intervalul [-128..128] deoarece în acest interval sunt 257 de valori ! Cu alte cuvinte cineva a trebuit să aleagă una dintre cele două variante şi totodată să facă precizarea că numerele -128 şi +128 nu pot coexista între limitele aceluiaşi interval de reprezentare al aceluiaşi tip de dată! (reamintim că în limbaj de asamblare *tip de dată = dimensiune de reprezentare*)

În acest sens este de observat şi impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât **shortint** cât şi **byte** în Turbo Pascal acceptă valoarea 80h (-128 ca *shortint* şi +128 ca *byte*) însă 80h **nu poate avea două interpretări distincte în cadrul aceluiaşi tip de dată !** Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 şi +128 ca fiind prezente în cadrul aceluiaşi tip de dată !

Ca urmare, s-a luat decizia ca intervalul acceptat al valorilor cu semn reprezentabile pe 1 octet să fie intervalul [-128..+127] (care este exact domeniul de valori şi a tipului de dată **shortint** din Turbo Pascal): deci **+128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet !**

Totuşi, după cum putem verifica foarte uşor, instrucţiunile **mov ah, 128** şi **mov ah,-128** sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în *ah* a configuraţiei binare 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fără semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea cu semn. Simpla încărcare a unui registru cu o anumită configuraţie binară nu presupune şi necesitatea interpretării respectivei configuraţii într-un anumit fel. Sarcina interpretării acelei configuraţii drept cu semn sau fără semn va cădea în sarcina instrucţiunilor ce urmează şi care vor folosi ca operanzi aceste valori. De exemplu, utilizarea lui IMUL în loc de MUL va provoca interpretarea configuraţiei binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui DIV în loc de IDIV va provoca interpretarea aceluiaşi operand ca fără semn ş.a.m.d.

În cazul **cmp 80h,0** se efectuează 80h–0 = 80h = 10000000b (128 - 0 = 128 în interpretarea fără semn) fără a fi nevoie de o cifră de transport împrumutată pentru a putea efectua scăderea, deci nu avem depăşire în interpretarea fără semn şi astfel CF = 0. În interpretarea cu semn a operanzilor şi a rezultatului final avem -128 - 0 = -128 ∈ [-128..127] deci nu avem depăşire nici în interpretarea cu semn şi astfel OF = 0.

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 şi 1 (şi apoi 0 şi -1) şi ce efecte are asupra flagurilor instrucţiunea cmp în fiecare dintre situaţii.

Situaţia **cmp 1,0** (evidenţiată 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 discuţiilor din exemplele anterioare.

Situaţia **cmp 0,1** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,1 cu ah=0) va efectua scăderea fictivă 0-1 = -1 = 11111111b:

```
0 - 00000001b  =   1 00000000 –
                     00000001
                   0 11111111
```

Efectul asupra flag-urilor va fi CF = SF = PF = AF = 1 şi ZF = OF = 0. Justificarea valorilor din CF şi SF este şi aici evidentă pe baza discuţiilor din exemplele anterioare iar OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar -1 ∈ [-128..127].

Situaţia **cmp -1,0** (evidenţiată 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 şi 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 ∈ [-128..127]. CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situaţia **cmp 0,-1** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,-1 cu ah = 0) va efectua scăderea fictivă 0 – (-1) = +1 = 00000001b:

```
0 - 11111111b  =   1 00000000 –
                     11111111
                   0 00000001
```

Efectul asupra flag-urilor va fi CF = AF = 1 şi OF = SF = ZF = PF = 0. SF = 0 deoarece bitul de semn este 0. OF=0 deoarece 0 – (-1) = +1 ∈ [-128..127]. CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica şi aşa: 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 şi astfel se semnalează depăşire în cazul interpretării fără semn, deci CF = 1.

v). Cazurile studiate anterior (i-iv) s-au referit la operaţii de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucţiunii cmp. Să analizăm în continuare şi cazul unei depăşiri furnizate de operaţia de adunare revenind astfel la discuţia asupra aplicării regulii RDA:

```
mov ah,126  ;126 = 01111110b = 7eh (aceeaşi 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 spaţiului de reprezentare al rez.
```

SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operaţiei efectuate este strict negativ = -128).

OF = 1 deoarece:
- justificare *tehnică* - conform RDA se adună două numere de acelaşi 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 ∉ [-128..127] deci se semnalează *signed overflow* şi ca urmare OF=1.

vi). Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situaţia în care programatorul îşi iniţializează operanzii cu anumite valori iniţiale dorite (cu semn sau fără semn, conform necesităţilor problemei în cauză) şi se aşteaptă la obţinerea unor rezultate sau reacţii în conformitate cu valorile furnizate. Atenţie însă! De obicei aceste valori au o dublă interpretare posibilă şi nu vor fi interpretate în orice situaţie sub forma furnizată la iniţializare!

Utilizarea ulterioară a unor instrucţiuni care forţează prin modul de lor de acţiune interpretarea complementară (cu semn/fără semn) celei de la iniţializare poate provoca apariţia unor situaţii 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 ţine cont în permanenţă de dubla interpretare posibilă a configuraţiilor 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)
             (şi se setează corespunzător OF=ZF=0 şi CF=SF=1)
```

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 aşa cum precizează valorile de la iniţializare? 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 comparaţiei efectuate în baza 10 nu este

dedus din acţiunea instrucţiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucţiuni ulterioare care vor avea ele rolul de a interpreta în unul din cele 4 moduri de mai sus comparaţia efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucţiunilor corespunzătoare de salt condiţionat:

jl et1 ; evident că 200 $\not<$ -1 deci la prima vedere pare că nu este îndeplinită condiţia necesară pentru efectuarea saltului... să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparaţiei ca fiind cu semn (deci -55) aceasta însemnând implicit şi faptul că scăderea este interpretată ca (-56 – (-1)) deci şi operanzii vor fi amândoi interpretaţi cu semn... cum -56 < -1 iată că şi intuitiv condiţia se verifică (pe lângă justificarea tehnică a îndeplinirii condiţiei de salt SF≠OF) şi deci saltul se va efectua ! Deci chiar dacă programatorul a furnizat la iniţializare valorile 200 şi -1, utilizarea instrucţiunii JL a provocat interpretarea comparaţiei ca fiind între -55 şi -1 şi nu între 200 şi -1! (explicaţia de aici şi faptul că saltul se va efectua vă poate ajuta să "demonstraţi" unor colegi cum 200 poate fi mai mic decât -1 !!!)

ja et2 ; deoarece 200 > -1 în acest caz ne-am aştepta ca saltul să se efectueze... însă utilizarea instrucţiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparaţie corectă aici este comparaţia lui 200 cu 255 şi cum 200 $\not>$ 255 condiţia nu este îndeplinită şi deci saltul nu se va efectua (iată deci cum se poate "demonstra" că 200 nu este superior valorii -1 !!!). Ca o confirmare, se poate vedea că nici condiţia tehnică impusă de JA nu este îndeplinită: ar trebui să avem CF=ZF=0, însă în cazul nostru CF=1 deci saltul nu se va efectua.

jb et3 ; intuitiv 200 < 255, iar tehnic CF=1 deci saltul se efectuează
jg et4 ; intuitiv -56 $\not>$ -1, iar tehnic deşi ZF=0 nu este îndeplinită şi condiţia SF = OF deci
; saltul nu se va efectua

Ca urmare din cele 4 situaţii teoretic posibile de mai sus, vom întâlni concret numai două:
- comparaţie fără semn (200 cu 255) - impusă de "above" sau "below"
- comparaţie cu semn (-56 cu -1) – impusă de "less than" sau "greater than"

Nu putem aşadar compara de fapt pe 200 cu -1 aşa cum au fost specificate valorile la iniţializare şi nici pe -56 cu 255 deoarece **interpretarea este ori cu semn ori fără semn pentru ambii operanzi!**

vii). Am studiat în exemplele anterioare modalitatea de reacţie (de interpretare) a procesorului 80x86 legată de noţiunea de depăşire în cazul operaţiilor de adunare şi de scădere. Când şi cum semnalează însă procesoarele din familia 80x86 depăşirea la înmulţire şi respectiv la împărţire ?

"Depăşirea" la înmulţire. Instrucţiunile MUL şi IMUL setează CF=1 şi OF=1 dacă "jumătatea" superioară a produsului (octetul superior dacă este vorba despre produs-cuvânt sau cuvântul superior dacă este vorba despre produs-dublucuvânt) este o valoare diferită de zero. Aceasta este definiţia noţiunii de "depăşire la înmulţire" în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincţie între MUL şi IMUL şi de aceea nici între CF şi OF. Ori vor fi amândouă flag-

urile setate la valoarea 1 cu semnificaţia de "depăşire la înmulţire" în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biţi:

```
mov al, 5
mov bl,170
mul  bl        ;AX := AL * BL = 5 * 170 = 850 = 0352h şi vom avea CF=1 şi OF=1
               ;deoarece octetul superior AH = 03 ≠ 0.
```

Varianta cu IMUL va furniza:

```
mov al, 5
mov bl,170     ;170 = 0aah = -86 în interpretarea cu semn
imul bl        ;AX := AL * BL = 5 * (-86) = - 430 = 0fe52h şi vom avea CF=1 şi
               ;OF=1 deoarece octetul superior AH = 0feh ≠ 0.
```

În cazul unor operanzi pe 16 biţi putem avea de exemplu:

```
val1 DW 2000h
val2 DW 0100h
...
mov ax, val1
mul val2       ;DX:AX = 00200000h şi vom avea CF=1 şi OF=1 deoarece jumătatea
;superioară a produsului DX:AX, adică registrul DX conţine valoarea 0020h ≠ 0.
```

Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potenţială pierdere de informaţie ca şi în cazul celorlalte depăşiri - adunare, scădere sau împărţire. Aceasta deoarece chiar dacă înmulţim valorile maximale posibil a fi reprezentate pe dimensiunea operanzilor (255 * 255 pentru octeţi şi respectiv 65535 * 65535 pentru cuvinte) tot nu se depăşeşte dublul dimensiunii de reprezentare a operanzilor, adică spaţiul pe care îl avem oricum la dispoziţie prin definiţie, deoarece 255 * 255 = 65025 < 65535 (numărul maximal fără semn reprezentabil pe un cuvânt) iar 65535 * 65535 = 4 294 836 225 < 4 294 967 295 (numărul maximal fără semn reprezentabil pe un dublucuvânt).

În cazul înmulţirii cu semn (instrucţiunea IMUL) justificarea este similară: 127 * 127 = 16129 < 32767 (numărul maximal cu semn ce poate fi reprezentat pe 1 cuvânt), iar 32767 * 32767 = 1 073 676 289 < 2 147 483 647 (numărul maximal cu semn reprezentabil pe un dublucuvânt).

Depăşirea în cazul înmulţirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanzi octeţi (respectiv cuvinte) produsul nu încape tot într-un octet (respectiv într-un cuvânt) ci este realmente nevoie de o dimensiune dublă pentru memorarea rezultatului. În acest sens, a se vedea şi capitolul 1, în care din punct de vedere matematic s-a specificat clar că înmulţirea nu provoacă de fapt depăşire, tocmai din cauza alocării unui spaţiu suficient pentru reprezentarea produsului. În concluzie, se poate spune că din punct de vedere matematic singura operaţie care nu provoacă depăşire este înmulţirea, însă procesoarele 80x86 promovează totuşi noţiunea de "depăşire la înmulţire" pentru a diferenţia între situaţiile în care produsul încape într-un spaţiu de dimensiunea operanzilor şi în care nu.

Situaţiile în care produsul încape pe dimensiunea operanzilor vor fi caracterizate de setările CF = OF = 0 (nu avem deci depăşire la înmulţire). Iată un exemplu:

```
mov al, 5
mov bl, 51
mul bl        ; AX := AL * BL = 5 * 51 = 255 = 00ffh şi vom avea CF=0 şi OF=0
              ; deoarece octetul superior AH = 0.
```

Depăşirea la împărţire. În cazul împărţirii, specificarea acestei operaţii sub forma

(I)DIV *operand*

presupune că operandul specificat este împărţitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biţi) iar deîmpărţitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărţitorul este cuvânt). Efectuarea operaţiei are ca efect:

AX : operand pe 8 biţi = câtul în AL şi restul în AH;
DX:AX / operand pe 16 biţi = câtul în AX şi restul în DX;

În cazul împărţirii depăşirea apare atunci când rezultatul împărţirii nu încape în spaţiul rezervat conform definiţiei pentru reprezentare, mai exact, când câtul nu încape în AL sau respectiv AX. Într-o astfel de situaţie, procesorul 80x86 emite o întrerupere 0, execuţia 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 funcţie de tipul de procesor şi/sau de SO instalat). Pare ciudat la prima vedere că o împărţire 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 împărţire care matematic se poate efectua. Secvenţa

```
mov ax,60000
mov bl,2
div bl
```

ar trebui să furnizeze din punct de vedere matematic câtul 30000. Însă conform definiţiei împărţirii DIV acest cât 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 împărţirea de mai sus nu se poate nici ea efectua (similar cu o situaţie de tip div 0) şi ca urmare înţelegem acum decizia proiectanţilor de a trata tot prin emiterea unei întreruperi 0 şi o situaţie de genul celei de mai sus. Să remarcăm în acest sens şi faptul că mesajul "*Divide overflow*" (depăşire la împărţire) este acceptat în acest context ca similar unui "*Divide by zero*".

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentaţi este de a confunda exprimările *"numere cu semn"* şi *"numere fără semn"* cu exprimările *"numere negative"* şi 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 enunţul unei probleme care cere efectuarea unei anumite acţiuni *"dacă numărul v este (strict) negativ"*? În primul rând vom concluziona 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 configuraţiei 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 şi testăm valoarea sa printr-o instrucţiune adecvată de salt condiţionat. Secvenţa

```
mov al,v          ;pentru a nu afecta destructiv conţinutul variabilei v
shl al,1          ;shift stânga cu 1 poziţie 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 instrucţiunea **cmp** pentru o comparaţie 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 ca la nivelul efectuării operaţiilor de adunare sau scădere procesorul 80x86 nu diferenţiază între adunări/scăderi cu semn sau fără semn (tehnic vorbind ele se efectuează drept operaţii binare cu rezultat interpretabil **ulterior** drept cu semn sau fără). Totuşi, în momentul în care se pune problema exprimării în baza 10 a unei operaţii de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect operanzii operaţiei respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obţinut ? Mai concret:

```
     00000101 +     (= 5 în ambele interpretări)
     11111110       (= 254 fără semn şi -2 în interpretarea cu semn)
(1)  00000011       (= 3 în ambele interpretări ale configuraţiei pe 8 biţi)
```

reprezintă 5 + 254 = 259 (= 1 00000011 – configuraţie pe 9 biţi !) sau reprezintă 5 + (-2) = 3 ? După cum vom vedea şi aici răspunsul este că putem interpreta în ambele moduri şi să justificăm astfel ca două reacţii separate modul de setare al flag-urilor CF şi 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 consecinţă tehnică a

modului de efectuare a operaţiei binare de adunare). Ca urmare în interpretarea fără semn avem depăşire (evident, deoarece 259 > 255, adică decât numărul maxim reprezentabil pe 1 octet).

Ce se întâmplă cu OF ? Rularea secvenţei

```
mov al, 5     ; = 5 în ambele interpretări
mov bl, 254   ; = -2 în interpretarea cu semn
add al, bl    ; AL := AL+BL = 5+(-2) = 3
```

nu setează flagul OF la valoarea 1, deci situaţia de mai sus nu este considerată "depăşire" în interpretarea cu semn! Din punct de vedere al justificării modului de setare a flag-ului OF secvenţa de mai sus ar fi mai corectă dacă ar fi scrisă:

```
mov al, 5
mov bl, -2
add al, bl    ; deci 5 + (-2) = 3
```

şi este evident că în această interpretare nu este vorba despre nici o depăşire (şi de aceea şi OF = 0).

Să ne reamintim în acest context şi exemplele date la prezentarea RDA şi RDS de la paginile ??-??: adunarea 100 + 50 = 150 va semnala depăşire (*signed overflow* - conform RDA), iar scăderile 130 - 42 (interpretată ca -126 - 42 = -168 ∉ [-128..127]) şi 42 -130 (interpretată ca scăderea 42-(-126) = +168 ∉ [-128..127]) produc la rândul lor *signed overflow* şi ca urmare OF=1.

4.3.3. Instrucţiuni de ciclare

Cu ajutorul instrucţiunilor de salt condiţionat se pot construi cicluri (bucle). Un ciclu nu este altceva decât un bloc de instrucţiuni care se termină cu o instrucţiune de salt condiţionat (în acest sens se cunosc foarte bine construcţiile de tip **for, while** şi **repeat** din unele limbaje de nivel înalt).

Limbajul de asamblare 80x86 prevede instrucţiuni speciale pentru realizarea ciclării. Ele sunt: **LOOP, LOOPE, LOOPNE** şi **JCXZ**. Sintaxa lor este

<instrucţiune> etichetă

Instrucţiunea **LOOP** comandă reluarea execuţiei blocului de instrucţiuni ce începe la *etichetă*, atâta timp cât valoarea din registrul CX este diferită de 0. Se efectuează întâi decrementarea registrului CX şi apoi se face testul şi eventual saltul. Saltul este "scurt" (max. 127 octeţi - atenţie deci la "distanţa" dintre LOOP şi etichetă!).

Ca exemplu să luăm tipărirea caracterelor unui şir dat:


```
data segment
        Test  DB     'Acesta este un exemplu'
        SF    LABEL  BYTE
        .     .      .
cod segment
        .     .      .
        mov   cx, SF-Test
        mov   bx,OFFSET Test
Tipareste:
        mov   dl,[bx]        ;preia următorul caracter
        inc   bx             ;punctează spre următorul caracter
        mov   ah,2           ;funcţia de tipărire a unui caracter
        int   21h            ;activarea tipăririi
        loop Tipareste       ;decrementează CX şi revine la  eticheta Tipareste
                             ;dacă mai sunt caractere
```

În cazul în care condiţiile de terminare a ciclului sunt mai complexe se pot folosi instrucţiunile **LOOPE** şi **LOOPNE**.

Instrucţiunea **LOOPE** (***LOOP** while **Equal***) diferă faţă de LOOP prin condiţia de terminare, ciclul terminându-se fie dacă CX=0, fie dacă ZF=1. În cazul instrucţiunii **LOOPNE** (***LOOP** while **Not Equal***) ciclul se va termina fie dacă CX=0, fie dacă ZF=0. Chiar dacă ieşirea din ciclu se face pe baza valorii din ZF, decrementarea lui CX are oricum loc.

Să presupunem de exemplu că dorim să reţinem într-un vector caracterele citite de la tastatură până când se apasă tasta *<Enter>* şi atâta timp cât numărul caracterelor citite nu depăşeşte 128. Acest lucru se poate obţine prin secvenţa:

```
        .     .      .
data segment
Vector        DB     128    DUP(?)
        .     .      .
cod segment
        .     .      .
        mov   cx,128
        lea   bx,Vector
Bucla:
        mov   ah,1           ;funcţia DOS pentru citire caracter
        int   21h            ;activarea funcţiei
        mov   [bx],al        ;memorarea tastei apăsate
        inc   bx             ;pregătirea vectorului pentru următorul caracter
        cmp   al,0dh         ;a fost <ENTER>?
        loopne Bucla         ;dacă nu, predă controlul instrucţiunii de după .
                 .     .     ;Bucla
```

LOOPE mai este cunoscută şi sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută şi sub numele de **LOOPNZ**. Aceste instrucţiuni se folosesc de obicei precedate de o instrucţiune CMP sau SUB.

O altă instrucţiune folosită pentru controlul ciclării este **JCXZ** (*Jump if CX is Zero*). Această instrucţiune realizează saltul la eticheta operand numai dacă CX=0, fiind utilă în situaţia în care se doreşte testarea valorii din CX înaintea intrării într-o buclă. Exemplul următor se referă la iniţializarea cu 0 a unui şir de octeţi, CX conţinând lungimea acestui şir. Instrucţiunea JCXZ se foloseşte pentru a se evita intrarea în ciclu dacă CX=0:

```
        .       .       .
    jcxz  MaiDeparte        ;dacă CX=0 se sare peste buclă
Bucla:
    Mov   BYTE PTR [si],0   ;iniţializarea octetului curent
    inc   si                ;trecere la octetul următor
    loop  Bucla             ;reluare ciclu sau terminare
MaiDeparte:     .       .       .
```

Dar să vedem de ce se acordă totuşi atenţie sporită situaţiei în care se începe execuţia unei bucle cu CX=0. La întâlnirea instrucţiunii LOOP cu CX=0, CX este decrementat, obţinându-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 instrucţiunea JCXZ acţionează rapid şi eficient exact în acest sens.

Este important să precizăm aici faptul că <u>nici una dintre instrucţiunile de ciclare prezentate nu afectează flag-urile</u>.

```
                    dec  cx
loop Bucla   şi     jnz  Bucla
```

deşi semantic echivalente, nu au exact acelaşi efect, deoarece spre deosebire de LOOP, instrucţiunea DEC afectează indicatorii OF, ZF, SF şi PF.

4.3.4. Instrucţiunile CALL şi RET

Apelul unei proceduri se face cu ajutorul instrucţiunii **CALL**, acesta putând fi *apel direct* sau *apel indirect*. Apelul direct are sintaxa

CALL *operand*

Asemănător instrucţiunii JMP şi instrucţiunea **CALL** transferă controlul la adresa desemnată de operand. În plus faţă de aceasta, înainte de a face saltul, instrucţiunea CALL salvează în stivă adresa următoarei instrucţiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalenţa


```
CALL operand                [ push CS ] (numai dacă este operand far)
A: . . .          ⇔         push offset A
                            jmp  operand
```

Terminarea execuţiei secvenţei apelate este marcată de întâlnirea unei instrucţiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucţiunea de la această adresă. Sintaxa instrucţiunii RET este

RET [*n*]

unde *n* este un parametru opţional. El indică eliberarea din stivă a *n* octeţi aflaţi sub adresa de revenire. Vom detalia acest mecanism în capitolul 8. Instrucţiunea RET poate fi far sau near. Ca efect, avem echivalenţele

```
                            B  dw   ?
RET n                       .     .     .
(revenire near)   ⇔         pop  B
                            add  sp,n
                            jmp  B

                            B  dd   ?
RET n                       .     .     .
(revenire far)    ⇔         pop  word ptr B
                            pop  word ptr B+2
                            add  sp,n
                            jmp  B
```

De cele mai multe ori, instrucţiunile CALL şi RET apar în următorul context

```
nume  PROC
.     .     .
      ret  n
nume  ENDP
.     .     .
CALL  nume
```

Directivele PROC şi ENDP au fost prezentate în 3.3.4. Apelul şi revenirea sunt implicit far sau near, după cum procedura *nume* este declarată FAR sau respectiv NEAR.

Instrucţiunea 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
```

Rezumând, operandul destinaţie al unei instrucţiuni CALL poate fi:

- numele unei proceduri FAR sau NEAR
- numele unui registru în care se află o adresă NEAR
- o adresă de memorie NEAR sau FAR (ca şi în cazul instrucţiunii JMP).

În principiu, un macro şi un apel de macro poate fi înlocuit cu o procedură şi respectiv apelul ei. Se pune întrebarea: ce este mai bine să alegem pentru îndeplinirea unei sarcini ? Un macro sau o procedură ? Dacă primordial este obţinerea unui cod minim se va alege varianta procedurală, deoarece pentru o subrutină codul este asamblat o singură dată. Din contră, un macro nu realizează o reducere a dimensiunilor programului obiect, ci numai a dimensiunilor programului sursă ! În schimb, prin utilizarea macrourilor se produce un cod mai rapid deoarece se evită apelurile instrucţiunilor CALL şi RET. Mai mult, flexibilitatea utilizării macrourilor rezultă şi din posibilităţile de "croire" a unui macro pentru a genera după dorinţă coduri diferite pentru apeluri diferite. Marele avantaj al macrourilor în limbajul de asamblare rezidă în faptul că sunt singurele structuri parametrizabile ale acestuia, permiţând astfel adaptarea apelurilor unui aceluiaşi macro în funcţie de un context definibil prin parametri (a se vedea în acest sens 3.3.7.)

Să facem observaţia (foarte importantă!) că spre deosebire de procedurile din limbajele de programare de nivel înalt, subrutinele scrise în limbaj de asamblare nu pot avea în mod explicit parametri.

În cazul unui text sursă scris integral în limbaj de asamblare apare problema conservării valorilor din regiştri de dinainte de apel, valori pe care subrutina are libertatea să le modifice, dar care ar putea fi necesare după revenire. O soluţie ar fi ca la intrarea în subrutină regiştrii să fie salvaţi în stivă, iar la ieşire să se refacă starea lor iniţială. Această variantă consumă însă mult timp şi cere un număr destul de mare de instrucţiuni scrise în acest scop. O altă variantă ar fi să acceptăm ideea că valorile iniţiale pot fi oricând alterate şi în consecinţă să nu ne mai bazăm pe posibilitatea de păstrare a valorilor iniţiale. Aceasta ar fi o variantă restrictivă care ne constrânge să limităm folosirea eficientă a regiştrilor.

Se adoptă ca soluţie introducerea de către utilizator în fiecare subrutină a unor linii de comentariu ce conţin suficiente informaţii pentru precizarea regiştrilor afectaţi, pentru a şti deci şi care sunt cei pe a căror valoare iniţială ne mai putem baza. Aceste informaţii devin deosebit de utile pentru programator în scopul gestionării eficiente a resurselor programului scris.

4.4. INSTRUCŢIUNI PE ŞIRURI

4.4.1. Generalităţi privind şirurile şi instrucţiunile pe şiruri

Instrucţiunile pe şiruri sunt nişte instrucţiuni puternice, ele având ca efect simultan (eventual repetat) atât accesarea memoriei cât şi incrementarea sau decrementarea unor regiştri pointer. Aşa cum le arată şi numele, aceste instrucţiuni se folosesc pentru manipularea şirurilor de octeţi sau

cuvinte, ele fiind mai scurte (ca şi cod rezultat) şi mai rapide în execuţie decât combinaţiile echivalente de instrucţiuni MOV, INC şi LOOP.

Un *şir* în sensul 8086 este caracterizat de următoarele atribute:

a). tipul elementelor (octeţi sau cuvinte).
b). adresa primului element din şir.
c). direcţia de parcurgere (de la adrese mici spre mari sau de la adrese mari spre mici).
d). numărul de elemente.
Din punct de vedere al instrucţiunilor pe şiruri, un şir poate fi şir sursă sau şir destinaţie. Instrucţiunile pe şiruri sunt în număr de 10 şi se împart în trei categorii:

- instrucţiuni care folosesc un şir sursă şi un şir destinaţie (MOVSB, MOVSW, CMPSB, CMPSW).
- instrucţiuni care folosesc numai un şir sursă (LODSB, LODSW).
- instrucţiuni care folosesc numai un şir destinaţie (STOSB, STOSW, SCASB, SCASW).

Fiecare dintre aceste instrucţiuni pretinde să-i fie pregătite în prealabil caracteristicile şirurilor cu care operează.

a). *Tipul* este indicat prin ultima literă a numelui instrucţiunii: **B** pentru elemente octeţi şi **W** pentru elemente cuvinte. Instrucţiunile ce folosesc două şiruri presupun că ambele şiruri sunt de acelaşi tip.

b). *Adresa primului element dintr-un şir* este o adresă far memorată în regiştri, astfel:

- în DS:SI pentru şirurile care sunt sursă;
- în ES:DI pentru şirurile care sunt destinaţie;

c). *Direcţia de parcurgere* este indicată de flagul DF, astfel:

- DF=0 impune ca ordinea de parcurgere să fie de la adrese mici spre adrese mari. În acest caz, adresa primului element este adresa cea mai mică din şir.

- DF=1 impune ca ordinea de parcurgere să fie de la adrese mari spre adrese mici. În acest caz, adresa primului element este adresa cea mai mare din şir.

Poziţionarea prealabilă a lui DF se face folosind instrucţiunile CLD sau STD. Instrucţiunile care folosesc două şiruri presupun că ambele şiruri sunt parcurse în aceeaşi direcţie.

d). *Numărul de elemente*, atunci când se doreşte a fi exploatat, trebuie trecut în registrul CX.

Vom prezenta instrucţiunile pe şiruri sub forma a două grupe funcţionale:
* instrucţiuni utilizate pentru transferul de date (**LODS, STOS** şi **MOVS**)
* instrucţiuni utilizate pentru consultarea şi compararea datelor (**SCAS** şi **CMPS**)

4.4.2. Instrucţiuni pe şiruri pentru transferul de date

Aceste instrucţiuni sunt asemănătoare instrucţiunii MOV, dar ele realizează mai mult şi operează mai rapid.

Instrucţiunea **LODS** se prezintă sub două forme: **LODSB** şi **LODSW**. Ea încarcă un octet sau respectiv un cuvânt din memorie în registrul acumulator.

LODSB	AL <-- <DS:SI> if DF=0 inc(SI) else dec(SI)	_
LODSW	AX <-- <DS:SI> if DF=0 SI<--SI+2 else SI<--SI-2	_
STOSB	<ES:DI> <-- AL if DF=0 inc(DI) else dec(DI)	_
STOSW	<ES:DI> <-- AX if DF=0 DI<--DI+2 else DI<--DI-2	_
MOVSB	<ES:DI> <-- <DS:SI>; if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}	_
MOVSW	<ES:DI> <-- <DS:SI>; if DF=0 {SI<--SI+2; DI<--DI+2} else {SI<--SI-2; DI<--DI-2}	_

Instrucţiunea **LODSB** încarcă octetul de adresă DS:SI în AL şi 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 instrucţiunea **CLD**) atunci SI este incrementat, iar dacă DF=1 (setarea acestuia cu 1 făcându-se cu instrucţiunea **STD**) SI este decrementat. Această regulă impusă de valoarea din DF este valabilă pentru toate instrucţiunile pe şiruri ce afectează regiştri pointer. De exemplu,

```
cld
mov   si,0
lodsb
```

va încărca AL cu conţinutul octetului de deplasament 0 din cadrul segmentului de date şi apoi se va incrementa SI cu 1, acţiuni echivalente cu

```
mov   si,0
mov   al,[si]
inc   si
```

Instrucţiunea LODSB este însă mai rapidă şi cu 2 octeţi mai scurtă decât echivalentul

```
mov   al,[si]
inc   si
```

Instrucţiunea **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
```

încarcă AX cu valoarea cuvântului de memorie de deplasament 10 din cadrul segmentului de date, decrementând apoi SI cu 2.

Instrucţiunea **STOS** este complementara instrucţiunii LODS, ea transferând valoarea octet sau cuvânt din acumulator la locaţia de memorie de adresă ES:DI, incrementând sau decrementând apoi corespunzător pe DI. Şi instrucţiunea STOS se prezintă sub două forme: **STOSB** şi **STOSW**.

Instrucţiunea **STOSB** copiază octetul din AL la octetul de adresă ES:DI, incrementând sau decrementând DI în funcţie 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 0FFFEh.

Instrucţiunea **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.

Instrucţiunile LODS şi STOS funcţionează eficient împreună pentru copierea de şiruri. Subrutina COPIERE de mai jos realizează copierea şirului terminat cu 0 care începe la DS:SI în şirul ce începe la adresa ES:DI :

```
;Subrutină pentru copierea unui şir terminat cu 0 în altul
;Intrări:      adresa de început a şirului sursă      DS:SI
;              adresa de început a şirului destinaţie  ES:DI
;Ieşiri : -
;Regiştri afectaţi : AL, SI, DI

Copiere    PROC
      cld          ;parcurgerea se va face crescător deci se va impune incrementarea
```

```
iar:    lodsb          ;preia caracterul sursă
        stosb          ;memorează la destinaţie
        cmp  al,0      ;a fost 0 ?
        jnz  iar       ;dacă nu, se continuă
        ret            ;dacă da, se revine din procedură
Copiere ENDP
```

Dacă cunoaştem direct numărul de octeţi (cuvinte) ce trebuie copiaţi (copiate) se poate proceda astfel:

```
        mov  cx, DIM_SIR_IN_CUVINTE
        mov  si, OFFSET SirSursa
        mov  ax, SEG SirSursa
        mov  ds, ax
        mov  di, OFFSET SirDest
        mov  ax, SEG SirDest
        mov  es, ax
        cld
Bucla:
        lodsw
        stosw
loop Bucla
```

O modalitate mai eficientă de transfer a unui octet sau cuvânt dintr-o locaţie de memorie în alta este prin folosirea instrucţiunii **MOVS**.

Instrucţiunea **MOVS** poate fi privită ca o combinaţie a instrucţiunilor LODS şi STOS, ea preluând octetul (**MOVSB**) sau cuvântul (**MOVSW**) de la DS:SI şi depunând această valoare la adresa ES:DI. Nu este folosit ca intermediar nici un registru, deci nici o valoare a acestora nu va fi afectată. Folosind MOVSW, ciclul de mai sus devine

```
Bucla: movsw
       loop Bucla
```

4.4.3. Instrucţiuni pe şiruri pentru consultarea şi compararea datelor

Instrucţiunea **SCAS** (care are şi ea două variante: **SCASB** şi **SCASW**) caută în memorie o anumită valoare particulară (octet sau respectiv cuvânt).

Instrucţiunea **SCASB** compară valoarea din AL cu valoarea octet adresată de ES:DI, setând flagurile în acord cu rezultatele comparării (la fel ca şi o instrucţiune CMP), DI fiind apoi incrementat sau decrementat.

SCASB	CMP AL,<ES:DI> if DF=0 inc(DI) else dec(DI)	OF, SF, ZF , AF, PF, CF
SCASW	CMP AX,<ES:DI> if DF=0 DI<--DI+2 else DI<--DI-2	OF, SF, ZF , AF, PF, CF
CMPSB	CMP <DS:SI>,<ES:DI>; if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}	OF, SF, ZF , AF, PF, CF
CMPSW	CMP <DS:SI>,<ES:DI>; if DF=0 {SI<--SI+2; DI<--DI+2} else {SI<--SI-2; DI<--DI-2}	OF, SF, ZF ,AF, PF, CF

De exemplu, secvenţa următoare caută prima literă 'a' din cadrul şirului *Text*:

```
data segment
     Text  DB     'comparare date',0
     LungimeSir   EQU   ($-Text)
data ends

cod  segment
     .      .      .
     mov  ax,data
     mov  es,ax
     mov  di,OFFSET Text      ;ES:DI pointează la începutul lui Text
     mov  al,'a'              ;caracterul de căutat
     mov  cx,LungimeSir       ;lungimea şirului de consultat
     cld                      ;se stabileşte direcţia de căutare "spre înainte"
                                   (DI se va incrementa)
     CautareA:
          scasb               ;se potriveşte AL cu ES:DI?
          je GasitA           ;dacă da, s-a terminat căutarea
     loop CautareA            ;dacă nu, se continuă
;dacă se ajunge aici nu s-a găsit caracterul 'a'

     GasitA:
          dec di              ;se decrementează DI pentru a indica
                              ;offsetul caracterului 'a' găsit
                                   .      .      .
```

Indiferent de succesul sau insuccesul comparaţiei efectuate, DI este incrementat după execuţia instrucţiunii SCASB, ceea ce face necesară decrementarea sa dacă după găsirea lui 'a' dorim ca DI să conţină offset-ul acestui caracter. Bucla CautareA de mai sus este echivalentă ca efect cu

```
CautareA:   cmp  es:[di],al
            je GasitA
            inc  di
loop CautareA
```

singura diferenţă ca ordine de efectuare a acţiunilor apărând în faptul că SCASB realizează incrementarea lui DI înaintea execuţiei instrucţiunii JE. Dacă aici incrementarea o facem explicit, atunci trebuie să o efectuăm după instrucţiunea JE pentru a nu afecta flagurile setate de către CMP!

Legat de aceasta să facem aici precizarea că instrucţiunile pe şiruri NU setează flagurile în urma acţiunii asupra regiştrilor SI, DI sau CX. Instrucţiunile LODS, STOS şi MOVS nu afectează nici un flag, iar SCAS şi CMPS modifică flagurile doar ca rezultat al comparaţiilor pe care le efectuează.

Instrucţiunea **SCASW** compară conţinutul registrului AX cu cuvântul de adresă ES:DI, incrementând sau decrementând DI cu 2, în funcţie de valoarea din DF. Secvenţa de mai jos utilizează instrucţiunea 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        ;pune 0 în AX pentru a căuta un element nenul
        std                 ;căutarea începe de la sfârşit
        repe  scasw         ;se repetă căutarea până la primul element nenul sau
                            ;până la epuizarea elementelor tabloului
        jne  AmGasit
        .     .             .;dacă se ajunge aici, tabloul conţine numai elemente nule
AmGasit:inc  di
        inc  di             ;se actualizează DI pentru a puncta spre elementul găsit
        .     .             .
```

Instrucţiunea **CMPS** are ca rol efectuarea de comparări de şiruri de octeţi sau cuvinte. Execuţia unei instrucţiuni CMPS are ca efect compararea locaţiilor de memorie de adrese DS:SI şi respectiv ES:DI, urmată de incrementarea sau decrementarea regiştrilor SI şi DI. Flagurile vor fi actualizate corespunzător pentru a reflecta rezultatul comparării.

Instrucţiunea **CMPSB** realizează compararea la nivel de octet, iar **CMPSW** la nivel de cuvânt, prima incrementând sau decrementând SI şi DI cu 1 iar ultima cu 2. În exemplul următor se compară două tablouri ce conţin 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 regiştrii SI şi DI pentru
        dec si        ;a puncta spre elementul prin care diferă
        dec di
        dec di
        .     .     .
```

4.4.4. Execuţia repetată a unei instrucţiuni pe şiruri

Pentru execuţia repetată a unei instrucţiuni pe şiruri, limbajul de asamblare dispune de variante echivalente instrucţiunii de ciclare LOOP. Acestea sunt oferite de aşa numitele *prefixe de instrucţiune*. Sintaxa utilizării lor este

prefix_de_instructiune instructiune_pe_sir

În principiu este vorba despre un singur prefix de instrucţiune, şi anume **REP**. Vom vedea mai jos însă, că în cazul utilizării instrucţiunii SCAS sau CMPS apar două variante posibile de REP.
Prefixul de instrucţiune **REP** impune execuţia repetată a instrucţiunii pe care o prefixează, până când valoarea din CX devine 0. Dacă de la început avem CX=0 atunci instrucţiunea respectivă este inoperantă. Astfel, ciclul

```
Bucla: instructiune_pe_sir
       loop Bucla
```

este echivalent cu instrucţiunea

```
rep instructiune_pe_sir
```

În cazul utilizării prefixelor cu instrucţiunile SCAS sau CMPS condiţia verificată se completează, ţinându-se cont de valoarea flagului ZF. Acest lucru face ca prefixul REP să fie prezent în două variante.

Forma **REP** (echivalentă cu formele **REPE** - ***REP**eat while **E**qual* şi **REPZ** - ***REP**eat while **Z**ero*) provoacă execuţia repetată a instrucţiunilor SCAS sau CMPS până când CX devine 0 sau până când apare o nepotrivire (caz în care ZF va primi valoarea 0).

Asemănător, **REPNE** (***REPeat while Not Equal***, formă echivalentă cu **REPNZ** -***REPeat while Not Zero***) provoacă execuţia repetată a instrucţiunii SCAS sau CMPS până când CX devine 0 sau până când apare o potrivire (caz în care ZF va primi valoarea 1).

De ce există această deosebire doar în cazul instrucţiunilor SCAS şi CMPS ? Foarte simplu: pentru că sunt singurele care afectează vreun flag (şi anume ZF), permiţând rafinarea condiţiilor pe baza valorii flagului afectat. Rezultă deci, că în cazul instrucţiunilor LODS, STOS şi MOVS toate cele cinci mnemonici prezentate au acelaşi efect: repetarea cât timp CX <> 0.

Faţă de variantele cu LOOP, avantajul evident al folosirii acestor prefixe este comprimarea scrierii.

4.4.5. Utilizarea de operanzi pentru instrucţiuni pe şiruri

În exemplele date până acum am folosit numai formele explicite ale instrucţiunilor pe şiruri în funcţie de dimensiunile de reprezentare a datelor. De exemplu, am folosit LODSB şi LODSW dar nu am utilizat LODS.

Limbajul de asamblare 80x86 acceptă utilizarea instrucţiunilor neexplicite dacă se prevăd operanzi pentru specificarea dimensiunii de reprezentare a argumentelor, ceea ce va permite alegerea corectă a instrucţiunii de executat (de exemplu, LODSB sau LODSW).

Instrucţiunea MOVS de mai jos este echivalentă cu MOVSB:

```
data  segment
      Sir1     LABEL  BYTE    DB    'Sirul sursa'
      LungSir1 EQU  ($-Sir1)
      Sir2     DB  50 DUP (?)
data  ends
      .        .       .
cod   segment
      mov   ax, data
      mov   ds, ax
      mov   es, ax
      mov   si, OFFSET Sir1
      mov   di, OFFSET Sir2
      mov   cx, LungSir1
      cld
      rep  movs es:[Sir2],[Sir1]
      .        .       .
```

Asamblorul va alege varianta de instrucţiune MOVS în funcţie de dimensiunea de reprezentare a operanzilor, aici aceasta fiind octetul.

Să facem precizarea că această informaţie este folosită numai pentru determinarea variantei de instrucţiune necesară, situaţia fiind oarecum asemănătoare unei declaraţii. Codul corespunzător operanzilor nu este asamblat în codul executabil, cel asamblat aici fiind de fapt cel corespunzător instrucţiunii **rep movsb**.

Am precizat aceasta pentru a evidenţia faptul că prezenţa operanzilor într-o instrucţiune pe şiruri nu scuteşte programatorul de responsabilitatea setării adreselor de lucru. De exemplu, forma lods [Sir] nu încarcă automat adresa lui *Sir* la DS:SI, acest lucru rămânând în sarcina programatorului.

4.5. UN EXEMPLU COMPLET DE PROGRAM

În această secţiune vom prezenta un exemplu complet de program. În acest scop ne-am oprit asupra determinării numerelor prime (şi impare) până la un n dat. Pentru economie de memorie, punem în şir doar numerele impare. Descrierea algoritmului preferăm să o facem prin programul Pascal care urmează, program care tipăreşte toate numerele prime şi se termină cu tipărirea numărului acestora.

```
program prime;
const  n = 65535;    ni  = (n - 1) div 2;    ni2 = ni div 2;
var    S: array[0..ni-1] of byte;
       i, j, pas: word;

begin
       for i := 0 to ni-1 do S[i] := 1;
       i := 0;
       while i <= ni2 do begin
              while S[i] = 0 do i := i + 1;
              pas := i + i + 3;
              j := i + pas;
              while j <= ni do begin
                     S[j] := 0;
                     j := j + pas
              end;
              i := i + 1
       end;
       j := 0;
       for i := 0 to ni do
              if S[i] = 1 then begin
                     j := j + 1;
                     writeln(i+i+3:10);
              end;
       writeln(j:10);
end.
```

Vom transpune acum acest program într-unul echivalent scris în limbaj de asamblare. Evident, vom face uz de toate facilităţile oferite de acesta din urmă. De aceea, în loc de a reţine câte un octet pentru fiecare poziţie din ciur, vom reţine doar câte un bit. Variabila i din programul Pascal este înlocuită aici cu registrul bx, care va conţine numărul bitului curent. Numărul cuvântului curent este conţinut în registrul di. Variabila j este înlocuită cu registrul dx. Tipărirea este realizată cu ajutorul funcţiei 9 a întreruperii 21h, despre care vom vorbi în capitolul următor. Aici trebuie doar remarcat faptul că operaţia de conversie în şir 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 condiţionat
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 biţi 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 biţi 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
```

```
        and     cx,7
        ror     al,cl
        and     al,0feh         ;S[j] := 0;
        rol     al,cl
        mov     byte ptr [S+di],al
        add     dx,pas          ;j := j + pas
        jo      Alti
        jmp     whilej

Alti:
        add     bx,1            ;i := i + 1
        jmp     whilei
exit:
        lea     si,S
        xor     bx,bx           ;i := 0;
        mov     pas,bx          ;pas va conţine numărul de numere prime

AltCuv:
        lodsw
        mov     cx,16
AltBit:
        shr     ax,1
        jnc     PesteBit

        push    bx
        add     bx,bx
        add     bx,3            ;bx := i + i + 3
        call    conv
        pop     bx
        inc     pas

PesteBit:
        inc     bx
        cmp     bx,ni
        jae     Stop
        loop    AltBit
        jmp     AltCuv
Stop:
        mov     bx,pas
        call    conv
        mov     ax,4c00h
        int     21h
```



```
conv    proc    near            ;tipăreşte numărul din bx
                                ;se foloseşte schema lui Horner
        push    ax
        push    cx
        push    dx

        lea     di,Rez
        mov     cx,10
        mov     al,' '
        rep     stosb           ;umple cu spaţii
        std
        lea     di,Rez+9
        mov     ax,bx
Horner:
        cmp     ax,0
        jz      Tiparire
        xor     dx,dx
        div     Zece
        xchg    al,dl
        add     al,'0'
        stosb                   ;depune o cifră
        xchg    al,dl
        jmp     Horner
Tiparire:
        cld
        mov     ah,9            ;vezi funcţia 09h a întreruperii 21h
        lea     dx,Rez

        int     21h

        pop     dx
        pop     cx
        pop     ax

        ret

conv    endp

end     Start
```

CAPITOLUL 5

ÎNTRERUPERI

5.1. PROBLEME GENERALE PRIVIND ÎNTRERUPERILE

O întrerupere este o acţiune a microprocesorului prin care acesta anunţă apariţia unui eveniment. Mai concret, întreruperea este un semnal electric transmis sistemului de calcul (SC) prin care acesta este anunţat de apariţia unui eveniment particular.

Acţiunile pe care le efectuează sistemul de calcul la apariţia unei întreruperi sunt:

1. suspendarea programului în curs de desfăşurare;
2. lansarea în execuţie a unei rutine specializate, numită *Rutină de Tratare a Întreruperii (RTI)* sau *Handler de întrerupere*, care deserveşte întreruperea;
3. eventual, reluarea execuţiei programului suspendat (depinzând de tipul de întrerupere).

Cauzele apariţiei acestor evenimente pot fi de 2 tipuri:

a). *externe* (apăsarea unor combinaţii de taste, iniţierea sau terminarea unor operaţii de I/O);
b). *interne* (împărţirea la 0, tentativa de adresare a unei zone de memorie inexistente, tentativa de execuţie a unei instrucţiuni având un cod inexistent, depăşirea capacităţii de reprezentare a unui rezultat).

De obicei, după tratarea unei întreruperi externe programul se reia, după o întrerupere internă nu!

La apariţia 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 proiectanţii sistemului de operare (SO);
- scrise de utilizatori;

Pentru localizarea rapidă a RTI se foloseşte vectorizarea întreruperilor: asocierea fiecărei întreruperi cu o locaţie de memorie dublucuvânt cu adresă fixă, unde se va memora adresa far a RTI corespunzătoare întreruperii.

Tabela RTI (vectorul de întreruperi) se află în memorie la adresa 0000:0000. Primii 256 x 4 = 1024 octeţi conţin aceste adrese (zona se numeşte *tabela vectorilor de întrerupere* - TVI). Pentru întreruperea k, adresa RTI(k) se găseşte la adresa 0000 : k*4.

Acest tablou cu adrese se iniţializează în momentul încărcării sistemului de operare. În timpul funcţionării sistemului, este posibila modificarea unora dintre adrese. Modificările pot apare fie accidental, fie intenţionat. Modificarea accidentală conduce de cele mai multe ori la blocarea sistemului, fiind necesară reiniţializarea lui. Modificarea voită o vom numi *deturnare a întreruperii*, şi ne vom ocupa de aceasta în capitolul 6.

5.2. CLASIFICAREA ÎNTRERUPERILOR

La nivelul arhitecturii 80x86 apar trei tipuri de evenimente numite în documentaţii întreruperi:

a). *întreruperi hardware* – întreruperi generate în mod automat ca răspuns la apariţia unor cauze de tip extern. Acestea sunt deci cauzate de un eveniment extern hardware (extern microprocesorului!), cum ar fi de exemplu semnalele de la periferice, ţinând aşadar de sistemul de intrare-ieşire (I/O system) şi fiind astfel întreruperi BIOS (**B**asic **I**nput **O**utput **S**ystem). RTI corespunzătoare sunt încărcate în memorie la pornirea sistemului de calcul din fişierele ROM-BIOS.

b). *excepţii* - întreruperi generate în mod automat, ca răspuns la apariţia unor cauze de tip intern. Exemple de cauze: împărţirea prin 0, încercarea de execuţie a unui cod de instrucţiune inexistent, accesarea unei zone de memorie interzise (*memory protection fault*).

c). *întreruperi software (software interrupts* sau *traps)* – acestea presupun un transfer de control al execuţiei, iniţiat de programator către o rutină specială (handler). Mijlocul prin care programatorul iniţiază o astfel de acţiune este instrucţiunea INT. Aceste întreruperi se numesc întreruperi software tocmai pentru că ele sunt invocate soft printr-o instrucţiune specificată explicit de către programator.

Datorită acestui specific putem da o definiţie alternativă acestui tip de întreruperi: acele întreruperi ce pot fi iniţiate numai de către programator prin INT sunt *întreruperi soft*.

Vom prezenta şi analiza în continuare cele mai importante întreruperi din fiecare categorie.

5.2.1. Întreruperi hardware

În documentaţii recunoaştem întreruperile hardware prin referirea la IRQ (*Interrupt Request*), aceasta fiind o cerere de întrerupere nativă la nivelul unui dispozitiv numit PIC (*Programmable Interrupt Controller* – Intel 8259A). Exemple.: IRQ 0 = INT 8 , IRQ 1 = INT 9 etc.

Un excelent tutorial oferind detalii despre funcţionarea acestui dispozitiv precum şi despre întreruperile hard găsim la: http://www.delorie.com/djgpp/doc/ug/interrupts/inthandlers1.html.

Cele mai cunoscute şi frecvent utilizate (emise) întreruperi hardware sunt prezentate în cele ce urmează.

INT 8 este întreruperea hard de ceas (*the system timer*). Este recunoscută prin IRQ 0. Se produce de 18.2 ori/secundă. Nu se deturnează de obicei, dacă e cazul se deturnează cea soft (1Ch). RTI corespunzătoare efectuează următoarele:

- reţine nr. de cicluri de ceas (*timer ticks*) la adresa 0000:046Ch pentru a gestiona în bune condiţii ora sistem.
- decrementează un contor de la adresa 0000:0440h (*Diskette Drive Motor Off Counter*) până când devine 0, moment în care motorul de antrenare este pus pe OFF şi octetul de la adresa 0000:043fh este actualizat pentru a reflecta aceasta.
- generează INT 1Ch (întreruperea soft de ceas).

INT 9 este întreruperea de tastatură (*keyboard interrupt*). Este recunoscută prin IRQ 1. Această întrerupere este generată de tastatură, la fiecare apăsare şi eliberare a unei taste. Acţiunile concrete ale RTI corespunzătoare sunt în esenţă: BIOS-ul răspunde prin citirea scan-codului corespunzător tastei, convertirea sa la codul ASCII corespunzător (a se vedea perechile corespunzătoare Scan code/ASCII code în documentaţii), urmată de memorarea acestei perechi de atribute în bufferul de tastatură, acesta fiind localizat la adresa 0000:041Ch. Începând de la această adresă şi până la deplasamentul 043eh (adică 20h = 32 octeţi) se pot reţine 16 caractere (pentru fiecare caracter câte o pereche scan code/ASCII code). Nu este de ajuns memorarea numai a unui cod ASCII pentru 1 caracter, deoarece există mai multe tipuri de tastaturi şi poziţia unui caracter în configuraţia tastaturii trebuie caracterizată prin atributul scan code. Acesta este aşadar un cod asociat unei taste în funcţie de poziţia ei pe tastatură.

Dacă s-a apăsat una dintre tastele Ctrl, Alt sau Shift, se actualizează corespunzător octeţii de la adresele 0000:0417h (Shift Status byte) şi 0000:0418h (Extended Shift Status byte).

Atenţie! – a nu se dezactiva întreruperea 9, deoarece atunci SC nu va mai răspunde nici la CTRL + ALT + DELETE !!!.

INT 0Bh şi **INT 0Ch** – întreruperi 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 informaţiei comparativ cu cea paralelă? În transmisia serială există o singură linie de date iar informaţia este transmisă succesiv bit cu bit (un singur bit odată). Deşi lentă în comparaţie cu o transmisie paralelă ce permite transmiterea simultană a mai multor biţi (8,16,32 sau 64) această tehnică a transmisiei seriale este adecvată transmiterii informaţiei la distanţe 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 interfaţa (portul) COM2, iar INT 0Ch (IRQ 4) se activează când este vorba despre o comunicare serială pe interfaţa (portul) COM1.

Gestionarea comunicaţiei seriale la nivelul arhitecturii 80x86 este asigurată de către dispozitivul UART (*Universal Asynchronous Receiver/Transmitter*) 8250 (sau compatibil cu acesta) care generează o întrerupere (IRQ 3 sau IRQ 4, depinzând de linia serială pe care are loc comunicarea) într-una din următoarele patru situaţii: un caracter soseşte pe o linie serială, dispozitivul UART a terminat transmisia unui caracter şi solicită un altul, apariţia unei erori sau solicitarea unei modificări de stare. RTI corespunzătoare va trebui să determine cauza exactă a întreruperii prin interogarea dispozitivului UART.

INT 0Dh şi **INT 0Fh** - întreruperi ce deservesc porturile paralele. După cum aminteam mai sus interfeţele paralele permit transmisia simultană a mai multor biţi (8,16,32 sau 64). Principalul avantaj al interfeţelor paralele este astfel viteza de transmisie. Aceasta este însă obţinută prin costuri suplimentare reprezentate de necesitatea unor cablări adiţionale pentru a asigura extinderea canalelor de date. Din cauza acestor costuri, interfeţele paralele sunt de obicei limitate la conexiuni scurte ce au sub 1 m lungime.

Iniţial, întreruperile INT 0Dh (IRQ 5) şi INT 0Fh (IRQ 7) au fost proiectate pentru a deservi porturile paralele LPT1, LPT2 (*Line Printer*), însă imediat după aceasta, IBM a proiectat o interfaţă pentru imprimantă (*printer interface card*) care nu este compatibilă cu aceste întreruperi! Ca urmare, astăzi ele nu mai sunt utilizate pentru imprimante ci preponderent pentru plăci SCSI şi plăci de sunet.

INT 0Eh → întreruperea de dischetă (IRQ 6 - *Diskette Drive interrupt*)
INT 76h → întreruperea de hard-disk (IRQ 14 - *Hard Disk Controller*)

Unităţile de dischetă şi harddisk generează fiecare câte o întrerupere la încheierea unei operaţii de citire/scriere. Aceste întreruperi sunt utile pentru gestiunea aplicaţiilor în sistemele de operare multitasking (OS/2, Linux, Windows): în timp ce are loc o operaţie I/O cu discheta sau hard-disk-ul, microprocesorul poate executa părţi din alte procese; când un disc şi-a încheiat operaţia

curentă de citire/scriere va întrerupe acţiunea curentă a microprocesorului pentru a-i semnala acestuia posibilitatea revenirii la procesul anterior.

INT 70h - The Real-Time Clock Interrupt (IRQ 8). Această întrerupere este activată de CMOS de 1024 ori/secundă în vederea asigurării bunei funcţionări a ceasului real al sistemului.

INT 75h - întreruperea de unitate în virgulă flotantă (*FPU Interrupt* - IRQ 13) este o întrerupere generată de coprocesorul matematic la orice situaţie de excepţie de tip virgulă flotantă (*floating-point exception*).

5.2.2. Excepţii

Datorită faptului că şi acestea sunt întreruperi BIOS sunt clasificări care încadrează excepţiile la întreruperi hard (care sunt şi ele BIOS). Nu este însă corect pentru că după tratarea unei întreruperi hard programul se reia întotdeauna, în timp ce după tratarea excepţiilor, de obicei programul NU se reia!

INT 0 - întreruperea împărţirii la zero (*Zero Divide interrupt*). Întreruperea 0 este generată de fiecare dată când apare o aşa numită *condiţie de împărţire la zero*. INT 0 poate fi emisă în trei situaţii distincte:

i). depăşirea rezultatului (câtului) la împărţire atunci când utilizăm DIV sau IDIV;

```
mov ax,600
mov bh,2
(i) div bh      ; se efectuează ax/bh, cu câtul în AL şi restul în AH
```

Câtul ar trebui să fie 300 şi să fie obţinut în AL, însă valoarea 300 nu încape 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 împărţirii 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 şi restul în DX
```

Acum valoarea 300 încape în AX şi nu se mai emite INT 0.

ii). încercarea de efectuare a unei împărţiri la zero:

```
mov ax, 600
mov bh,0
div bh          ; se va emite INT 0 deoarece se încearcă împărţirea la BH=0!
```

iii). Emiterea explicită a acestei întreruperi prin invocare software sub forma INT 0:

```
.......
add ax,2
INT 0          ; se emite explicit de către programator INT 0
........
```

RTI în cazul INT 0 afişează "*Divide by zero*" şi predă controlul SO MS-DOS.

Observaţie: De fapt toate întreruperile, indiferent de categoria din care fac parte pot fi activate şi SOFT prin specificarea explicită la nivelul codului sursă a instrucţiunii INT n, unde n este numărul întreruperii ce se doreşte a fi emisă. Asta nu le face însă întreruperi software! A se vedea definiţia alternativă pe care am dat-o acestora: întreruperile software sunt acele întreruperi ce pot fi iniţiate numai de către programator prin INT! Excepţiile şi întreruperile hardware pot fi invocate şi software, însă caracteristica lor de bază este că ele sunt emise în mod automat la apariţia evenimentului corespunzător asociat.

INT 1 (*Single Step*). Această întrerupere este iniţiată de procesor după fiecare instrucţiune maşină, dacă flagul TF = 1. Se foloseşte la depanare, execuţiile pas cu pas ale programelor în cadrul depanatoarelor fiind posibile tocmai datorită emiterii INT 1 după fiecare linie de cod sursă.

INT 2 - întreruperea nemascabilă (*Non-Maskable Interrupt* - NMI). Întreruperile pot fi dezactivate (mascate) prin instrucţiunea CLI (*Clear Interrupts*). INT 2 este singura întrerupere ce nu poate fi mascată, ea fiind generată de fiecare dată când apare o condiţie nemascabilă, ca de exemplu o eroare de paritate a memoriei (*memory parity error*).

INT 3 (*Breakpoint interrupt*). Această întrerupere este folosită de către depanatoare pentru stabilirea de puncte de întrerupere a execuţiei (*breakpoints*) în cursul depanării programelor.

INT 4 (*Overflow*). Această întrerupere se emite când are loc o depăşire în cadrul unor operaţii aritmetice. Mai precis, se emite când se execută instrucţiunea INTO (*interrupt on overflow*) şi OF = 1. Dacă OF = 0, INTO se traduce în NOP (*no operation*).

Scrierea unui handler pentru INT 4 oferă programatorilor o modalitate facilă pentru gestionarea condiţiilor de depăşire aritmetică. Prin execuţia unei instrucţiuni INTO după câte o operaţie aritmetică expusă depăşirii, se poate asigura tratarea adecvată a situaţiilor de depăşire.

Instrucţiunea INTO are următorul efect:

```
INTO ↔ if (OF=1) PUSHF
                 TF:=0
                 IF:=0
                 CALL FAR 0000:0010h
```


INT 6 – (*invalid opcode*) – cod de instrucţiune ilegal. Această întrerupere se emite la încercarea de execuţie a unui cod de instrucţiune inexistent. De exemplu:

```
add ax,2     ; ok, se execută fără probleme
a db 199     ;această linie de cod, deşi la prima vedere ar trebui să aparţină numai
```

unui segment de date, poate apărea şi în segmentul de cod, semnificaţia ei fiind în acest caz interpretarea valorii atribuite (aici 199) drept cod de instrucţiune; am generat aici codul 199 care nu reprezintă un cod de instrucţiune valid şi ca urmare se va emite la această linie INT 6, al cărei handler va afişa mesajul "*Illegal instruction*".

5.2.3. Întreruperi software

O întrerupere software este invocată numai de către programator prin apelul instrucţiunii INT (cu excepţia întreruperilor 05h, 19h, 1Bh). Diferenţa între invocarea unei întreruperi prin instrucţiunea INT şi apelul far al unei proceduri cu instrucţiunea CALL este faptul că instrucţiunea INT pune şi flagurile în stivă înainte de adresa de revenire, astfel încât se va folosi instrucţiunea IRET pentru revenire din rutină (spre deosebire de instrucţiunea RET, aceasta scoate din stivă şi registrul de flaguri).

Întreruperile soft se clasifică în:

- întreruperi BIOS (rutinele de tratare a acestor întreruperi sunt încărcate în memorie la pornire din fişierele ROMBIOS)
- întreruperi DOS (rutinele de tratare a acestor întreruperi sunt încărcate în memorie la pornire din fişierele BDOS)

În plus, unele întreruperi pot fi caracterizate prin faptul că nu au un scop iniţial declarat sau utilizatorii îşi pot scrie propriile rutine de tratare a lor (întreruperi utilizator).

Principalele întreruperi BIOS sunt:

05h Se emite la apăsarea tastei PrintScreen în scopul trimiterii conţinutului ecranului la imprimantă. Instrucţiunea BOUND (ce verifică dacă valoarea unui index de tablou se află între limitele specificate) apelează şi ea această întrerupere dacă condiţiile verificate nu sunt îndeplinite.

10h Servicii de lucru cu ecranul, în mod text şi în mod grafic. Funcţiile acestei întreruperi pot fi folosite pentru setarea modului video (funcţia 00h), setarea dimensiunii şi a poziţiei cursorului (funcţiile 01h, 02h), citirea şi scrierea de caractere şi atribute (funcţiile 08h, 09h) ş.a.m.d.

11h Returnează lista echipamentelor BIOS instalate în sistem: numărul de porturi paralele, dacă modemul intern este instalat, dacă adaptorul de jocuri este instalat, numărul de porturi seriale, numărul unităţilor de dischetă, modul video iniţial, dacă coprocesorul matematic este instalat, dacă sunt unităţi de dischetă instalate.

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 dispoziţie servicii de lucru cu harddisk-ul şi cu discheta: resetarea unui disc (funcţia 00h), obţinerea stării unei operaţii făcute asupra discului (funcţia 01h), citirea şi scrierea de sectoare de pe un disc fix sau dischetă într-un (respectiv dintr-un) buffer din memoria internă (funcţia 02h, respectiv 03h), verificarea sectoarelor (funcţia 04h) ş.a.m.d.

14h Permite accesul la porturile seriale. Această întrerupere pune la dispoziţie funcţii de iniţializare a unui port serial (funcţia 00h), transmitere sau primire a unui caracter (funcţiile 01h şi 02h) şi obţinere a stării unui port serial (funcţia 03h).

15h Pune la dispoziţie funcţii de acces la memoria extinsă, de citire a dispozitivelor de tip joystick ş.a.m.d.

16h Această întrerupere oferă servicii de manipulare a tastaturii, dintre care: citirea unui caracter din buffer-ul tastaturii (funcţia 00h), returnarea stării unor taste (Caps Lock, Scroll Lock, Num Lock, ... – funcţia 02h).

17h Această întrerupere oferă servicii de manipulare a imprimantei: tipărirea unui caracter (funcţia 00h), iniţializarea imprimantei (funcţia 01h) şi returnarea stării imprimantei (funcţia 02h).

18h Activează interpretorul ROM BASIC. Astăzi mai puţin 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 combinaţiei de taste Ctrl-Alt-Del.

1Ah Servicii legate de ceasul sistem. Există două astfel de servicii: citirea ceasului (funcţia 00h) şi setarea ceasului (funcţia 01h).

1Bh Această întrerupere este apelată la apăsarea combinaţiei 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 execuţia programului curent şi 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 acţiune, 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 unităţilor 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. Secţiunea următoare va fi destinată exclusiv acestei întreruperi.

Alte întreruperi DOS:

20h Acesta este unul dintre apelurile care pot terminarea execuţia unui program. Memoria ocupată va fi eliberată ca efect al acestui apel.

25h Permite citirea fizică de pe disc de la o anumită locaţie de memorie, începând cu un anumit sector, într-o anumită locaţie de memorie.

26h Permite scrierea fizică pe disc dintr-o anumită locaţie de memorie, începând cu un anumit sector.

27h Termină execuţia 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 nedocumentate vom reveni ulterior cu explicaţii.

2Eh (nedocumentată). Execută o comandă DOS ca şi când ar fi dată de la prompter.

2Fh Funcţiile acestei întreruperi se ocupă cu: multiplexarea resurselor sistemului, gestiunea memoriei extinse (XMS) dacă aceasta există, controlul programelor TSR ş.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 funcţiile 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 execuţie.

Principalele funcţii DOS

Aşa cum am mai arătat, principala întrerupere DOS este **21h**. Ea înmagazinează practic întreaga componentă BDOS a sistemului de operare DOS. Mai mult, sarcinile unora dintre întreruperile mai sus amintite au fost preluate şi uneori extinse de către unele dintre funcţiile acestei întreruperi.

Vom exemplifica, pe categorii, unele dintre funcţiile DOS mai importante. O prezentare exhaustivă a lor ar necesita mult spaţiu tipografic. Documentaţiile DOS fac o astfel de prezentare, ca şi o serie de programe de tip HELP, accesibile pe orice calculator.

Funcţii de gestiune a memoriei:

48h Alocă un bloc de memorie şi returnează un pointer spre începutul acelei zone de memorie.

49h Eliberează o zonă de memorie pentru a o face disponibilă altor programe.

4ah Ajustează spaţiul de memorie alocat.

Funcţii de gestiune a proceselor:

4Bh Încarcă un program pentru a fi executat sub controlul unui program existent. La terminarea execuţiei programului apelat, controlul se întoarce în programul apelant.

4Ch Termină execuţia unui program întorcându-se în command.com sau în rutina apelantă, cu un anumit cod de retur (error level) setat în AL.

31h Termină execuţia unui program lăsându-l rezident în memorie. Prin intermediul lui AL se va transmite codul de retur.

4Dh Această funcţie este folosită pentru obţinerea codului de retur al unui proces fiu lansat de către un program prin apelul funcţiei 4Bh.

26h Copiază PSP-ul programului curent la o anumită adresă în memorie, apoi actualizează noul PSP pentru a fi folosit de către un nou program.

62h Obţine adresa de început a PSP-ului în memorie.

Funcţii specifice discului:

19h Returnează codul (0=A, 1=B, ...) discului implicit.

33h Returnează codul (numărul) discului folosit la încărcarea SO.

1Ah Setează adresa zonei de transfer disc (DTA), această zonă urmând a fi folosită de operaţii care folosesc FCB (File Control Block). Este responsabilitatea programatorului să se asigure că dimensiunea acestei zone este suficient de mare pentru operaţiile ce urmează a fi făcute.

1Bh Furnizează informaţii despre tabela de alocare a fişierelor (FAT).

Funcţii specifice directoarelor şi fişierelor:

39h Creează un nou director folosind discul şi calea specificată.

3Ah Şterge un director specificând calea spre acesta.

3bh Schimbă directorul curent în cel specificat.

47h Obţine un string ASCIIZ reprezentând calea spre directorul curent.

56h Schimbarea numelui unui fişier.

4Eh Caută în directorul specificat sau în cel curent primul nume de fişier care se potriveşte cu o specificare generică.

4Fh Caută în directorul specificat sau în cel curent următorul nume de fişier care se potriveşte cu o specificare generică.

41h Şterge un fişier din directorul specificat sau din cel curent.

3Dh Deschide un fişier din directorul specificat sau din cel curent. Dacă deschiderea s-a făcut cu succes, returnează un identificator al fişierului (*handle*) pe 16 biţi pentru accesarea fişierului de acum înainte.

3Eh Închide un fişier care a fost deschis cu succes. Identificatorul fişierului devine astfel liber pentru a fi folosit ca identificator al altui fişier. Dacă fişierul a fost modificat, data şi ora ultimei modificări vor fi actualizate.

3Fh Citeşte un anumit număr de octeţi dintr-un fişier deschis cu succes.

40h Scrie un anumit număr de octeţi într-un fişier deschis cu succes.

Intrări / ieşiri cu periferice de tip caracter:

01h Citeşte un caracter de la intrarea standard şi îl afişează la ieşirea standard.

02h Afişează un caracter la ieşirea standard.

09h Tipăreşte un şir de caractere la ieşirea standard. Acest şir de caractere trebuie să conţină ca şi marcaj de sfârşit de şir caracterul '$'. Acest caracter nu va fi tipărit.

0Ah Citeşte de la intrarea standard un şir de caractere până la tastarea lui *Enter*.

Informaţii despre sistem:

30h Furnizează versiunea sistemului de operare.

38h Tratează parametrii dependenţi de regiunea geografică.

2Bh Setează data curentă reţinută de ceasul sistem.

2Dh Setează ora curentă reţinută de ceasul sistem.

Diverse alte funcţii:

35h Obţine adresa unui handler de întrerupere, sub forma adresă de segment:offset

25h Modifică adresa unui handler de întrerupere.

44h Ansamblu de funcţii destinat lucrului la nivel fizic cu diverse tipuri de periferice.

34h Returnează numărul proceselor curente active (nedocumentată).

52h Atribuie valori unor variabile DOS (nedocumentată).

5.3. CÂTEVA OBSERVAŢII ASUPRA ÎNTRERUPERILOR 8086

a) Înşiruirea principalelor întreruperi 8086, precum şi a principalelor funcţii DOS relevă următorul fapt. Numai o parte dintre ele deservesc apariţia unui eveniment neobişnuit, intern sau extern. Acestea, evident că vor fi activate la apariţia evenimentu-lui, fără a se putea prevedea momentul apariţiei. Vom spune despre handler-ele din această categorie că sunt activate prin evenimente.

În schimb, cealaltă parte a handler-elor oferă aceleaşi 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 instrucţiuni speciale de apel de întreruperi. În 5.4 vom trata aceste instrucţiuni.

b) Funcţiile DOS ale întreruperii 21h execută unele sarcini pe care le execută şi alte întreruperi. De exemplu, funcţia 4ch a întreruperii 21h - terminarea unui program, este realizată şi de către întreruperea 20h. Din punct de vedere istoric, funcţiile DOS au apărut ultimele. Ele sunt mai performante şi mai fiabile decât întreruperile similare, motiv pentru care în prezent sunt folosite aproape exclusiv numai acestea.

c) După cum s-a văzut deja, există aşa-numitele *întreruperi nedocumentate*. Acestea sunt rezervate spre folosire doar de către proiectanţii 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 puţin "obscure" pentru muritorii de rând. Aceste întreruperi sau funcţii constituie ţinta preocupărilor pentru mulţi programatori pasionaţi. Nu trebuie uitat însă faptul că proiectanţii sistemului de operare îşi 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 "viruşi", folosesc de multe ori aceste numere de întreruperi pentru scopurile lor distructive.

5.4. INSTRUCŢIUNI SPECIFICE LUCRULUI CU ÎNTRERUPERI

După cum am arătat în 5.2, un handler de întrerupere poate fi apelat şi direct prin intermediul instrucţiunii INT. Instrucţiunea **INT**, cu sintaxa:

INT n

provoacă activarea handlerului corespunzător întreruperii cu numărul *n*.

Ea realizează patru acţiuni succesive:
- pune în stivă flagurile;
- pune în stivă adresa FAR de revenire;
- pune 0 în flagurile TF şi IF;
- apelează, prin adresare indirectă, handlerul asociat întreruperii.

De multe ori, în aplicaţii este utilă simularea acestei instrucţiuni sau a unei părţi din ea. Iată o secvenţă care realizează simularea ei.

```
AdRTI dd    ...          ;Presupunem că conţine 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
```

```
        push  ax            ;Se anulează toate flagurile pentru a anula şi TF şi IF
        popf
        jmp   AdRTI         ;Salt indirect la handler
REV: - - - - - - - -
```

Instrucţinea **INTO** (care nu are operanzi) se comportă în două moduri, în funcţie de valoarea flagului OF:

- dacă OF = 1, atunci este echivalentă cu INT 4 (se apelează handlerul întreruperii de depăşire aritmetică);
- dacă OF = 0, atunci este echivalentă cu NOP (instrucţiunea inefectivă, nu a apărut depăşire).

De aceea, orice instrucţiune care ar putea provoca depăşire este indicat a se programa astfel:

```
        add   ax,b
        into                ;dacă se bănuieşte că operanzii adunării
                            ;precedente ar putea conduce la depăşire.
```

Instrucţiunile **STI şi CLI** (care nu au operanzi) acţionează asupra flagului de întreruperi IF şi prin acesta indică procesorului cum să se comporte la apariţia unei întreruperi.

Instrucţiunea CLI (Clear Interrupt) interzice procesorului să recunoască apariţia vreunei întreruperi. Apare de obicei la începutul unui handler pentru a interzice perturbarea activităţii acestuia.

Instrucţiunea STI (Set Interrupt) permite procesorului să recunoască apariţia unei întreruperi. Are rolul de a anula efectul instrucţiunii CLI.

Întreruperile *nemascabile* nu ţin cont de flagul IF!

Instrucţiunea **IRET** (care nu are operanzi), provoacă revenirea dintr-o întrerupere. Ea este ultima instrucţiune executată în cadrul handlerului. Efectul ei este invers instrucţiunii INT, adică:

- reface flagurile din stivă;
- revine la instrucţiunea a cărei adresă FAR se află în vârful stivei.

De multe ori, în aplicaţii este utilă simularea acestei instrucţiuni sau a unei părţi din ea. Iată o secvenţă care realizează simularea ei.

```
AdRev dd    ?               ;Păstrează temporar adresa de revenire.
- - - - - - - - - - - -
        popf                ;Reface flagurile
        pop   word ptr AdRev     ;Depune offsetul de revenire
        pop   word ptr AdRev+2   ;Depune segmentul de revenire
        jmp   AdRev              ;Salt indirect pentru revenire
```


5.5. FORMATELE COM ŞI EXE

Sub sistemul de operare MS-DOS există trei tipuri de fişiere care sunt lansabile în execuţie: fişiere de tip .BAT, de tip .EXE şi de tip .COM. Fişierele de tip BAT sunt fişiere text care conţin comenzi DOS şi eventual directive care controlează ordinea de execuţie a comenzilor. La lansarea în execuţie a unui fişier de tip BAT sunt executate una după alta toate comenzile trecute în fişier, respectându-se semnificaţiile directivelor. Fişierele COM şi EXE sunt fişiere binare ce conţin instrucţiuni în limbaj maşină. În această secţiune ne vom ocupa de acestea două din urmă.

Din punct de vedere istoric primele fişiere apărute sunt cele de tip COM. Fişierele de tip COM sunt în general fişiere mici având lungimea de maximum 64 kocteţi.

Fişierele EXE sunt fişiere mari. Cu ajutorul lor se pot descrie programe complexe ,de aceea ele tind să devină generale, păstrându-se cele de tip COM numai acolo unde ele devin indispensabile.

5.5.1. Prefixul unui program executabil (PSP)

Imaginea în memorie a unui program de tip EXE sau COM începe cu un antet numit *PSP* (*Program Segment Prefix*). În momentul încărcării programului, imaginea lui în memorie este completată cu acest tabel special care are 256 octeţi. Informaţiile din PSP sunt utilizabile direct de către sistemul de operare DOS şi indirect de către utilizator. În figura 5.1 este indicată structura tabelei PSP. În utilitarul Norton Guide găsim informaţii detaliate despre această structură în cadrul meniului *DOS/ PSP Description.*

După cum se observă, în PSP există o serie de zone care în prezent sunt mai puţin folosite. Ele au fost introduse la prima versiune de DOS, pentru asigurarea compatibilităţii cu sistemul de operare CP/M (de unde s-a inspirat MICROSOFT pentru DOS V1.0). Începând cu DOS V2.0 ele nu se mai folosesc, dar sunt păstrate pentru a se asigura compatibilitatea programelor executabile DOS din versiunile mai noi cu cele din prima versiune.

Deplasament	Lungime	Semnificaţie
00h	2	Codul instrucţiunii INT 20h - terminare program
02h	2	Adresa de sfârşit a memoriei ocupate de program
04h	1	rezervat
05h	1	Codul instrucţiunii INT 21h -- funcţii DOS
06h	2	Memoria disponibilă (număr octeţi) în cadrul segmentului
08h	2	rezervat
0Ah	4	Adresa FAR a RTI 22h – aceasta furnizează adresa de revenire din programul curent
0Eh	4	Adresa FAR a RTI 23h - se utilizează pentru restaurarea vechiului handler dacă programul curent a deturnat cumva întreruperea 23h (*CTRL+Break*)

12h	4	Adresa FAR a RTI 24h - se utilizează pentru restaurarea vechiului handler dacă programul curent a deturnat cumva întreruperea 24h (*Critical error interrupt handler*)
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 şi FCB2, câte 16 octeţi pentru accesarea fişierelor standard de intrare şi ieşire (se evita utilizarea lor la momentul actual – păstraţi 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 transmişi în linia de comandă la lansarea în execuţie 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, informaţii 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)* şi PSP-ul conţine un pointer la începutul acestuia (la deplasament 2ch). Segmentul de mediu conţine şiruri 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 şirul de caractere *Valoareparam*. Construcţia se termină cu un octet conţinând valoarea zero.

Se ştie că o comandă DOS are forma:

```
...>numecomanda arg1,...,argn
```

Porţiunea arg1,...,argn se numeşte *coada liniei de comandă* şi ea este memorată în jumătatea a doua a tabelei PSP.

5.5.2. Structura unui program EXE

Un program de tip EXE poate avea oricâte segmente de tip cod, date sau stivă. Acestea pot fi prezente toate în memorie sau numai o parte dintre ele. Prezenţa opţională a unor segmente se realizează prin aşa-numitul 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 execuţiei unui program este activ un segment de cod, un segment de date şi un segment de stivă. Toate segmentele sunt plasate de regulă după PSP, însă ordinea lor nu este importantă. Stiva nu are implicit 64 kocteţi ca la programele de tip COM, ci poate fi dimensionată de către programator. În figura 5.2 este ilustrat un program EXE care are câte un singur segment din fiecare tip şi sunt plasate în ordinea cod, date, stivă.

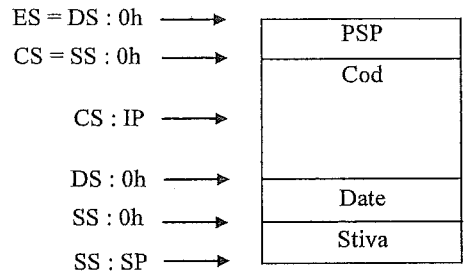


Fig. 5.2. Structura unui program EXE în memorie

În momentul încărcării în memorie a unui program EXE, se creează PSP, şi se depun în memorie segmentele. Se încarcă CS cu adresa segmentului de cod ce trebuie să fie activ primul. Prima instrucţiune executabilă poate să apară oriunde în segmentul de cod punctat de CS, CS:IP va puncta spre această instrucţiune.

La încărcare ES şi DS se iniţializează şi punctează la începutul PSP. Ulterior, programatorul are obligaţia să încarce (cel puţin) registrul DS cu adresa segmentului de date curent. De aceea în figura 5.2, DS:0h apare în două poziţii.

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 aceeaşi valoare ca şi 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 poziţii.

Dăm în continuare programul E.ASM, care tipăreşte numărul unităţilor de dischetă din sistem. El va avea declarată o stivă de 512 octeţi (200h).

```
stiva segment    stack 'stack'
      db    200h dup (?)
stiva ends

date  segment    'data'
      t1    db   'In sistem exista $'
      t2    db   ' unitati de discheta $'
      t3    db   ' Nu exista unitati de discheta instalate in sistem $'
date  ends

code  segment    'code'
      assume     cs:code, ds:date, ss:stiva
start:
      mov   ax, date
      mov   ds, ax        ;Baza segment de date

      int   11h           ;bitul 0 din configuraţia lui AX precizează dacă există unităţi de
                          ;dischetă instalate (valoarea 1) sau nu (valoarea 0)
      mov   bx, ax        ;reţinem în bx valoarea lui ax, pentru a păstra configuraţia obţinută
      and   bx, 1         ;forţăm la zero valoarea biţilor 1-15 din configuraţia lui AX,
                          ;păstrând neschimbată valoarea bitului 0
      jz    NuExista      ;dacă valoarea din AX este 0, înseamnă că nu există nici o unitate de
                          ;dischetă instalată
      lea   dx, t1        ;altfel, pregătim afişarea numărului unităţilor de dischetă din sistem
      mov   ah, 9
      int   21h           ;tipărire pe ecran a mesajului t1
                          ;în urma apelului întreruperii 11h, dacă bitul 0 din configuraţia lui
                          ;AX este 1, biţii 6 şi 7 din această configuraţie reprezintă numărul de
                          ;unităţi de dischetă din sistem, mai puţin una
      and   al, 0C0h      ;păstrăm în al valoarea biţilor 6 şi 7, iar valoarea celorlalţi biţi o
                          ;forţăm la zero
      mov   cl, 6
      shr   al, cl        ;shiftăm spre dreapta cu 6 poziţii configuraţia din al pentru a obţine
                          ;în al numărul format din cei doi biţi
      add   al,1+'0'      ;se adună valoarea 1 la valoarea din al pentru a obţine în al numărul
                          ;de unităţi de dischetă din sistem; deoarece acest număr este format
                          ;dintr-o singură cifră (e reprezentat pe 2 biţi), îl vom transforma în
                          ;caracterul corespunzător cifrei adunând la valoarea sa codul ASCII
                          ;al caracterului 0
      mov   dl, al        ;vom tipări caracterul astfel obţinut cu funcţia 02h a întreruperii 21h
      mov   ah, 02h
      int   21h           ;se tipăreşte pe ecran numărul de unităţi de dischetă din sistem
```



```
      mov   ah, 9         ;tipărire pe ecran a mesajului t2
      lea   dx, t2
      int   21h
      jmp Sfarsit

NuExista:
      mov   ah, 9         ;tipărire pe ecran a mesajului t3
      lea   dx, t3
      int   21h

Sfarsit:
      mov   ax, 4c00h
      int   21h           ;terminarea programului
code  ends
      end   start
```

La lansarea în execuţie cu TD a lui E.EXE, regiştrii au avut valorile:

DS = ES = 628B CS = 62BE IP = 0000
SS = 629B SP = 01FE

După execuţia primelor două instrucţiuni, registrul DS a primit valoarea 62BB.

Un acelaşi program, în format EXE are dimensiuni mult mai mari decât echivalentul lui în format COM. Acest fapt se petrece deoarece fiecare fişier EXE începe cu un *antet* disc. Acesta este necesar pentru a se putea manipula mai multe segmente ale programului EXE, pentru a fi fixate valorile implicite ale regiştrilor la intrare, a se defini segmentele curente la lansarea în execuţie etc. Structura antetului EXE pe disc este dată în figura 5.3.

Deplasament	Lungime	Semnificaţie
00h	2	Semnătura EXE: 5A4Dh (codul 'MZ' Mark Zbirkowski)
02h	2	Lungime fişier mod 512
04h	2	Lungime fişier div 512
06h	2	Numărul total al adreselor relocabile (NAR)
08h	2	Lungimea în paragrafe (multiplu de 16) a antetului
0Ah	2	Numărul minim de paragrafe cerute în plus (0000)
0Ch	2	Numărul maxim de paragrafe cerute în plus (FFFE)
0Eh	2	Poziţia relativă a segmentului SS (paragrafe)
10h	2	Valoarea iniţială a registrului SP
12h	2	Suma de control (Checksum) a fişierului
14h	2	Valoare iniţială a registrului IP
16h	2	Poziţia relativă a segmentului CS (paragrafe)

18h	2	Adresa disc a tabelei de relocare (TR) (uzual 1Ch)
1Ah	2	Număr de suprapunere (pentru overlay) (2 octeţi)
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 puţin mecanismul de încărcare şi lansare în execuţie a unui program EXE. Ordinea segmentelor este fixată de către editorul de legături. Prima acţiune 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.

Aşa cum am văzut în capitolul 3, unele instrucţiuni citează ca operand un nume de segment. De exemplu, secvenţa:

```
mov  ax, SegmentDeDate
mov  ds, ax
```

încarcă registrul DS cu adresa segmentului cu numele SegmentDeDate. Editorul de legături plasează în codul primei instrucţiuni 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 **operaţie de relocare**. Aceasta este necesară deoarece deplasamentele sunt determinabile ca şi constante la momentul asamblării, însă adresele de segment nu. Ca urmare, elementele relocabile sunt de fapt operanzii din cadrul programului care reprezintă adrese de segment şi care trebuie acum ajustate cu o valoare rezultată din plasamentul concret în memorie al programului EXE (valoarea StartSeg). De exemplu în cadrul instrucţiunilor:

mov ax, *data* sau mov bx, *seg a*

elementele relocabile sunt *data* şi respectiv *seg a*.

Octeţii 6 şi 7 din antet conţin numărul total de astfel de citări ale numelor de segmente (număr notat NAR – Numărul Adreselor de Relocare - în figura 5.3). Octeţii 24 şi 25 (adresa 18h) din antet indică poziţia în antet (notată TR în figura 5.3), identică cu poziţia în fişierul disc, a începutului unei aşa 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 şi păstra integritatea antetului disc. Astfel, se consideră întregul fişier ca o succesiune de cuvinte. Conţinuturile negate ale acestor cuvinte sunt însumate modulo 65536, rezultatul obţinut fiind acest checksum. La încărcarea programului în memorie, sistemul de operare DOS repetă acest calcul şi compară cu ceea ce este în checksum, refuzând încărcarea în caz de neconcordanţă.

După acest control, se creează PSP. Apoi sunt încărcate segmentele şi se actualizează adresele relocabile. În sfârşit, sunt încărcate din antet regiştrii CS, SS, IP şi SP aşa după cum am arătat mai sus. Ca efect al încărcării CS:IP programul este lansat în execuţie.

Descriind într-o manieră algoritmică procesul de încărcare în memorie a unui program EXE, putem identifica următorii paşi:

1. Se creează în memorie o structura de date PSP (*Program Segment Prefix*) de lungime 256 de octeţi.

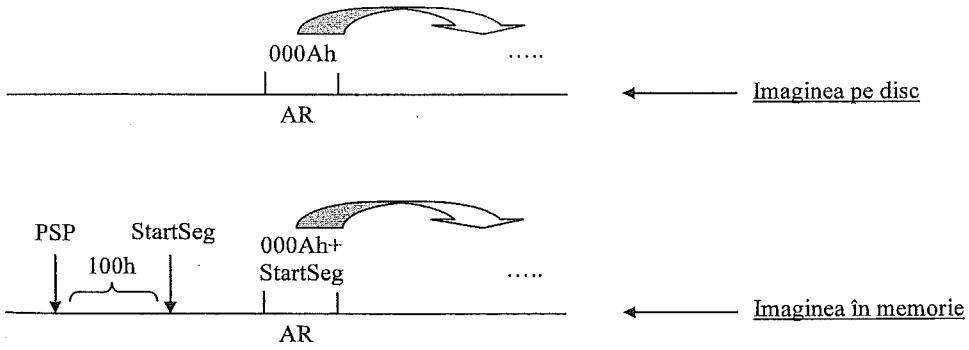
2. Se alege o adresă de la care începând se va încărca programul (de obicei aceasta este sfârşitul PSP). Fie această adresă StartSeg:

StartSeg := adresa PSP + 100h

3. Se încarcă la StartSeg porţiunea din fişierul EXE de după antetul de pe disc.

4. Se efectuează operaţia de relocare: pentru fiecare adresă de relocare AR (adică pentru fiecare adresă unde se găseşte un element ce trebuie relocat), la conţinutul adresei AR+StartSeg se adună StartSeg; în notaţie C acest lucru se poate exprima

[AR+StartSeg] += StartSeg



5. Se alocă dacă este posibil memoria necesară în plus în funcţie de valorile din antet (de obicei pentru heap).

6. Se iniţializează regiştrii astfel:

CS := CS relativ (valoarea din antetul EXE) + StartSeg
IP := IP iniţial (valoarea din antetul EXE)
SS := SS relativ (valoarea din antetul EXE) + StartSeg
SP := SP iniţial (valoarea din antetul EXE)
ES := DS := adresa de început a PSP

5.5.3. Structura unui program COM

Un fişier de tip COM are o structură simplă. El conţine imaginea (binară) a conţinutului ce va fi încărcat în memorie după PSP pentru a obţine un program lansabil în execuţie. În figura 5.4 este ilustrată imaginea din memorie a unui program COM, cu indicarea valorilor iniţiale ale regiştrilor de segment şi a registrului IP. Un program COM nu are antet pe disc! Numai programele EXE au aşa ceva.

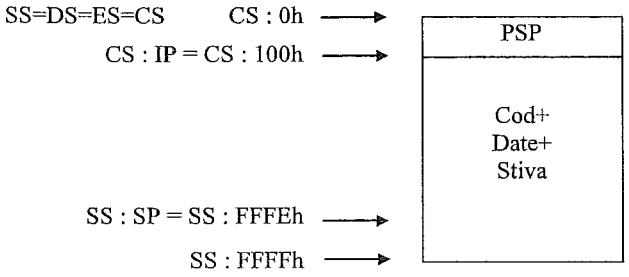


Fig. 5.4. Imaginea unui program COM în memorie

În momentul încărcării în memorie a unui program COM se creează PSP, se iniţializează toţi regiştrii de segment ca în figura 5.4 şi se dă controlul instrucţiunii aflate la adresa 100h, adică primul octet care se află după PSP. Pointerul de stivă SP are cea mai mare adresă posibilă, adică FFFEh, în ideea că programul are 64 kocteţi iar stiva urcă spre adrese mici.

Pentru ca un program scris în limbaj de asamblare să fie de tip COM trebuie îndeplinite următoarele cerinţe:

1. Programul trebuie să conţină un singur segment (toate referirile la cod, date şi stivă se fac în cadrul aceluiaşi segment); în consecinţă directiva assume este de forma:

```
assume cs:<nume>,ds:<nume>
```


unde <nume> este acelaşi atât pentru CS cât şi pentru DS.

2. Imediat după declaraţia de segment trebuie sa apară o directivă:

```
ORG   100h
```

indicând faptul că instrucţiunile şi datele încep de la adresa 100h. După ORG urmează o instrucţiune etichetată care trebuie să fie prima instrucţiune executabilă. Numele acestei etichete trebuie să apară în linia END a programului.

3. Datele pot fi plasate oriunde între instrucţiuni, singura condiţie (dependentă numai de programator) este să nu interfereze datele cu instrucţiunile. Pentru aceasta programatorul trebuie să izoleze zonele de date prin instrucţiuni de salt corespunzătoare.

4. Regiştrii de segment sunt iniţializaţi automat (ca în figura 5.4), deci utilizatorul nu mai trebuie să-i încarce cu valori iniţiale (în sensul celor discutate în 3.3.1.2).

5. În cadrul programului nu trebuie să apară elemente relocabile, adică operanzi nume de segmente.

Să considerăm programul echivalent COM al celui EXE din secţiunea anterioară, care tipăreşte numărul unităţilor de dischetă din sistem. Textul programului C.COM care face acest lucru este dat mai jos.

```
        code  segment
        assume      cs:code,ds:code
        org   100h

start:
        jmp   inceput       ;Aşa se evita interferarea datelor cu codul
t1      db    'In sistem exista $'
t2      db    ' unitati de disc $'
t3      db    ' Nu exista unitati de discheta instalate in sistem $'

inceput:
        int   11h
        mov   bx, ax
        and   bx, 1
        jz    NuExista

        lea   dx, t1
        mov   ah, 9
        int   21h
```

```
        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 condiţiile de mai sus), cu ajutorul comenzii DOS EXE2BIN:

...>EXE2BIN C.EXE C.COM

Dacă programul EXE nu respectă condiţiile impuse unuia COM şi 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 şi .COM

În ceea ce priveşte depanarea unui program .EXE, folosirea opţiunilor /zi pentru asamblare şi /v pentru linkeditare,

```
>tasm /zi prb
>tlink /v prb
```

provoacă includerea de informaţii simbolice pentru depanare la nivelul codului obiect şi respectiv la nivelul celui executabil. Includerea informaţiei simbolice înseamnă reţinerea 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 şi nu doar prin adresa care o reprezintă. În mod implicit, fără specificarea opţiunilor de mai sus, asamblorul nu include informaţiile simbolice în formatul obiect pe care îl generează.

De exemplu, dacă avem definită variabila

```
a dw 5
```

fişierele .obj şi .exe vor conţine informaţia simbolică ce face asocierea între numele a şi 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 instrucţiunea:

```
mov ax, a
```

vom obţine astfel la depanare:

```
mov ax, a
```

în loc de

```
mov ax, [0005h]
```

Includerea acestor informaţii simbolice pentru depanare în fişierul .exe face ca dimensiunea acestuia să fie mai mare decât dacă nu ar conţine aceste informaţii.

În ceea ce priveşte depanarea unui program .COM, care nu poate avea o dimensiune mai mare de 64 kocteţi (din care 256 octeţi sunt folosiţi pentru PSP), dimensiunea acestuia nu permite includerea informaţiilor simbolice pentru depanare. Acesta 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 de o parte avem fişierul pexe.asm din care vom obţine un program .EXE şi fişierul pcom.asm din care vom obţine un program .COM. Informaţiile obţinute la depanare sunt prezentate comparativ.

pexe.asm

```
assume cs:code, ds:data
data segment
        a db 5
        b db 7
data ends
code segment
start:
        mov ax, data
        mov ds, ax

        mov al, a
        add al, b

        mov ax, 4C00h
        int 21h
code ends
end start
```

Obţinere pexe.exe:

```
tasm /zi pexe
tlink /v pexe
td pexe
```

Depanare pexe.exe
(la nivel simbolic)

```
mov ax, data
mov ds, ax

mov al, a
add al, b

mov ax, 4C00h
int 21h
```

pcom.asm

```
code segment
assume cs:code
org 100h
start:
        jmp RealStart
        a db 5
        b db 7

RealStart:
        mov al, a
        add al, b

        mov ax, 4C00h
        int 21h
code ends
end start
```

Obţinere pcom.com:

```
tasm /zi pcom
tlink /t pcom
td pcom
```

Depanare pcom.com
(fără includerea informaţiilor simbolice)

```
jmp 0105

mov al, [0103]
add al, cs:[0104]

mov ax, 4C00
int 21
```


CAPITOLUL 6

REDIRECTAREA ÎNTRERUPERILOR

Sistemele 8086 folosesc un mecanism vectorizat (cu indirectare) de gestiune a întreruperilor. În loc de a memora rutinele de tratare a întreruperilor (RTI sau handler) la adrese fixe în memorie, proiectanţii sistemului de operare MS-DOS au ales varianta memorării adreselor acestor rutine într-un vector aflat la o locaţie de memorie cunoscută (0000:0000). Acest mod de gestiune a întreruperilor permite un lucru important şi anume posibilitatea de a redefini comportamentul unui handler în timpul funcţionării sistemului de operare. Pentru acest lucru este suficientă memorarea unei noi adrese pentru rutina de tratare a întreruperii în tabela de vectori.

6.1. REDIRECTAREA ÎNTRERUPERILOR

În general redirectarea unei întreruperi se face pentru a implementa o funcţionalitate care lipseşte în cadrul sistemului de operare, pentru a îmbunătăţi sau optimiza unele rutine BIOS sau pentru a modifica comportamentul sistemului de operare în cazuri particulare. Paşii necesari pentru redirectarea unei întreruperi implică în general:

- Scrierea noii rutine de întrerupere;
- Salvarea adresei vechiului handler (pentru refacerea acesteia la terminare sau pentru apelul handler-ului original);
- Modificarea adresei handler-ului în tabela de vectori;
- Utilizarea directă sau indirectă a rutinei de întrerupere modificate;
- Restaurarea handler-ului original (opţional – doar atunci când dorim să readucem comportamentul sistemului la cel original);

Modificarea adresei rutinei de tratare a unei întreruperi se poate face în două moduri:

1. direct prin memorarea noii adrese FAR în tabela vectorilor de întrerupere;
2. prin intermediul funcţiei DOS 25h;

Modificarea direct în tabela de vectori de întreruperi se poate face cu o secvenţă de cod similară cu cea descrisă mai jos. Această secvenţă de cod iniţializează vectorul întreruperii 77h cu adresa rutinei *HandlerNou*:

```
mov ax, 0
mov es, ax
cli
mov word ptr es:[4*77h], offset Newhandler
mov word ptr es:[4*77h+2], seg NewHandler
sti
```

Secvenţa de cod anterioară dezactivează întreruperile 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 întreruperea pe care o modificăm, exact între cele două instrucţiuni MOV WORD PTR, transferul se face la noul deplasament, însă în cadrul vechiului segment. Dezactivarea întreruperilor (sti) este necesară atunci când se modifică rutinele de tratare a întreruperilor ce pot fi generate asincron. În această categorie intră de obicei întreruperile ce deservesc un echipament hardware, întreruperile de ceas, etc. Acestea sunt generate automat de către sistem la apariţia unui eveniment (ceas, tastatură etc). Ele pot apare în orice moment şi de aceea se numesc *asincrone*. După actualizarea noii adrese a rutinei de tratare a întreruperii se reactivează întreruperile pentru a permite funcţionarea corectă a sistemului. Având în vedere că adresa unui handler se reprezintă pe 4 octeţi (un cuvânt pentru offset şi un cuvânt pentru adresa de segment), rezultă că deplasamentul în cadrul tabelei pentru întreruperea 77h se află la adresa 4*77h. Conform convenţiei 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 noului handler, iar la adresa 4*77h+2 adresa de segment a acestuia.

Modificarea adresei RTI cu ajutorul funcţiei DOS 25h se face printr-o secvenţă 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 întreruperii
mov dx, seg NewHandler
mov ds, dx
mov dx, offset NewHandler
int 21h
pop ds                      ;reface registrul DS (înapoi spre segmentul de date)
```

Secvenţa de cod de mai sus depune în registrul DS adresa de segment a noului handler, în DX deplasamentul noului handler, în AL numărul întreruperii şi apelează funcţia DOS 25h. În acest caz funcţia DOS 25h dezactivează întreruperile în decursul modificării adresei RTI. Chiar dacă această secvenţă de cod pare mai complicată decât cea care accesează direct tabela vectorilor de întrerupere, ea este mai sigură. Aceasta deoarece multe dintre aplicaţiile DOS monitorizează modificările aduse tabelei vectorilor de întrerupere prin intermediul DOS (funcţia 25h). Modificarea directă a tabelei de vectori scurt-circuitează mecanismul de monitorizare şi astfel aceste aplicaţii nu îşi mai pot da seama de modificările apărute, fapt ce poate duce la funcţionarea lor incorectă.

Pentru a putea reface, sau atunci când este nevoie apela handler-ul original al unei întreruperi, este necesar să salvăm adresa acestuia înainte de a face redirectarea spre rutina noastră de tratare. Următoarele două secvenţe de cod ilustrează modul în care putem realiza acest lucru, atât prin accesul direct la tabela vectorilor de întrerupere cât şi prin utilizarea funcţiei 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 spaţiu de stocare pentru adresa vechiului handler în segmentul de cod (oldint77), secvenţa de mai sus salvează în dublul cuvânt de la adresa oldint77, adresa rutinei de tratare a întreruperii 77h. Acelaşi lucru se poate face folosind funcţia DOS 35h:

```
oldint77   dd ?
...
...
mov ax, 3577h
int 21h                            ;se obţine în ES:BX adresa handler-ului curent
mov word ptr cs:oldint77, bx
mov cs:[oldint77+2], es
```

În general există puţine cazuri în care înlocuim complet funcţionalitatea unei rutine de tratare de întrerupere. De cele mai multe ori modificăm doar comportamentul unei funcţii a întreruperii sau modificăm comportamentul în anumite cazuri speciale (de exemplu rescriem handlerul întreruperii de tastatură 09h pentru a putea intercepta activarea unor anumite combinaţii 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 întreruperii. Acesta este un alt motiv pentru care este necesară salvarea adresei handler-ului original. Procesul duce la un mecanism de *înlănţuire* a apelurilor către handler-ele de întrerupere. Acest mecanism este extrem de util pentru buna funcţionare a programelor *TSR* (*Terminate and Stay Resident*) după cum vom vedea în continuare. Această partajare/înlănţuire a vectorilor de întrerupere este simplu de implementat în cazul în care fiecare RTI salvează adresa handler-ului vechi de întrerupere. Pentru a apela vechiul handler de întrerupere atunci când acesta este salvat în variabila dublucuvânt oldint77 vom proceda în felul următor:

```
pushf                      ;plasarea flag-urilor pe stivă
call dword ptr cs:oldint77
....                       ;cod executat după apelul handler-ului original
iret                       ;terminarea execuţiei handler-ului curent (cel nou) şi
                           ;întoarcerea în contextul aplicaţiei întrerupte
```

în cazul în care dorim să apelăm vechiul handler şi apoi să continuăm execuţia handler-ului nostru (folosind eventual efectul produs de handler-ul original), sau:

```
jmp dword ptr cs:oldint77          ;salt la handler-ul original al întreruperii
```

în cazul în care după apelul handler-ului original controlul se redă aplicaţiei întrerupte fără a mai reveni în codul handler-ului nostru. În primul caz se poate remarca simularea unei instrucţiuni *int* prin plasarea flag-urilor pe stiva şi apelul far al handler-ului. Este necesară plasarea flag-urilor pe stivă întrucât handler-ul original se termină printr-o instrucţiune *iret* (echivalentă cu o secvenţă: *popf + retf*). Plasarea flag-urilor pe stivă permite astfel terminarea corectă a handler-ului original. În cazul în care apelul este de tip *jump*, nu mai este necesară salvarea flag-urilor pe stivă întrucât acestea sunt deja plasate pe stivă la apelul iniţial de către sistem al handler-ului prin instrucţiunea *int* corespunzătoare sau la generarea întreruperii de către sistem.

Se poate observa de asemenea că atât la apelul de tip *call* cât şi la cel de tip *jump* adresa vechiului handler este specificată complet – raportată la segmentul de cod CS. Aceasta deoarece la intrarea în handler în afară de perechea de regiştri CS:IP care pointează la instrucţiunea curentă, restul regiştrilor au valorile pe care le aveau în aplicaţia întreruptă (în cazul unui handler care gestionează un eveniment sistem sau o întrerupere hardware) sau cele pregătite special la apelul întreruperii (în cazul în care întreruperea a fost generată cu un apel *int*). Este greşit să presupunem la intrarea în handler-ul nostru că registrul DS este setat spre segmentul care memorează variabilele din cadrul codului nostru.

6.2. PROGRAME TSR

Aplicaţiile MS-DOS obişnuite au o natură *tranzientă* (temporară). Ele sunt încărcate în memorie, executate iar apoi se termină urmând ca DOS să folosească memoria pentru următorul program care este lansat în execuţie. Programele *rezidente* sunt aplicaţii care urmează în general aceeaşi regulă cu excepţia dealocării memoriei. La terminarea programului, acesta nu returnează sistemului MS-DOS întreaga cantitate de memorie alocată. O parte a programului rămâne încărcată în memorie gata de a fi activată de un alt program în viitor. Acest tip de aplicaţii sunt cunoscute şi sub numele de aplicaţii *Terminate and Stay Resident (TSR)*, şi sunt folosite în general pentru a introduce facilităţi de multitasking în cadrul unui sistem de operare monotask. Până la apariţia sistemelor de operare din familia Windows, aplicaţiile rezidente erau singura metodă prin care mai multe aplicaţii puteau să coexiste în memorie în acelaşi timp. Chiar dacă apariţia sistemelor din familia Windows a diminuat sau înlăturat în unele cazuri necesitatea programelor rezidente şi a procesării în fundal oferită de acestea, ele mai sunt încă folosite la aplicaţiile native DOS, antiviruşi, drivere DOS, *patch-uri live*, etc.

6.3. HARTA DE MEMORIE DOS ŞI PROGRAMELE TSR

La demararea (procesul de boot) al sistemului de operare DOS (pe un calculator care rulează nativ sistemul MS-DOS) acesta iniţializează anumite zone de memorie predefinite şi încarcă codul executabil al sistemului de operare. Memoria care rămâne liberă după încărcarea sistemului de operare este folosită pentru rularea programelor utilizator, în general în regim monotask. În cadrul

sistemelor de operare din familia Windows NT, se foloseşte mecanismul de „maşină virtuală" pentru rularea aplicaţiilor DOS şi a interpretorului de comenzi. Sistemul de operare gazdă (Windows NT) simulează o maşină hardware virtuală în cadrul căreia rulează apoi sistemul de operare MS-DOS. Deşi există unele diferenţe din punct de vedere al accesării componentelor hardware, conceptele pe care le vom descrie în cadrul acestui capitol sunt valabile şi pentru sistemele DOS emulate virtual. Acolo unde acest lucru nu este valabil vom diferenţia explicit acest lucru.

Harta de memorie a unei maşini ce rulează sistemul de operare DOS arată, după încărcarea acestuia, similar cu cea prezentată în figura de mai jos:

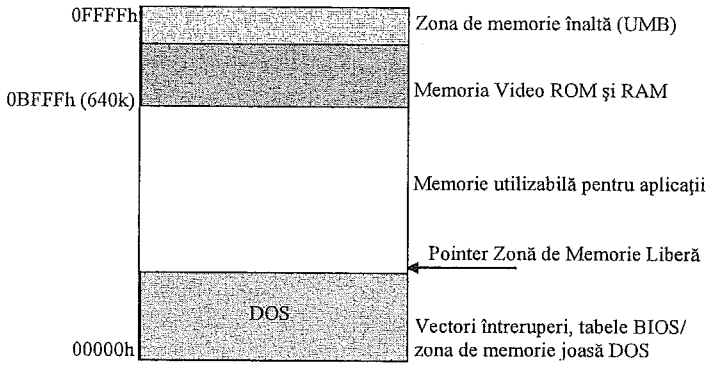


Fig. 6.1. Harta de memorie a sistemului de operare DOS după încărcare, fără aplicaţii active.

DOS gestionează un pointer care indică zona liberă de memorie după încărcarea sistemului de operare. Această zonă se află după spaţiul de memorie alocat tabelei vectorilor de întrerupere, tabelelor BIOS şi DOS şi este limitată superior de graniţa celor 640K practic adresabili pe sistemele DOS.

Notă:
Procesoarele 8086, 80186 dispuneau la introducerea lor pe piaţă de bus-uri de adresă cu 20 canale (20 biţi). Consecinţa directă a fost limitarea memoriei maxime adresabile la un spaţiu de adresă cuprins între 0 şi 2^{20}=1024Kb=1Mb. În mod real adresarea se face folosind componentele ***segment*** *şi* ***offset****, ambele reprezentate pe 16 biţi. Fiecare segment conţine astfel 2^{16} octeţi = 64K. Din cei 20 biţi a unei adrese fizice mai rămân 4 pentru stabilirea adresei de segment din cei 16 biţi a registrului de segment. Restul combinaţiilor între regiştrii de segment şi offset au ca rezultat valori de adrese fizice ce pot fi obţinute deja cu politica descrisă mai sus. Mai precis din cele 2^{32}combinaţii posibile de valori segment+offset doar 2^{20} sunt unice. Din spaţiul maxim de 1Mb adresabil, zona de memorie de peste 640Kb are în cadrul MS-DOS întrebuinţări speciale.*

Nativ MS-DOS nu poate rula mai multe aplicaţii simultan (este un sistem de operare monotask). MS-DOS întotdeauna încarcă în memorie programul de executat la adresa indicată de pointerul spre *Spaţiul de Memorie Liberă* şi îi alocă întreaga zonă de memorie liberă, până la adresa 0BFFFh. Practic întreaga memorie RAM disponibilă este alocată programului în curs de execuţie. Harta de alocare a memoriei MS-DOS atunci când în sistem rulează o aplicaţie este prezentată în figura 6.2.

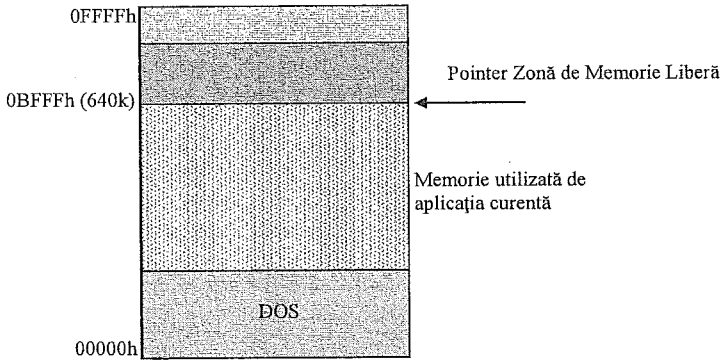


Fig. 6.2. Harta de memorie a sistemului de operare DOS cu o aplicaţie activă în sistem.

Atunci când o aplicaţie se termină prin invocarea funcţiei DOS 4Ch, sistemul de operare eliberează întreaga zonă de memorie alocată acesteia şi resetează pointerul *Zonei de Memorie Liberă* imediat deasupra componenţei MS-DOS din zona joasă de memorie. Funcţia DOS 31h schimbă comportamentul la terminarea unei aplicaţii în modul ilustrat în figura 6.3.

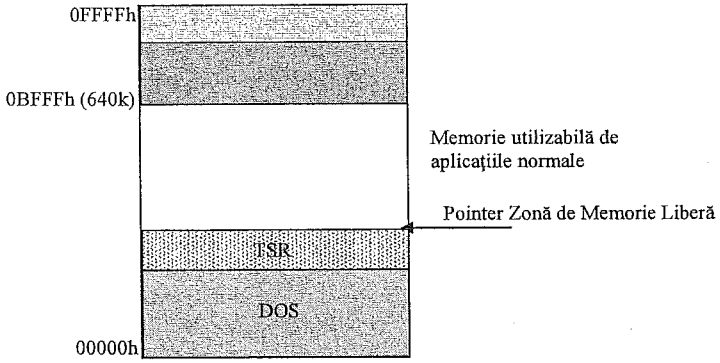


Fig. 6.3. Harta de memorie DOS după terminarea unei aplicaţii rezidente.

În acest caz DOS face o excepţie: nu eliberează întreaga cantitate de memorie. Acest apel de tip *terminate and stay resident* menţine alocat în memorie blocul de la începutul zonei alocate aplicaţiei având dimensiunea stabilită în registrul DX. Dimensiunea acestui bloc de memorie este specificată în paragrafe (un paragraf = 16 octeţi). La apelul funcţiei 31h, DOS setează de fapt pointerul *Zonei de Memorie Liberă* astfel încât să indice locaţia de memorie aflată la DX*16 octeţi „deasupra" PSP-ului programului. La execuţia unei alte aplicaţii în sistem, DOS va aloca doar memoria care începe la noua poziţie a pointerului spre *Zona de Memorie Liberă* indicată în figura 6.3. Acest lucru permite protejarea spaţiului de memorie în care se află încărcat programul TSR. Harta alocării memoriei după încărcarea unui program TSR şi a unei aplicaţii normale DOS este ilustrată în figura următoare:

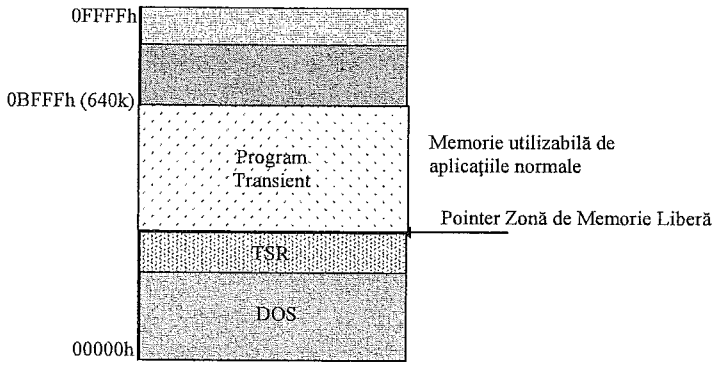


Fig. 6.4. Harta alocării memoriei DOS după încărcarea unui TSR şi a unei aplicaţii tranziente.

La terminarea execuţiei aplicaţiei tranziente (care este încărcată „peste" TSR), DOS va elibera doar memoria alocată acesteia, protejând încă o dată spaţiul de memorie alocat TSR-ului.

Aspectul cel mai important la utilizarea funcţiei DOS 31h este calcularea corectă a numărului de paragrafe care trebuie să rămână rezidente. Cea mai mare parte a programelor TSR conţin două secţiuni: o secţiune *tranzientă* şi o secţiune *rezidentă*. Partea tranzientă conţine datele, programul principal şi rutinele de suport care se execută atunci când lansăm programul în linie de comandă. Această porţiune de cod nu se mai execută în general niciodată după instalarea programului TSR în memorie. În consecinţă 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 puţin pentru restul programelor care vor rula în sistem.

Partea rezidentă este cea care rămâne în memorie şi implementează funcţionalităţile 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 sa fie una similară cu cea ilustrată în figura 6.5. Pentru ca un TSR să poată funcţiona corect este necesar să organizăm codul

şi datele de aşa natură încât secţiunile rezidente ale programului să fie încărcate la adresele mici de memorie, iar cele tranziente la adresele înalte de memorie. În general toate asambloarele oferă facilităţi care permit controlul ordinii de încărcare a segmentelor în memorie. Cea mai simplă soluţie pentru ordinea de încărcare a segmentelor în memorie şi cea care funcţionează cu toate asambloarele este aceea de a organiza atât datele cât şi codul într-un singur segment care apare primul în fiecare fişier sursă al programului. În acest caz link-editorul va comasa definiţiile din toate fişierele ale segmentului respectiv într-un singur segment care va fi primul încărcat în memorie.

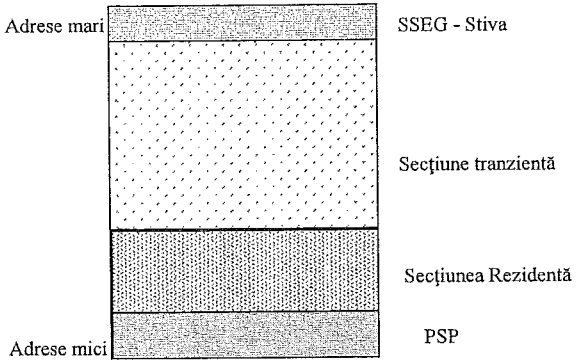


Fig. 6.5. Organizarea în memorie a unui program TSR.

Înainte de a trece la alte aspecte legate de programele TSR mai trebuie să amintim aici un ultim detaliu legat de gestiunea memoriei – accesarea datelor în secţiunea rezidentă după instalarea programului. Procedurile din cadrul unui program TSR devin active ca urmare a apelului direct dintr-un alt program sau prin intermediul unei întreruperi hardware. După intrarea în secţiunea rezidentă, rutina apelată poate presupune că anumiţi regiştri conţin parametri cu anumite valori. Nu putem însă face nici o presupunere aici privitoare la valorile stocate în regiştrii de segment. Singurul registru de segment care conţine o valoare semnificativă semantic este registrul de cod CS. Restul regiştrilor de segment vor trebui iniţializaţi cu valori semnificative pentru aplicaţia TSR. La terminarea rutinei TSR aceasta va restaura valorile regiştrilor de segment pentru a permite execuţia corectă a aplicaţiei care a fost întreruptă.

6.4. TSR-URI ACTIVE ŞI TSR-URI PASIVE

Microsoft identifică două tipuri mari de programe TSR: active şi pasive. Un TSR pasiv este activat printr-un apel explicit *call* din aplicaţia care rulează în sistem. Un TSR activ este acela care răspunde la întreruperile hardware generate de sistem pe parcursul funcţionării acestuia.

TSR-urile sunt cu mici excepţii rutine de tratare de întrerupere. TSR-urile active sunt în general RTI-uri ale unor întreruperi hardware generate de sistem, iar cele pasive sunt handlere ale unor întreruperi generate prin apeluri de tip *int*.

TSR-urile pasive reprezintă echivalentul bibliotecilor de programe din limbajele de nivel înalt. Ele extind de obicei serviciile implementate la nivel DOS sau BIOS. De exemplu dacă dorim să redirectăm toate caracterele trimise de o aplicaţie către imprimantă vom redirecta întreruperea 17h şi vom intercepta toate datele destinate imprimantei. Putem să adăugăm funcţionalităţi noi unei rutine BIOS prin înlănţuirea unei RTI proprii în lanţul de handlere ale vectorului de întrerupere corespunzător. De exemplu putem implementa o funcţie BIOS nouă pentru interfaţa video prin redirectarea întreruperii 10h şi gestiunea unui nou cod de funcţie în registrul AH. Pe de altă parte TSR-urile pasive pot implementa un set nou de servicii implementate la nivel BIOS prin rescrierea unei RTI pe un vector neutilizat de BIOS. Un exemplu concludent în această direcţie este implementarea serviciilor de gestiune a mouse-ului. Acestea sunt implementate de un driver de tip TSR folosind întreruperea 33h.

TSR-urile active sunt implementate pentru a defini direct serviciile unei întreruperi hardware, sau pentru a permite activarea periodică a TSR-ului prin înlănţuirea în vectorul de întrerupere a unei întreruperi hardware. Aşa cum am văzut deja în secţiunile anterioare există o multitudine de întreruperi hardware care sunt generate de către sistem la apariţia unui eveniment. Dintre acestea amintim întreruperea de ceas, întreruperea de tastatură, etc. La fiecare eveniment specific aceste întreruperi sunt activate şi permit TSR-ului care s-a înlănţuit în vectorul acestora de întrerupere să fie activat. Un exemplu concludent în acest sens sunt programele de tip *pop-up* care se *lipesc* în general de întreruperea de tastatură. Apăsarea unei taste apelează handler-ul întreruperii 9h şi permite activarea TSR-ului. Acesta citeşte de obicei tasta sau combinaţia de taste apăsate şi le compară cu o secvenţă care îi este cunoscută. În cazul în care combinaţia de taste este cea specifică TSR-ului acesta se activează. Viruşii sunt alte exemple de programe TSR active. Pentru a putea să îşi ducă la îndeplinire activităţile ne-ortodoxe ei se înlănţuie în vectorii unor întreruperi ale căror apeluri le permite să fie activaţi. În acest sens majoritatea viruşilor se inserează în vectorii întreruperii 21h şi 13h pentru a detecta orice încercare de scriere/citire dintr-un fişier sau pe/de pe disc. De fiecare dată când aceste operaţii sunt executate ei îşi inserează codul malefic în cadrul fişierului (mai ales dacă este vorba de un fişier executabil).

Spunem în general că un TSR este activ dacă el foloseşte măcar o componentă activă: cel puţin una dintre rutinele TSR-ului este activă.

Vom da în continuare un exemplu de program TSR activ care simulează activitatea unui *screen-saver* (economizor de ecran). El va detecta perioadele de timp în care tastatura este inactivă pe o durată mai mare de 10 secunde şi va afişa un mesaj corespunzător pe ecran. Desigur un adevărat economizor ecran va afişa în general o imagine animată sau va dezactiva ieşirea video pentru a nu utiliza inutil monitorul calculatorului.

Pentru a implementa funcţionalităţile acestui program vom redirecta întreruperea 1Ch (întrerupere software de timp) care este apelată automat de către sistem de 18.2 ori pe secundă.

Întreruperea 1Ch este varianta software a întreruperii 8h (întreruperea hardware de ceas a sistemului). Ea este oferită în general pentru a facilita programatorilor executarea unor acţiuni periodice. Spre deosebire de întreruperea 8h a cărei redirectare necesită *colaborarea* cu controller-ul de întreruperi al sistemului şi programarea acestuia, întreruperea 1Ch nu necesită acest lucru. Folosim întreruperea 1Ch pentru a măsura intervalul de timp scurs între interacţiunile utilizatorului cu tastatura. Vom redirecta de asemenea întreruperea 9h de tastatură pentru a determina momentele în care utilizatorul apasă o tastă. Întreruperea 9h este generată atât la apăsarea unei taste cât şi la eliberarea acesteia. Codul scan citit la eliberarea unei taste este codul scan al tastei respctive la care se adaugă valoarea 80h.

În continuare prezentăm codul sursă al programului TSR:

IdleKeybMon.asm

```
cseg segment para
assume cs:cseg,ds:cseg, es:cseg

; necesar pentru a putea calcula dimensiunea codului rezident
RezStart equ $

start:
   jmp begin                          ;salt peste secţiunea de date rezidentă

msg         db 13,10,'Tastatura inactiva in ultimele 10 sec',13,10
msg_len     EQU $-msg

t_out_sec   EQU 10                     ;numărul de secunde de inactivitate
timeout     EQU t_out_sec * 18         ;numărul de apeluri ale într 1Ch corespunzător

crt_timeout dw 0                       ;numărul curent de secunde scurs de la ultima apăsare
                                       ;a unei taste
; Aceste două variabile vor conţine adresele originale ale întreruperilor 1Ch şi 9h
oldInt1C    dd ?
oldint9     dd ?

;Handler întrerupere 1Ch (timp).
;Această rutină incrementează un contor la fiecare apel. Atunci când valoarea contorului depăşeşte
;o valoare prag se generează afişarea unui mesaj pe ecran. De fiecare dată este apelat handler-ul
;original al întreruperii 1Ch pentru a permite şi celorlalte aplicaţii TSR potenţiale să îşi desfăşoare
;corect activitatea

int1c proc far
   cli
   inc word ptr cs:crt_timeout         ;incrementează contorul de apeluri
```



```
   cmp cs:crt_timeout,timeout          ;dacă contorul depăşeşte numărul de apeluri
                                       ;echivalent unui interval de t_out_sec se cere afişarea
                                       ;mesajului pe ecran.
   jb orig1c                           ;în caz contrar trece controlul la secţiunea de cod care
                                       ;apelează handlerul original

;salvăm pe stivă regiştrii pe care îi modificăm pentru a-i putea reface după afişarea mesajului.
;Handler-ele întreruperilor generate de sistem trebuie să păstreze în general nealterat mediul, adică
;regiştrii şi stiva pentru a permite reluarea corectă a execuţiei aplicaţiilor întrerupte.

   push ax
   push bx
   push cx
   push bp
   push es

   push cs
   pop es
   mov ah, 03h                         ;Funcţia 03h a întreruperii 10h (video)
   mov bh, 0                           ;numărul paginii video (mod de afişare text)
   int 10h                             ;Determină poziţia actuală a cursorului pe ecran raportând-o
                                       ;în perechea de regiştri DH (linia) şi DL (coloana)
   mov ax,1301h                        ;Funcţia 13h a INT 10h afişează pe ecran la poziţia
                                       ;determinată (linie,coloană) mesajul specificat în ES:BP.
                                       ;AL=1 – subserviciul 1 presupune actualizarea cursorului şi
   mov bl,07h                          ;utilizarea atributului (culoare font/fond=alb/negru)
                                       ;specificat în BL pentru tipărirea caracterelor şirului.
   mov bh,0                            ;numărul paginii video (mod de afişare text)
   mov cx, msg_len
   mov bp, offset msg
   int 10h
   mov cs:crt_timeout, 0               ;reiniţializează contorul de apeluri la 0
   pop es                              ;reface regiştrii modificaţi
   pop bp
   pop cx
   pop bx
   pop ax

orig1c:
   sti
   jmp dword ptr cs:oldint1c           ;apel la handler-ul original al întreruperii 1Ch (fără revenire
                                       ;la rutina curentă)
int1c endp
```

```
;Handlerul întreruperii 9h (tastatură)
;Această întrerupere este generată la fiecare apăsare sau eliberare a unei taste. La fiecare apăsare
;resetăm contorul de apeluri incrementat de către întreruperea 1Ch. Întrucât ne aflăm într-o regiune
;critică este necesar să dezactivăm întreruperile pe durata execuţiei acestui handler. Nu vrem ca o
;nouă întrerupere 9h să fie generată pe parcursul execuţiei handler-ului, care să întrerupă chiar
;execuţia în curs a handler-ului într. 9h. Pe măsură ce handler-ul întreruperii 1Ch incrementează
;contorul de apeluri (timp), la fiecare apăsare a unei taste resetăm acest contor. Folosim portul 60h
;pentru citirea fără consumarea codului scan de la controller-ul de tastatură.

int9 proc far
   cli                          ;dezactivăm întreruperile
   push ax                      ;salvăm pe stivă registrul AX pe care îl vom modifica
   in al, 60h                   ;citim din codul scan al tastei acţionate
   cmp al,80h                   ;verificăm că este vorba de apăsarea unei taste şi nu
                                ;eliberarea acesteia
   pop ax                       ;refacem conţinutul registrului AX modificat la citirea tastei
   ja orig9                     ;în cazul în care este vorba de o tasta eliberată trecem la
                                ;apelul handler-ului original al întreruperii 9h
   mov cs:crt_timeout,0         ;s-a apăsat o tastă => resetăm contorul de timp
orig9:
                                ;reactivăm întreruperile. Handler-ul original al într 9h le va
                                ;dezactiva singur dacă are nevoie
   sti
   jmp dword ptr cs:oldint9     ;salt fără revenire la RTI originală a întreruperii 9h
int9  endp
                                ;marchează sfârşitul secţiunii rezidente de cod
RezEnd equ $

msg_success db     'IdleKeybMon - monitorizeaza timpii morti la tastatura',13,10,'$'

                                ;programul principal care instalează TSR-ul în memorie.
                                ;Acesta face parte din secţiunea tranzientă de cod a TSR-ului.
begin:
                                ;iniţializează regiştrii de segment utilizaţi. DS conţine iniţial
                                ;adresa PSP-ului în cazul unui program de tip EXE.
   push cs
   pop ds
   mov ax, 351Ch                ;salvăm adresa handler-ului original al întreruperii 1Ch
   int 21h
   mov word ptr [oldInt1c],bx
   mov word ptr [oldInt1c+2],es
   mov ax,3509h                 ;salvăm adresa handler-ului original a întreruperii 9h
   int 21h
```



```
   mov word ptr [oldint9], bx
   mov word ptr [oldint9+2], es

   mov ax,251ch                 ;redirectăm întreruperea 1Ch către handler-ul nostru
   mov dx, offset int1c
   int 21h

   mov ax,2509h                 ;redirectăm întreruperea 9h către handler-ul nostru
   mov dx, offset int9
   int 21h

   mov ah,09h                   ;afişăm mesajul corespunzător instalării cu succes.
   lea dx, msg_success
   int 21h

;Calculăm dimensiunea secţiunii rezidente în paragrafe. Aceasta se face prin calcularea numărului
;de octeţi (RezEnd-RezStart) din secţiunea rezidentă la care se adaugă dimensiunea PSP-ului (256
;octeţi = 100h octeţi) totul rotunjit în sus la un multiplu de 16. Prin împărţirea la 16 obţinem
;numărul întreg corect de paragrafe ocupat de secţiunea rezidentă a TSR-ului

   mov dx, (RezEnd - RezStart + 0Fh + 100h)/16
   mov ax, 3100h                ;terminare program cu lăsare rezidentă a DX paragrafe
   int 21h

cseg ends
end start
```

6.4.1. Problema funcţiilor DOS non reentrante

O problemă spinoasă şi care dă multă bătaie de cap programatorilor de aplicaţii TSR este cea a codului reentrant, sau mai bine zis a codului non reentrant. Ea rezultă din natura asincronă a generării întreruperilor hardware precum cele care sunt activate la apăsarea unei taste, întreruperea de ceas, semnalizare date disponibile pe portul serial sau paralel, etc. Întrucât handlerele sunt activate de întreruperi hardware (asincrone) este posibil ca sistemul să fie în mijlocul execuţiei a aproximativ oricărei secvenţe de cod, fără ca noi să putem determina sau controla acest lucru. Problemele apar dacă programul TSR decide să apeleze cod „extern", precum funcţii DOS, rutine BIOS sau rutine ale unui alt TSR. De exemplu, aplicaţia principală care rulează în prim-plan se poate afla pe parcursul unui apel DOS în momentul în care întreruperea de ceas activează un TSR, fapt care întrerupe apelul DOS la un moment de timp în care CPU încă execută cod al funcţiei DOS. Dacă în acest moment TSR-ul activat încearcă să apeleze o funcţie DOS, acest lucru va duce la existenţa a două apeluri de funcţii DOS simultan. Din păcate, DOS nu este *reentrant* (nu permite ca mai multe apeluri DOS să fie activate în acelaşi timp), ceea ce creează de obicei probleme care duc la blocarea sistemului. Acesta este doar un exemplu care ilustrează posibilităţile de a genera

apeluri *reentrante*. În realitate, de fiecare dată când un TSR apelează alte rutine decât cele implementate direct în TSR trebuie să fim conştienţi de posibilitatea apariţiei 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 şi printr-un eveniment asincron (generarea unei întreruperi hardware) atunci nu există probleme din punct de vedere al reentranţei codului TSR-ului.

Aşa cum am descris în exemplul de mai sus, DOS este probabil problema cea mai spinoasă din punct de vedere al programării de TSR-uri şi reentranţei 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 specialiştii 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) şi când nu. Motivul care stă la baza rezolvării în acest mod al problemei: problema reentranţei 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 funcţiile DOS fără probleme.

MS-DOS furnizează un flag special pe un octet (denumit flag InDOS) care conţine valoarea zero dacă DOS nu este activ şi o valoare diferită de zero dacă DOS se află în decursul procesării unei cereri. Prin testarea valorii flag-ului InDOS o aplicaţie TSR poate să îşi „dea seama singură" dacă poate apela în siguranţă funcţii DOS. Dacă flag-ul InDOS conţine valoarea zero TSR-ul poate face apel la funcţii DOS. Dacă valoarea InDOS conţine o altă valoare atunci există pericolul de a ne suprapune apelul cu o funcţie DOS în curs. Există o funcţie DOS (GetInDosFlagAddress) care returnează adresa flag-ului InDOS. Pentru utilizarea acestei funcţii trebuie încărcat registrul AH cu valoarea 34h şi apelat DOS, care va returna adresa flag-ului InDOS în perechea de regiştrii 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
```

De fapt există două flag-uri care trebuie testate: InDOS şi CritError (flag-ul de eroare critică DOS). Ambele trebuie să conţină valoarea zero pentru a putea apela în siguranţă funcţii 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 şi sunt suprascrise la fiecare apel al unei funcţii DOS. Dacă alegem să facem un apel DOS atunci când acest flag este setat, cel mai probabil din cauza faptului că aplicaţia ce rulează în prim-plan a apelat o funcţie DOS care nu s-a putut executa corect, vom suprascrie detaliile legate de eroarea critică şi vom înşela în acest fel aplicaţia întreruptă. Aceasta nu îşi va da seama că apelul DOS executat a eşuat, fapt care va duce cu siguranţă la funcţionarea ei incorectă în continuare. Flag-ul de eroare critică DOS se află stocat pentru versiunile DOS 3.1 şi 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: "Întoarce-te mai târziu pentru e executa TSR-ul, atunci când funcţia MS-DOS activă întoarce controlul la programul utilizator". Cum putem însă implementa în practică acest lucru ? Dacă TSR-ul nostru este activat de întreruperea de tastatura la apăsarea unei combinaţii de taste şi 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 soluţie ar fi aceea de a obliga utilizatorul să activeze în mod repetat combinaţia de taste până când prinde un interval de timp între două apeluri DOS. Această soluţie nu este însă aplicabilă în practică din motive pe care cititorul le poate bănui cu uşurinţă. A doua soluţie este aceea de a redirecta în cadrul TSR-ului întreruperea de ceas, pe lângă cea de tastatură. Atunci când la apăsarea combinaţiei de taste descoperim în handler-ul întreruperii de tastatură că DOS este activ setăm un flag (CererеTSR) la nivelul TSR-ului care ne indică că am avut o cerere de activare a TSR-ului şi 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-ul original al întreruperii de ceas. Dacă flag-ul este setat atunci verifică flag-ul InDOS şi CritError. Dacă DOS este activ/ocupat RTI de ceas redă controlul handler-ului original. După ce DOS redă controlul aplicaţiei utilizator prima întrerupere 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 funcţii DOS. Desigur, primul lucru pe care TSR-ul îl va face la activare este acela de a reseta flag-ul intern CererеTSR astfel încât următoarele întreruperi de ceas să nu restarteze TSR-ul.

Soluţia de mai sus este valabilă în toate cazurile cu o singură excepţie. Există unele funcţii DOS care necesită un interval de timp nedefinit de execuţie. Este vorba de exemplu de funcţia 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 funcţii DOS. Implementarea DOS a acestor funcţii se bazează pe o buclă internă DOS care aşteaptă ca utilizatorul să apese o tastă. Şi până când utilizatorul face acest lucru flag-ul InDOS va rămâne nenul. Dacă TSR-ul pe care l-am implementat trebuie să scrie date într-un fişier la intervale regulate de timp, acest lucru nu va fi posibil dacă utilizatorul tocmai a plecat să ia masa în timp de DOS aşteaptă apăsarea unei taste sau introducerea unui şir de caractere de la tastatură.

Din fericire DOS furnizează o soluţie şi pentru această problemă prin implementarea unei întreruperi de *timp de inactivitate*. Cât timp DOS este într-un ciclu de aşteptare a unei operaţii cu un dispozitiv I/O (intrare-ieşire), apelează în mod continuu întreruperea 28h. Prin redirectarea vectorului întreruperii 28h, TSR-ul nostru poate determina cazurile în care DOS se află în bucle de aşteptare. Atunci când DOS apelează int 28h putem face în siguranţă apeluri DOS către funcţii ale căror număr de identificare este mai mare decât 0Ch. Pentru funcţiile care au identificator mai mic de 0Ch DOS foloseşte stiva aflată la o adresă fixă. Pentru a nu distruge stiva unui apel în curs printr-un alt apel este interzisă apelarea acestor funcţii DOS. De cele mai multe ori folosim întreruperea 21h pentru a afişa şiruri de caractere pe ecran (funcţia 09h). Conform remarcilor de mai sus, utilizarea acestei funcţii nu este posibilă în timpul unei întreruperi 28h. Pentru a face posibil apelul oricărei funcţii DOS atunci când DOS se află într-o buclă de aşteptare putem salva conţinutul anterior al stivei DOS într-o zonă de memorie din cadrul TSR-ului şi să refacem conţinutul stivei DOS după terminarea execuţiei curente a acestuia, sau atunci când nu mai facem

apeluri la funcţii DOS. Un exemplu de redirectare a întreruperii 28h cu salvarea stivei pentru a putea utiliza orice funcţie DOS este prezentat în fragmentul de cod de mai jos:

```
...
stiva_ss    dw cseg
      dw 256 dup(0)
stiva_sp      dw stiva_sp

save_ss    dw ?
save_sp      dw ?
...
int28 proc
   pushf
   call dword ptr cs:oldint28          ;simulează apel int 28h
   cli
   cmp byte ptr cs:CerereTSR,1         ;verifică cerere TSR
   jne int28_end

   mov word ptr cs:save_sp, sp         ;salvează stiva DOS curentă (adresa)
   mov word ptr cs:save_ss, ss
   mov ss, cs:stiva_ss                 ;setează stiva curentă la cea alocată în cadrul
   mov sp, cs:stiva_sp                 ;TSR-ului (256 octeţi) în segmentul de cod
   push cx
   push si
   push ds
   mov cx, 64
   mov ds, cs:save_ss
   mov si, cs:save_sp
 rep_save_stiva:                       ;salvează 64 cuvinte din stiva DOS originală
   push word ptr [si]                  ;în stiva nou alocată. Aceasta deoarece DOS
   inc si                              ;va folosi tot stiva sa internă la fiecare apel
   inc si
   loop rep_save_stiva

   mov byte ptr cs:flag,0
   ....
; Secvenţă de cod care foloseşte funcţii DOS sau apel la rutina principală a TSR-ului.
   ....

   mov cx,64
   mov ds, cs:save_ss
   mov si, cs:save_sp
   add si, 128
```



```
rep_restore_stiva:                     ;reface cele 64 de cuvinte salvate din stiva
   dec si                              ;DOS
   dec si
   pop word ptr [si]
   loop rep_restore_stiva

   pop ds
   pop si
   pop cx
   mov ss, cs:save_ss                  ;reface adresa stivei curente
   mov sp, cs:save_sp
int28_end:
   sti
   iret
int28 endp
```

Vom furniza spre sfârşitul acestui capitol un exemplu complet de program TSR care utilizează o mare parte din tehnicile descrise mai sus pentru a putea apela funcţii DOS.

6.4.2. Problema întreruperilor BIOS non reentrante

DOS nu este singura componentă non reentrantă a unui sistem. Există şi rutine BIOS care se află în aceeaşi categorie. Problema care se pune aici este generată de faptul că BIOS nu furnizează un flag InBIOS care să permită determinarea faptului că BIOS este activ sau nu. Din fericire rutinele BIOS nu sunt atât de sensibile la acest fenomen. Există unele funcţii ale unor întreruperi BIOS non-reeentrante şi există o multitudine de funcţii/întreruperi BIOS care pot fi apelate fără probleme din TSR-uri. Pentru cele care sunt non-reentrante singura soluţie este implementarea de rutine *wrapper* în jurul acestora. O astfel de rutină redirectează întreruperea BIOS fără a-i modifica cu nimic comportamentul în afara faptului că la intrarea în rutină setează un flag *InBIOS* implementat la nivel de TSR şi la ieşirea din RTI resetează acest flag. Un exemplu de astfel de *wrapper* este descris în secvenţa de cod de mai jos.

```
int17 proc far
   inc CS:InBIOS
   pushf
   call dword ptr CS:OldInt17
   dec CS:InBIOS
   iret
int17 endp
```

6.5. ÎNTRERUPEREA MULTIPLEX (INT 2Fh)

Atunci când instalăm un TSR care conţine componente pasive trebuie să alegem un vector de întrerupere pe care îl vom utiliza pentru a comunica cu rutinele pasive din cadrul TSR-ului. Pentru aceasta avem la îndemână două soluţii:

1. Alegerea la întâmplare a unui vector de întrerupere;
2. Alegerea unui vector de întrerupere deja definit în cadrul sistemului - care implementează o funcţie specifică;

Prima soluţie este nepotrivită din mai multe considerente. Dacă vectorul respectiv este deja folosit de către sistem, atunci rescrierea lui poate să ducă la disfuncţionalităţi în funcţionarea sistemului de operare. Chiar dacă introducem doar o funcţie suplimentară întreruperii alese, este posibil ca acea funcţie sa mai fie aleasă şi de alţi implementatori de TSR-uri sau este posibil ca funcţia să fie aleasă de chiar implementatorii sistemului de operare în viitor pentru implementarea unui serviciu specific.

Pentru a doua soluţie 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 îi vom redirecta/adăuga una sau mai multe funcţii. Pe de altă parte, dacă implementăm un driver de dispozitiv sau un TSR care nu extinde servicii existente problema alegerii vectorului de întrerupere prin care să comunicăm cu TSR-ul rămâne. Din fericire MS-DOS furnizează o soluţie şi în acest caz: întreruperea multiplex 2Fh. Această întrerupere a fost rezervată pentru a furniza un mecanism general de instalare, testarea prezenţei şi comunicare cu un TSR.

Pentru a utiliza întreruperea multiplex, aplicaţiile stochează un număr de identificare în registrul AH şi fac apel la INT 2Fh. Fiecare TSR din lanţul de handlere 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 lanţul vectorilor întreruperii 2Fh.

Desigur, această metodă rezolvă doar o parte a problemei, aceea a alegerii unui vector de întrerupere. Numărul de identificare trebuie să fie cunoscut atât de entitatea care verifică cât şi 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 aceluiaşi 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 aplicaţia care testează prezenţa TSR-ului?"

Problema enunţată mai sus poate fi rezolvată printr-un mic artificiu. Folosim prin convenţie funcţia (comanda) zero pentru a testa dacă un TSR este instalat sau nu. Orice aplicaţie va executa întotdeauna această funcţie pentru a determina dacă TSR-ul este instalat în memorie înainte de a executa alte funcţii ale TSR-ului. În mod normal funcţia zero va returna valoarea zero în registrul AL dacă TSR-ul nu este prezent în memorie sau 0FFh dacă TSR-ul este prezent. Desigur prezenţa unei valori 0FFh în AL ne indică doar prezenţa *unui* TSR instalat, dar nu ne garantează că TSR-ul

respectiv este cel pe care îl căutăm noi. Prin extinderea convenţiei făcute până acum însă, vom putea determina corect şi exact prezenţa TSR-ului care ne interesează în memorie. Pentru aceasta să presupunem că funcţia zero întoarce pe lângă valoarea din AL un pointer la string de identificare in regiştrii ES:DI. Prin compararea string-ului de la adresa ES:DI cu semnătura cunoscută de aplicaţia care testează prezenţa TSR-ului se poate detecta exact prezenţa unui TSR anume în memorie.

6.6. INSTALAREA UNUI TSR

Deşi 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 aplicaţii TSR care să funcţioneze cu mai multe copii simultan în memorie. Aceasta deoarece fiecare copie redirectează aceleaşi întreruperi şi îndeplineşte aceleaşi sarcini. Mai mult, instalarea de *N* ori, N>1 a unui TSR în memorie implică consumul unei cantităţi de N ori mai mare de memorie. În majoritatea cazurilor un astfel de scenariu nu poate decât să ducă la risipă de memorie şi chiar la blocarea sistemului. În consecinţă fiecare TSR trebuie să verifice la instalare existenţa unei copii deja instalare anterior. În cazul în care se detectează o copie deja instalată de obicei se afişează un mesaj informativ şi TSR-ul refuză instalarea.

Secvenţa de cod prezentată în continuare determină prezenţa unui TSR în memorie şi furnizează numărul de identificare al acestuia. Această secvenţă de cod ne dă răspunsul şi la a doua întrebare. Rutina determină în cazul în care TSR-ul nu este activ un număr de identificare liber şi resetează CF la terminare.

```
; Scanează toate ID-urile posibile între 255 şi 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, 0FFh                    ;verifică ID-urile posibile de la 255 la 0
```

```
CicluID:
   mov ah, cl                  ;depune în AH ID-ul curent
   push cx                     ;salvează CX pe stiva deoarece îl modificăm
   mov al, 0                   ;codul comenzii (funcţiei) – "Are you There?"
   int 2Fh
   cmp al, 0                   ;verifică dacă există TSR pentru ID-ul din AH
   pop cx
   je NextID                   ;dacă nu încercăm următorul ID
   push cx
   mov si, offset Sign         ;dacă da comparăm semnătura Sign cu cea
   mov cx, Sign_len            ;stocată la adresa ES:DI (TSR)
   cld
   rep cmpsb
   pop cx                      ;refacem CX pentru a putea relua bucla corect
   jne NextID                  ;dacă semnăturile nu coincid continuăm bucla
   jmp Installed               ;dacă semnăturile coincid TSR-ul nostru este instalat
NextID:
   mov FuncID, cl              ;acest ID este liber – folosit pentru a găsi ID-urile
   loop CicluID                ;nefolosite. Reluăm ciclul atât timp cât CX>0
   jmp NotInstalled            ;am verificat toate ID-urile. TSR-ul nu este instalat

Installed:
   Stc                         ;TSR instalat. Întoarcem:CF setat şi în reg. CL ID-ul
   mov FuncID, cl              ;Valoarea de returnat este cea a var. FuncID.
   ret
NotInstalled:                  ;TSR-ul nu este instalat.
   clc                         ;Întoarcem CF – resetat şi în CL un ID liber
   mov cl, FuncID              ;dacă CL=0 nu mai sunt ID-uri diponibile
   ret
check_installed endp
```

Rutina de mai sus verifică într-un ciclu toate numerele de identificare posibile între 255 şi 0. În cazul în care pentru un ID se detectează prezenţa unui TSR (AL=0FFh) se testează semnătura specifică a TSR-ului. La fiecare iteraţie rutina stochează în FuncID ultimul număr de identificare liber găsit. La terminarea rutinei flag-ul CF este setat dacă TSR-ul a fost găsit în memorie şi registrul CL conţine numărul de identificare al TSR-ului. Dacă TSR-ul nu a fost găsit în memorie CF este resetat şi în CL se întoarce un număr de identificare liber. Dacă numărul de identificare liber întors de rutină este zero, acest lucru semnifică că nu mai sunt numere de identificare libere (sunt prea multe TSR-uri instalate). Este sarcina apelantului să verifice valoarea întoarsă de rutina descrisă mai sus şi să ia o decizie în consecinţă.

Desigur fiecare TSR care se instalează şi foloseşte convenţiile descrise mai sus trebuie să redirecteze întreruperea 2Fh pentru implementarea acestor funcţionalităţi.

6.7. DEZINSTALAREA UNUI TSR

Dezinstalarea unui TSR este mai complicată decât procesul de instalare. Codul de dezinstalare trebuie să ducă trei sarcini la îndeplinire:

- Să oprească toate activităţile în curs ale programului TSR;
- Să refacă toţi vectorii de întrerupere modificaţi de TSR;
- Să dealoce memoria alocată astfel încât aceasta să poată fi utilizată de alte aplicaţii;

Dificultatea majoră o constituie ultimele două puncte. Nu este întotdeauna posibil să refacem vectorii de întrerupere modificaţi şi nu se poate dealoca întotdeauna memoria TSR-ului. Să vedem care sunt motivele care fac imposibile cele două activităţi.

În cazul în care codul TSR-ului încearcă doar să readucă la valorile originale vectorii de întrerupere poate să apară o problemă cu grave repercusiuni în funcţionarea sistemului. Dacă utilizatorul încarcă în memorie şi alte TSR-uri după TSR-ul nostru, este posibil ca aceste TSR-uri să redirecteze aceleaşi întreruperi, sau măcar unele dintre ele. Acest lucru duce la crearea unui lanţ precum cel ilustrat în figura de mai jos:

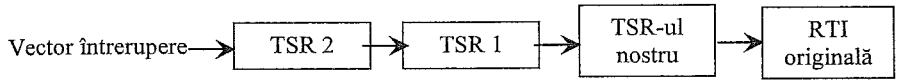


Fig. 6.6. Lanţ de RTI pentru mai multe TSR-uri instalate pe aceeaşi întrerupere.

În figura 6.6 am reprezentat grafic două TSR-uri care au fost instalate după "TSR-ul nostru". Ambele au redirectat aceeaşi întrerupere. Dacă în situaţia prezentă TSR-ul nostru reface vectorul întreruperii la cel original, salvat la instalare, configuraţia finală va fi cea prezentată în figura 6.7.

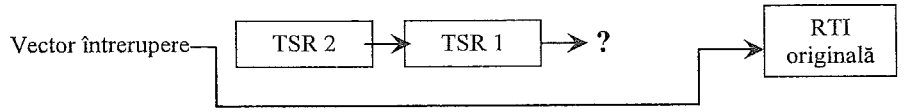


Fig. 6.7. Refacere RTI originală după instalarea mai multor TSR-uri pe aceeaşi întrerupere.

În figura 6.6 fiecare TSR a salvat adresa handler-ului original, *din punctul său de vedere*. Fiecare handler apelează RTI salvată, astfel încât orice întrerupere generată trece prin întreg lanţul de RTI-uri. În momentul în care TSR-ul nostru readuce vectorul de întrerupere la valoarea salvată (adresa originală) de el, practic RTI din TSR 1 şi TSR 2 sunt scoase din lanţ şi prin aceasta din uz. La generarea ulterioară a întreruperii, rutinele ei de tratare din TSR 1, respectiv TSR 2 nu vor mai fi apelate. Acest lucru duce în general la funcţionarea defectuoasă a celor două TSR-uri implicate. Ceea ce este şi mai rău este faptul că doar handler-ele întreruperilor comun redirectate de cele trei TSR-uri vor fi dezafectate, restul întreruperilor redirectate în TSR 1 şi TSR 2 rămânând active. Este

greu de prezis care va fi comportamentul sistemului într-o astfel de situaţie. Depinde de funcţionalităţile implementate de cele două TSR-uri rămase.

Cum putem rezolva această problemă? Soluţia cea mai simplă şi cea mai eficace atunci când detectăm un astfel de caz este să tipărim un mesaj de atenţionare şi să refuzăm dezinstalarea TSR-ului. Aceasta este o hibă bine cunoscută legată de TSR-uri şi în general utilizatorii care instalează TSR-uri sunt conştienţi (sau ar trebui să fie) de faptul că dezinstalarea lor trebuie făcută în ordine inversă instalării. Desigur se mai pune încă problema modalităţii prin care putem detecta dacă utilizatorul a mai instalat şi 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 respectiva întrerupere după noi. În caz contrar nu putem permite dezinstalarea TSR-ului.

Desigur, faptul că nici o întrerupere nu a fost redirectată *după* TSR-ul nostru nu înseamnă că nu există şi 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 aplicaţiile care vor rula mai târziu în sistem. Aceasta datorită schemei de alocare şi gestiune a memoriei de către MS-DOS. Situaţia 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 consecinţă după dealocarea memoriei utilizate de către TSR-ul nostru, pointerul spre zona de memorie liberă (utilizabilă) va rămâne neschimbat, pentru a proteja TSR1. Fenomenul descris mai sus se numeşte *fragmentarea memoriei* şi poate să apară de fiecare dată când procesele de alocare şi dealocare a memoriei alocate TSR-urilor nu sunt făcute în regim LIFO (*Last In First Out*).

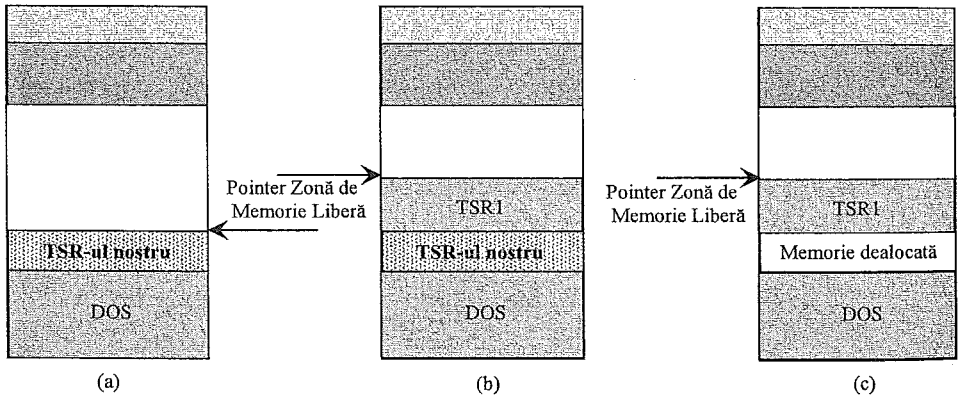


Fig. 6.8. Pointerul de Memorie Liberă în timpul alocării şi dealocării TSR-urilor.

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 şi în figură, zona de memorie devenită liberă după dezinstalarea TSR-ului nostru nu poate fi folosită până când nu se dezinstalează şi TSR1.

Pentru dealocarea memoriei programului TSR avem nevoie de apeluri ale funcţiei 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 găseşte la deplasamentul 2Ch în PSP. Acesta este de altfel şi unul dintre motivele pentru care orice program TSR dezinstalabil trebuie să salveze adresa sa PSP. Următorul bloc de memorie este cel al programului rezident însăşi. Acesta începe la adresa de început a PSP-ului în memorie. Secvenţa de cod care permite dealocarea memoriei unui TSR este prezentată în continuare:

```
; Secvenţa de cod de mai jos presupune că variabila PSP a fost iniţializată cu adresa PSP-ului
; acestui program înainte de apelul care lasă TSR-ul rezident
mov es, PSP
mov es, es:[2Ch]        ;încarcă în ES adresa de segment a variabilelor de mediu
mov ah, 49h
int 21h                 ;dealocă blocul variabilelor de mediu
mov es, PSP             ;încarcă în ES adresa de segment a PSP-ului (programul însuşi)
mov ah, 49h             ;dealocă programul TSR din memorie
int 21h
```

La fel cum codul de instalare a unui TSR este înglobat în acelaşi cod sursă cu programul TSR, codul de dezinstalare face parte tot din partea tranzientă a aceluiaşi program. De obicei acelaşi 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 noţiunile prezentate în paragrafele precedente printr-un program TSR complet. TSR-ul denumit KeybMon permite monitorizarea tastaturii şi înregistrarea tuturor tastelor apăsate pe parcursul rulării sale. La apăsarea unei combinaţii de taste de activare (F9), KeybMon va afişa pe ecran toate tastele apăsate de la ultima activare. TSR-ul verifică la instalare dacă exista deja o copie în memorie, caz în care refuză o nouă instalare. El poate fi dezinstalat prin lansarea aplicaţiei cu parametrul */u* în linia de comandă. TSR-ul verifică dacă procesul de dezinstalare este posibil şi nu intră în conflict cu alte TSR-uri instalate după el.

```
cseg segment para
assume cs:cseg,ds:cseg, es:cseg
RezStart equ $

start:
   jmp begin

maxh        EQU 2000            ;numărul maxim de taste apăsate care îl înregistrăm

key_codes
db '!!!!$',' ESC$',' 1$',' 2$',' 3$',' 4$',' 5$',' 6$',' 7$',' 8$',' 9$',' 0$',' -$',' =$'
db 'BKSP$',' TAB$',' q$',' w$',' e$',' r$',' t$',' y$',' u$',' i$',' o$',' p$',' [$',' ]$'
db 'ENT$','!!!!$',' a$',' s$',' d$',' f$',' g$',' h$',' j$',' k$',' l$',' ;$'," '$",' `$'
db '!!!!$',' \$',' z$',' x$',' c$',' v$',' b$',' n$',' m$',' ,$',' .$',' /$','!!!!$',' GR*$'
db '!!!!$',' SPC$','!!!!$',' F1$',' F2$',' F3$',' F4$',' F5$',' F6$',' F7$',' F8$',' F9$',' F10$'
db '!!!!$','!!!!$'
                                ;tablou cu titlul tastelor indexat după codul SCAN al fiecărei
                                ;taste (1=ESC, etc)
keys    db maxh dup(0)          ;tablou în care stocăm codurile SCAN ale tastelor apăsate

kindex     dw -1                ;contorul curent de taste apăsate

msg     db 'Taste înregistrate:',13,10,'$'
                                ;mesajul afişat la apăsarea combinaţiei de taste (F9) de
                                ;activare a TSR-ului înainte de afişarea pe ecran a numelui
                                ;tuturor tastelor apăsate.

inDos_OFFS  dw ?                ;adresa indicatorului InDOS
inDos_SEGdw ?
flag     db 0                   ;indică cerere de activare TSR atunci când InDOS>0

stiva_ssdw cseg                 ;stiva în care vom salva 64 de cuvinte din stiva DOS
     dw 256 dup(0)              ;O vom folosi atunci când TSR-ul este activat
stiva_sp   dw stiva_sp

save_ss    dw ?                 ;aici salvăm regiştrii de adresa a vechii stive DOS
save_sp       dw ?

FuncID     db 0                 ;Codul de identificare al TSR-ului
PSP      dw 0                   ;variabilele în care salvăm adresa PSP-ului şi vectorii
oldint2F   dd ?                 ;originali ai întreruperilor redirectate
oldint28   dd ?
oldInt9 dd ?
```



```
;semnătura TSR-ului – utilizată de către copia tranzientă a KeybMon pentru a verifica dacă acesta
;este instalat în memorie
sign      db 'Keybord Monitor'
sign_len   equ $-sign

; Int9      Noul handler al întreruperii 9h. Sistemul apelează această rutină de fiecare dată când este
;           apăsată o tastă. Rutina culege non-distructiv codul scan al tastei apăsate din controller-ul
;           de tastatură. Dacă s-a apăsat tasta F9 – se citeşte şi şterge codul scan din controller-ul de
;           tastatură. În continuare dacă DOS nu este activ lansează rutina de afişare a tastelor
;           memorate, altfel setează variabila flag astfel încât să putem prelua controlul atunci când
;           DOS nu mai este activ. Pentru orice altă tastă apăsată, rutina salvează codul scan în tabloul
;           keys şi predă controlul rutinei originale de tratare a întreruperii.
;           Folosim porturile 60h şi 61h pentru interacţiunea cu controller-ul de tastatură.

int9 proc far
   cli                          ;dezactivează întreruperile
   push ax
   in al,60h                    ;culege nedistructiv codul scan din controller-ul de
                                ;tastatură
   cmp al, 43h
   je read_key                  ;dacă e tasta F9 ?
   cmp al,80h                   ;codurile scan>80h semnifică eliberarea unei taste.
   ja orig9                     ;Nu este nevoie să le tratăm. Salt la rutina originală
   cmp cs:[kindex], maxh        ;verificăm dacă mai avem spaţiu de stocare
   je orig9                     ;dacă nu redăm controlul handler-ului int9 original
   inc cs:[kindex]              ;mai avem spaţiu=> memorăm codul scan în keys
   push bx                      ;salvăm regiştrii BX, DS pe care îi modificăm
   push ds
   push cs
   pop ds
   mov bx, offset keys
   add bx, kindex               ;salvăm codul scan în tabloul keys pe prima poziţie
   mov byte ptr [bx],al         ;liberă
   pop ds                       ;refacem regiştrii modificaţi
   pop bx

orig9:                          ;apel rutină originală a int 9 fără revenire
   sti                          ;activăm întreruperile şi apelăm oldint9
   pop ax
   jmp dword ptr cs:oldint9

;În general apăsarea unei taste generează apelul int 9h. Handler-ul original al int 9h citeşte codul
;scan al tastei apăsate din portul 60h şi îl transformă în codul ASCII corespunzător. Apoi plasează
```

;atât codul scan cât şi 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 ;paşi:
; 1) Citeşte 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) Converteşte codul scan la corespondentul ASCII şi plasează ambele coduri în bufferul de
; tastatură
; 5) Trimite secvenţa EOI – 20h (End of Interrupt – Sfârşit de întrerupere) controller-ului
; programabil de întreruperi 9259 (portul 20h). Atât timp cât acesta nu primeşte secvenţa EOI nu
; mai generează alte apeluri int 9.

```
read_key:                          ;dacă tasta apăsata este F9
   cli                             ;dezactivăm întreruperile
   in al,61h                       ;consumăm codul scan din controller-ul de tastatură
   mov ah,al                       ;Simulăm acţiunea standard a int 9 = consumare
   or al,80h                       ;a codului scan cu ştergerea acestuia din controller
   out 61h, al                     ;Controller-ul nu va mai semnaliza apăsarea acestei
   xchg ah,al                      ;taste. Ea a fost citită deja dpdv al controller-ului de
   out 61h, al                     ;de către sistemul de operare.
   mov al, 20h                     ;trimitem secvenţa de sfârşit de întrerupere către
   out 20h, al                     ;contreller-ul programabil de întreruperi 8259.
   push ds
   push dx
   lds dx, dword ptr cs:inDos_OFFS ;verificăm dacă DOS este activ
   cmp byte ptr dx, 0
   jne later                       ;dacă DA setăm variabila flag şi terminăm execuţia
                                   ;handler-ului în ideea de a fi reactivaţi după.
   call write_keys                 ;dacă NU, apelăm rutina de afişare a tastelor
   jmp end_int9                    ;memorate şi terminăm execuţia handler-ului

later:
   mov byte ptr cs:flag,1

end_int9:                          ;revenire din int 9 în cazurile în care handler-ul
   pop ds                          ;a gestionat întreruperea (tasta F9 apăsată)
   pop dx
   pop ax
   sti
   iret
int9 endp
```


; Int28 Noul handler al întreruperii 28h (DOS Idle).Sistemul apelează această rutină de fiecare
; dată când DOS este într-o buclă de aşteptare I/O. Rutina apelează handler-ul original, cu
; revenire, pentru a permite şi celorlalte aplicaţii să detecteze starea DOS şi verifică apoi
; starea variabilei flag. În cazul în care aceasta este setată, salvăm stiva DOS pentru a
; permite revenirea corectă la apelul DOS întrerupt şi apelăm rutina de afişare a tastelor
; memorate de la ultimul apel *write_keys*. Se resetează variabila *flag* (am tratat cererea de
; activare). După afişarea tastelor memorate refacem stiva DOS şi redăm controlul aplicaţiei
; întrerupte

```
int28 proc
   pushf                           ;simulare apel de tip call al int28 originală
   call dword ptr cs:oldint28
   cli                             ;dezactivare întreruperi.
   cmp byte ptr cs:flag,1          ;verificăm dacă avem cerere de activare
   jne int28_end                   ;dacă NU redăm controlul aplicaţiei întrerupte

   mov word ptr cs:save_sp, sp     ;dacă DA – salvăm 64 de cuvinte din stiva DOS şi
   mov word ptr cs:save_ss, ss     ;adresa acesteia
   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          ;resetăm variabila flag
   call write_keys                 ;apelăm rutina de afişare a tastelor memorate

   mov cx,64                       ;refacem primele 64 de cuvinte din stiva DOS
   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
```

```
   pop ds                          ;refacem regiştrii modificaţi
   pop si
   pop cx
   mov ss, cs:save_ss              ;refacem adresa stivei DOS
   mov sp, cs:save_sp
int28_end:
   sti                             ;activăm întreruperile pentru a permite funcţionarea
   iret                            ;corectă a sistemului şi revenim din întrerupere
int28 endp

; Int2F  Noul handler al întreruperii 2Fh (Multiplex). Noua rutină verifică dacă codul de funcţie
;        transmis este egal cu codul de identificare al TSR-ului. În caz afirmativ verifică codul
;        subfuncţiei (comenzii) transmise în AL. Subfuncţia acceptată este 00 – Verifică prezenţă.
;        Aceasta întoarce în ES:DI adresa far a semnăturii TSR-ului şi 0FFh (prezent) în AL.

int2F proc
   cmp ah,cs:FuncID                ;verifică ID TSR
   jne orig2F                      ;dacă nu apelăm int 2F original
   cmp al, 0                       ;verifică subfuncţia (0)
   jne orig2F                      ;dacă NU apelăm int 2F original
   push cs                         ;întoarce adresa semnăturii în ES:DI şi 0FFh în AL
   pop es
   mov di, offset sign
   mov al,0FFh
   iret                            ;terminare tratare int 2F
orig2F:
   jmp dword ptr cs:oldint2F
int2F endp

; Write_keys  Rutina de afişare a tastelor memorate. Această rutină este apelată la activarea
;             TSR-ului şi afişează pe ecran numele tastelor apăsate de la ultima activare. La
;             fiecare apel se re-iniţializează tabloul codurilor scan memorate.

write_keys proc
   push cs                         ;afişare preambul
   push offset msg
   call write_str

   push ax                         ;salvăm regiştrii modificaţi
   push cx
   push dx
   push si
   push ds
```



```
   push cs                         ;iniţializăm registrul DS
   pop ds
   mov cx,0
cycle:                             ;afişăm pentru fiecare cod scan memorat numele
   cmp cx, kindex                  ;tastei corespunzătoare
   jg wk_end                       ;Mai avem taste memorate ? Nu – salt la terminare
   mov si, offset keys             ;încărcăm în SI deplasamentul tabloului de coduri
   add si, cx                      ;scan şi facem corecţia pentru elementul curent
   lodsb                           ;încărcăm în AL codul scan al tastei curent procesate
   mov dl,5                        ;regăsim deplasamentul numelui tastei în funcţie de
   xor ah,ah                       ;codul scan (în tabloul key_codes). Fiecare nume
   mul dl                          ;este stocat pe 5 caractere=> adresa =cod_scan*5
   mov dx, offset key_codes        ;în DX avem deplasamentul şirului de caractere
   add dx, ax                      ;care reprezintă numele tastei apăsate
   push ds
   push dx
   call write_str                  ;afişăm numele tastei pe ecran
   inc cx                          ;trecem la următorul cod scan memorat
   jmp cycle
wk_end:
   mov kindex,-1                   ;reiniţializăm tabloul de coduri scan memorate
   pop ds                          ;refacem regiştrii modificaţi
   pop si
   pop dx
   pop cx
   pop ax
   ret                             ;revenire din rutină
write_keys endp

; Write_Str Rutina de afişare a unui şir de caractere terminat în $ pe ecran. Rutina preia adresa
;           de segment şi deplasamentul şirului de afişat de pe stivă şi foloseşte funcţia DOS 09h
;           pentru afişare. Rutina nu alterează regiştrii.

write_str  proc near
   push bp
   mov bp,sp                       ;pregăteşte stiva pentru recuperare parametri
   push ax                         ;salvează regiştrii modificaţi
   push dx
   push ds
   mov dx, [bp+6]                  ;recuperează deplasamentul şirului
   mov ds, dx
   mov dx, [bp+4]                  ;recuperează adresa de segment a şirului
   mov ah,09h
```