

Alexandru Vancea

Darius Bufnea

Adrian Dărăbant

Florian Boian

Anca Andreica

Andreea Navroschi

Arhitectura calculatoarelor.
Limbajul de asamblare 80x86.

Editura RISOPRINT

Cluj-Napoca • 2017

© RISOPRINT

Toate drepturile rezervate autorilor & Editurii Risoprint

Editura **RISOPRINT** este recunoscută de C.N.C.S.
(Consiliul Național al Cercetării Științifice).
www.risoprint.ro www.cnscs-ufiscd.ro

•••••

Opiniile exprimate în această carte aparțin autorului și nu reprezintă punctul de vedere al Editurii Risoprint. Autorul își asumă întreaga responsabilitate pentru forma și conținutul cărții și se obligă să respecte toate legile privind drepturile de autor.

Toate drepturile rezervate. Tipărit în România. Nicio parte din această lucrare nu poate fi reproducă sub nicio formă, prin niciun mijloc mecanic sau electronic, sau stocată într-o bază de date fără acordul prealabil, în scris, al autorilor.

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 Andreica, Adrian Dărăbant, Andreea Navroschi - Cluj-Napoca; Risoprint, 2005

400 p.; 17 x 24 cm.

Bibliogr.

ISBN 973-651-037-2

I. BOIAN, FLORIAN
II. BUFNEA, DARIUS

III. ANDREICA, ANCA
IV. DĂRĂBANT, ADRIAN
V. NAVROSCHI, ANDREEA

Director editură: GHEORGHE POP

Coperta: ILEANA ȘERBĂNUȚ

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

PREFATĂ

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 atât 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

C U P R I N S

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 octetilor 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	3.2.1.4. Operanzi cu adresare indirectă.....	78
2.2.1. Ceasul sistem (The System Clock).....	39	3.2.2. Utilizarea operatorilor.....	79
2.2.2. "Dimensiunea" unui microprocesor sau răspunsul la întrebarea "ce înseamnă calculator pe n biți?".....	41	3.2.2.1. Operatori aritmetici.....	80
2.3. Memoria.....	42	3.2.2.2. Operatorul de indexare.....	81
2.3.1. Memoria internă.....	43	3.2.2.3. Operatori de deplasare de biți.....	81
2.3.2. Memoria externă / secundară.....	45	3.2.2.4. Operatori logici pe biți.....	81
2.3.3. Ierarhia memoriei.....	48	3.2.2.5. Operatori relaționali.....	82
2.4. Dispozitive periferice.....	50	3.2.2.6. Operatorul de specificare a segmentului.....	82
2.4.1. Magistrale – structuri de interconectare.....	51	3.2.2.7. Operatori de tip.....	83
2.4.2. Magistrale I/O interne și interfețele asociate.....	53	3.3. Directive.....	85
2.4.3. Magistrale I/O externe și interfețele asociate.....	55	3.3.1. Directive standard pentru definirea segmentelor.....	85
2.4.4. Componentele unui calculator personal.....	57	3.3.1.1. Directiva SEGMENT.....	85
2.5. Performanțele unui sistem de calcul.....	61	3.3.1.2. Directiva ASSUME și gestiunea segmentelor.....	87
2.6. Arhitectura microprocesorului 8086.....	61	3.3.2. Directive pentru definirea datelor.....	94
2.6.1. Structura microprocesorului.....	61	3.3.3. Directivele LABEL, EQU, =	97
2.6.2. Reștrigări generali EU.....	62	3.3.4. Directiva PROC.....	99
2.6.3. Flagurile.....	63	3.3.5. Blocuri repetitive.....	99
2.6.4. Reștrigări de adresă și calculul de adresă.....	64	3.3.6. Directiva INCLUDE.....	101
2.6.5. Reprezentarea instrucțiunilor mașină.....	66	3.3.7. Macrouri.....	102
2.6.6. Adrese FAR și NEAR.....	66	 4. INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE.....	107
2.6.7. Calculul offsetului unui operand. Moduri de adresare.....	67	4.1. Manipularea datelor.....	108
 3. ELEMENTELE LIMBAJULUI DE ASAMBLARE.....	69	4.1.1. Instrucțiuni de transfer al informației.....	108
3.1. Formatul unei linii sursă.....	71	4.1.1.1. Instrucțiuni de transfer de uz general.....	108
3.2. Expresii.....	73	4.1.1.2. Instrucțiuni de transfer de intrare-ieșire.....	112
3.2.1. Moduri de adresare.....	73	4.1.1.3. Instrucțiuni de transfer al adreselor.....	113
3.2.1.1. Utilizarea operanzilor imediați.....	74	4.1.1.4. Instrucțiuni asupra flagurilor.....	116
3.2.1.2. Utilizarea operanzilor registru.....	76	4.1.2. Instrucțiuni de conversie.....	117
3.2.1.3. Utilizarea operanzilor din memorie.....	77	4.1.3. Impactul reprezentării little-endian asupra accesării datelor.....	119
		4.2. Operații.....	122

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

7.2.2. Harta memoriei Turbo Pascal.....	257	8.3.1.6. Întoarcerea unui rezultat.....	282
7.2.3. Cod de apel al subprogramelor Pascal.....	259	8.3.1.7. Revenirea din procedură.....	282
7.2.3.1. Transmiterea parametrilor.....	259	8.3.1.8. Directivele ARG și LOCAL.....	283
7.2.3.2. Tipuri de apel al subprogramelor.....	260	8.3.2. Interfața dintre Turbo Assembler și Turbo Pascal.....	285
7.2.3.3. Exemplu.....	261	8.3.2.1. Directiva de compilare \$L și subprogramele <i>external</i>	285
7.2.4. Cod de intrare în subprogramele Pascal.....	262	8.3.2.2. Reguli de utilizare a regiștrilor.....	288
7.2.4.1. Întoarcerea rezultatului de către funcții.....	262	8.3.2.3. Transmiterea și accesarea parametrilor.....	288
7.2.4.2. Exemplu.....	262	8.3.2.4. Întoarcerea rezultatului de către funcții.....	289
7.2.5. Cod de ieșire din subprogramele Pascal.....	264	8.3.2.5. Alocarea de spațiu pentru datele locale.....	289
7.2.6. Proceduri și funcții imbricate în Turbo Pascal.....	265	8.3.2.6. Utilizarea directivelor de segment simplificate.....	290
7.3. Implementarea subprogramelor în Borland C.....	267	8.3.2.7. Exemplul 1.....	291
7.3.1. Pointeri far și near.....	267	8.3.2.8. Exemplul 2.....	293
7.3.2. Cod de apel al subprogramelor C.....	268	8.3.3. Interfața dintre Turbo Assembler și Borland C++.....	301
7.3.3. Cod de intrare în subprogramele C.....	269	8.3.3.1. Cerințe ale editorului de legături privind definirea modułului asamblare.....	301
7.3.4. Cod de ieșire din subprogramele C.....	270	8.3.3.2. Transmiterea parametrilor.....	304
8. PROGRAMAREA MULTIMODUL.....	271	8.3.3.3. Întoarcerea de valori.....	306
8.1. Directiva MODEL. Directive de segment simplificate.....	271	8.3.3.4. Convenții privind utilizarea regiștrilor.....	306
8.2. Cerințele unui modul asamblare la legarea cu un alt modul.....	274	8.3.3.5. Exemplu.....	306
8.2.1. Directiva PUBLIC.....	274	9. PROGRAMAREA LOW LEVEL ÎN LIMBAJELE PASCAL ȘI C.....	309
8.2.2. Directiva EXTRN.....	275	9.1. Inserarea de cod mașină în textul sursă Pascal.....	309
8.2.3. Directiva GLOBAL. Legarea de module asamblare.....	276	9.1.1. Instrucțiunea inline.....	309
8.3. Legarea de module asamblare cu module scrise în limbaje de nivel înalt.....	278	9.1.2. Directiva inline.....	311
8.3.1. Etapele legării unui modul asamblare cu un modul scris în limbaj de nivel înalt.....	278	9.2. Asamblare inline.....	312
8.3.1.1. Cerințe ale editorului de legături.....	278	9.2.1. Asamblorul inline al lui Borland Pascal 6.0.....	312
8.3.1.2. Intrarea în procedură.....	279	9.2.1.1. Instrucțiunea asm.....	312
8.3.1.3. Nealterarea valorilor unor regiștri.....	279	9.2.1.2. Instrucțiunile asamblorului.....	312
8.3.1.4. Transmiterea și accesarea parametrilor.....	279	9.2.1.3. Expresii.....	315
8.3.1.5. Alocarea de spațiu de memorie pentru datele locale.....	281	9.2.1.4. Proceduri și funcții assembler.....	321

9.2.2. Asamblorul inline al lui Borland C++.....	327
9.2.2.1. Instrucțiunea asm.....	327
9.2.2.2. Exemplu.....	330
9.3. Proceduri și funcții imbricate în Borland Pascal.....	331
9.4. Accesarea reștricțiilor și apelarea de intreruperi.....	335
9.4.1. Borland Pascal 6.0.....	335
9.4.2. Borland C++.....	336
9.5. Scrierea de rutine de tratare a intreruperilor în limbajele Pascal și C.....	337
9.5.1. Proceduri interrupt în Pascal.....	337
9.5.2. Funcții interrupt în C.....	340
10. EXTENSII x86.....	341
10.1. Memoria înaltă (<i>high memory</i>) și memoria extinsă.....	342
10.2. Procesorul 80386 și modul de lucru protejat.....	355
10.3. Noi instrucțiuni aduse de urmașii procesorului 8086.....	361
10.4. Exemplu de program care lucrează cu procesorul în mod protejat.....	365
11. PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS.....	371
11.1. Modul de adresare protejat și microprocesoare pe 32 de biți.....	371
11.2. Programare în limbaj de asamblare sub Windows.....	372
11.3. Microsoft Macro Assembler.....	373
11.4. Modalități de folosire a limbajului de asamblare sub Windows.....	374
11.4.1. Programe scrise în întregime în limbaj de asamblare.....	374
11.4.2. Inserare de cod sursă asamblare în cadrul limbajelor de nivel înalt (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 esențial în utilizarea unui sistem de calcul este cunoașterea tipurilor de date primitive cu care lucrează sistemul și modul în care acesta și le reprezintă. Datele care se prelucră sunt fie numerice, fie nenumerice. La rândul lor, prelucrările numerice le putem separa în calcule numai asupra numerelor întregi și calcule în care numerele inițiale, rezultatele intermedie și finale nu sunt neapărat întregi. Deși a doua categorie de calcule o include pe prima, în sistemele de calcul ele sunt tratate ca și două categorii distințe, din rațiuni care vor fi lămurite mai târziu. Să exemplificăm aceste trei categorii de date.

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

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

$7 : 3 = 2,3333 \dots$ și vom numi aceasta *împărțire reală*

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

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

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

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

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

1.2. NUMERE ÎNTREGI

1.2.1. Baze de numerație

În viața 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 știe, numerele pot fi reprezentate și în alte baze de numerație. În știința și practica informatică, pe lângă baza 10 sunt frecvent folosite bazele de numerație 2, 8 și 16.

Dacă se lucrează în *baza 8* atunci sunt folosite doar cifrele 0, 1, 2, 3, 4, 5, 6 și 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 și 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 și 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 și 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 și 16 reprezintă același număr: 39549 (reprezentat în baza 10!).

Apare firesc întrebarea: cum se convertește 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 prezintăm o formă simplificată de conversie folosind ca intermediar baza 10 și care constă din doi pași:

- 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 simțim obligați să precizăm că alegerea ca intermediar a bazei 10 s-a făcut pentru că suntem obișnuiți să facem calcule în această bază! Dacă cineva sau ceva (un calculator) știe să facă calcule în altă bază, o poate lua ca intermedier pe aceea!

Pentru ușurință operării cu bazele de numerație 10, 16, 2, 8, recomandăm înșurirea 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 numerație

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 coeficienții polinomului sunt valorile cifrelor numărului în baza din care se convertește, iar argumentul este valoarea bazei în care este reprezentat numărul. Atât valorile cifrelor, cât și valoarea bazei se vor reprezenta în baza 10 (tabelul de mai sus ne ajută). Numărul obținut 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:

$$\begin{aligned} (9A7D)_{16} &= (9*16^3 + 10*16^2 + 7*16^1 + 13*16^0)_{10} = \\ &\quad (\text{rescriem sub o formă mai ușoară de calculat}) \\ &= ((9*16 + 10)*16 + 7)*16 + 13 = ((154*16 + 7)*16 + 13) = (2471*16 + 13) = 39549 \end{aligned}$$

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

$$d = i * c + r$$

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

De exemplu, să convertim numărul 39549 în baza 8:

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

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

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

Reluăm aceleași calcule pentru trecerea în baza 2:

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

Deci avem:

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

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

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

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

Deci avem:

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

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

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

Deci avem:

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

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

Deci avem:

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

1.2.3. Conversii rapide între bazele 2, 8, 16

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

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

Un efect similar se poate observa dacă se compară prima linie de la conversia în baza 16 cu primele patru linii de la conversia în baza 2 etc. Aceste similarități nu sunt întâmplătoare, ele decurg din faptul că $8 = 2^3$ și $16 = 2^4$. Plecând de la aceste observații, se pot defini niște reguli simple de conversie între bazele 2 și 8, respectiv 2 și 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 și 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 părții întregi nu are exact trei cifre, se completează cu zerouri la stânga pentru partea întreagă. Apoi se înlocuiește 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 înlocuiește fiecare cifră octală cu triada binară corespunzătoare ei (fiecare cifră octală se va înlocui cu exact trei cifre binare!).

Să considerăm câteva exemple:

$$\begin{array}{ll} (115175)_8 & = (001\ 001\ 101\ 001\ 111\ 101)_2 \\ (100\ 101\ 110)_2 & = (456)_8 \\ (11\ 100\ 111\ 000\ 001\ 101)_2 & = (347015)_8 \\ (1153)_8 & = (001\ 001\ 101\ 011)_2 \end{array}$$

În același spirit, avem următoarele două reguli practice de trecere între bazele 2 și 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 părții întregi nu are exact patru cifre, se completează cu zerouri la stânga pentru partea întreagă. Apoi se înlocuiește 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 înlocuiește fiecare cifră hexazecimală cu tetradă binară corespunzătoare ei (fiecare cifră hexazecimală se va înlocui cu exact patru cifre binare!).

Să considerăm câteva exemple:

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

O regulă practică de trecere între bazele 8 și 16 este aceea al folosirii bazei 2 ca intermediu. Pentru a nu opera cu șiruri nesfârșite 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 intermedii în baza 2 se pot organiza astfel:

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

Lăsăm pe seama cititorului să generalizeze, pentru baze de numerație 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 ȘI ORDINI DE PLASARE

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

1.3.1. Dimensiunea reprezentării

Fiind vorba de calcule efectuate cu o mașină, se impun firesc o serie de restricții legate de reprezentarea întregilor (și nu numai). Cea mai importantă dintre ele este dimensiunea a reprezentării, adică numărul maxim de cifre binare (numărul de biți) 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 și 64. Sistemele care folosesc alte dimensiuni de reprezentare sunt atât de rare, încât nu merită să ne ocupăm de ele.

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

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



Fig. 1.1. Dimensiuni de reprezentare la adunare și scădere

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

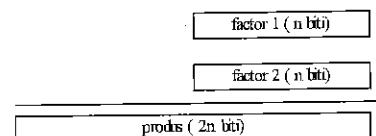


Fig. 1.2. Dimensiuni de reprezentare la înmulțire

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

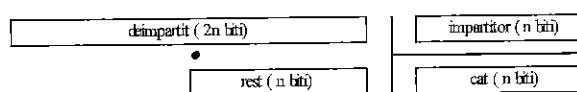


Fig. 1.3. Dimensiuni de reprezentare la împărțire

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

1.3.2. Organizarea și memorarea datelor

1.3.2.1. Bit, octet, locație, adresă

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

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

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

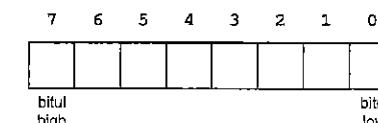


Fig. 1.4. Numerotarea bițiilor în cadrul unui octet

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

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

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

Fig. 1.5. Succesiunea octetilor în memorie

Ca și terminologie privind conținutul unui octet, vorbim de *bitul 0* al octetului sau *bitul cel mai puțin semnificativ*, bitul de *rang minim*, bitul *cel mai din dreapta*, bitul *low* etc. Bitul 1 are rangul / semnificația următoare bitului 0 și.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 bițiilor în octet de la dreapta spre stânga ca în figura 1.4.? Pentru faptul că în scrierea numerelor, cifrele (binare) se scriu, în scrierea obisnuită de la dreapta spre stânga în ordinea crescătoare a puterilor bazei.

Aici este momentul să clarificăm două noțiuni fundamentale: **adresă – adresare și conținut – numerotare conținut**. În ceea ce privește **adresarea**: este unanim acceptată convenția că adresele octetelor în memorie cresc de la stânga la dreapta (vezi figura 1.5.); deci la adresa relativă, octetul de referință este *la stânga* octetului curent. În ceea ce privește **conținutul** unei locații de memorie: numerotările entităților dintr-un conținut se fac de la dreapta spre stânga, aşa cum am văzut la numerotarea bițiilor unui octet (vezi figura 1.4); în cele ce urmează vom mai vedea astfel de numerotări și pentru alte tipuri de continuturi.

Entitatea octet este folosită practic de către toate instrucțiunile de prelucrare și de schimb cu exteriorul ale unui sistem de calcul. În contextul comunicărilor trebuie reținut postulatul că: octetul are aceeași reprezentare, indiferent de sistemul de calcul. Cu alte cuvinte, se spune că octetul este portabil. În același context, terminologia de mai sus este valabilă pentru octet indiferent de sistemul de calcul.

Accesul la biții unui octet se poate face prin intermediul unor instrucțiuni specializate. În particular, sunt situații în care se folosește ca și entitate de execuție *semoctetul (nibble)*. Un semioctet este format din patru biți, alăturați. Vorbim de *semoctet low*, sau *semoctet mai puțin semnificativ*, sau *semoctet drept*, sau cifră hexazecimală dreaptă, sau *nibble drept* etc. Analog, vorbim de *semoctet high*, sau *semoctet semnificativ*, sau *semoctet stâng*, sau cifră hexazecimală stângă, sau *nibble stâng* etc. Schematic, această împărțire apare ca în figura 1.6.

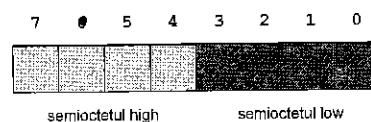


Fig. 1.6. Semioctetii componente ai unui octet

Prelucrările fundamentale sunt efectuate pe octeți și pe grupuri de octeți consecutivi. În particular, operațiile cu numere întregi se pot efectua fie pe octeți, fie pe grupuri de octeți consecutivi.

Cap. 1. Reprezentarea datelor.

Prin definiție, o succesiune de octeți consecutivi de dimensiune fixată, privită ca o entitate de sine stătătoare formează o **locatie** sau **unitate de prelucrare**.

Prin definiție, **adresa unei locații** este egală cu adresa primului octet component al locației (cu cea mai mică adresă a octetilor ce o compun).

Dimensiunea unei locatii este egală cu numărul de octeti care o compun.

Dimensiunea și denumirea pe care o poartă o locație, variază de un tip de sistem de calcul la altul. Spre exemplu, la unele sisteme, cum ar fi cele din familia IBM-PC:

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

La astfel de sisteme vorbim de locații octet, locații cuvânt și locații dublucuvânt. În figura 1.7. prezentăm un cuvânt cu subdiviziunile (subentitățile) lui, iar în figura 1.8. prezentăm un dublucuvânt cu subdiviziunile (subentitățile) lui.

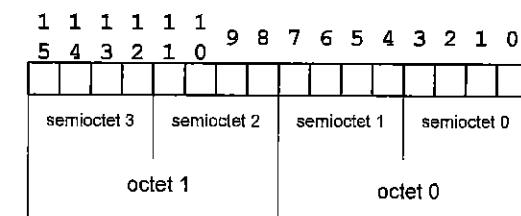


Fig. 1.7. Un cuvânt IBM-PC

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

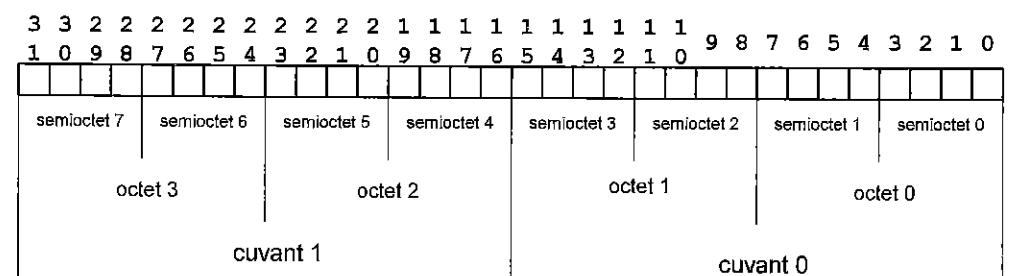


Fig. 1.8. Un dublu cuvânt IBM-PC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Plasarea *little-endian*, în care octetul cu cea mai mică adresă din locație va conține octetul cu numărul 0 al reprezentării, octetul cu adresa următoare va conține octetul 1 al reprezentării s.a.m.d. (octetul „end” al reprezentării are adresa cea mai „little”).
- Plasarea *big-endian*, în care octetul cu cea mai mare adresă din locație va conține octetul 0 al reprezentării, octetul cu adresa precedentă va conține octetul 1 al reprezentării s.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 locație de patru octeți. Pentru această dimensiune a locației, reprezentările lui în bazele 16 și 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 octetelor scrise atât în hexazecimal, cât și în binar:

Big-endian:	00 00000000	00 00000000	04 00000100	01 00000001
	Adresa B	Adresa B+1	Adresa B+2	Adresa B+3
Little-endian:	01 00000001	04 00000100	00 00000000	00 00000000
	Adresa L	Adresa L+1	Adresa L+2	Adresa L+3

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 octetelor 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	
-1	IEEE, dublă precizie	BF	F0	00	00	00	00	00	
1	Turbo Pascal	00	00	00	00	81			
-1	Turbo Pascal	80	00	00	00	00	81		

Cap.1. Reprezentarea datelor.

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.

Dată	Standard de reprezentare	Exemple de plasări little-endian							
		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	81			
-1	Turbo Pascal	80	00	00	00	00	81		

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

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

Big-endian:	00 00000000	00 00000000	00 00000000	02 00000010
	Adresa B	Adresa B+1	Adresa B+2	Adresa B+3
Little-endian:	02 00000010	00 00000000	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 mașini big-endian adresa octetului cu bitul de semn coincide cu adresa locației. Cu notațiile din figura 1.10., bitul de semn, la mașina big-endian, se află la adresa **B**, indiferent de faptul că se va prelucra o locație de 4 octeți, de 2 octeți sau de 1 octet. În cazul unei mașini little-endian, bitul de semn se află în ultimul octet al locației (cel cu cea mai mare adresă), aşa că pentru a-l obține procesorul trebuie să calculeze adresa acestei octeți: ea este **L** pentru locația de 1 octet, **L+1** pentru locația de 2 octeți și **L+3** pentru cea de 4 octeți.

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

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

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

Și acum câteva cuvinte "exotice" legate de arhitecturile big-endian și little-endian. Conversia datelor între cele două arhitecturi este cunoscută în literatură sub numele de problema '**NUXIP**'. De ce? '**UNIX**' este numele unui sistem de operare celebru. Să presupunem că într-un sir de 4 octeți se dorește încărcarea cuvântului '**UNIX**'. Un programator "isteț", în loc să comande încărcarea pe rând în cele patru locații succesive a căte unui caracter pentru a obține succesiunea ('U','N','I','X'), face numai două încărcări a căte unui întreg pe doi octeți: mai întâi încarcă sirul '**UN**' privit ca un întreg, apoi în locația întreagă următoare încarcă sirul '**IX**' privit tot ca un întreg! În acest mod a făcut "economie" de două instrucțiuni.

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

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

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

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

1.3.2.4. Unități de capacitate a memoriei

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

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

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

Spre a avea o idee asupra capacitateilor de memorare actuale, un calculator personal obișnuit are memoria internă între 256 Mo și 4 Go. Capacitatea unui hard – disc actual este între 20 Go și 512 Go. Pentru comparație, o pagină de format A4 – echivalentă cu o pagină dintr-o carte, conține cu puțin peste 3000 de caractere: litere mari / mici, cifre, semne speciale. Așa cum vom vedea mai târziu, memorarea unui text se face un caracter pe 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 comparație: un CD obișnuit (nu DVD) are capacitatea de 700 Mo. Într-un astfel de CD se poate memora lista tuturor abonaților telefonici Romtelecom din Romania, cu toate informațiile necesare: nume, prenume, adresa completă, telefon.

1.4. CODIFICAREA CARACTERELOR

Să considerăm mulțimea *caracterelor tipăribile*, existente la fiecare imprimantă, tastatură, care apar pe ecran etc. Este vorba de literele (cele 26 din alfabetul englez) mari și mici, cifrele, semnele speciale etc. Pentru codificarea acestor caractere în vederea prelucrării lor automate, organizațiile internaționale 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 condiții. După cum se va vedea, existența acestor condiții 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 și FELIX-C folosesc acest cod.

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

În contextul cerințelor de internaționalizare impuse de existența Internet, în ultimii 10 ani se impune din ce în ce mai mult standardul de codificare UNICODE. Acesta codifică un caracter pe doi octeți, pentru a se permite codificarea simbolurilor și a caracterelor folosite în scrierile majorității 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*: spațiul, virgula, punctul, +, -, _, \$, & și a.m.d

Cap.1. Reprezentarea datelor.

5. Un set de *caractere funcționale* care nu apar la tipărire / afișare, ci doar dirijează tipărirea / afișarea.

Caracterele funcționale apar în tabelele de definiție sub forma unor grupuri de litere, ca de exemplu CR, LF, TAB, FF, BEL, BS și.a.m.d. CR provoacă deplasarea dispozitivului de afișare (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 poziția în cadrul rândului. În funcție de sistem, pentru a separa două linii dintr-un text se folosește fie LF, fie succesiunea de caractere CR LF. TAB este caracterul de tabulare, deci avansul dispozitivului la poziția următorului stop de tabulare (de obicei peste 5-8 caractere). FF (Form Feed) provoacă trecerea la pagina (ecranul) următoare (următor). BEL provoacă emisarea unui semnal sonor, iar BS (backspace) provoacă deplasarea dispozitivului de afișare (tipărire) cu o poziție spre stânga, în vederea ștergerii (supraimprimării) ultimului caracter.

Condițiile de codificare pe care le respectă standardul ASCII sunt:

- Toate caracterele funcționale au codul mai mic decât codul caracterului spațiu.
- Codul caracterului spațiu 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
0	0 000	000	(null)	32	20	040	 	Space	64	40	100	@	B
1	1 001	301	(start of heading)	33	21	041	!	!	65	41	101	A	A
2	2 002	302	(start of text)	34	22	042	"	"	66	42	102	B	B
3	3 003	303	(end of text)	35	23	043	#	#	67	43	103	C	C
4	4 004	304	(end of transmission)	36	24	044	$	\$	68	44	104	D	D
5	5 005	305	(enquiry)	37	25	045	%	?	69	45	105	E	E
6	6 006	306	(acknowledge)	38	26	046	&	@	70	46	106	ʞ	F
7	7 007	307	(bell)	39	27	047	'	'	71	47	107	G	G
8	8 010	308	(backspace)	40	28	050	({	72	48	110	H	H
9	9 011	309	(horizontal tab)	41	29	051))	73	49	111	I	I
10	10 012	310	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J
11	11 013	311	(VT vertical tab)	43	2B	053	+	+	75	4B	113	K	K
12	12 014	312	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L
13	13 015	313	(carriage return)	45	2D	055	-	:	77	4D	115	M	M
14	14 016	314	(shift out)	46	2E	056	.	.	78	4E	116	N	N
15	15 017	315	(shift in)	47	2F	057	/	/	79	4F	117	O	O
16	16 020	316	(data link escape)	48	30	060	0	0	80	50	120	P	P
17	17 021	317	(device control 1)	49	31	061	1	1	81	51	121	Q	Q
18	18 022	318	(device control 2)	50	32	062	2	2	82	52	122	R	R
19	19 023	319	(device control 3)	51	33	063	3	3	83	53	123	S	S
20	20 024	320	(device control 4)	52	34	064	4	4	84	54	124	T	T
21	21 025	321	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U
22	22 026	322	(synchronous idle)	54	36	066	6	6	86	56	126	V	V
23	23 027	323	(end of trans. block)	55	37	067	7	7	87	57	127	W	W
24	24 030	324	(cancel)	56	38	070	8	8	88	58	130	X	X
25	25 031	325	(end of medium)	57	39	071	9	9	89	59	131	Y	Y
26	26 032	326	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z
27	27 033	327	(escape)	59	3B	073	;	;	91	5B	133	[[
28	28 034	328	(file separator)	60	3C	074	<	<	92	5C	134	\	\
29	29 035	329	(group separator)	61	3D	075	=	-	93	5D	135]	J
30	30 036	330	(record separator)	62	3E	076	>	>	94	5E	136	^	^
31	31 037	331	(unit separator)	63	3F	077	?	?	95	5F	137	_	_

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 și în octal. Pe prima coloană apar caracterele funcționale, incluzând atât grupurile de litere prin care se simbolizează și semnificația pe scurt a fiecărui. Pe celelalte coloane apare în plus modul în care caracterele se pot specifica în limbajul HTML, limbaj de bază în descrierea paginilor Web. Din aceeași sursă, în figura 1.12. se prezintă extinderea IBM prin folosirea celui de-al 8-lea bit.

128	¢	144	£	161	ƒ	177	¤	193	Ł	209	₱	225	฿	241	₩
129	¤	145	₪	162	₪	178	₪	194	₪	210	₪	226	₪	242	₪
130	€	146	₭	163	₭	179	₭	195	₭	211	₭	227	₭	243	₭
131	฿	147	₵	164	₵	180	₵	196	₵	212	₵	228	₵	244	₵
132	₭	148	₵	165	₵	181	₵	197	₵	213	₵	229	₵	245	₵
133	₼	149	₵	166	₵	182	₵	198	₵	214	₵	230	₵	246	₵
134	₼	150	₵	167	₵	183	₵	199	₵	215	₵	231	₵	247	₵
135	₼	151	₵	168	₵	184	₵	200	₵	216	₵	232	₵	248	₵
136	₼	152	₵	169	₵	185	₵	201	₵	217	₵	233	₵	249	₵
137	₼	153	₵	170	₵	186	₵	202	₵	218	₵	234	₵	250	₵
138	₼	154	₵	171	₵	187	₵	203	₵	219	₵	235	₵	251	₵
139	₼	156	₵	172	₵	188	₵	204	₵	220	₵	236	₵	252	₵
140	₼	157	₩	173	₩	189	₩	205	₩	221	₩	237	₩	253	₩
141	₼	158	₩	174	₩	190	₩	206	₩	222	₩	238	₩	254	₩
142	₼	159	₣	175	₣	191	₣	207	₣	223	₣	239	₣	255	₣
143	₼	160	₼	176	₼	192	₼	208	₼	224	₼	240	=		

Source: www.LookupTables.com

Fig. 1.12. Codul ASCII extins

1.5. CODIFICAREA NUMERELOR ÎNTREGI

1.5.1. Convenție cu semn și convenție fără semn

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

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

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

Cap. 1. Reprezentarea datelor.

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

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

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

1.5.2. Bitul de semn; codul complementar

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

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

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

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

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

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

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

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

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

Pentru început vom defini operația de complementare, numită de multe ori și operația de schimbare de semn.

Definiție: Pentru **complementarea unui număr întreg reprezentat pe n biți, mai întâi se inversează valorile tuturor biților (valoarea 0 devine 1 și valoarea 1 devine 0) din locația de reprezentare, după care se adaugă 1 la valoarea obținută.**

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 neschimbați biții începând din dreapta reprezentării binare până la primul bit 1 inclusiv, restul biților se inversează până la bitul $n-1$ inclusiv,

sau

Se scade binar conținutul (evidenț binar) al locației de complementat din 100...00, unde numărul de după cifra binară 1 are atâtea zerouri căci biți are locația de complementat.

sau

Se scade hexazecimal conținutul (evidenț hexazecimal) al locației de complementat din 100..00, unde după cifra hexazecimală 1 apar atâtea zerouri căci cifre hexazecimale are locația de complementat.

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

De exemplu, dacă vrem să complementăm o locație de un octet și care conține numărul $(18)_{10}$:

Locația inițială:	00010010
După inversarea biților	11101101
Se adaugă 1	11101101+
	<u>00000001</u>
Complementul:	11101110

Deci numărul $(18)_{10}$, adică $(12)_{16}$, adică $(00010010)_2$, are ca și complement numărul $(11101110)_2$, adică $(EE)_{16}$, adică $(238)_{10}$.

Aplicând regula de scădere binară, avem:

Locația inițială:	00010010
Complementul:	11101110

Aplicând regula de scădere hexazecimală, avem:

Locația inițială:	12
Complementul:	EE

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

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

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

Locația inițială:	EE
Complementul:	12

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

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

Se ignoră acest transport:	11101110
1	00000000

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

Locația inițială:	00000000
După inversarea biților	11111111
Se adăugă 1	11111111+
(se ignoră ultimul transport)	00000001
Complementul:	00000000
Locația inițială:	10000000
După inversarea biților	01111111
Se adăugă 1	01111111+
	00000001
Complementul:	10000000

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

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

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

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

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

2	127	007F	0000000011111111
1	-128	80	10000000
2	-128	FF80	1111111100000000
2	128	0080	0000000100000000
2	32767	7FFF	0111111111111111
2	-32767	8001	1000000000000001
2	-32768	8000	1000000000000000
4	-32768	FFFF8000	11111111111111000000000000000000
4	32768	00008000	00000000000000001000000000000000
1	18	12	00010010
2	18	0012	00000000000010010
1	-18	EE	11101110
2	-18	FFEE	1111111111101110
4	39549	00009A7D	00000000000000001001101001111101
4	-39549	FFFF6583	1111111111111110110010110000011
4	985437	000F095D	00000000000011110000100101011101
4	-985437	FFF0F6A3	11111111111100001111011010100011

1.5.3. Operări aritmetice; conceptul de depășire

1.5.3.1. De ce codul complementar?

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

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

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

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

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

1.5.3.2. Conceptul de depășire

Așa după cum am arătat mai sus, reprezentarea numerelor întregi se face în locații cu dimensiune fixată apriori. În mod natural, trebuie să ne punem problema: ce se întâmplă când rezultatul nu începe în spațiul care-i este rezervat?

Generic vorbind, *depășirea* apare atunci când rezultatul unui calcul nu începe în spațiul care-i este rezervat.

Specific vorbind, condițiile de apariție a unei depășiri apar în mod diferit în funcție de context. Contextul depinde, pe de o parte dacă se utilizează convenția fără semn sau convenția cu semn. Pe de altă parte, contextul depinde de tipul operației: aditivă sau multiplicativă.

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

În cele ce urmează vom aborda, pe rând tipurile de operații și în cadrul acestora vom semnaliza situațiile de depășire.

1.5.3.3. Adunări și scăderi

Operațiile de adunare și de scădere sunt efectuate de calculator exact după algoritmii cunoscuți din clasa I primară, cu singura deosebire că operațiile sunt efectuate în baza 2. În schimb, datorită regulilor de dimensionare și a faptului că operația este efectuată de mașină și nu de om, există posibilitatea de apariție a depășirilor.

Pentru cazul operațiilor fără semn, există două reguli care provoacă depășire:

R1) Dacă rezultatul adunării nu începe pe n biți, atunci apare depășire la adunare și rămân în rezultat numai biții de la ordinul 0 la ordinul $n-1$, iar bitul de ordin n se pierde.

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

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

Dăm, în continuare, câteva exemple de operații de adunare și scădere analizate în convenția fără semn. Pentru fiecare operație, vom scrie mai întâi operanții în baza 10, apoi în baza 16 și apoi în baza 2. Dimensiunea locațiilor este fie de un octet, fie de doi octeți.

Baza 10	Baza 16	Baza 2	Observații
18+ 18 36	12+ 12 24	00010010+ <u>00010010</u> 00100100	
243- 18 225	F3- 12 E1	11110011- <u>00010010</u> 11100001	
243+ 18 261	F3+ 12 05	11110011+ <u>00010010</u> 00000101	Depășire la adunare, conform regulii R1!
18- 243 -225	12- F3 1F	00010010- <u>11110011</u> 00011111	Depășire la scădere, conform regulii R2!
575+ 650 1225	023F+ 028A 04C9	0000001000111111+ <u>0000001010001010</u> 0000010011001001	
650- 575 75	028A- 023F 004B	0000001010001010- <u>0000001000111111</u> 0000000001001011	
65535+ 90 65625	FFFF+ 005A 0059	1111111111111111+ <u>0000000001011010</u> 0000000001011001	Depășire la adunare, conform regulii R1!
575- 650 -75	023F- 028A FFB5	0000001000111111- <u>0000001010001010</u> 1111111101101011	Depășire la scădere, conform regulii R2!

Să ne ocupăm în continuare de operațiile de adunare și scădere în convenția 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 secțiunea 1.5.3.1.). Pentru a surprinde situațiile de depășire este suficient să studiem efectuarea de sume ale unor numere din intervalul $[-2^{n-1}, 2^{n-1}-1]$ și de a depista aici situațiile în care poate să apară depășire, adică neîncadrarea sumei în intervalul admis $[-2^{n-1}, 2^{n-1}-1]$ pentru suma pe n biți.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Suma în baza 10	Suma în cod complementar	Transport sau depășire	Diferența în baza 10	Diferența în cod complementar	Împrumut sau depășire
$(-7) + 5 = -2$	$\begin{array}{r} 1001+ \\ \underline{0101} \\ 1110 \end{array}$		$5 - 7 = -2$	$\begin{array}{r} 0101- \\ \underline{0111} \\ 1110 \end{array}$	
$(-4) + 4 = 0$	$\begin{array}{r} 1100+ \\ \underline{0100} \\ 0000 \end{array}$	Transport	$4 - 4 = 0$	$\begin{array}{r} 0100- \\ \underline{0100} \\ 0000 \end{array}$	
$2 + (-7) = -5$	$\begin{array}{r} 0010+ \\ \underline{1001} \\ 1011 \end{array}$		$2 - 7 = -5$	$\begin{array}{r} 0010- \\ \underline{0111} \\ 1011 \end{array}$	Împrumut
$5 + (-2) = 3$	$\begin{array}{r} 0101+ \\ \underline{1110} \\ 0011 \end{array}$	Transport	$5 - 2 = 3$	$\begin{array}{r} 0101- \\ \underline{0010} \\ 0011 \end{array}$	
$3 + 4 = 7$	$\begin{array}{r} 0011+ \\ \underline{0100} \\ 0111 \end{array}$				
$(-4) + (-1) = -5$	$\begin{array}{r} 1100+ \\ \underline{1111} \\ 1011 \end{array}$		$(-4) - 1 = -5$	$\begin{array}{r} 1100- \\ \underline{0001} \\ 1011 \end{array}$	
$5 + 2 = 7$	$\begin{array}{r} 0101+ \\ \underline{0010} \\ 0111 \end{array}$				

$(-5) + 2 = -3$	1011+ <u>0010</u> 1101		$2 - 5 = -3$	0010- <u>0101</u> 1101	Împrumut
$(-8) + 7 = -1$	1000+ <u>0111</u> 1111		Nu se poate scrie $7-8$ pe 4 biți, deoarece 8 nu aparține intervalului $[-8,7]$.		
$5 + 4 = 9$	0101+ <u>0100</u> 1001	Depășire!			
$(-7) + (-6) = -13$	1001+ <u>1010</u> 0011	Depășire!	$(-7) - 6 = -13$	1001- <u>0110</u> 0011	Depășire!
$(-6) + (-4) = -10$	1010+ <u>1100</u> 0110	Depășire!	$(-6) - 4 = -10$	1010- <u>0100</u> 0110	Depășire!
$7 + 7 = 14$	0111+ <u>0111</u> 1110	Depășire!			
$(-8) + (-8) = -16$	1000+ <u>1000</u> 0000	Depășire!			

1.5.3.4. Înmulțiri și împărțiri

Înmulțirea și împărțirea pe n biți impun următoarele restricții de dimensiune a locațiilor:

Înmulțirea pe n biți presupune că ambii factori sunt reprezentați pe câte n biți, iar produsul lor va fi reprezentat pe $2 * n$ biți. În cazul convenției cu semn, factorii înmulțirii vor fi reprezentați în cod complementar. Produsul va fi reprezentat tot în cod complementar și va respecta regula semnelor.

Ca o consecință imediată a acestei dimensiuni, rezultă că operatia de înmulțire nu provoacă depășire! Într-adevăr, în convenția fără semn cea mai mare valoare posibilă pe n biți este $2^n - 1$, pătratul acestui valorii este $2^{2n} - 2^{n+1} + 1$, număr care începe pe $2n$ biți. În convenția cu semn, numărul -2^{n-1} are valoarea absolută cea mai mare, iar pătratul acestuia este 2^{2n-2} , număr care se poate reprezenta pe $2n-1$ biți, deci în cod complementar acest număr se poate reprezenta pe $2n$ biți.

Împărțirea pe n biți (oarecum invers față de înmulțire), impune condiția ca deîmpărțitul să fie reprezentat pe $2 * n$ biți, iar împărtitorul pe n biți. Operația furnizează două rezultate: câțul reprezentat pe n biți și restul reprezentat tot pe n biți. În cazul convenției cu semn, deîmpărțitul și împărtitorul se vor reprezenta în cod complementar. Atât câtul, cât și restul vor fi, de asemenea,

reprezentate în cod complementar. Câtul împărțirii va respecta regula semnelor. Important! Restul împărțirii va fi, în valoare absolută mai mic decât valoarea absolută a împărtitorului și va avea același semn ca și deîmpărțitul! De exemplu, $-7 : 3$ dă câtul -2 și restul -1 , adică $-7 = (-2) * 3 + (-1)$. Rugăciunile să comparați acest rezultat cu cel impus de teorema împărțirii cu rest din aritmetică, care impune ca restul să fie un număr pozitiv, deci în aritmetică avem $(-7) = (-3) * 3 + 2$, deci câtul -3 și restul 2 !

Operația de împărțire semnalează eroare la împărtitor zero (Divide by zero!). În cazul neîncadrării în dimensiuni semnalează depășire! Spre exemplu, împărțirea fără semn $1000 : 3$ se poate, formal, efectua, deoarece deîmpărțitul se poate reprezenta pe 16 biți și împărtitorul se poate reprezenta pe 8 biți. La această operație va apărea depășire, deoarece câtul este 333 și nu se poate reprezenta pe 8 biți. Restul împărțirii este 1 și se poate reprezenta pe un octet. Pentru a evita o astfel de situație, programatorul poate decide să facă operația pe 16 biți, adică să reprezinte deîmpărțitul pe 32 de biți și împărtitorul pe 16 biți, urmând să obțină câtul și restul pe câte 16 biți (333 începe pe 16 biți și astfel nu vom mai avea depășire).

Să exemplificăm mai întâi operațiile de înmulțire și împărțire fără semn.

Înmulțirea fără semn a numerelor întregi reprezentate binar se desfășoară conform algoritmului cunoscut, doar că se folosește baza 2. Pentru comoditate, vom considera $n = 4$. Să considerăm factorii 11 și 13. Înmulțirea lor în baza 2 este:

$$\begin{array}{r} 1011 \times \\ 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ \hline 10001111 \end{array}$$

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

Plecând de la acest exemplu, se poate observa că:

- 1) Înmulțirea generează o serie de produse parțiale care sunt adunate. Există câte un produs parțial pentru fiecare cifră a împărtitorului.
- 2) Produsele parțiale sunt fie deînmulțitul, fie 0.
- 3) Produsele parțiale nenule pot fi adunate unul către unul. Ele sunt obținute deplasând deînmulțitul spre stânga cu câte o poziție.
- 4) Înmulțirea a două numere de câte n biți generează un produs care ocupă $2 * n$ biți (motiv pentru care s-a impus restricția de dimensiune amintită mai sus).

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

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

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

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

Să reamintim câteva observații privind înmulțirile și împărțirile cu semn.

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

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

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

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

1.5.4. Conversia la o locație de alte dimensiuni

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

În fapt este vorba de patru operații:

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

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

8 biți:	16 biți: extensie cu semn	32 biți: extensie cu semn	16 biți: extensie cu zero	32 biți: extensie cu zero
80	FF80	FFFFF80	0080	00000080
10000000	111111110000000	111111111111111 111111110000000	0000000010000000	000000000000000000000000
28	0028	00000028	0028	00000028
00101000	0000000000101000	000000000000000000000000 00000000000101000	00000000000101000	00000000000000000000000000000000
9A	FF9A	FFFFF9A	009A	0000009A
10011010	111111110011010	111111111111111 111111110011010	00000000100110100	00000000000000000000000000000000
7F	007F	0000007F	007F	0000007F
01111111	000000001111111	000000000000000000000000 000000001111111	00000000000000000000000000000000	00000000000000000000000000000000
--	1020	00001020	--	00001020
	0001000000100000	000000000000000000000000 0001000000100000		00000000000000000000000000000000
--	8088	FFFF8088	--	00008088
	1000000010001000	111111111111111 1000000010001000		00000000000000000000000000000000

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

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

16 biți:	8 biți: contracție cu semn	8 biți: contracție cu zero
FF80 111111110000000	80 10000000	Depășire! Se pierd 8 biți 1
0028 000000000101000	28 00101000	28 000101000
FF9A 111111110011010	9A 10011010	Depășire! Se pierd 8 biți 1
FE40 111111100100000	Depășire! Se schimbă bitul de semn	Depășire! Se pierd 8 biți 1
0100 000000010000000	Depășire! Se schimbă bitul de semn	Depășire! Se pierd 8 biți 1
0088 0000000010001000	Depășire! Se schimbă bitul de semn	88 10001000

C A P I T O L U L 2

ARHITECTURA SISTEMELOR DE CALCUL

2.1. DEFINIȚIA 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 definiție similară a unui sistem de calcul (*The American Heritage Dictionary of the English Language*, 2000) este: un dispozitiv care efectuează calcule, în special o mașină electronică programabilă care execută operații aritmétice, logice sau care asamblează, stochează, corelează sau efectuează un alt tip de procesare a informației, cu viteză ridicată.

Funcțiile de bază ale unui SC sunt:

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

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

Structura unui sistem de calcul este:

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

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

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

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

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

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

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

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

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

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

ISA definește:

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

Organizarea unei mașini se referă la:

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

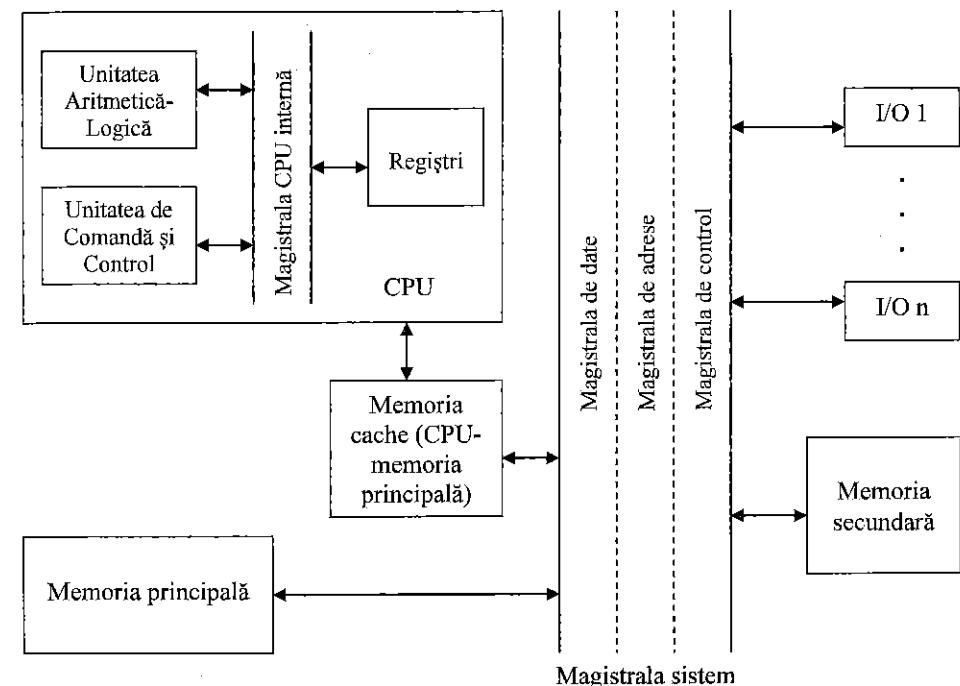


Fig. 2.1. Arhitectura unui sistem de calcul

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

Caracteristicile arhitecturii von Neumann sunt:

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

2.2. UNITATEA CENTRALĂ (CENTRAL PROCESSING UNIT – CPU)

Procesorul controlează modul de operare a calculatorului și execută funcțiile de procesare a datelor.

Funcțiile unui CPU sunt:

- obținerea instrucțiunilor care trebuie executate;
- obținerea datelor necesare instrucțiunilor;
- procesarea datelor (execuția instrucțiunilor);
- furnizarea rezultatelor obținute.

Putem identifica două componente de bază la nivelul unui CPU:

- Unitatea Aritmetică-Logică (*Arithmetic Logic Unit – ALU*);
- Unitatea de Comandă și Control (*Control Unit – CU*);

iar pentru îndeplinirea funcțiilor de mai sus, CPU mai are nevoie de:

- regiștri – aceștia sunt dispozitive de stocare temporară a datelor și informațiilor de control (instrucțiunile), de capacitate mică și viteză de acces mare;
- magistrale interne CPU – dispozitive pentru comunicare între componentele CPU și comunicare cu exteriorul, pentru traficul de informații.

Unitatea Aritmetică-Logică execută operații aritmetice și logice asupra datelor.

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

Pentru execuția unei instrucțiuni, aceasta și datele necesare trebuie aduse din memoria secundară sau de la un dispozitiv de intrare în memoria principală. CU coordonează *ciclul de execuție a instrucțiunii*:

- citirea instrucțiunii din memoria principală (FETCH);
- decodificarea acesteia (DECODE – tipul instrucțiunii, numărul de argumente, etc);
- obținerea operandelor necesare instrucțiunii (READ MEMORY);
- execuția ei (EXECUTE; dacă e cazul, ALU primește controlul pentru efectuarea operației aritmetice sau logice implicate);
- furnizarea rezultatelor în regiștri sau în memorie (STORE).

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

Cunoscând structura unui CPU și pașii din ciclul de execuție a unei instrucțiuni, putem contura structura funcțională a procesorului: *modulul de control și calea de date (datapath)*. Modulul de

Cap.2. Arhitectura sistemelor de calcul.

control este răspunzător de comunicarea cu exteriorul CPU (preluarea și interpretarea instrucțiunilor și de transmiterea rezultatelor), precum și de controlul execuției instrucțiunilor (emiterea semnalelor de control către calea de date și recepționarea semnalelor de stare dinspre aceasta). Din calea de date fac parte componente de stocare (regiștri de date), unități funcționale (ALU) și căi de comunicare. Practic, calea de date definește mulțimea operațiilor efectuate asupra datelor în vederea execuției unei instrucțiuni și drumul parcurs de datele prelucrate în decursul execuției operațiilor.

2.2.1. Ceasul sistem (The System Clock)

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

Frecvența de ceas este de fapt numărul de cicluri de ceas pe secundă și este măsurată în hertzii.

Observație: 1 Hz = 1 ciclu de ceas / secundă
frecvență = 1 / perioada

Toate operațiile efectuate de către microprocesor sunt sincronizate cu ceasul sistem; aceasta înseamnă că procesorul nu poate efectua de o manieră secvențială operații mai rapid decât frecvența de tact a ceasului (a unității 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 cantă de timp sesizabilă de către procesor. Toate activitățile și instrucțiunile executate de procesor sunt executate în general în multipli ai ciclului de ceas (din punct de vedere al duratei de execuție). Fiecare instrucțiune mașină necesită un număr fix, cunoscut (în general) de cicluri procesor. Există diferențe în durata execuției instrucțiunilor mașină în funcție de natura parametrilor și mediul lor de stocare. Astfel o instrucțiune **MOV destinație, sursă** se execută între 2 și 14 cicluri de ceas procesor în funcție de natura argumentelor destinație și sursă care pot fi: regiștri generali, regiștri de segment, constante, adrese de memorie. Fiecare combinație posibilă a acestor tipuri de parametri va rezulta într-o durată de execuție specifică în cicluri de ceas procesor. Pentru fiecare combinație posibilă, durata de execuție în cicluri procesor este cunoscută. Pentru a afla numărul de cicluri procesor pentru fiecare instrucțiune mașină, împreună cu derivațiile rezultate din tipul parametrilor se poate consulta documentația oferită de *Norton Guide*.

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

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

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

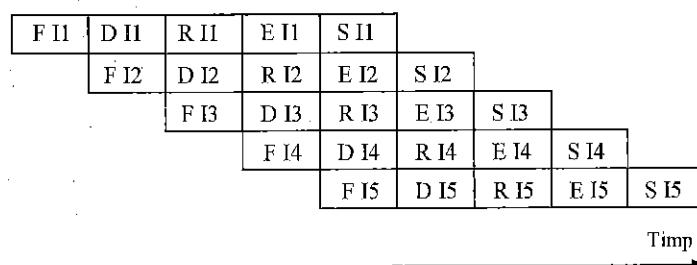


Fig. 2.2. Mecanismul de paraleлизare în execuția instrucțiunilor (pipeline)

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

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

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

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

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

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

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

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

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

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

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

Micropresor	Data Bus	Regiștri
8088	8 biți	16 biți
80386sx	16 biți	32 biți
Pentium	64 biți	32 biți

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)*: locațiile pot fi accesate (în citire sau scriere) în orice ordine (aleator) indiferent de ultima locație accesată. Exemple: oricare chip de memorie (memorie "on-chip"), precum memoria principală (DRAM, SRAM), memoria flash și.a.;

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

Observație: La încărcarea sau căutarea în memoria cache a unei informații, pe baza adresei acesteia se determină care este poziția 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 locației din memoria cache depinde de modul de organizare a acesteia. De exemplu: *maparea directă* – unei informații de la o anumită adresă îi corespunde o locație bine determinată din memoria cache, locație calculată cu ajutorul unei funcții de transformare (poate fi funcție de dispersie de tip modulo); *maparea asociativă pe mulțime* – o anumită informație poate fi încărcată în memoria cache în oricare din pozițiile dintr-o mulțime bine determinată.

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

a.4. *memorie cu acces direct*: spre deosebire de accesul secvențial, poziționarea 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 secțiunea 2.3.2.).

Observație: 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). Diferența constă în timpul de acces: în cazul memoriei cu acces aleator timpul de acces este același, indiferent de adresa datelor accesate; în schimb, memoria cu acces direct (vezi dispozitivele disc – secțiunea 2.3.2.) folosește un mecanism fizic de poziționare la o anumită adresă, fapt care introduce un

timp de întârziere în accesarea datelor => timp de accesare variabil, în funcție de poziția curentă a mecanismului de accesare și de poziția datelor pe disc.

b. din punct de vedere al volatilității:

b.1. *memorie volatilă* (de scurtă durată): conținutul său se pierde la îndepărțarea sursei de curent. Cel mai elovent exemplu îl constituie în acest sens memoria principală a SC (care conține datele și instrucțiunile utilizate curent de CPU);

b.2. *memorie non-volatile sau remanentă* (de lungă durată): conținutul se păstrează și 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 spațiile de stocare de date accesibile CPU fără utilizarea canalelor de comunicație de intrare / ieșire. Din această categorie fac parte memoria principală, memoria cache (dintre CPU și memoria principală), ROM și registrii, toate aceste dispozitive putând fi direct accesate de către CPU.

Memoria principală (main memory, primary memory), cunoscută în general sub numele de memorie RAM, conține date care sunt utilizate curent de către procesor – instrucțiuni ale programelor care sunt executate și date cu care acestea operează. Aceste informații 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 memorilor RAM, precum DRAM sau SRAM. În general, capacitatea memoriei principale a unui sistem de calcul este cuprinsă între 1 MB și 4 GB.

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

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

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

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

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

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

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

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

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

2.3.2. Memoria externă / secundară

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

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

Structura unui volum disc

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

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

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

Din cele menționate mai sus rezultă următorii parametri (constante) ai unui dispozitiv disc: numărul de discuri, numărul capetelor de citire / scriere pentru un disc (nunărul de fețe active ale unui disc), numărul de piste de pe o față, numărul de sectoare de pe o pistă și numărul de octeți dintr-un sector.

Cunoscând valorile parametrilor unui dispozitiv disc, o metodă de adresare a informațiilor (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 și numărul sectorului din cadrul pistei (unde pistă e determinată de cap și cilindru). De exemplu, o dischetă de 3.5 inch este configurată în general la următorii parametri: 2 capete de citire / scriere (numerotate 0 și 1), 80 cilindri / disc (numerotați 0 – 79), 18 sectoare / pistă (numerotate 0 – 17), 512 octeți / sector.

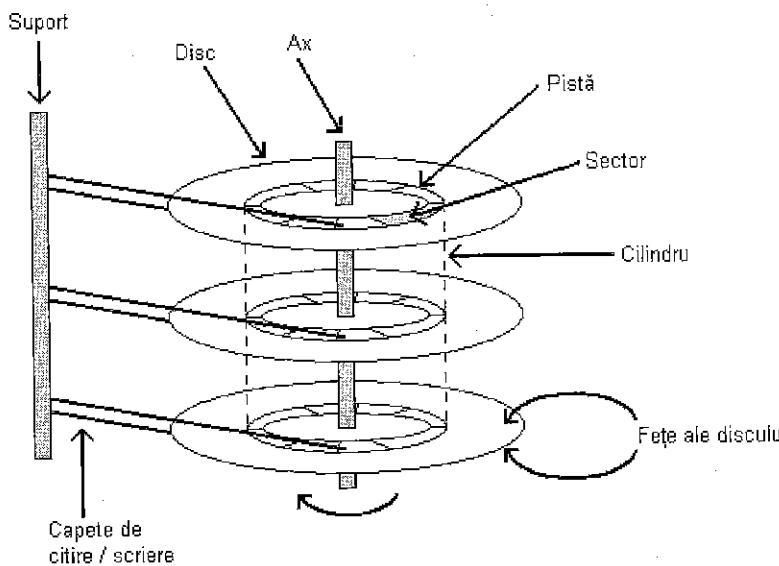


Fig. 2.3. Structura unui volum disc

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

$$\text{TimpAcces} = \text{TimpCăutare} + \text{TimpRotire} + \text{TimpTransfer}$$

unde:

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

La TimpAcces mai putem adăuga TimpController ce reprezintă întârzierea produsă de controllerul dispozitivului, ca interfață de comunicare între CPU și disc.

Performanțele dispozitivelor disc au rate de evoluție 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 diskurile - 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 performanțe ale acestor dispozitive în prezent: 7200-10000-15000 RPM (rotații pe minut), timp de acces mediu = 8-15 ms, 50-150 operații I/O / secundă;
- dischetele (floppy disk) - permit acces direct la date, au preț de producție și achiziționare foarte scăzut și sunt portabile; cele mai utilizate sunt cele de 1.44 megabytes (1 MB = 2^{20} bytes);
- benzile magnetice - permit acces secvențial la date, asigură o capacitate mare de stocare, sunt ieftine și sunt folosite în general pentru arhivare și memorarea copiilor de siguranță (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 producția și 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 și 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

Motivatie

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

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

Pentru a gestiona cât mai eficient accesul la date, un sistem de calcul definește un sistem complex al memoriei, în care combină memorie de capacitate mică, dar rapidă, și memorie de capacitate mare, însă de viteză redusă. Drecht 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:

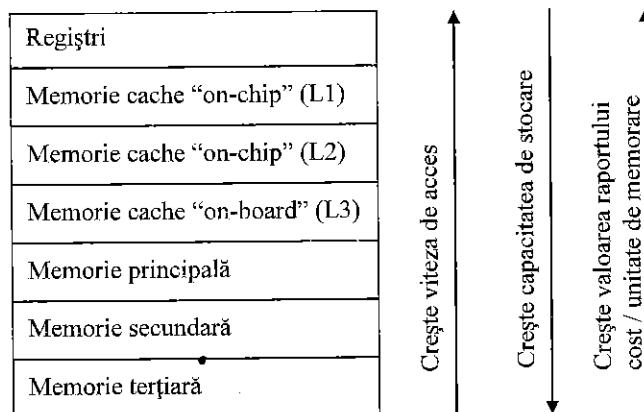


Fig. 2.4. Ierarhia memoriei

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 memorile de capacitate mare, dar de viteză de acces mai mică.

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

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

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

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

Memoria cache

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

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

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

există și 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. Secvența de căutare este:

- CPU solicită o instrucțiune sau un operand; fie I informația solicitată.
- Se caută în primul nivel cache, cel mai apropiat de CPU.
- Dacă I este găsit \Rightarrow cache hit (succes); se oprește căutarea pe acest nivel.
- Dacă I nu este găsit \Rightarrow cache miss (eșec); se continuă căutarea pe nivelul cache secundar.
- În cazul în care pe toate nivelele cache s-a raportat eșec, se caută I în memoria principală.

Ca și criterii de măsură a performanței unei memorii cache se folosesc

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

$miss\ rate = nr.miss / nr.referințe\ la\ memorie$ (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 interfață între aceasta și CPU: poate fi pe unul, două sau trei nivele; acest cache este gestionat de către hardware
- memoria cache între memoria principală și 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 tabelii de pagini folosită pentru realizarea corespondenței de adrese între memoria principală și memoria virtuală
- memorii cache gestionate de componente soft: cache DNS (pentru corespondențe dintre nume de domenii și 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 execuție dezvoltate).

În general, performanța unei memorii cache depinde de parametrii de organizare și funcționare 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 spațiu liber în cache (*block replacement policy*), scrierea datelor modificate (*cache write policy* – imediat sau la dealocarea blocului).

2.4. DISPOZITIVE PERIFERICE

Dispozitivele periferice asigură interfață dintre utilizator și sistemul de calcul sau dintre sistemul de calcul și alte sisteme fizice. Un asemenea dispozitiv este caracterizat de tipul de comportament (intrare, ieșire, stocare), partener (utilizator uman sau sistem fizic) și performanță.

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

Tipurile de dispozitive periferice întâlnite sunt:

- dispozitive de intrare: tastatură, mouse, scanner
- dispozitive de ieșire: imprimantă, monitor
- dispozitive de intrare sau ieșire: modem, placă de rețea
- 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 informație (date, instrucțiuni, 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 informația ca sir de biți (bit după bit). Magistralele parallele transportă simultan informație prin mai multe fire, mărinindu-se astfel rata de transfer. De exemplu, pe o magistrală de 16 biți se pot transmite simultan doi octeți. Acest lucru nu înseamnă că pe o magistrală paralelă viteza de transfer a informației va fi neapărat mai mare decât pe una serială. Dimpotrivă, datorită costurilor mari implicate de transmisia paralelă a datelor, în ultimul timp se remarcă renunțarea la magistralele paralele și concentrarea pe magistrale seriale care să lucreze la frecvențe de transfer mari (de exemplu: magistrala serială S-ATA are o frecvență de transfer mai mare decât magistrala paralelă IDE/ATA; similar pentru interfața serială USB în raport cu oricare interfață paralelă aflată în uz).

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

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

- *magistrale de date (data bus)* – transportă informație de tip dată sau instrucțiuni; performanța depinde în primul rând de dimensiune (de exemplu: magistrale de 8, 16, 32, 64 biți)
- *magistrale de adrese (address bus)* – informația comunicată este adresa locației de memorie pe care componenta solicitantă dorește să o acceseze (în citire sau scriere); dimensiunea magistralei determină capacitatea maximă de memorie adresabilă din sistem (de exemplu: sistemele de calcul cu regiștri pe 8 biți dețin magistrale de 16 biți, de

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

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

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

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

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

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

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

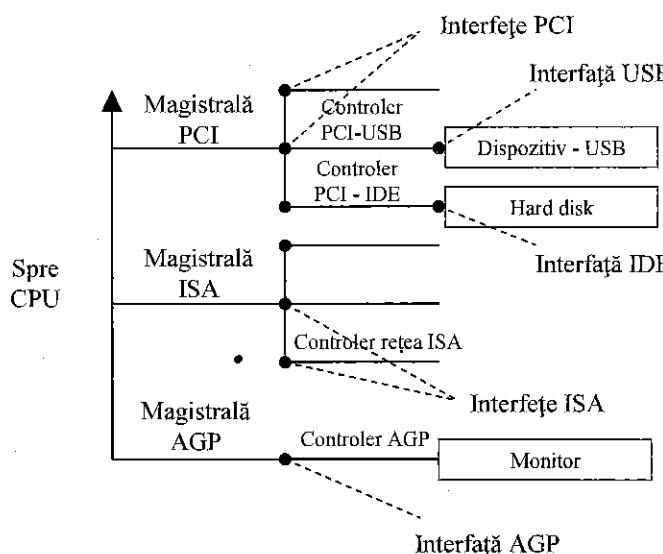


Fig. 2.5. Magistrale de I/O

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

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

2.4.2. Magistrale I/O interne și interfețele asociate

Magistrala ISA (Industry Standard Architecture)

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

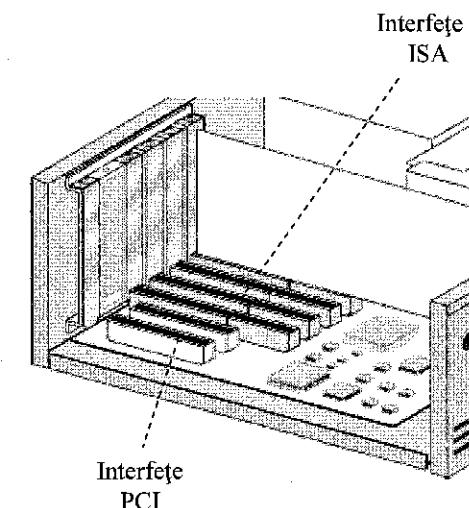


Fig. 2.6. Interfețele ISA și PCI din punct de vedere fizic în cadrul unui calculator personal

Magistrala PCI (Peripheral Component Interconnect)

A fost introdusă de Intel la sfârșitul anilor '90 și se bucură de succes chiar și astăzi. Este o magistrală pe 32 de biți și operează

la frecvențe începând cu 33 MHz. Aceste valori permit viteze teoretice de până la 132 Megaoceteți pe secundă ($33 \cdot 10^6 \cdot 4$ octeți). Majoritatea controlerelor existente în momentul de față se conectează la această magistrală: controlere IDE/ATA pentru conectarea hard disk-urilor, controlere de rețea, controlere multimedia de captură, sunet, etc. Este specifică și altor tipuri de calculatoare, nu numai calculatoarelor personale, cum ar fi Power Macintosh.

Magistrala AGP (Accelerated Graphics Port)

S-a născut din nevoie de lățime de bandă mai mare spre controlerul video. Aplicații precum jocurile cereau o lățime de bandă pe care magistrala PCI nu o putea oferi. Făcând un calcul simplu, un joc care rulează la rezoluția 1600 x 1200, cu 32 biți de culoare per pixel, pentru a putea reda 30 de cadre pe secundă are nevoie de o lățime de bandă spre controlerul video de $1600 \cdot 1200 \cdot 4$ octeți • 30 ≈ 230 Megaoceteți pe secundă, cu mult peste posibilitățile oferite de magistrala PCI. De fapt numele de magistrală AGP este folosit impropriu, AGP fiind un canal de comunicație punct la punct. Magistrala AGP 1x este pe 32 de biți și lucrează la frecvențe de 66 MHz, oferind o viteză de aproximativ 266 Megaoceteți pe secundă – dublu față de viteză oferită de magistrala PCI. Această lățime de bandă este dedicată în întregime controlerului video, în timp ce un controler video PCI era nevoie să partajeze această viteză cu alte controlere PCI, mari consumatoare de lățime de bandă, cum ar fi controlere IDE/ATA. Magistralele AGP 8x actuale, permit atingerea unor viteze de 8 ori mai mare decât specificațiile AGP inițiale (2133 Megaoceteți pe secundă).

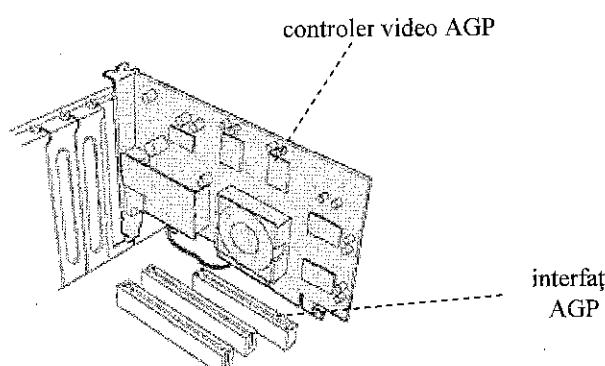


Fig. 2.7. Interfața AGP și un controler video AGP

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

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

2.4.3. Magistrale I/O externe și interfețele asociate

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

Interfața 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 și unitățile optice. Primele specificații ATA, foloseau procesorul (așa numitul *programmed input/output mode* sau mod PIO) pentru a transfera informația octet cu octet de pe dispozitivul periferic de stocare în memoria internă prin intermediul regiștrilor procesorului. Acest mod de lucru, permitea obținerea de viteze relativ mici de până la 16.7 Megaoceteți pe secundă și, mai grav, rețineau procesorul ocupat în timpul acestui transfer. Această viteză a crescut însă odată cu folosirea DMA (*Direct Memory Access*), permitându-se astfel copierea informației în blocuri de pe dispozitivul de stocare direct în memoria internă fără a mai implica procesorul în acest transfer. Vitezele actuale ale interfețelor ATA 133 se apropie cel puțin teoretic de limitele magistralei PCI de 132 Megaoceteți pe secundă pentru transferul datelor (să nu uităm că informația citită de pe dispozitivul de stocare trebuie să treacă prin magistrala PCI via controler).

Specificațiile ATA descriu și modul de adresare a dispozitivului de stocare. Ultimele specificații prevăd o adresare pe 48 de biți, lucru care permite folosirea teoretică a dispozitivelor de stocare cu capacitate de până la 32 de Teraocteți. Au existat situații când proiectanții de controlere IDE au subestimat viteză de creștere a capacitatii unităților de stocare, modele noi de astfel de dispozitive fiind incompatibile cu controlerile mai vechi (astfel de bariere au fost întâlnite la trecerea la capacitate de stocare de peste 504 Megaoceteți, 8, 32, 137 Gigaoceteți).

Fizic, pe un controler IDE/ATA se pot conecta doar două dispozitive de stocare, unul operând în mod *master* și celălalt în mod *slave*. Majoritatea calculatoarelor personale, fiind dotate cu două controlere IDE (numite *primary* și *secondary* IDE), pot folosi la un moment dat doar 4 dispozitive periferice IDE. Dacă se dorește 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, unități optice, unități de bandă magnetică. Poate fi însă folosită și 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 și în stațiile și serverele Sun. Acest tip de magistrală, poate fi folosită atât ca magistrală internă, cât și ca magistrală externă. În calculatoarele personale, a fost folosită doar ca magistrală externă, comunicând cu procesorul doar prin intermediul unui controler ISA sau PCI prin magistrala internă corespunzătoare tipului de controler. Funcțional, o magistrală SCSI operează la frecvențe cuprinse între 5 MHz (conform primelor specificații SCSI) și 80 de MHz, dimensiunea magistralei fiind de 8 sau 16 biți. O magistrală SCSI pe 16 biți operând la 80 MHz poate atinge viteze de până la 320 Megaoceteți pe secundă în transferul datelor.

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

Interfața serială

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) și terminalele virtuale. Controlerul serial la calculatoarele personale de azi este pe cale de dispariție, majoritatea perifericelor care se conectau la calculator pe această interfață conectându-se în prezent prin intermediul unei interfețe USB. Acolo unde este încă prezent, controlerul serial este legat de magistrala ISA sau PCI. Viteza maximă atinsă pe un port serial este foarte mică, 128000 biți /secundă ≈ 16 kiloocteți / secundă.

Interfața paralelă

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

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

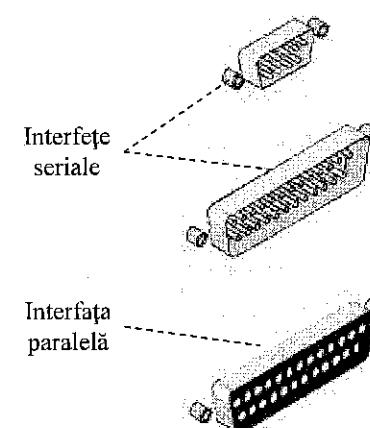


Fig. 2.8. Interfețe seriale și paralele

USB (Universal Serial Bus)

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

2.4.4. Componentele unui calculator personal

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

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

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

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

- eventual interfețe ISA;
- cel puțin două interfețe USB dacă placa de bază are un asemenea controller;
- una sau două interfețe seriale și o interfață paralelă (vezi paragraful 2.4.3.);
- cel puțin trei interfețe audio (intrare, ieșire și microfon) dacă placa de bază are un controller (placă) de sunet;
- interfață de rețea în cazul prezenței unui asemenea controller pe placa de bază;
- interfețe PS/2 pentru atașarea unei tastaturi și a unui mouse;
- interfață specială pentru atașarea unui joystick (așa numitul *game port*) sau a altor periferice destinate jocurilor.

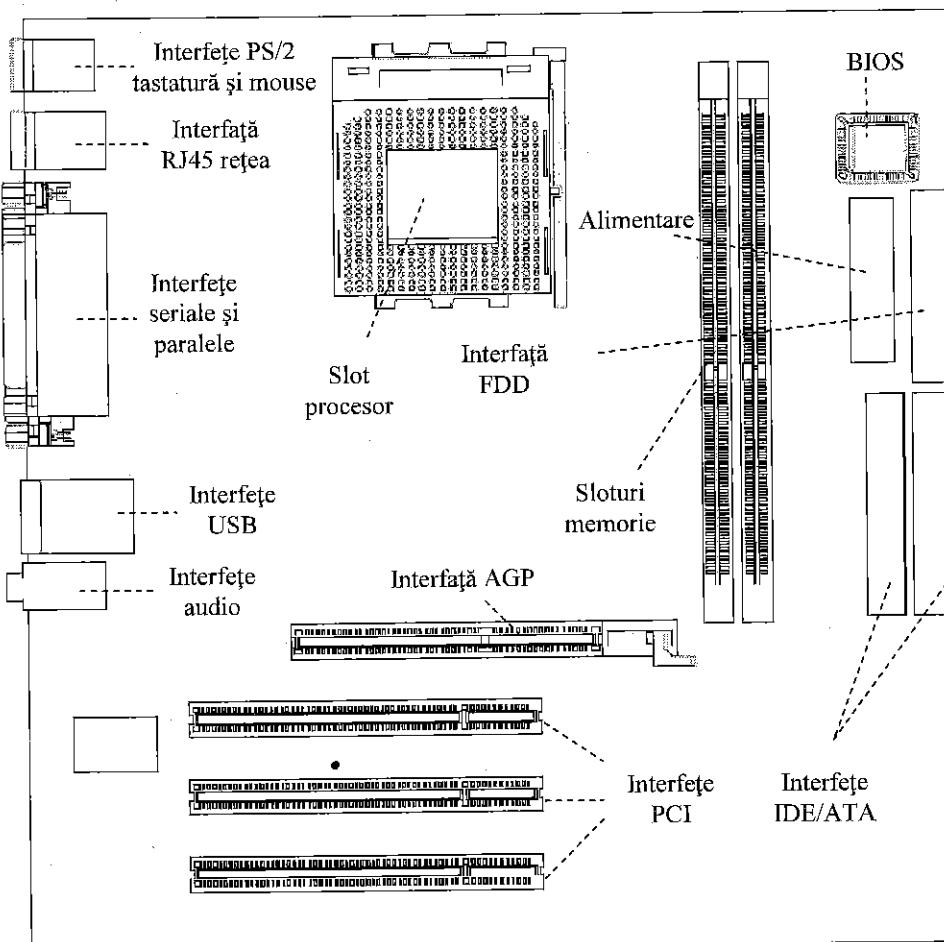


Fig. 2.9. Placă de bază a unui calculator personal

Aceste interfețe sunt ilustrate în figura 2.9. Există situații în care echipamentul periferic nu se atașează direct interfeței oferită de controller sau de placa de bază, fiind necesar un cablu (sau "panglică") prelungitor care este parte integrantă a interfeței fizice de conectare oferită de controllerul respectiv. Spre exemplu, un hard disk nu se conectează direct interfeței IDE/ATA de pe placa de bază, fiind necesar un cablu (numit eronat uneori controller IDE) prelungitor de la interfața IDE a plăcii de bază la hard disk. De asemenea, un mouse serial se poate conecta la controllerul serial prin intermediul unei cabluri prelungitoare.

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

- interfețele PCI (vezi figura 2.6.): permit conectarea oricărui controller care comunică cu procesorul prin intermediul magistralei PCI. Majoritatea controllerelor 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 comunicații fără fir);
- interfața AGP (vezi figura 2.7.) permite conectarea unui controller video. Un controller video din zilele noastre poate fi considerat un calculator în calculator, fiind dotat cu propriul procesor grafic și propria memorie. Uneori, procesorul grafic și memoria controllerului video pot fi comparabile ca putere și dimensiune cu procesorul și memoria sistemului (acest lucru se datorează în primul rând cifrelor mari de afaceri din industria jocurilor pe calculator).
- interfața IDE/ATA permite conectarea hard disk-urilor și unităților optice (CD-RW, DVD-ROM, etc). Pentru conectarea acestor echipamente la această interfață este necesară o "panglică" suplimentară. Este permisă adăugarea pe un singur controller a două astfel de echipamente periferice, unul operând în mod master și celălalt în mod slave. Acest mod de operare se configureră cu ajutorul unor jumperi prezenti la fiecare hard disk sau unitate optică IDE.

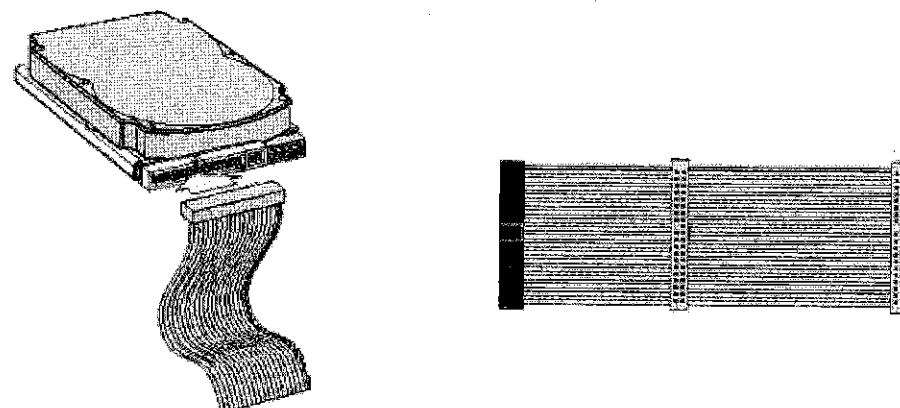


Fig. 2.10. Hard disk și cablu de conectare IDE/ATA

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

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

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

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

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

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

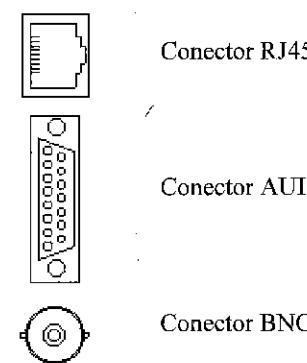


Fig. 2.11. Interfețe de rețea

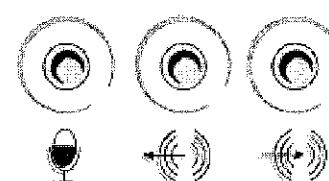


Fig. 2.12. Interfețe audio

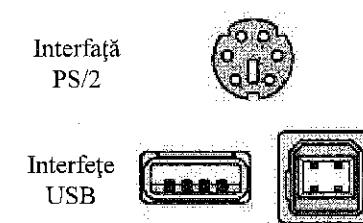


Fig. 2.13.
Interfețe USB și PS/2

2.5. PERFORMANȚELE UNUI SISTEM DE CALCUL

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

- *timpul de execuție (Execution Time)* = timpul în care este executată o sarcină, timpul de răspuns
- *rata de execuție (Throughput, Bandwidth)* = numărul de sarcini rezolvate într-un anumit interval de timp (zi, oră, secundă, milisecundă...),
de unde rezultă că îmbunătățirea performanței poate fi privită sub două aspecte:
 - reducerea timpului de rezolvare a unei sarcini
 - creșterea ratei de execuție.

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

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

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

2.6. ARHITECTURA MICROPROCESORULUI 8086

2.6.1. Structura microprocesorului

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

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

Cele două componente lucrează în paralel, în sensul că în timp ce EU execută instrucțiunea curentă, BIU pregătește instrucțiunea următoare. Cele două acțiuni sunt sincronizate, în sensul că cea care termină prima așteaptă după cealaltă.

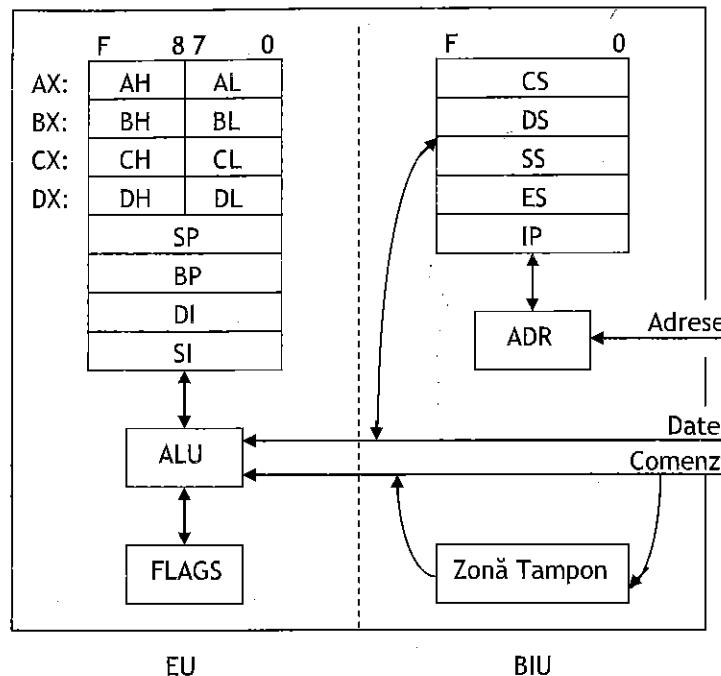


Fig. 2.14. Arhitectura microprocesorului 8086

2.6.2. Registrii generali EU

Registrul **AX** este *registratorul acumulator*. El este folosit de către majoritatea instrucțiunilor ca unul dintre operanți.

Registrul **BX** este folosit în principal ca *registrator de bază*. Folosirea lui în această accepție va fi descrisă în secțiunea 2.6.7.

Registrul **CX** este folosit în principal ca *registrator de numărare (registrator contor)* pentru instrucțiunile care au nevoie de indicații numerice.

Registrul **DX** este un *registrator de date*. Împreună cu registrul AX se folosește în calculele ale căror rezultate depășesc dimensiunea unui cuvânt.

Fiecare dintre registrii AX, BX, CX, DX au capacitatea de 16 biți. Fiecare dintre ei poate fi privit în același timp ca fiind format prin concatenarea (alipirea) a doi (sub)registri. Subregistru superior conține cei mai semnificativi 8 biți (partea HIGH) ai registratorului de 16 biți din care face parte. Există astfel registrii AH, BH, CH, DH. Subregistru inferior conține cei mai puțin semnificativi 8 biți (partea LOW) ai registratorului de 16 biți din care face parte. Există astfel registrii AL, BL, CL, DL.

Registrii **SP** și **BP** sunt registri destinați lucrului cu *stiva*. O stivă se definește ca fiind o zonă de memorie în care se pot depune succesiv valori, extragerea lor ulterioră făcându-se în ordinea inversă depunerii.

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

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

Registrii **DI** și **SI** sunt *registrii de index* utilizati de obicei pentru accesarea elementelor din siruri de octeți sau de cuvinte. Denumirile lor (*Destination Index* și *Source Index*) precum și rolurile lor vor fi clarificate în capitolul 4, secțiunea 4.4.2.

2.6.3. Flagurile

Un *flag* este un indicator reprezentat pe un bit. O configurație a *registratorului de flaguri* indică un rezumat sintetic a execuției fiecărei instrucțiuni. Pentru 8086 acest registrator (notat FLAGS în figura 2.14) are 16 biți dintre care sunt folosiți numai 9. Structura în detaliu a registratorului FLAGS este dată în figura 2.15.

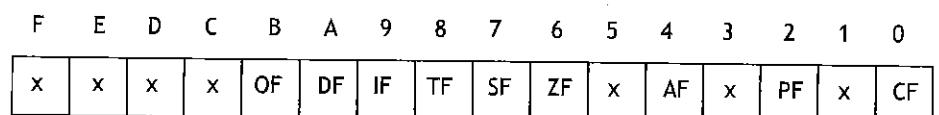


Fig. 2.15. Structura registratorului de flaguri 8086

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

$$\begin{array}{r}
 1001\ 0011 + \\
 0111\ 0011 \\
 \hline
 1\ 0000\ 0110
 \end{array}$$

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

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

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

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

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

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

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

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

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

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

2.6.4. Registrii de adresă și calculul de adresă

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

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

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

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

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

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

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

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

$$a_4 a_3 a_2 a_1 a_0 := S_3 S_2 S_1 S_0 0 + O_3 O_2 O_1 O_0$$

unde $a_4 a_3 a_2 a_1 a_0$ este adresa calculată (scrisă în hexazecimal). Deci configurația de 16 biți $s_3 s_2 s_1 s_0$ care localizează începutul segmentului este înmulțită cu 16 (i se adaugă o cifră 0 în baza 16, sau 4 cifre 0 în baza 2) și la rezultat se adună valoarea offsetului $o_3 o_2 o_1 o_0$. Acest calcul este efectuat de către componenta **ADR** din BIU.

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

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

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

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

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

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

Arhitectura 8086 permite existența a patru tipuri de segmente:

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

- 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 execuției este declarat activ către un singur segment din fiecare tip. Regiștrii CS (Code Segment), DS (Data Segment), SS (Stack Segment) și ES (Extra Segment) din BIU rețin adresele de început ale segmentelor active, corespunzător fiecărui tip. Deci regiștrii CS, DS, SS și ES conțin (de fapt) adresele de început ale segmentelor active: de cod, de date, de stivă și suplimentar. Registrul IP conține offsetul instrucțiunii curente în cadrul segmentului de cod curent, el fiind manipulat exclusiv de către BIU.

2.6.5. Reprezentarea instrucțiunilor mașină

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

numeinstrucțiune destinație, sursă

Formatul intern al unei instrucțiuni este variabil, el putând ocupa între 1 și 6 octeți. Semnificația generală a octetelor reprezentării este următoarea:

Primul octet, numit *cod* identifică instrucțiunea de executat.

Al doilea octet, numit *octetmod* specifică pentru unele dintre instrucțiuni natura și locul operanzilor (registrator, memorie, constantă întreagă etc.). Unele instrucțiuni folosesc acest octet pentru a reprezenta în el o *constantă scurtă (short)*, adică o constantă care poate lua valori între -128 și 127.

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

Următorii maximum patru octeți, dacă apar, identifică fie o adresă de memorie, fie o constantă reprezentată pe mai mult de un octet. De aici se deduce o restricție suplimentară privind operanzi: 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 și NEAR

Așa cum am văzut în secțiunea 2.6.4., pentru a adresa o locație din memoria RAM sunt necesare două cuvinte: unul care să indice segmentul, altul care să indice offsetul în cadrul segmentului. Pentru a simplifica referirea la memorie, microprocesorul preia, în lipsa unei alte specificări, adresa segmentului din unul dintre regiștrii de segment CS, DS, SS sau ES. Alegera implicită a unui registrator de segment se face după niște reguli proprii instrucțiunii folosite.

Prin definiție, o adresă în care se specifică doar offsetul, urmând ca segmentul să fie preluat implicit dintr-un registrator 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 conține 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, și reprezentarea adreselor respectă principiul reprezentării little-endian expus în capitolul 1, paragraful 1.3.2.3: partea cea mai puțin semnificativă are adresa cea mai mică, iar partea cea mai semnificativă are adresa cea mai mare.

Faptul că o instrucțiune folosește o adresă FAR sau NEAR se reflectă în conținutul octetilor care codifică instrucțiunea. În capitolul 4 vom vedea cum poate fi specificată natura adresei în limbaj de asamblare, lăsând astfel pe seama asamblorului codificarea instrucțiunii.

2.6.7. Calculul offsetului unui operand. Moduri de adresare

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

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

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

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

- conținutul unuia dintre regiștrii BX sau BP;
- conținutul unuia dintre regiștrii 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 și indexată etc.

La instrucțiunile de salt mai apar două tipuri de adresări.

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

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

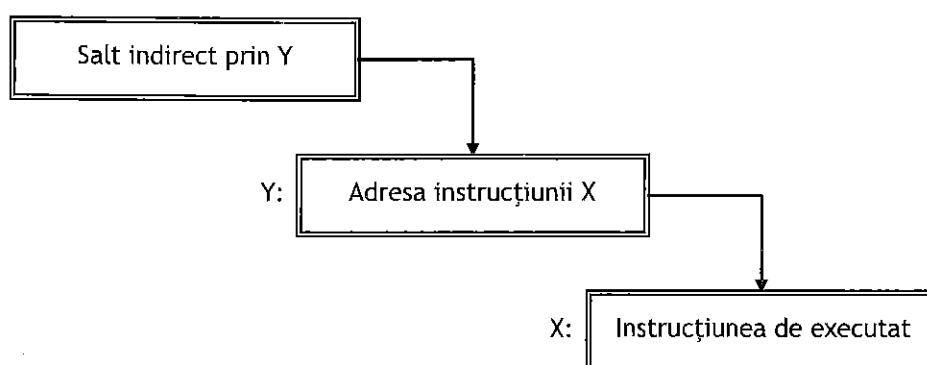


Fig. 2.16. Adresarea indirectă

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

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

CAPITOLUL 3

ELEMENTELE LIMBAJULUI DE ASAMBLARE

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

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

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

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

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

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

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.

* **instrucțiuni** - scrise sub forma unor mnemonice care sugerează acțiunea, fiecare instrucție a limbajului de asamblare corespunzând unei instrucții mașină. Asamblorul generează octeți care codifică instrucția respectivă.

* **directive** - sunt indicații date asamblorului în diverse scopuri cum ar fi: relații între modulele obiect, definirea unor segmente, indicații de asamblare condiționată, machete de macrogenerare, controlul listingului, etc. Directivele care indică asamblorului să încarce o zonă de memorie cu un conținut dorit de programator sau care rezervă unui număr de octeți în vederea folosirii ulterioare se mai numesc și *pseudoInstrucții* sau *directive de generare a datelor*.

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

Introducerea noțiunii de contor de locații se va dovedi utilă în scopul identificării/utilizării dinamice a unui punct curent de execuție în cadrul unui program (segment de cod/instrucții) sau a adreselor unor locații de memorie care tocmai au fost alocate în cadrul unor segmente de date. Acest concept al contorului de locații propriu fiecărui segment de memorie este relevant pentru a înțelege modul în care trebuie să privim execuția unor programe în limbaj de asamblare: **execuția unui program în limbaj de asamblare nu este altceva decât o generare de octeți corespunzătoare instrucțiunilor și directivelor specificate în codul sursă de către programator!**

Într-adevăr, un program în execuție va genera în cadrul unui segment de cod octeți reprezentând codurile instrucțiunilor ce trebuie executate, iar în cadrul unui segment de date se vor genera corespunzător în memorie octeți ce vor reflecta alocarea datelor declarate de către programator, în funcție de tipul de date specificat. În acest sens este important de reținut că în cadrul limbajului de asamblare noțiunea de tip de dată este echivalentă cu dimensiunea de reprezentare a datei respective (vizuire low-level asupra noțiunii de tip de date specifică limbajelor de asamblare). Vom reveni practic asupra tuturor acestor aspecte și asupra efectelor lor pe tot parcursul volumului

deoarece este foarte important să punctăm semantica asociată instrucțiunilor și directivelor din exemplele ce se vor prezenta prin referirea la noțiunile și concepții definite aici.

3.1. FORMATUL UNEI LINII SURSA

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 \$ și ? folosite singure au un înțeles special: simbolul '\$' desemnează valoarea contorului de locații iar simbolul '?' este folosit pentru rezervarea de spațiu de memorie fără initializare. 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 secvențelor de instrucții (deci în cadrul unor segmente de cod) cu scopul de a defini destinația de transfer ale controlului în cadrul unui program. O etichetă de cod este de fapt echivalentul noțiunii de *etichetă* ce apare la nivelul unui limbaj de programare de nivel înalt, semantica ei fiind definirea destinației unor instrucții de tip *goto*. La definirea lor în cadrul programelor, etichetele ce prefixează instrucții (etichetele de cod) trebuie să fie urmate de caracterul '!'.

2). **etichete de date**, care identifică simbolic unele locații de memorie, din punct de vedere semantic ele fiind echivalentul noțiunii 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 unică). Ca operand, o aceeași 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 instrucției sau directivei ce urmează etichetei.

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

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

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

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

Spre exemplu în cadrul instrucțiunilor:

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

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

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

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

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

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

Operanții sunt parametri care definesc valorile ce vor fi prelucrate de instrucțiuni sau de directive. Ei pot fi regiștri, constante, etichete, expresii, cuvinte cheie sau alte simboluri. Semnificația operanților depinde de mnenomica instrucțiunii sau directivei asociate.

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

Parantezele drepte precizează că prezența elementului respectiv este optională. Așadar, putem avea linii cu toate cele patru elemente prezente, după cum putem avea și 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 mulți operanți care sunt combinați pentru a descrie o valoare sau o locație de memorie. *Operatorii* indică modul de combinare a operanților în scopul formării expresiei. Expresiile sunt evaluate în momentul asamblării (adică, valorile lor sunt determinabile la momentul asamblării, cu excepția acelor părți care desemnează conținuturi de regiștri și care vor fi determinate la execuție).

3.2.1. Moduri de adresare

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

Relativ la modurile de adresare, cele trei tipuri de operanzi sunt **operanzi imediați**, **operanzi registru** și **operanzi în memorie**. Valoarea operanzilor este calculată în momentul asamblării pentru operanzi imediați, în momentul încărcării programului pentru adresarea directă și în momentul execuției pentru operanzi registru și cei adresati indirect (vezi 2.6.7.).

Instrucțiunile care au doi sau mai mulți operanzi operează întotdeauna de la dreapta spre stânga. Operandul din dreapta este operandul *sursă*. El specifică datele care vor fi folosite în operație, dar nu și modificate. Operandul din stânga este operandul *destinație*. El specifică datele care vor fi folosite și, probabil, modificate de către instrucție.

3.2.1.1. Utilizarea operanzilor imediați

Operanzi imediați sunt formați din date numerice constante cunoscute sau calculabile la momentul asamblării.

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

Constantele întregi reprezintă valori întregi. Ele pot fi utilizate în declarații sau ca operanzi imediați.

Constantele întregi se specifică prin valori binare, octale, zecimale sau hexazecimale. Baza de numerație se dă printr-un specificator al bazei de numerație după ultima cifră a numărului, astfel: pentru numere binare specificatorul B, pentru cele octale -Q sau O, pentru numere zecimale - D și 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 și 9. De exemplu, OABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol. Cifrele hexazecimale de la A la F precum și specificatorii bazei de numerație pot fi atât litere mari cât și mici. Exemple: 123d (echivalent cu 123), 94fah, 63701o, 01101011b, 0FAB2h, etc.

Dacă nu se folosește nici un specificator asamblorul interpretează întregul folosind baza de numerație implicită. Inițial, baza de numerație 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 sir** 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 instrucțiuni există o limită maximă în ceea ce privește dimensiunea de reprezentare a valorilor imediate (de obicei 8, 16 sau 32 biți). Constantele și 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 instrucțiuni.

Datele imediate nu sunt admise ca operand destinație (așa cum nici 2:=N nu este o instrucție 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 inițializarea variabilelor codificate binar zecimal (BCD). Cu *numere reale* se poate opera numai în prezența unui coprocessor matematic.

Deplasamentul unei etichete – valoare constantă determinabilă la momentul asamblării.

Este foarte important de semnalat faptul că pe lângă constantele numerice "clasice" echivalente ca semnificație celor din limbajele de nivel înalt, la momentul asamblării, asamblorul poate determina și o altă categorie de valori care rămân constante pe tot parcursul executiei programului și anume deplasamentele etichetelor de date și de cod. De aceea, de exemplu, o instrucție 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 ; distanță de 8 octeți față de începutul segmentului de date

pe baza ordinii de declarare a variabilelor și a tipului de dată (dimensiunea de reprezentare) specificat la declarare pentru variabila v – lucruri ce rămân constante relativ la execuția programului și astfel calculabile la momentul asamblării. În mod similar, o instrucție 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 octeți mai jos față de poziția 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 și vor rămâne valori constante pe timpul executiei și ce este și mai important aceste valori sunt determinabile la momentul asamblării! Aceasta se întâmplă deoarece octetii corespunzători segmentelor sunt generati la momentul asamblării (se poate să fie

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

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

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

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

3.2.1.2. Utilizarea operanzilor registru

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

Operanzii-registru sunt formați din datele memorate în regiștri. Modul de *adresare directă* în cazul regiștrilor înseamnă folosirea valorii reale din interiorul registrului în momentul execuției instrucțiunii (de exemplu **mov ax,bx** – provoacă transferul valorii din registrul *bx* în registrul *ax*). De asemenea, regiștrii pot fi folosiți *indirect* pentru a indica locațiile de memorie (de exemplu instrucțiunea **mov ax,[bx]** va provoca transferul cuvântului de memorie de la adresa desemnată de *bx* spre 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 și noțiunea clasica de *variabilă* cunoscută din limbajele de nivel înalt) se împart în două grupuri: operanzi cu *adresare directă* și operanzi cu *adresare indirectă*.

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

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

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

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

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

Observație.

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

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

3.2.1.4. Operanzi cu adresare indirectă

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

În cazul microoprocesoarelor 8086 numai patru registrii pot fi folosiți în adresarea indirectă: BX, BP (*registri de bază*), DI și SI (*registri index*). Orice tentativă de a folosi alți registrii, diferiți de cei patru de mai sus, într-o instrucțiune care accesează memoria indirect, va produce o eroare.

Registrii de bază sau index pot fi folosiți 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., și anume:

$$(1) \quad [registru_de_bază + registru_index + deplasament]$$

unde toate cele trei componente sunt optionale, 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 și 6 sunt deplasamente. Asamblorul calculează deplasamentul real adunând *table* și 6, pentru a obține deplasamentul total.

Operanzi *registru_de_bază* și *registru_index* sunt folosiți de obicei pentru a indica o adresă de memorie referitoare la un tablou.

În ceea ce privește *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, registru 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 asamblări sau moduri de asamblare permit și alte modalități de a specifica operanzi cu adresare indirectă. Orice operator care indică adunarea (+, [], .) poate fi folosit pentru a combina deplasamentul cu registrii 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 registrii trebuie să fie registru de bază, iar celălalt trebuie să fie registru index. Următoarele instrucțiuni sunt incorecte:

mov ax, table [bx] [bp]	;illegal - doi registri de bază!
mov ax, table [di] [si]	;illegal - doi registri index!

Deci, să reținem că pentru adresarea indirectă, esențială este specificarea între paranteze drepte a cel puțin 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 și analiza operanziilor. Unii operatori lucrează cu constante întregi, alții cu valori întregi memorate, iar alții cu ambele tipuri de operanzi.

Este importantă înțelegerea diferenței dintre operatori și instrucțiuni. Operatorii efectuează calcule cu valori constante determinabile la momentul asamblării. Instrucțiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuției. De exemplu, operatorul de adunare (+) efectuează adunarea în momentul asamblării, în timp ce instrucțiunea ADD efectuează adunarea în timpul execuției.

Expresiile sunt evaluate conform următoarelor reguli:

- Operațiile cu prioritatea cea mai mare sunt efectuate primele.
- Operațiile cu aceeași prioritate se execută de la stânga la dreapta.
- Ordinea de prioritate poate fi modificată prin folosirea parantezelor. Operațiile din paranteze se efectuează întotdeauna înaintea oricărora operații adiacente.

În tabelul de mai jos sunt prezentate în ordinea priorității operatorii ce pot fi folosiți în cadrul expresiilor limbajului de asamblare 8086. Operatorii de pe aceeași linie au prioritate egală.

maximă	
1	0, [], <-, 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	SHORT, TYPE, SMALL, LARGE
minimă	

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

3.2.2.1. Operatori aritmetici

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

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

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

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

3.2.2.2. Operatorul de indexare

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

[expresie_1] [expresie_2]

Ca efect, se adună expresie_1 cu expresie_2. Restricțiile privind adunarea operanzilor păstrați în memorie ce se aplică la operatorul de adunare sunt valabile și pentru operatorul de indexare. De exemplu, nu se pot aduna doi operanzi în memorie adresați în mod direct. Expressia *eticheta_1* [*eticheta_2*] nu este admisă dacă ambii sunt operanzi în memorie.

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

3.2.2.3. Operatori de deplasare de biți

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

expresie SHR cu_cât și expresie SHL cu_cât

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

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

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

3.2.2.4. Operatori logici pe biți

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

OPERATOR	SINTAXA	SEMNIFICATIE
NOT	NOT expresie	complementare biți
AND	expr1 AND expr2	ȘI 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 relationali

Un operator relațional compara (cu semn!) două expresii și întoarce valoarea adevărat (-1) când condiția 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 sir de biți, având toti valoarea 1. Expresiile evaluate au ca rezultat valori constante. Numele și 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) și 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 funcție de un anumit segment. Sintaxa este:

segment:expresie

unde *segment* poate fi specificat în mai multe moduri. El poate fi unul dintre registri 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.) și 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 descriși operatorii limbajului de asamblare care specifică sau analizează tipurile unor expresii și a unor operanzi păstrați în memorie.

Operatorul PTR

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

tip PTR expresie

Operatorul forțează ca *expresie* să fie tratată ca având dimensiunea de reprezentare *tip*, fără însă a-i modifica definitiv (destructiv) valoarea în sensul precizat de conversia dorită. De aceea, operatorul PTR este considerat un *operator de conversie (temporar) nedistructiv*. Pentru operanții păstrați în memorie, *tip* poate fi BYTE, WORD, DWORD, QWORD sau TBYTE, având dimensiunile de reprezentare 1, 2, 4, 8 și respectiv 10 biți. Pentru etichetele de cod el poate fi NEAR (adresă pe 2 octeți) sau FAR (adresă pe 4 octeți).

Operatorul PTR este folosit pentru a da posibilitatea instrucțiunilor să-și "vadă" operanții ca fiind de tipul indicat în cazul în care aceștia 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 semnificația 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 și segment sunt egale cu valoarea curentă a contorului de locații și al cărui tip este specificat de operator. Sintaxa lui este:

THIS *tip*

Pentru operanții păstrați în memorie, *tip* poate fi BYTE, WORD, DWORD, QWORD sau TBYTE. Pentru etichete, *tip* poate fi NEAR sau FAR. Operatorul THIS se folosește de obicei cu directivele EQU sau = pentru crearea unor etichete sau variabile. Rezultatul este similar cu cel obținut 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 inițializarea 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 și LOW

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

HIGH expresie și **LOW expresie**

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

Operatorii SEG și OFFSET

Sintaxele acestor doi operatori sunt:

SEG expresie și **OFFSET expresie**

unde *expresie* adresează direct o locație de memorie.

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

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

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

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

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

3.3. DIRECTIVE

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

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

3.3.1. Directive standard pentru definirea segmentelor

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

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

3.3.1.1. Directiva SEGMENT

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

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

[*instrucțiuni*]

nume ENDS

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

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

Argumentul optional *aliniere* specifică multiplul numărului de octeți 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* lipsește, atunci se consideră implicit că este vorba despre o alinieră de tip PARA.

Argumentul optional *combinare* controlează modul în care segmente cu același 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 același nume, obținându-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 același nume. Se obține un segment având dimensiunea egală cu cea a celui mai mare segment având același nume.
- **AT <expresie>** - impune ca segmentul să fie încărcat în memorie la adresa reprezentată de valoarea expresiei.
- **STACK** - segmentele cu același nume vor fi concatenate. În faza de execuție segmentul rezultat va fi segmentul stivă.
- **MEMORY** - segmentele cu acest atribut vor fi așezate în memorie în spațiul 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 același nume nu vor fi combinate, fiecare primind o zonă de memorie separată.

Observații.

1. Pentru programele care se intenționează a deveni programe .COM (vezi 5.5.3.) nu trebuie specificată alinieră.
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 spațiu pentru stivă în continuarea unuia dintre segmentele existente.

Argumentul *utilizare* se folosește numai în cazul microprocesoarelor 80386 și 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 aceeași clasă vor fi plasate într-un bloc contigu 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 și gestiunea segmentelor

Directiva **ASSUME** stabilește 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 și oricără dintre acestea putând să lipsească. Fiecare dintre *nume1*, *nume2*, *nume3* și *nume4* este un nume de segment sau cuvântul rezervat **NOTHING**.

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

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

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

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

*Unde și cum intervine însă exact informația 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 informația de asociere furnizată prin ASSUME. În cadrul parcurgerii textului sursă, asamblorul va întâlni o anume etichetă, o va identifica drept deplasament în cadrul acelui segment, însă pentru referirea la o adresă completă (segment:deplasament) va fi necesar ca în acel moment să prefixeze acel offset cu adresa de segment corespunzătoare. Care este însă aceasta? Tocmai aici intervine ASSUME: *asamblorul va căuta în textul sursă locul în care apare definitia etichetei, va identifica numele segmentului în care apare această definiție de etichetă după care va recurge imediat la ASSUME pentru a vedea cu ce fel de registrul de segment a fost asociat numele de segment identificat și astfel va prefixa eticheta cu registrul de segment corespunzător detectat*. Oricare referiri (accesări) a unei etichete i se aplică mecanismul descris. Recomandăm a se urmări aplicarea acestui mecanism în exemplele care urmează pentru a înțelege pe deplin necesitatea furnizării de 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 și astfel directiva ASSUME nu va mai fi aplicată și ca urmare nu mai este necesară. Însă acest lucru se întâmplă numai pentru acele adrese utilizate explicit ca adrese FAR (de exemplu: FAR PTR a, ds:b sau data:v). Celelalte adrese (adică cele NEAR) trebuie să facă obiectul unor asocieri implicate cu adrese de segment (și de aceea este nevoie de ASSUME) iar limbajul de asamblare specifică faptul că obiectul unor astfel de asocieri implicate pot fi numai registri de segment. De aceea de exemplu nu sunt posibile specificări prin ASSUME de tipul:

ASSUME data:d1 sau ASSUME 07Abh:d2

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

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

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

x:
c ends

ASSUME CS:d

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

d segment

etd: jmp y ;salt near necondiționat la eticheta locală y (aici este nevoie de ASSUME pentru a se putea compune corect adresa fizică CS:y)

y: jmp far ptr etc ;salt far necondiționat la eticheta etc din cadrul segmentului de cod c

d ends

end start

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

c segment

start: jmp far ptr etd ;salt far necondiționat la eticheta etd din cadrul segmentului de cod d

etc: ... ;nu se impune aici prezența unei directive ASSUME deoarece eticheta locală etc este referită doar ca etichetă FAR din segmentul d

c ends

d segment

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

... ;(aceeași observație ca mai sus și 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, operațiile la nivel de stivă se rezumă la introducerea sau scoaterea de cuvinte în și respectiv din vârful stivei.

Direcția ASSUME pentru SS este necesară numai în cazul în care se definește un segment de stivă și 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 privește 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 dorește 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 și ES să fie asociati la același segment de date.

În ceea ce privește încărcarea regiștrilor de segment, precizăm următoarele:

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

Anticipând instrucțiunea 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 regiștrii de segment nu pot fi încărcați direct, ci numai printr-un intermediar (registr 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 înțelege necesitatea utilizării unei directive ASSUME pentru definitivarea adreselor de accesare a etichetelor definite în program și evită astfel eroare de sintaxă de genul celei de mai sus, utilizarea directivei ASSUME se poate constitui și într-o potențială sursă de eroare logice! Acest lucru se întâmplă atunci când deși este prezentă o directivă ASSUME, specificările asociierilor între regiștri de segment și numele de segmente din program nu sunt urmate și de o încărcare corespunzătoare cu adresele acestor segmente a regiștrilor de segment asociati prin ASSUME. Să luăm ca exemplu o variantă a codului său anterior, în care vom inversa asocierile specificate prin ASSUME, însă nu vom și actualiza corespunzător acțiunile de încărcare ale regiștrilor DS și 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 inițiale pentru DS și ES!

c segment

```
start: mov ax,d1 ;încărcarea registrului DS cu adresa segmentului de date d1
       mov ds,ax ;deși ASSUME a fost modificată, acțiunile de încărcare ale
                  ;registriilor DS și ES nu sunt modificate corespunzător cu
                  ;ASSUME !
       mov ax,d2 ;încărcarea registrului ES cu adresa segmentului de date d2
       mov es,ax
       mov al,a1 ;accesare a1 relativ la ES (adică mov al, ES:a1, deoarece segmentul
                  ;d1 în cadrul căruia apare eticheta a1 a fost asociat prin ASSUME
                  ;cu registrul ES)
```

;numai că ES a fost încărcat mai sus cu adresa de început a segmentului d2 și nu cu adresa
;de început a segmentului d1 unde se află a1, deci în AL se va încărca conținutul locației de
;memorie de deplasament a1 din cadrul segmentului d2 ceea ce va reprezenta o eroare
;logică, deoarece a1 trebuie raportat la segmentul d1 și nu la segmentul d2!!

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

;numai că DS a fost încărcat mai sus cu adresa de început a segmentului d1 și nu cu adresa
;de început a segmentului d2 unde se află a2, deci în AH se va încărca conținutul locației de
;memorie de deplasament a2 din cadrul segmentului d1 ceea ce va reprezenta o eroare
;logică, deoarece a2 trebuie raportat la segmentul d2 și nu la segmentul d1!!

c ends
end start

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

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

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

```
d segment
  vsp : . ;definire vsp
  vss : . ;definire vss
d ends

ns segment
  . . ;rezervare spațiu stivă
ns ends
vs segment stack 'STACK'
  . . ;rezervare spațiu stivă
vs ends
c segment
  assume ds:d, ss:vs

start: mov ax,d
       mov ds,ax ;este activă stiva veche
       mov vsp,sp
       mov ax,ss
       mov vss,ax
       mov ax,ns
       mov ss,ax
       mov sp,200h-2
       mov ax,vss ;este activă stiva nouă
       mov ss,ax
       mov sp,vsp
c ends
end start
```

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

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

END [adresa_start]

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

multe module fiecare modul are în final o directivă END. Însă numai directiva END a modulului ce conține instrucțiunea de la care programul trebuie să-și înceapă execuția va conține specificarea adresei de start. Semnificația acestor reguli este clară și naturală - fiecare program trebuie să aibă neapărat un început, și numai unul, indiferent de numărul modulelor ce îl compun. Eventualele excepții de la aceste reguli pot apărea la nivelul legării cu module scrisă în alte limbi 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 și alocarea spațiului de memorie necesar. Pe scurt deci, *definire date = declarare* (specificarea atributelor) + *alocare* (rezervarea spațiului de memorie necesar).

În cadrul limbajului de asamblare, *definirea unei date* se realizează prin utilizarea directivei 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 și 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 octeți corespunzătoare directivelor și instrucțiunilor întâlnite (reamintim în acest context rolul *contorului de locații* - \$). Ca urmare, asamblorul va genera și octeții ce corespund directivelor de definire a datelor întâlnite realizând în acest fel de fapt acțiunea de alocare a datelor în cadrul segmentului respectiv.

În același timp, pentru datele alocate se pot specifica valorile inițiale. Datele pot fi specificate ca numere, siruri de caractere sau expresii evaluate ca fiind constante. Asamblorul transformă aceste valori constante în octeți, cuvinte sau alte unități de date. Datele codificate sunt scrise în fișierul obiect în momentul asamblării.

Forma generală a unei linii sursă în cazul unei declarații 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 referată data. Acest nume are asociat un tip și 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 inițializa zonele de date rezervate prin declarația respectivă. Dacă apare numai caracterul '?' atunci zona de date corespunzătoare va fi numai rezervată, nu și inițializată.

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

Tip_data este o directive 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 octeți (QWORD - utilizate pentru memorarea constantelor reale)
- DT - date de tip 10 octeți (TWORD - utilizate pentru memorarea constantelor BCD)

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

De exemplu, secvența de mai jos definește și inițializează cinci variabile de memorie:

```
data segment
    varb DB 'd' ;1 octet
    varw DW 101b ;2 octeți
    vard DD 2bfh ;4 octeți
    varq DQ 3070 ;8 octeți (1 quadword)
    vart DT 100 ;10 octeți
data ends
```

După o directive de definire a datelor pot să apară mai multe valori, permitându-se astfel declararea și inițializarea de tablouri. De exemplu, declarația

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

crează un tablou de 5 întregi reprezentați pe cuvinte având valorile respectiv 1,2,3,4,5. Dacă valorile de după directive nu încap pe o singură linie se pot adăuga oricătre linii este necesar, linii ce vor conține numai directive și valorile dorite. Exemplu:

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

Operatorul DUP se folosește pentru definirea unor blocuri de memorie inițializate repetitiv cu o anumită valoare. De exemplu

```
Tabzero DW 100h DUP (0)
rezervă 256 de cuvinte pentru tabloul Tabzero inițializându-le cu 0, iar
```

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

crează un tablou de 80 de octeți inițializați fiecare cu codul ASCII al caracterului 'a'.

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

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

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

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

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

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

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

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

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

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

data ends
```

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

3.3.3. Directivele LABEL, EQU, =

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

nume LABEL tip

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

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

BYTE	NEAR	FAR
WORD	QWORD	PROC
DWORD	TBYTE	UNKNOWN

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

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

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

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

<pre>data segment tempvar LABEL UNKNOWN DB ?,? data ends</pre>	<pre>code segment mov tempvar,ax ;utilizare ca și cuvânt add dl,tempvar ;utilizare ca și octet . . .</pre>
--	--

Reamintim că o altă soluție 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.

Direcția EQU permite atribuirea, în faza de asamblare, unei valori numerice sau sir de caractere unei etichete fără alocarea de spațiu de memorie sau generare de octeți. 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 sir de caractere. Cu alte cuvinte nu se admite redefiniri de etichete decât în cazul echivalării lor inițiale cu siruri 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 echivalente prin directiva EQU cu constantele din limbajele de programare de nivel înalt.

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

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

Direcția = este echivalentă directivei EQU cu excepția următoarelor aspecte:

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

3.3.4. Directiva PROC

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

În limbajul de asamblare 8086 definirea unei subruteine 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 lipsește, 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 și din alte segmente de cod.

Directiva ENDP marchează sfârșitul unei subruteine ce începe cu **PROC**. Sintaxa ei este

<nume_procedură> ENDP

Între cele două directive se vor scrie instrucțiunile și directivele corpului procedurii. Problema subruteinilor este reluată în capitolul 4 în cadrul prezentării instrucțiunilor de apel (CALL) și respectiv de revenire (RET) dintr-o subrutină (4.3.4).

3.3.5. Blocuri repetitive

Un *bloc repetitive* este o construcție prin care i se cere asamblorului să genereze în mod repetat o configurație de octeți. Prezentăm în continuare trei tipuri de blocuri repetitive: **REPT**, **IRP** și **IRPC**.

Un bloc repetitive delimitat de directivele **REPT** și **ENDM** are următoarea sintaxă de definire:

```
REPT contor
      secvență
ENDM
```

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

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

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

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

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

De asemenea, blocurile repetitive pot fi imbricate. Secvența

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

generează de 10 ori secvența specificată.

Directiva **IRP** are sintaxa

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

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

IRP	<i>parametru,<0,1,4,9,16,25></i>	db 0
	db param	db 1
		db 4
ENDM		generează secvența
		db 9
		db 16
		db 25

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

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

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

De exemplu	IRPC	<i>nr,1375</i>
		db nr
	ENDM	

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

3.3.6. Directiva INCLUDE

Directiva **INCLUDE** are sintaxa

```
INCLUDE numefisier
```

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

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

iar fișierul *instr2.asm* conține codul

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

rezultatul asamblării fișierului *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
```

Fișierele incluse pot conține la rândul lor alte directive INCLUDE, și.a.m.d. până la orice nivel, instrucțiunile respective fiind inserate corespunzător pentru crearea în final a unui singur cod sursă.

3.3.7. Macrouri

Un *macro* este un text parametrizat căruia î se atribuie un nume. La fiecare întâlnire a numelui, asamblorul pune în codul sursă textul cu parametrii actualizați. Operația este cunoscută și sub numele de *expandarea* macroului. Se poate face o analogie cu directiva INCLUDE prezentată anterior. Față de fișierele incluse, macrourile prezintă un grad sporit de flexibilitate permitând transmiterea de parametri și existența de etichete locale.

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

```
nume MACRO [parametru [,parametru]...]
    corp instrucțiuni
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 crește consistența exemplelor de mai jos vom utiliza și instrucțiuni care vor fi prezentate abia în următorul capitol. Oricum, în contextul acestei secțiuni nu este esențială semnificația lor precisă. Dorim doar promovarea unei baze de discuție asupra structurii programelor pentru a putea scoate în evidență semnificația elementelor prezentate aici.

Pentru înmulțirea 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 secvența

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

Macrourile pot conține blocuri repetitive. În acest sens putem lua ca exemplu chiar macroul **inmcu4** de mai sus, care se poate scrie 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 definește o etichetă și în program există mai mult de un apel al aceluiași 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      ;și aici apare Etich!

```

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

```

scade MACRO
  LOCAL Etich
  jcxz  Etich
  dec   cx
Etich:
ENDM

```

cele două apeluri consecutive de mai sus fiind transluate în

```

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

```

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

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

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

```

push_reg MACRO rlitera
  push rliterax
ENDM

```

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

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

```

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

```

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

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

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

```

push_reg MACRO      registru
  push registru
ENDM

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

```

macroul ***push_toate_reg*** conține un bloc repetitiv, care la rândul lui conține o invocare a macroului ***push_reg***.

CAPITOLUL 4

INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE

Pentru ca cititorul să poată experimenta instrucțiunile prezentate în acest capitol, prezentăm pentru început forma generală (sau cel puțin 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  
... ;definiții 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 funcției 4ch a întreruperii 21h pentru a provoca încheierea  
               ;execuției programului curent  
code ends  
end start ;precizarea sfârșitului textului sursă (directivea END) împreună cu  
           ;definiția punctului de intrare în program – eticheta de cod "start"
```

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

```
.model small  
.data  
.code  
Start:  
       mov ax,@Data  
       mov ds,ax  
       ...  
       mov ah,4ch  
       int 21h  
end Start ;apelarea funcției 4ch a întreruperii 21h
```

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

nume.ASM

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

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

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

4.1. MANIPULAREA DATELOR

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

4.1.1. Instrucțiuni de transfer al informației

4.1.1.1. Instrucțiuni de transfer de uz general

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

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

MOV *destinație, sursă*

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

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

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

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

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

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

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

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

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

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

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

PUSH *s* și **POP** *d*

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

De exemplu, să presupunem că SP conține inițial valoarea 1000h și să urmărim evoluția conținutului stivei precum și a regiștrilor AX, BX și SP pe parcursul execuției secvenței

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

La început avem situația

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

După mov ax,1 și push ax avem

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

După mov bx,2 și push bx vom avea

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

După execuția instrucțiunii pop ax vom avea

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

iar după execuția instrucțiunii pop bx vom obține configurația

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

Încărcarea unui regisztr de segment se poate face și prin intermediul instrucțiunilor PUSH și POP. De exemplu, secvența de mai sus de încărcare a regisztrului 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.

Instrucțiunea XCHG permite interschimbarea conținutului a doi operanzi de aceeași dimensiune (octet sau cuvânt), cel puțin unul dintre ei trebuind să fie regisztr. Sintaxa ei este

XCHG *operand1, operand2*

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

```
xchg ax,bx
```

schimbă conținutul regisztrilor AX și BX, operație echivalentă cu

<pre>push ax mov ax,bx pop bx</pre>	<pre>push bx mov bx,ax pop ax</pre>
iar	
xchg al,varmem (sau xchg varmem,al)	

schimbă conținutul regisztrului AL cu cel al variabilei octet *varmem*.

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

XLAT [*tabelă_de_translatare*]

tabelă_de_translatare este adresa directă a unui sir de octeți. Instrucțiunea pretinde la intrare adresa fară a tabelii de translatare furnizată sub unul din următoarele două moduri:

- DS:BX (implicit, dacă lipsește operandul)
- regisztr_segment:BX, dacă operandul instrucțiunii XLAT este prezent, regiszrul segment fiind determinat pe baza directivei ASSUME corespunzătoare operandului.

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

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

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

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

```
.data
    TabHexa db '0123456789ABCDEF'
.code
    mov bx, offset TabHexa
    mov al, numar
    xlat TabHexa ;sau doar xlat fără parametru dacă s-a
                   ;specificat ASSUME ds:data
```

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

4.1.1.2. Instrucțiuni de transfer de intrare-iesire

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

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

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

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

copiază un octet din portul 1000 în AX.

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

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

scrie valoarea 63 la portul I/O 500.

4.1.1.3. Instrucțiuni de transfer al adreselor

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

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

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

```
lea ax,v
```

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

```
mov ax, offset v
```

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

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

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

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

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

```
varp dd 12345678h
```

```
lds bx,varp ;se transferă conținutul 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 reținem că instrucțiunea LDS a asigurat doar transferul valorii de tip dublucuvânt din *varp* în DS:BX, acționând în acest sens ca un fel de "instrucțiune *mov* pe 4 octeți" (instrucțiunea *mov* poate avea doar operanzi octet sau cuvânt!). Utilizarea valorii dublucuvânt transferate pe post de adresă să a făcut numai în cadrul instrucțiunii *mov al,[bx]*, care datorită adresării indirekte specificate a provocat interpretarea operandului sursă *[bx]* drept "conținutul de la adresa DS:BX" și nu "conținutul registrului bx", ca și în cazul specificării *mov ax,bx*.

Iată în continuare un exemplu care demonstrează utilitatea instrucțiunilor 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 conține 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 conține 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 instrucțiuni prezentate în cadrul acestei secțiuni, instrucțiunea LEA este de fapt singura ce corespunde cu adevărat titlului secțiunii, fiind într-adevăr o instrucțiune ce transferă o adresă. Este vorba despre adresa near (offsetul) a operandului specificat. Următoarele două instrucțiuni (LDS și LES) sunt însă net diferite de LEA, ele transferând de fapt nu adrese ale unor entități din memorie, ci conținuturi ale unor locații de memorie ce au posibilitatea de a fi interpretate ulterior drept adrese (similar deci ca semantică cu noțiunea de variabile pointer din limbajele de programare de nivel înalt). Ca urmare, în momentul aplicării instrucțiunilor LDS și LES asamblorul va face doar un transfer de dublucuvinte, nefiind sigură interpretarea ulterioară a acestor valori drept adrese! Deși 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 instrucțiuni (LDS și 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 instrucțiuni drept niște "superinstrucțiuni MOV" capabile să opereze direct cu dublucuvinte (instrucțiunea MOV poate lucra doar cu octeți sau cuvinte). De aceea poziția noastră este că LDS și LES ar trebui tratate mai degrabă alături de instrucțiunea MOV în cadrul secțiunii "Instrucțiuni de transfer de uz general". Pe de altă parte, este de înțeles de ce în literatura de specialitate ele sunt tratate în cadrul secțiunii "instrucțiuni de transfer al adreselor": ele sunt într-adevăr instrucțiuni de transfer, permitând interpretarea și 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 instrucțiunilor LDS sau/și LES.

Să reținem deosemenea că instrucțiunile LDS și LES sunt singurele instrucțiuni ale limbajului de asamblare ce acceptă ca ambii operanzi sa fie dublucuvinte și această chiar de o manieră explicită! (continutul <mem> dublucuvânt se transferă în dublucuvântul DS:reg sau respectiv ES:reg).

O să vedem în cadrul secțiunii dedicată operațiilor aritmétice că și instrucțiunile mul, imul, div și idiv acceptă ca un operand să fie dublucuvânt, însă acesta e predefinit (DX:AX) și ca urmare din punct de vedere al programatorului acesta este limitat tot la specificarea unui operand octet sau cuvânt pe post de înmulțitor sau împărțitor!

Am ținut să facem aceste observații deoarece din experiența noastră didactică se remarcă printre studenți o rată foarte înaltă de confuzie asupra mecanismelor implicate de LEA în raport cu LDS și LES. Din cauza tratării lor sub aceeași titulatură de "instrucțiuni de transfer al adreselor" și datorită relativi ușurință de a înțelege efectul instrucțiunii LEA, există tentația de a deduce că LDS și LES sunt similare lui LEA, în sensul că ele nu vor încărca numai deplasamentul operandului sursă (ca LEA) ci (se interpretează greșit!) vor transfera *intreaga adresă (segment:deplasament)* a

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

Revenind și reanalizând exemplul anterior în care am avut

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

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

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

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

4.1.1.4. Instrucțiuni asupra flagurilor

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2. Instrucțiuni de conversie

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

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

Instrucțiunile de conversie prezentate în această secțiune sunt considerate ca fiind de tip **distructiv**, deoarece efectul lor este modificarea conținutului unor registri (mai precis a registrilor AH și DX). Aceste instrucțiuni sunt **CBW** (Convert Byte to Word) și respectiv **CWD** (Convert Word to Doubleword).

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

Instrucțiunea **CBW** convertește 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 conținutul 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, instrucțiunea **CWD** extinde cuvântul cu semn din AX în dublucuvântul cu semn DX:AX. Exemplu:

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

obține valoarea -10000 în DX:AX.

Este evident deci că aceste două instrucțiuni (CBW, CWD) sunt **instrucțiuni de conversie cu semn**. Care sunt atunci instrucțiunile (echivalente lor) de **conversie fără semn**? Răspuns: nu există în cadrul limbajului de asamblare instrucțiuni speciale de conversie fără semn, deoarece extinderea dimensiunii de reprezentare în cazul fără semn înseamnă **intotdeauna simpla completare cu zerouri nesemnificative** (de asemenea **conversie distructiv!**). În acest scop, o simplă instrucțiune MOV realizează scopul dorit! Exemplu:

```
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 instrucțiuni 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!) și ca urmare nu se poate folosi un simplu MOV în acest caz.

Ca urmare, să reținem: **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 instrucțiunilor special prevăzute în acest sens: CBW și CWD.

4.1.3. Impactul reprezentării little-endian asupra accesării datelor.

După cum am văzut în capitolul 1 (secțiunea 1.3.2.3.) arhitecturile de tip 80x86 utilizează modalitatea **little-endian** de reprezentare a datelor. Întrebarea care se pune este cât de conștient trebuie să fie programatorul despre particularitățile de reprezentare ale unor date atunci când elaborează un cod sursă, când și în ce fel trebuie să țină cont acesta de detaliiile de reprezentare little endian ale datelor pe care le accesează?

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octetilor drept octeți și nu drept secvențe de octeți interpretate ca și cuvinte sau dublucuvinte, accesarea de cuvinte ca și cuvinte și nu ca perechi de octeți, accesarea de dublucuvinte ca și dublucuvinte și nu ca secvențe de octeți sau de cuvinte) atunci instrucțiunile limbajului de asamblare vor ține cont în mod automat în cadrul manipulării datelor de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiție programatorul nu trebuie să intervină suplimentar în nici un fel pentru a asigura corectitudinea accesării și 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 octetilor în BX va fi, însă
           ;înversată față de reprezentarea în memorie, deoarece numai reprezentarea în
           ;memorie folosește reprezentarea little-endian! În registri datele sunt
           ;memorate conform reprezentării structurale normale, echivalente unei
           ;reprezentări big endian. Instrucțiunea mov realizează automat în mod
           ;corect acest transfer – cu inversarea corespunzătoare a ordinii octetilor 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 combinația de registri ES:DX valoarea dublucuvânt
           ;1234:5678h, mai precis, în ES se încarcă 1234h iar în DX se încarcă 5678h.
           ;Aceeași observație și aici: față de ordinea octetilor din memorie care este
```

;78h 56h 34h 12h, după acțiunea instrucției **les** ordinea octetelor în cadrul registrilor va fi cea structurală normală, adică 12h 34h 56h 78h.

Dacă însă se dorește accesarea sau interpretarea datelor sub o formă diferită față de modalitatea de definire (ex: octeți ai cuvintelor sau dublucuvintelor, cuvinte ale dublucuvintelor, secvențe de octeți interpretate drept cuvinte sau dublucuvinte) atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să își asume însă întreaga responsabilitate a interpretării și accesării corecte a datelor. Cu alte cuvinte, în astfel de situații programatorul este obligat să conștientizeze particularitățile de reprezentare little-endian (este vorba despre ordinea de plasare a octetelor în memorie) și să utilizeze modalități de accesare a datelor în conformitate cu această ordine. Exemplu:

```
assume cs:code, ds:data

data segment
    a dw 1234h          ;datorită reprezentării little-endian, în memorie octetii sunt
                        ;plasati astfel:
    b dd 11223344h      ;34h 12h 44h 33h 22h 11h
                        ;adresa      a   a+1  b   b+1  b+2  b+3
data ends

code segment

start:
    mov ax, data
    mov ds, ax
```

;să presupunem că dorim transferul primului octet din a în AL. În primul rând trebuie să ne fie clar că la nivelul arhitecturii 80x86 primul octet din structura lui a este 12h, iar primul octet din reprezentarea lui a este 34h (datorită ordinii de plasare little-endian). Ca urmare, trebuie să fie clar ce semnificație acordăm sintagmei "primul octet al lui a": primul octet din structură sau primul octet din reprezentare? Să presupunem că dorim primul octet din structură. Dacă am încerca să facem aceasta prin instrucție

```
    mov al, a            ;syntax error! a este cuvânt, iar AL este octet;
```

vom obține o eroare de sintaxă datorită dimensiunii diferite de reprezentare dintre operanzi instrucției **mov**. Ca urmare, va trebui să utilizăm o instrucție de conversie de tip prin care să selectăm doar 1 octet din cuvântul a. În acest scop folosim operatorul PTR (vezi 3.2.2.7.) prin care specificăm sub ce tip de dată dorim a fi interpretat operandul:

```
    mov al, byte ptr a  ;accesarea lui a drept octet, selectarea octetului de la adresa a și
                        ;transferul 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 găsește la adresa a+1. Ca urmare, instrucția care rezolvă ceea ce ne-am propus este:

```
    mov al, byte ptr a+1 ;accesarea lui a drept octet, efectuarea calculului de adresă a+1,
                        ;selectarea octetului de la adresa a+1 (octetul de valoare 12h) și
                        ;transferul său în registrul AL.
```

Simbolul '+' care apare în cadrul expresiei a+1 reprezintă un **operator** și nu o **instrucție**. Operatorii (vezi capitolul 3) efectuează calcule cu valori constante determinabile la momentul asamblării. Semantica sa nu reprezintă aici 12h+1 = 13h, adică nu înseamnă "conținutul lui a" +1, deoarece "conținutul unei zone de memorie" nu poate fi o valoare determinabilă la momentul asamblării (când zona destinată execuției programului nici măcar nu există!). În schimb, stă din capitolul 3 (secțiunea 3.1) că în limbajul de asamblare, valoarea asociată simbolului ce desemnează o variabilă (etichetă de date) este prin definiție adresa sa (deplasament), valoare determinabilă la momentul asamblării. Ca urmare, expresiei a+1 i se asociază de fapt ca înțeles "valoarea etichetei de date a" +1, reprezentând astfel o aritmetică de adrese.

Dacă s-ar fi dorit transferul în AL a "conținutului lui a" + 1 acest lucru nu putea fi exprimat prin utilizarea unui operator, ci trebuie făcut prin utilizarea instrucției **add** corespunzătoare:

```
    mov al, a            ;"conținutul lui a" se transferă în registrul AL
    add al, 1            ;aici se adună 1 la "conținutul lui a"
```

Pe baza celor discutate până acum și ținând cont de ordinea de mai sus a plasării în memorie a octetelor se pot ușor verifica următoarele rezultate:

```
    mov dx, word ptr b+2      ;dx:=1122h
    mov dx, word ptr a+4      ;dx:=1122h deoarece b+2 = a+4 , în sensul că aceste
                            ;expresii de tip pointer desemnează aceeași adresă și anume
                            ;adresa octetului 22h.
```

```
    mov dx, a+4              ;deoarece a este de tip cuvânt această instrucție este de
                            ;fapt echivalentă cu cea de mai sus, nefind necesară
                            ;utilizarea operatorului de conversie PTR.
```

```
    mov bx, word ptr b        ;bx:=3344h
    mov bx, word ptr a+2      ;bx:=3344h, deoarece ca adresa b = a+2.
```

```
    les cx, dword ptr a       ;es:cx:=3344h:1234h, deoarece dublucuvântul ce începe la
                            ;adresa a este format din octeți 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. OPERAȚII

După cum s-a văzut, transferul de informație joacă un rol esențial în funcționarea microprocesorului. La fel de important însă este să dispunem de mijloace de transformare a acestei informații. Operațiile aritmetice și logice reprezintă tocmai astfel de mijloace.

4.2.1. Operații aritmetice

Facilitățile de calcul aritmetic oferite de instrucțiunile mașină se dovedesc a fi surprinzător de rudimentare. De exemplu, nu se permite lucru simplu în aritmetică numerelor reale! Putem totuși realiza aceasta, dar numai prin intermediul scrierii unor programe specializate sau prin integrarea unui coprocesor matematic specializat în operații aritmetice de mare precizie. De asemenea, nu există instrucțiuni aritmetice sau logice care să poată manipula direct operanzi specificeți explicit reprezentări pe mai mult de 16 biți (sau 32 de biți în cazul microoprocesoarelor >= 80386). Operațiile primitive de care dispune un microprocesor 80x86 sunt adunarea, scăderea, înmulțirea și împărțirea valorilor întregi reprezentate pe 8, 16 sau eventual 32 de biți. Tabelul de pe pagina următoare conține instrucțiunile aritmetice, semnificația lor și efectul asupra flagurilor.

4.2.1.1. Adunarea și scăderea

Aceste operații coincid cu operațiile cunoscute din mulțimea numerelor întregi. Operanzii sunt reprezentăți în cod complementar (vezi 1.5.2.). Datorită proprietăților reprezentării în cod complementar, microprocesorul realizează adunările și scăderile "văzând" doar configurații de biți și nu numere cu semn sau fără. Regulile de efectuare a adunării și scăderii presupun adunarea de configurații binare, fără a fi nevoie de a interpreta operanziile drept cu semn sau fără semn anterior efectuării operației! Deci, la nivelul acestor instrucțiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucțiuni separate pentru adunarea/scăderea cu semn față de adunarea/scăderea fără semn. Adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații! După cum vom vedea acest lucru nu este valabil și pentru înmulțire și împărțire. În cazul acestor operații trebuie să știm apriori dacă operanzii vor fi interpretăți drept cu semn sau fără semn. De exemplu, fie doi operanzi A și B reprezentați fiecare pe câte un octet:

INSTRUCȚIUNI 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 tratați ca întregi fără semn	CF,OF modificări AF,PF,SF,ZF nedefiniți; dacă CF și OF sunt 1, atunci AH (DX) conține valori $\neq 0$
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 tratați ca întregi cu semn	CF,OF modificări AF,PF,SF,ZF nedefiniți; dacă CF și OF sunt 1, atunci AH (DX) conține valori $\neq 0$
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 nedefiniți La obținerea unui cât ce nu începe în AL (AX) se semnalează "depășire"
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 nedefiniți La obținerea unui cât ce nu începe în AL (AX) se semnalează "depășire"

A = 9Ch = 10011100b (= 156 în interpretarea fără semn și -100 în interpretarea cu semn)
B = 4Ah = 01001010b (= 74 atât în interpretarea fără semn cât și în interpretarea cu semn)

Microprocesorul realizează adunarea $C = A + B$ și obține

$$C = E6h = 11100110b (= 230 \text{ în interpretarea fără semn și } -26 \text{ în interpretarea cu semn})$$

Se observă deci că simpla adunare a configurațiilor de biți (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obținut, atât în interpretarea cu semn cât și în cea fără semn.

Instrucțiunea **ADD** are forma

ADD destinație, sursă

Ea adună cei doi operanzi, rezultatul fiind depus în operandul destinație, cu actualizarea corespunzătoare a flagurilor. Cei doi operanzi trebuie să fie reprezentăți ambii pe octet sau ambii pe cuvânt.

Instrucțiunea **SUB** are forma

SUB destinație, sursă

Aceasta scade operandul sursă din operandul destinație, păstrând rezultatul în operandul destinație. De exemplu, atribuirea $E := A + B - C$ (unde A, B, C și E sunt valori reprezentate pe cuvânt) se realizează prin secvența

```
mov ax, A
add ax, B
sub ax, C
mov E, ax
```

În urma efectuării unei operații de adunare sau scădere, 80x86 setează flag-ul C (*carry flag*) cu valoarea 0 (marcând desfășurarea normală a operației) sau 1 (marcând depășirea spațiului de reprezentare a operandului destinație de către valoarea rezultată în cazul unei adunări, respectiv necesitatea "împrumutului" unei unități în cazul scăderii).

Este evident că pentru programarea unei adunări sau scăderi pe mai mult de 16 biți trebuie să ținem cont de valoarea din CF. Este ceea ce fac instrucțiunile **ADC** și **SBB**. **ADC** are același efect ca și ADD numai că adună în plus și valoarea din CF. **SBB** are același efect ca și **SUB**, scăzând însă și "împrumutul" făcut de o eventuală scădere anterioară, împrumut semnalat în CF. Adunarea dublucuvântului din CX:BX la dublucuvântul din DX:AX se face astfel:

```
add ax,bx
adc dx,cx
```

Să luăm acum un exemplu mai complex care să ilustreze necesitatea și utilitatea instrucțiunii ADC. Fie atribuirea $x := x + y$, unde x și y sunt numere fără semn reprezentate în baza 2 pe căte 48 de biți. Convenim reprezentarea astfel încât cifra binară de rang 0 să aibă adresa cea mai mică, iar cifra binară de rang 47 să aibă adresa cea mai mare. Secvența de instrucțiuni ce realizează această atribuire este:

x	dw	1234h, 2345h, 5678h, 0
y	dw	4544h, 4545h, 6766h ; reprezentările hexa ale celor doi operanzi
mov	ax, y	
add	x, ax	; efectuăm 1234h + 4544h
mov	ax, y+2	
adc	x+2,ax	; efectuăm 2345h + 4545h + CF ;(CF - cifra de transport de la adunarea anterioară)
mov	ax, y+4	
adc	x+4,ax	; efectuăm 5678h + 6766h + CF
adc	x+6,0	; necesară pentru adunarea în final a cifrei de ;transport 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 ținând cont de eventuala cifră binară "împrumutată" la prima operație de scădere :

```
sub ax,bx
sbb dx,cx
```

Atragem atenția că pentru corecta funcționare a instrucțiunilor **ADC** și **SBB** trebuie să se aibă în vedere păstrarea valorii din CF între respectivele operații compuse. De exemplu, secvența

```
add ax,bx
sub si,si ; setează CF pe 0
adc dx,cx
```

nu va aduna corect CX:BX la DX:AX deoarece instrucțiunea **SUB** modifică potențial valoarea din CF între cele două operații ADD și ADC.

Datorită frecvenței ridicate a apariției în cadrul programelor a adunărilor cu 1 (incrementare) precum și a scăderilor cu 1 (decrementare) limbajul de asamblare conține instrucțiunile **INC** și respectiv **DEC** cu sintaxa

INC operand	și	DEC operand	echivalente ca efect deci cu
ADD operand,1	și	SUB operand,1	

Avantajele introducerii acestor instrucțiuni rezidă atât în codul compact rezultat (se reprezintă doar pe 1 octet comparativ cu variantele ADD sau SUB care necesită 3) cât și în viteza mult mai mare de execuție.

Instrucțiunea **NEG** are sintaxa

NEG destinație

și are ca efect schimbarea semnului valorii operandului (registru sau variabilă de memorie). De exemplu, în urma execuției secvenței

```
mov ax, 1
neg ax ;conținutul din ax devine -1 (16 cifre 1)
mov bx, ax
neg bx ;conținutul registrului bx devine 1
```

o să avem -1 în AX (în interpretarea cu semn) și 1 în BX. Facem precizarea că valoarea din AX se poate interpreta și ca valoare fără semn, aceasta fiind în acest caz 65535.

4.2.1.2. Înmulțirea și împărțirea

Limbajul de asamblare pune la dispoziția utilizatorilor instrucțiuni de înmulțire și împărțire cu semn și fără semn.

Forma generală a instrucțiunii de înmulțire fără semn este

MUL operand

unde *operand* este un registru general sau o variabilă de memorie. Instrucțiunea MUL realizează înmulțirea fără semn a valorii operandului cu valoarea din AL sau AX, acțiunea instrucțiunii 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 conținând cuvântul superior al produsului.

Deși înmulțirea furnizează rezultatul pe un spațiu dublu decât operanții, ea semnalează dacă produsul nu se poate reprezenta pe un spațiu de aceeași dimensiune cu operanții, punând CF=1 și OF=1. În caz contrar se pune CF=0 și OF=0.

De exemplu, calculul valorii E := A*B, cu operanții reprezentate pe octet, se poate efectua prin secvența:

- mov al, A
- mul B
- mov E, al

Instrucțiunea **mul ax** depune în DX:AX pătratul vechii valori din AX.

Pentru cazul în care se dorește realizarea de înmulțiri cu semn se utilizează instrucțiunea

IMUL operand

care are aceleași reguli de acțiune ca și MUL, cu precizarea că respectă regula semnelor cunoscută din aritmetică. Astfel, secvența

```
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 conținutului inițial din AL (0FEh = 254), care, înmulțit cu 10 ar produce în AX valoarea 254 * 10 = 2540 (09ECh).

Vom urmări în secvență de mai jos un mod de calcul al valorii Z := X * Y, unde X și Y sunt valori fără semn reprezentate respectiv pe 48 și 16 biți. Rezultă în mod natural că pentru reprezentarea lui Z sunt necesari 64 biți. Iată exemplul:

```
mov ax, X ;depune cuvântul cel mai puțin semnificativ al valorii X în AX
mul Y ;înmulțire cu valoarea din Y, rezultatul în DX:AX
mov Z, ax ;pune primele 2 cifre hexa obținute (cele mai puțin 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 înmulții
mul Y ;înmulțire cu valoarea din Y
add ax,bx ;se adună transportul anterior de la înmulțire
adc dx,0 ;și se ține cont de eventualul transport în urma adunării
mov Z+2,ax ;se actualizează rezultatul cu următoarele cifre
mov bx,dx ;retine transportul în BX

mov ax, X+4 ;preia cuvântul cel mai semnificativ al sursei
mul Y ;înmulțește corespunzător
add ax,bx ;adună transportul înmulțirii
adc dx,0 ;și ține cont de transport adunare
mov Z+4,ax ;depune valoarea obținută în AX ca cel de-al treilea cuvânt al rezultatului
mov Z+6,dx ;și pune cuvântul superior al ultimului rezultat obținut pe pozițiile ;corespunzătoare celor mai semnificative 2 cifre hexa ale rezultatului final
```

Împărțirea fără semn a două numere întregi se face cu ajutorul instrucțiunii DIV, care are forma

DIV operand

unde *operand* poate fi un registru de memorie sau o variabilă de memorie, el reprezentând împărțitorul. Deîmpărțitorul este determinat în funcție de dimensiunea de reprezentare a operandului împărțitor, astfel:

* dacă împărțitorul este octet, atunci deîmpărțitul este conținutul registrului AX. Câștul va fi depus în AL, iar restul în AH.

* dacă împărțitorul este reprezentat pe cuvânt, atunci deîmpărțitul este DX:AX. Câștul va fi depus în AX, iar restul în DX.

De exemplu,

```
mov ax,51
mov dl,10
div dl
```

realizează împărțirea conținutului lui AX la conținutul lui DL, memorându-se câștul 5 în AL și restul 1 în AH, iar

```
mov ax,2
mov dx,1
mov bx,10h
div bx
```

împarte 10002h din DX:AX la 10h din BX, memorând câștul 1000h în AX și restul 2 în DX.

Să observăm că, dacă n este lungimea de reprezentare a deîmpărțitului, câștul se impune a fi reprezentat pe n/2 biți. Dacă rezultatul obținut va necesita pentru reprezentare o lungime mai mare, atunci microprocesorul va semnaliza depășire prin generarea unei intreruperi 0 (împărțire prin 0). De exemplu, următoarea secvență generează o intrerupere 0 (vezi capitolul 5):

```
mov ax,0ffffh
mov bl,1
div bl
```

Pentru împărțirea cu semn disponem de instrucțiunea

IDIV *operand*

cu aceleași observații și reguli de funcționare ca și DIV, înținând cont în plus de semnul operanzilor. Spre deosebire de teorema fundamentală a împărțirii din aritmetică, aici avem $D = C * I + R$, unde:

- semnul lui C este dat de regula semnelor D/I
- semnul lui R coincide cu semnul lui D
- $|R| < |I|$

Astfel, în urma execuției secvenței

data segment	
Divizor	DW 100
code segment	

mov ax,-667	
cwd	;extinde numărul în DX:AX
idiv Divizor	

se va depune -6 în AX și -67 în DX.

4.2.1.3. Exemple și exerciții propuse.

a).

.....
mov ah,0
mov al, -5 ;echivalent cu **mov al, 11111011b** deci echivalent și cu **mov al, 251**
;echivalent și cu **mov al, 0fbh** în hexazecimal; f = 1111b și b=1011b

;ansamblul celor 2 instrucțiuni de mai sus este echivalent cu **mov ax,251** (echivalent în
;binar cu **mov ax, 0000000011111011b** și în hexazecimal cu **mov ax, 00fbh**) însă nu și
;cu **mov ax, -5** (echivalent în binar cu **mov ax, 111111111111011b** și în hexazecimal cu
;**mov ax, 0fffbh**) - diferența constă în completarea conținutului lui AH cu 8 cifre binare 0
;în cazul lui 251 – interpretare fără semn - și respectiv cu 8 cifre binare 1 în cazul lui -5
;adică în interpretarea cu semn)

mov bx,10

imul bx ;dx:ax := ax*bx = 251 * 10 = 2510 = 09CEh (în AX) și DX:=0

(deși aici prin **imul** e forțată interpretarea cu semn pentru AX, deoarece AX =
0000000011111011b , bitul de semn fiind 0, rezultă că AX = 251 în ambele interpretări)

mul bx ;idem – rezultatele sunt identice datorită observației de mai sus

mov cl, -100 (= 9ch = 10011100b = 156 în interpretarea fără semn!)

idiv cl ;AX=2510; AL (câștul):= AX idiv (-100) = -25 (=e7); AH (restul):=10 (0ah)
;conținutul lui AX este acum AX=0ae7h

imul cl ;AX:=AL*CL = (-25)*(-100) = 2500 (=09c4h)

add ax,10 ;AX:=AX+10 = 2500+10 = 2510 - refacerea valorii inițiale din AX:=2510;

div cl ;AX=2510; AL (câștul) := AX div 156 = 16 (=10h); AH (restul) := 14 (0eh)
;conținutul lui AX este acum AX=0e10h

Reluați discuția, analizați și justificați rezultatele furnizate în situația în care ultimele 5 instrucțiuni de mai sus devin (CL se înlocuiește cu CX):

```
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 obține acum ca rezultat? Comparați rezultatul furnizat cu cel obținut prin aplicarea operației **mul bx** în loc de **imul bx**.

c).
mov al, 251 ; 251 = 0fbh = -5 - deci instrucțione echivalentă cu **mov al, -5**

(suma valorilor absolute ale reprezentărilor cu semn și 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 și 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 = fffh ; restul în AH = 11 = 0bh
;deci conținutul lui AX este acum AX = 0bffh
```

Ce se întâmplă dacă **"imul bl"** este înlocuită cu instrucționea **"mul bl"**? Analizați și explicați comparativ rezultatele furnizate.

Ce observați că se întâmplă dacă **"div cl"** este înlocuită cu instrucționea **"idiv cl"**?

Încercați să (vă) explicați potențiala cauză a unui astfel de comportament (vezi și 5.2.2 – INT 0) prin efectuarea “pe hârtie” a operației **“idiv cl”** și analizând apoi încadrarea valorilor astfel obținute (cât și rest) în dimensiunile de reprezentare puse la dispoziție de o operație **div** sau **idiv**.

Dar dacă apoi **“div cl”** este înlocuită întâi cu **“div ex”** iar apoi cu **“idiv ex”**? Justificați și explicați rezultatele obținute. De ce în acest caz nu se mai manifestă situația apărută în cazul **“idiv cl”**?

4.2.2. Operări logice pe biți

Limbajul de asamblare dispune de un set de patru instrucții pentru realizarea de operații logice la nivel de bit: **AND, OR, XOR și NOT**.

AND d,s	<d> <-> <d> și <s>	CF,OF,PF,SF,ZF modificați; AF - nedefinit
OR d,s	<d> <-> <d> sau <s>	CF,OF,PF,SF,ZF modificați; AF - nedefinit
XOR d,s	<d> <-> <d> sau exclusiv <s>	CF,OF,PF,SF,ZF modificați; AF - nedefinit
NOT s	<s> <-> <s> negat complement față de 1	

Pentru doi biți, unul din sursă, altul din destinație, instrucțiunile logice produc noul bit destinație 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 instrucțiunile logice realizează aceste operații asupra perechilor corespunzătoare de biți. Operanții sursă și destinație trebuie să aibă ambele aceeași 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

Instrucțiunea AND este indicată pentru izolarea unui anumit bit sau pentru forțarea anumitor biți la valoarea 0. Astfel, dacă dintr-o configurație pe 8 biți

$$a = \text{x x x x x x x b}$$

dorim izolarea bitului i ($0 \leq i \leq 7$), vom crea expresia

$$\text{AND } a, 2^i$$

iar dacă dorim forțarea anumitor biți din a la valoarea 0, să zicem de exemplu biții 0, 2 și 3 (restul rămânând neschimbați) vom crea expresia

$$\text{AND } a, 11110010b \quad (= 242)$$

(dacă a conține inițial valoarea 01010110b de exemplu, atunci după execuția instrucțiunii de mai sus, în a se va afla valoarea 01010010b).

Instrucțiunea OR poate fi folosită printre altele pentru forțarea anumitor biți la valoarea 1. De exemplu, presupunând că variabila a conține inițial aceeași valoare de mai sus (01010110b), dacă dorim forțarea biților 0, 2 și 3 ai acestei valori la 1 vom folosi instrucțiunea OR astfel:

$$\text{OR } a, 00001101b \quad (=13)$$

Valoarea inițială din a este distrusă, în locul ei depunându-se noua valoare rezultată, adică 01011111b.

Instrucțiunea XOR este indicată pentru schimbarea valorii unor biți din 0 în 1 sau din 1 în 0. De exemplu, dacă dorim ca biții 4-7 din AL să-și schimbe valorile iar ceilalți să rămână neschimbați, atunci aceasta se poate realiza prin

$$\text{XOR } al, 11110000b$$

Dacă presupunem că inițial în AL am avut valoarea 01010101b, atunci după execuția instrucțiunii de mai sus în AL se va afla valoarea 10100101b (= A5h). Se observă că biții 1 comută valorile inițiale iar biții 0 le lasă neschimbate. De asemenea, instrucțiunea XOR poate fi utilizată pentru zerorizarea unui registru:

$$\text{XOR } ax, ax$$

Instrucțiunea NOT modifică biții operandului în valoarea lor complementară (0 în 1 și 1 în 0). Efectul este echivalent cu a efectua un XOR cu operandul sursă 0FFh. Exemplu:

```
mov bl, 10110001b
not bl           ;se modifică în 01001110b
xor bl, 0ffh     ;se revine la 10110001b
```

4.2.3. Deplasări și rotiri de biți

Micropresesoarele 80x86 dispun de o serie de instrucțiuni cu ajutorul cărora pot deplasa sau roti biții din cadrul unui operand în memorie sau registru.

În cadrul octetelor sau cuvintelor biții pot fi deplasati aritmetic sau logic sau rotiți la dreapta sau la stânga cu una sau mai multe poziții. Numărul de deplasări este fie 1, fie o valoare cuprinsă între 1 și 255, valoare specificată în registrul CL.

Instrucțiunile de *deplasare* de biți se clasifică în:

- Instrucțiuni de deplasare logică
 - stânga - **SHL**
 - dreapta - **SHR**
- Instrucțiuni de deplasare aritmetică
 - stânga - **SAL**
 - dreapta - **SAR**

Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:

- Instrucțiuni de rotire fără carry
 - stânga - **ROL**
 - dreapta - **ROR**
- Instrucțiuni de rotire cu carry
 - stânga - **RCL**
 - dreapta - **RCR**

Pentru a defini deplasările și rotirile să considerăm că și configurație inițială un octet $X = abcdefgh$, unde $a-h$ sunt cifre binare, h este cifra binară de rang 0, a este cifra binară de rang 7, iar k este valoarea existentă în CF ($CF=k$). Atunci,

```
SHL X,1 ;rezultă X = bcdefgh0 și CF = a
SHR X,1 ;rezultă X = 0abcdefg și CF = h
SAL X,1 ; identic cu SHL
SAR X,1 ;rezultă X = aabcdefg și CF = h
ROL X,1 ;rezultă X = bcdefgha și CF = a
ROR X,1 ;rezultă X = habcdefg și CF = h
RCL X,1 ;rezultă X = bcdefghk și CF = a
RCR X,1 ;rezultă X = kababcdefg și CF = h
```

Deplasările și rotirile pe 16 biți se fac analog. Deplasările și rotirile cu o valoare specificată în CL se fac prin aplicarea repetată a regulilor de mai sus.

Important! Se observă că, în toate cazurile, bitul ce părăsește configurația trece în CF.

INSTRUCȚIUNILE DE DEPLASARE ȘI ROTIRE DE BIȚI

SHL s,1	deplasare logică (aritmetică) stânga cu o poziție. CF conține bitul cel mai semnificativ deplasat. La dreapta se introduc zerouri.	OF,SF,ZF,PF,CF modificați AF - nedefinit
SHL s,cl	deplasare logică (aritmetică) stânga cu cl poziții. CF conține ultimul bit deplasat. La dreapta se introduc zerouri.	OF,SF,ZF,PF,CF modificați; AF - nedefinit Dacă CL≠1 , OF este nedefinit
SHR s,1	deplasare logică dreapta cu o poziție. La stânga se introduc zerouri. Bitul cel mai puțin semnificativ este deplasat în CF.	OF,SF,ZF,PF,CF modificați AF - nedefinit
SHR s,cl	deplasare logică dreapta cu cl poziții. La stânga se introduc zerouri. CF va conține ultimul bit deplasat.	OF,SF,ZF,PF,CF modificați AF – nedefinit; Dacă CL≠1, OF este nedefinit
SAR s,1	deplasare aritmetică dreapta cu o poziție. La stânga se extinde bitul de semn. Bitul cel mai puțin semnificativ este deplasat în CF.	OF,SF,ZF,PF,CF modificați AF - nedefinit
SAR s,cl	deplasare aritmetică dreapta cu cl poziții. La stânga se extinde bitul de semn. CF va conține ultimul bit deplasat.	OF,SF,ZF,PF,CF modificați AF – nedefinit ; Dacă CL≠1, OF este nedefinit
ROL s,1	rotire stânga cu o poziție. CF va conține bitul (cel mai semnificativ) rotit.	OF,CF
ROL s,cl	rotire stânga cu cl poziții. CF va conține ultimul bit rotit.	OF,CF ; Dacă CL≠1, OF este nedefinit
ROR s,1	rotire dreapta cu o poziție. CF va conține bitul cel mai puțin semnificativ al lui s.	OF,CF
ROR s,cl	rotire dreapta cu cl poziții. CF va conține ultimul bit rotit.	OF,CF ; Dacă CL≠1, OF este nedefinit
RCL s,1	rotire stânga cu carry cu o poziție.	OF,CF
RCL s,cl	rotire stânga cu carry cu cl poziții.	OF,CF ; Dacă CL≠1, OF este nedefinit
RCR s,1	rotire dreapta cu carry cu o poziție.	OF,CF
RCR s,cl	rotire dreapta cu carry cu cl poziții.	OF,CF ; Dacă CL≠1, OF este nedefinit

Cap.4. Instrucțiuni ale limbajului de asamblare.

Deplasările logice pot fi folosite pentru izolarea unor biți în interiorul octetelor (cuvintelor), iar deplasările aritmetice se pot utiliza pentru înmulțirea sau împărțirea numerelor binare cu puteri ale lui 2.

Instrucțiunile de deplasare afectează flagurile astfel:

- Flag-ul AF este întotdeauna nedefinit în urma unei operații de deplasare.
- Indicatorii PF, SF și ZF sunt actualizați ca și în cazul instrucțiunilor logice pe biți.
- Indicatorul CF conține întotdeauna valoarea ultimului bit deplasat din cadrul operandului.
- Conținutul indicatorului OF este întotdeauna nedefinit în cazul unor operații de deplasare a mai multor biți. 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 operația de deplasare.

Instrucțiunile de rotire afectează numai flagurile CF și OF. CF va conține întotdeauna valoarea ultimului bit rotit. Pentru operațiile de rotire cu mai mulți biți valoarea flagului OF este nedefinită. La rotirea unui singur bit OF este setat la valoarea 1 dacă bitul cel mai semnificativ își schimbă valoarea (schimbare de semn).

Să exemplificăm modul de acțiune al acestor instrucțiuni în câteva situații concrete.

De exemplu, dacă AL conține valoarea 10010110b (=96h = 150 în interpretarea fără semn, sau -6Ah = -106 în interpretarea cu semn), atunci după execuția instrucțiunii **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 poziția rămasă liberă în dreapta se introduce 0. Acțiunea instrucțiunii SHL în acest caz poate fi reprezentată schematic conform desenului din figura 4.1.

O utilizare frecventă a acestei instrucțiuni apare atunci când se dorește multiplicarea rapidă a operandului cu puteri ale lui 2. Deplasarea spre stânga obținută ca efect al instrucțiunii SHL este de fapt o înmulțire 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 conținutului 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** (deși ultimele versiuni de TASM de exemplu permit acum acest lucru!).

În schimb, se permite utilizarea registrului CL pentru a indica numărul de poziții cu care se dorește deplasarea, ceea ce face ca secvența de mai sus să fie echivalentă cu

```
mov cl,4
shl dx,cl
```

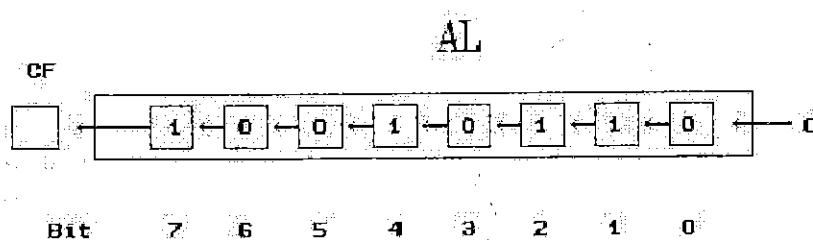


Fig. 4.1. Instrucțiunea SHL

Subliniem că variantele de multiplicare cu puteri ale lui 2 folosind SHL sunt mult mai rapide decât variantele echivalente folosind instrucțiunea MUL.

În acest sens, să urmărim secvența următoare care realizează înmulțirea cu 10 a valorii octetului var, fără a folosi instrucțiuni MUL sau IMUL:

```
xor ah,ah
mov cl,3
mov al,var
shl ax,cl      ;înmulțire cu opt
add al,var
adc ah,0        ;al = var * 9
add al,var
adc ah,0        ;al = var * 10
```

În ceea ce privește instrucțiunile de deplasare spre dreapta, presupunând că AL conține valoarea 10010110b (= 96h = -106), shr al,1 produce 01001011b (= 04Bh = 75), iar sar al,1 rezultă în 11001011b (= 0CBh = -53), având deci ca efect păstrarea semnului operandului. Acest lucru face ca instrucțiunea SAR să fie indicată pentru efectuarea împărțirilor cu semn la puteri ale lui 2. Spre exemplu,

```
mov bx,-4
sar bx,1
```

are ca rezultat memorarea valorii -2 în BX.

Figura 4.2. arată acțiunea instrucțiunii ROR AL,1 asupra valorii 10010110b. Rezultatul este valoarea 01001011b, în CF depunându-se valoarea inițială a ultimului bit, adică 0.

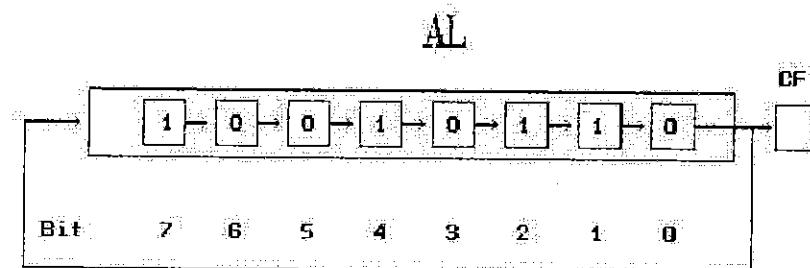


Fig. 4.2. Instrucțiunea ROR

Instrucțiunile ROR și ROL devin utile în probleme privind realinierea bițiilor în cadrul unui octet sau cuvânt. De exemplu, secvența

```
mov si,49f1h
mov cl,4
ror si,cl
```

face ca în SI să se afle în final 149Fh, mutând biții 0-3 în biții 12-15, biții 4-7 în biții 0-3, și.a.m.d.

Figura 4.3 arată rotirea spre dreapta a valorii 10010110b (= 96h = 150) din AL, participând și CF (care inițial conține valoarea 1). Instrucțiunea rcr al,1 produce rezultatul 11001011b (= 0CBh = 203) care este depus în AL. CF va conține valoarea de dinainte de rotire a bitului cel mai puțin semnificativ, adică 0.

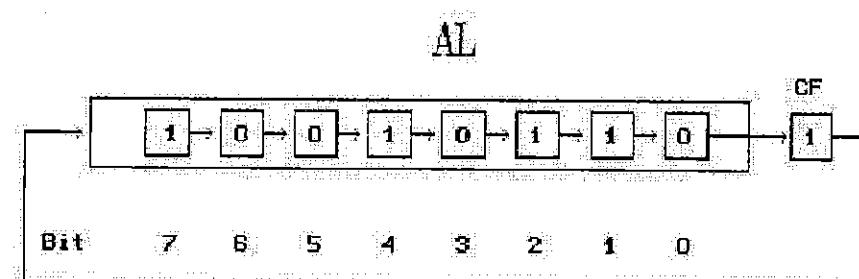


Fig. 4.3. Instrucțiunea RCR

Instrucțiunile RCR și RCL sunt utile pentru realizarea de deplasări de biți implicând operanzi reprezentați pe mai multe cuvinte. De exemplu, secvența 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 instrucțiuni de rotire (cu sau fără carry) cu câte o poziție este mai rapid și necesită mai puțină memorie decât o singură instrucție de rotire ce are $cu_căt = n$.

Precizare. În cazul utilizării de directive ce permit lucru cu instrucțiunile specifice adăugate, microprocesoarele 80x86 cu $x \geq 2$ admit în TASM pentru $cu_căt$ și o valoare constantă diferită de 1 (pentru orice instrucție de deplasare sau rotire). Totuși, există restricția ca această valoare să nu depășească 31.

4.3. RAMIFICĂRI, SALTURI, CICLURI

Un program scris într-un limbaj de programare se execută secvențial, adică instrucțiunile programului se execută una după alta în ordinea în care ele sunt scrisе. Apare în mod natural necesitatea "ruperii" acestei ordini, pentru a putea soluționa anumite cerințe specificate de problema de rezolvat. În limbajele de programare de nivel înalt cunoscute (Pascal, C, FORTRAN etc.) există în acest scop așa numitele *structuri de control* (if, goto, for, while etc.) care determină ordinea de execuție a instrucțiunilor unui program.

O proprietate de bază pe care trebuie să 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 (conditionat sau nu) la o instrucție, alta decât cea care urmează secvențial celei curente. Microprocesoarele 80x86 dispun și ele de instrucțiuni specializate care realizează transferuri ale controlului (salturi), prevăzând și instrucții care facilitează prelucrarea repetată a unui bloc de instrucții (cicluri).

În cursul execuției unui program, adresa următoarei instrucții de executat este indicată întotdeauna de CS:IP. Deci, instrucțiunile care modifică această adresă sunt de fapt instrucții de salt. La o execuție secvențială CS și IP sunt modificate automat de componenta BIU a microprocesorului.

4.3.1. Saltul necondiționat

În această categorie intră instrucțiunile JMP (echivalentul instrucției GOTO din alte limbi), CALL (apelul de procedură înseamnă transferul controlului din punctul apelului la prima instrucție din procedura apelată) și RET (transfer control la prima instrucție executabilă de după CALL).

JMP operand	Salt necondiționat la adresa determinată de operand	-
CALL operand	Transferă controlul procedurii determinată de operand	-
RET [n]	Transferă controlul instrucției de după CALL	-

4.3.1.1. Instrucție JMP

Instrucție de salt necondiționat JMP are sintaxa

JMP operand

unde *operand* este o etichetă, un registru sau o variabilă de memorie ce conține o adresă. Efectul ei este transferul necondiționat al controlului la instrucție ce urmează etichetei, la adresa conținută în registru, respectiv la adresa conținută în variabila de memorie. De exemplu, după execuția secvenței

```
mov ax,1
jmp AdunaDoi
AdunaUnu: inc ax
            jmp urmare
AdunaDoi: add ax,2
urmare:    .
```

registru AX va conține valoarea 3. Instrucțiunile inc și jmp dintre etichetele AdunaUnu și AdunaDoi nu se vor executa, decât dacă se va face salt la AdunaUnu de altundeva din program.

În mod normal instrucție JMP efectuează un salt NEAR (salt în cadrul aceluiași segment). Pentru a reduce codul generat, în cazul în care destinația saltului nu este mai departe de 127 octeți, există posibilitatea așa numitului "salt scurt". Un salt scurt se specifică sub forma

jmp SHORT PTR eticheta

Operatorul SHORT impune folosirea unei adrese pe 8 biți, adresă "scurtă", economisindu-se astfel 1 octet în reprezentarea instrucției JMP. Acest operator își 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 "întâia").

După cum am menționat, saltul poate fi făcut și la o adresă memorată într-un registru sau într-o variabilă de memorie. Exemple:

```
(1)    mov ax, OFFSET etich  
          jmp ax      ;operand registru  
etich: . . .
```

Să remarcăm faptul că esențial pentru exemplele date este prezentarea modalității în care offset-ul (deplasamentul) adresei destinație este permis a fi încărcat în operandul instrucțiunii JMP: în cazul (1) e vorba de o încărcare explicită prin mov a unui registru, în cazul (2) este vorba despre declararea cu inițializare a variabilei de tip cuvânt Salt, în cadrul căreia se realizează inițializarea variabilei *Salt* cu offset-ul etichetei *Dest* (1 cuvânt de memorie = 16 biți = 2 octeți = dimensiunea de reprezentare a deplasamentului). Dacă în cazul (1) dorim înlocuirea operandului destinație registru cu un operand destinație variabilă de memorie, o soluție posibilă este:

(1') b dw ?
...
mov b, offset etich
jmp b : salt NEAR – operand variabilă de memorie

Instrucția JMP poate fi folosită de asemenea pentru saltul într-un alt segment de cod (salt FAR - far jump). Pentru aceasta este necesară determinarea unei adrese de forma *segment:offset*.

Saltul FAR se poate realiza în patru moduri:

a). declarând eticheta destinație ca etichetă far prin directiva LABEL (vezi 3.3.3). De exemplu, instrucțiunea JMP de mai jos realizează un astfel de salt:

```
Cseg1 SEGMENT
    ASSUME cs:Cseg1

Dest LABEL FAR
    .
    .

Cseg1 ENDS
    .
    .

Cseg2 SEGMENT
    ASSUME cs:Cseg2

    jmp Dest ;salt FAR la adresa Dest în segmentul Cseg1

Cseg2 ENDS
    .
    .
```

b). specificând direct în instrucțiunea JMP tipul FAR PTR pentru eticheta destinație, sub forma: JMP FAR PTR *eticheta*. Acest mod de specificare al operandului destinație a fost utilizat în exemplele din 3.3.1.2 - directiva ASSUME și gestiunea segmentelor. Operatorul PTR a fost prezentat în 3.2.2.7.

c). prefixând explicit cu un registru de segment o adresă NEAR specificată indirect (vezi 3.2.1.4 – operanzi cu adresare indirectă), adică furnizarea ca operand al instrucțiunii JMP a unei expresii de forma: *reg segment: specificare offset*. Exemplu: jmp es:[bx+d1].

d). specificând ca operand o variabilă dublucuvânt care contine adresa far a destinației;

```
data segment
      Salt    DD Dest ;Salt:= segm:offset (Dest)
      .
      .

cod  segment
      .
      .
      jmp   Salt          ;salt FAR la eticheta Dest

code1 segment
      .
      .
      .

Dest : . . .
```

secvență echivalentă ca efect cu

```

data segment
    Salt DD ?

cod segment
    lea ax, Dest      ;ax:=offset(Dest)

    mov word ptr Salt, ax ;initializare cuvânt inferior al
                          ;variabilei dublucuvânt Salt cu offset(Dest)
                          ;- se ține cont de reprezentarea little-endian

    mov ax, cod

    mov word ptr Salt+2, ax ;initializare cuvânt superior al
                          ;variabilei dublucuvânt Salt (Salt+2 - 
                          ;aritmetică de pointeri) cu seg(Dest)
                          ;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 evidență deosebirile dintre un transfer *direct* și 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 conținutul 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
```

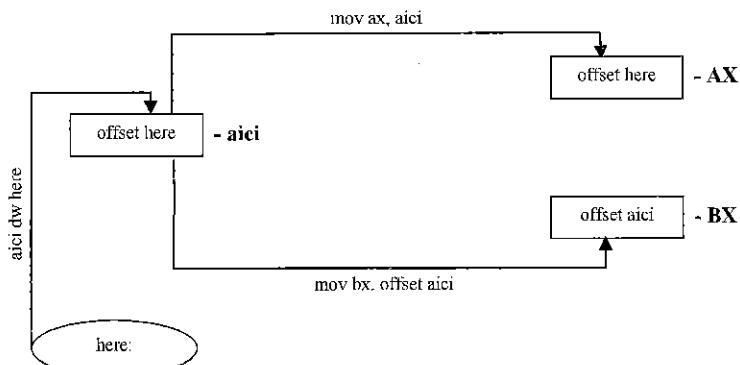


Fig. 4.4. Inițializarea variabilei `aici` și a regiștrilor AX și BX.

```
jmp aici      ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui
                ;here), deci instrucțiune echivalentă cu jmp here
jmp here      ;salt la adresa lui here (sau, echivalent, salt la eticheta here)
jmp ax        ;salt la adresa conținută în AX (adresare regisztr în mod direct), adică la here
jmp [bx]       ;salt la adresa conținută în locația de memorie a cărei adresă este conținută în
                ;BX (adresare regisztr în mod indirect - singura situație de apel indirect din
                ;acest exemplu)
```

în BX se află adresa variabilei `aici`, deci se accesează conținutul acestei variabile. În această locație de memorie se găsește offset-ul etichetei `here`, deci se va efectua saltul la adresa `here` - ca urmare, ultimele 4 instrucțiuni sunt toate echivalente cu `jmp here`

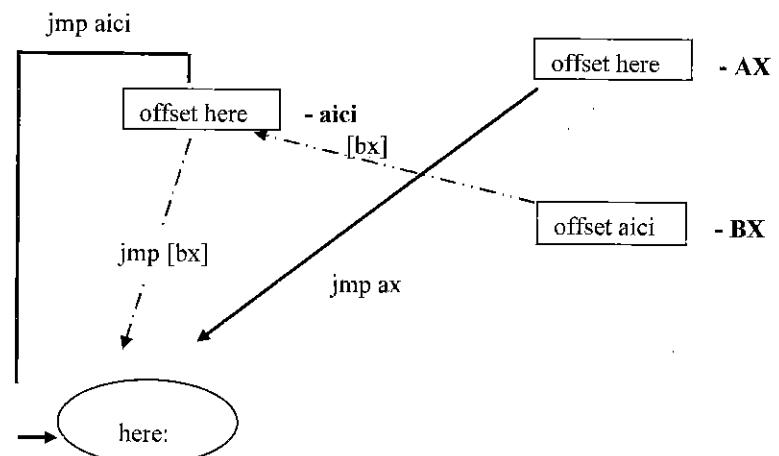


Fig. 4.5. Modalități alternative de efectuare a salturilor la eticheta `here`.

`jmp [ax]` ;eroare de sintaxă: “Illegal indexing mode”

Justificare: orice referire de tipul “[reg]” înseamnă interpretarea conținutului regisztrului specificat drept adresă de memorie al cărei conținut va fi accesat. Însă conform 2.6.7. și 3.2.1.4. în formula de specificare a unui deplasament (offset) au voie să apară doar regisztri BX sau BP (ca regisztri de bază) și SI sau DI (ca regisztri de index). Deci AX nu poate fi parte a unei astfel de formule de calcul și ca urmare acest fapt este semnalat ca eroare de sintaxă!

`jmp bx` ;salt (near) la adresa conținută în BX (adresare regisztr în mod direct)

în BX avem ca valoare offset-ul lui `aici` în cadrul segmentului `data` (vezi mai sus cum a fost încărcat BX) – deși corectă sintactic, o astfel de instrucțiune 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 destinație a salturilor

Acest exemplu ilustrează diferența dintre operanze din memorie care reprezintă adrese și operanze din memorie care reprezintă valoarea de la o adresă (conținutul 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 situațiile în care a apărut, acest nume a desemnat conținutul de la acea locație de memorie (eticheta de date `aici` ar fi fost interpretată ca 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 însăși.

4.3.2. Instrucțiuni de salt condiționat

O caracteristică necesară pe care trebuie să o prezinte orice limbaj de programare este posibilitatea de a lua decizii, ceea ce în plan concret revine la existența unor instrucțiuni de salt condiționat. Elementele care condiționează acest tip de salt sunt valorile flagurilor și conținutul registrului CX.

4.3.2.1. Comparări între operanzi

CMP <i>d,s</i>	comparare valori operanzi (nu modifică operanzi) (execuție fictivă <i>d - s</i>)	OF,SF,ZF,AF,PF și CF
TEST <i>d,s</i>	execuție fictivă <i>d AND s</i>	OF = 0, CF = 0 SF,ZF,PF - modificări AF - nef definite

Operanții instrucțiunii **CMP** pot fi de dimensiune octet sau cuvânt, fiind supuși acelașii restricții de asociere ca și în cazul instrucțiunii **MOV**.

CMP scade valoarea operandului sursă din operandul destinație, dar spre deosebire de instrucțiunea **SUB**, rezultatul nu este reținut, el neafectând nici una din valorile inițiale ale operanților. Efectul acestei instrucțiuni constă numai în modificarea valorii unor flaguri. Instrucțiunea **CMP** este cel mai des folosită în combinație cu instrucțiuni de salt condiționat.

Efectele execuției instrucțiunii **CMP** pot fi utilizate de exemplu atunci când dorim să facem comparații matematice. Acestea se pot face *cu semn* sau *fără semn*, în funcție de configurațiile de biți furnizate de către programator ca operanți.

Instrucțiunile de salt condiționat se folosesc de obicei în combinație cu instrucțiuni de comparare. De aceea, semnificațiile instrucțiunilor de salt rezultă din semnificația operanților unei instrucțiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucțiune **CMP** este de multe ori necesară determinarea relației de ordine dintre două valori. De exemplu, se pune întrebarea: numărul 11111111b (= FFh = 255 = -1) este mai mare decât 00000000b (= 0h = 0)? Răspunsul poate fi și da și nu! Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 și este evident mai mare decât 0. Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 și este mai mic decât 0.

Este valabil și aici ceea ce am subliniat la prezentarea instrucțiunilor de adunare și scădere: interpretarea operanților și a rezultatului comparației cu sau fără semn este la latitudinea programatorului! Cum se poate face concret însă această distincție și cum se poate ea provoca *la nivelul unui program?*

Instrucțiunea **CMP** nu face distincție între cele două situații, deoarece astăzi după cum am precizat și în 4.2.1.1. adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații

Cap.4. Instrucțiuni ale limbajului de asamblare.

binare) indiferent de semnul (interpretarea) acestor configurații. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn **operanții** scăderii fictive *d-s*, ci **rezultatul** final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine diverselor instrucțiuni de salt condiționat prezentate în secțiunea următoare (4.3.2.2) care vor prezenta categorii diferite de instrucțiuni pentru cele două tipuri posibile de interpretări.

Din semnificația flagurilor rezultă că pentru determinarea relației de ordine între operanți, după execuția instrucțiunii **CMP** sunt semnificative doar OF, SF, ZF și CF. Această problemă este detaliată în secțiunea următoare, unde vom vedea că instrucțiunile de salt condiționat diferă tocmai prin condiția asupra flag-urilor care este verificată de către fiecare instrucțiune în parte.

Instrucțiunea **TEST** este utilă pentru examinarea stării anumitor biți. În exemplul de mai jos se transferă controlul la eticheta **Acolo** dacă biții 1 și 5 ai registrului AL sunt 0. Starea celorlalți biți este ignorată:

test	al, 00100010b	;mascare prin AND
jz	Acolo	;salt dacă s-a obținut 0
Nuezero: . . .		
Acolo: . . .		

4.3.2.2. Salturi conditionate de flaguri

În tabelul 4.1. prezentăm instrucțiunile de salt condiționat împreună cu semnificația lor și cu precizarea valorilor flagurilor în urma căror se execută salturile respective. Precizăm că pentru toate instrucțiunile de salt sintaxa este aceeași, și anume

<instrucțiune_de_salt> etichetă

Semnificația instrucțiunilor de salt condiționat este dată sub forma "*salt dacă operand1 <> relație> față de operand2*" (unde cei doi operanți sunt obiectul unei instrucțiuni anterioare **CMP** sau **SUB**) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă și din condițiile ce trebuie verificate, instrucțiunile ce se află într-o aceeași linie a tabelului sunt echivalente.

Când se compară două numere *cu semn* se folosesc termenii "less than" (mai mic decât) și "greater than" (mai mare decât), iar când se compară două numere *fără semn* se folosesc termenii "below" (inferior, sub) și respectiv "above" (superior, deasupra, peste).

Instrucțiunile de salt condiționat efectuează întotdeauna salturi "scurte", adică adresa de destinație nu poate fi la distanță mai mare de 127 octeți față de instrucțiunea curentă. De exemplu, în urma asamblării secvenței

MNEMONICA	SEMNIFICATIE (salt dacă..<<relație>>)	Condiția 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 și 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 și 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 depășire	OF=1
JNO	nu există depășire	OF=0

Tabelul 4.1. Instrucțiunile de salt condiționat

Aici: .

DB 1000 DUP (?)

dec ax
jnz Aici

se va semnala eroare, deoarece eticheta Aici se află la peste 1000 de octeți distanță față 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 instrucțiune de salt condiționat pentru a lua decizia dacă trebuie sau nu să facă un salt necondiționat la eticheta Aici (un salt necondiționat nu este supus nici unei restricții referitoare la distanța de salt).

Pentru a facilita alegerea corectă de către programator a variantelor de salt condiționat în raport cu rezultatul unei comparații (adică, dacă programatorul dorește interpretarea rezultatului comparației cu semn sau fără semn) dăm următorul tabel:

Relația între operanzi ce se dorește a fi testată	Comparație cu semn	Comparație 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 instrucțiunilor de comparare în funcție de relația testată.

Comparând acest tabel cu tabelul 4.1 se poate deduce modul de poziționare a flagurilor de către instrucțiunea CMP. Studiul acestui tabel reiterează afirmația noastră anterioară: nu instrucțiunea CMP este cea care face distincție între o comparație cu semn și una fără semn! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine NUMAI instrucțiunilor de salt condiționat specificate ULTERIOR comparației efectuate.

Tabelul de mai sus îl considerăm foarte util pentru interpretarea rezultatelor instrucțiunilor aritmetice în general. Fără a mai efectua o instrucțiune CMP, considerând d rezultatul ultimei instrucțiuni aritmetice executate și punând $s=0$, tabelul rămâne valabil.

4.3.2.3. Exemple comentate

Modul de acțiune a instrucțiunilor de salt conditionat și instrucțiunea cmp

mov al,80h	;al := 128 = 10000000b = -128 ! (Interesant! – remarcăm faptul că datorită regulilor de reprezentare în cod complementar 128 și -128 au aceeași reprezentare binară și anume 10000000b!)
(*) cmp al,0	;instrucțiunea cmp nu interpretează în nici un fel valoarea din AL (ca fiind ;cu semn sau fără semn) ci doar realizează scăderea fictivă al-0 și afectează ;corespunzător flagurile: SF=1, CF=ZF=OF=PF=AF=0.
jl et	;utilizarea instrucțiunii JL (Jump if Less than) provoacă interpretarea ;comparației al<0 cu semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă SF≠OF și cum SF=1 iar OF=0 se decide îndeplinirea condiției ;și saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a ;stat la latitudinea programatorului care prin utilizarea instrucțiunii JL a ;decis că dorește să compare -128 cu 0 și cum -128 este "less than" 0 ;condiția a fost îndeplinită (echiv. cu jnge et). În contrast, jnl et sau jge ;et ;(care vor testa dacă SF=OF) NU vor fi îndeplinite și NU vor provoca saltul ;la eticheta specificată.
jb et	;utilizarea instrucțiunii JB (Jump if Below) provoacă interpretarea ;comparației al<0 fără semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă CF=1 și cum CF=0 se decide neîndeplinirea condiției deci nu ;se va face saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JB a decis că dorește să compare 128 cu 0 și cum 128 NU este "below" 0 condiția NU a fost îndeplinită (echivalent cu jnae et sau jc et).
jae et1	;se testează fără semn dacă al \geq 0 (128 \geq 0?) - CF=0 deci condiție ;îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la ;eticheta et1
jbe et2	;se testează fără semn dacă al \leq 0 (128 \leq 0?) – CF = ZF = 0 deci condiția ;(CF=1 sau ZF=1) NU este îndeplinită și ca urmare nu se va face saltul la ;eticheta et2 – rezultat consistent cu jb et, deoarece jbe implică jb ;(echivalent cu jna et2)

Cap.4. Instrucțiuni ale limbajului de asamblare.

ja et3	;se testează fără semn dacă al > 0 (128 > 0?) – CF = ZF = 0 deci condiția ;(CF=0 și ZF=0) este îndeplinită și ca urmare se va face saltul la eticheta et3 ;(echivalent cu jnbe et3) și rezultat consistent cu jbe et2, deoarece dacă ;jbe nu este îndeplinită atunci ja trebuie să fie.
je et4	;se testează dacă al = 0 (128 = 0 ?) – nu se pune problema semnului dacă se ;testează egalitatea! – cum ZF=0, condiția ZF=1 nu este îndeplinită deci nu ;se va efectua saltul la eticheta et4 (echivalent cu jz et4). În contrast, jne ;et4 sau jnz et4 (care vor testa dacă ZF=1) vor fi îndeplinite și vor provoca ;saltul la eticheta specificată.
jle et5	;se testează cu semn dacă al \leq 0 (-128 \leq 0?) – OF = ZF = 0 și SF=1 deci ;condiția (ZF=1 sau SF≠OF) este îndeplinită și ca urmare se va face saltul la ;eticheta et5 (echivalent cu jng et5) și rezultat consistent cu jl et, deoarece ;jle implică jl.
jg et6	;se testează cu semn dacă al > 0 (-128 > 0?) – OF = ZF = 0 și SF=1 deci ;condiția (ZF=0 și SF=OF) NU este îndeplinită și ca urmare NU se va face ;saltul la eticheta et6 (echivalent cu jnle et6) și rezultat consistent cu jle ;et5, deoarece dacă jg nu este îndeplinită atunci jle trebuie să fie.
jp et7	;se testează dacă PF=1 - PF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (echivalent cu jpe et7 – Jump if Parity Even). În contrast, jnp et7 ;(care testează dacă PF=0 – echivalentă cu jpo et7 – Jump if Parity Odd) va ;fi îndeplinită și saltul se va efectua.
jo et8	;se testează dacă OF=1 - OF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (nu există depășire). În contrast, jno et8 (care testează dacă OF=0) va ;fi îndeplinită și saltul se va efectua.
js et9	;se testează dacă în interpretarea cu semn <u>rezultatul</u> comparației are semn ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucțiunii ;CMP, nu este vorba de a interpreta cu semn sau fără semn <u>operanzei</u> ;scăderii fictive d-s, ci <u>rezultatul</u> final al acesteia !) adică testăm dacă SF=1 - ;condiție îndeplinită în cazul nostru și ca urmare saltul se va efectua !. În ;contrast, jns et9 (care testează dacă SF=0) NU va fi îndeplinită și saltul ;NU se va efectua.
cmp 0,al	;eroare de sintaxă : "Illegal immediate" deoarece sintaxa instrucțiunii cmp ;interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuși dorim forțarea unei comparații de acest tip (0-al) putem utiliza ;pe post de prim operand un registru inițializat cu valoarea 0.
mov bl,0	

`cmp bl, al`; realizează scăderea fictivă $bl-al$ ($0-al = 0-80h = 0-10000000b = 10000000b$) și afectează corespunzător flagurile: $CF=SF=OF=1$, $ZF=PF=AF=0$.

Exercițiu propus: Reluați discuția efectului tuturor instrucțiunilor de salt condiționat de mai sus (analizate pentru cazul comparației (*) `cmp al,0`) în condițiile în care această comparație e înlocuită de ultimele două instrucțiuni 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 discuție regulile practice de setare a acestor flag-uri. Aceste reguli generale sunt:

- SF ia valoarea bitului de semn al rezultatului obținut;
- CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obținut a provocat ($CF=1$) sau nu ($CF=0$) un transport în afara spațiului 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 și acest lucru se reflectă în CF)
- OF este setat la valoarea 1 dacă există depășire în interpretarea cu semn a rezultatului (“*OF is set if there exists a signed overflow*”), adică dacă rezultatul obținut nu se încadrează în intervalul de interpretare admis (acesta fiind $[-128..+127]$ dacă este vorba despre octeți și respectiv $[-32768..+32767]$ pentru cuvinte interpretate cu semn).

Ultimele două reguli derivă de fapt din modul de implementare a conceptului de depășire (overflow) la nivelul procesorului 80x86.

În cazul operațiilor/operanzilor fără semn depășirea va fi semnalată prin setarea indicatorului CF (carry flag). În cazul operațiilor/operanzilor cu semn depășirea va fi semnalată prin setarea indicatorului OF (overflow flag).

Cum să detectăm însă situațiile de depășire în cazul operațiilor de adunare și scădere ? Care sunt regulile practice de aplicat pentru a înțelege și a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ? În discuțiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (overflow flag) deoarece și datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situații din partea programatorilor ca fiind depășire sau nu.

Atragem însă atenția asupra a ceea ce se ignoră de multe ori în acest context și anume faptul că o situație de tipul $CF=1$ (cu $OF=0$) semnalează la rândul ei o depășire, însă pentru cazul numerelor interpretate fără semn.

Pentru ADUNARE: dacă se adună două numere de același semn și rezultatul este de semn diferit atunci se semnalează depășire ($OF=1$), în caz contrar nu ($OF=0$). Aceasta este deci ceea ce am putea numi regula depășirii la adunare (RDA) în cazul interpretării cu semn.

De exemplu, la nivel de octet, dacă vom considera adunarea $100 + 50 = 150$ vom obține depășire (!) cu semn (pare surprinzător, nu-i aşa ?). Justificare: $100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b)$. Operanții au același semn dar rezultatul este de semn diferit, deci conform RDA vom avea $OF=1$. Intuitiv, depășirea se poate justifica prin faptul că $150 \notin [-128..127]$ deci se obține o eroare de tip “*out of range*”. Deși 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 operanții (100 și 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 și 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 spațiului de reprezentare) deci nu avem depășire 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:

$$\begin{array}{r} 10010110 + \\ 10000010 \\ \hline 100011000 \end{array}$$

Se observă din reprezentarea binară că există un transport de cifră 1 în afara spațiului de reprezentare admis al celor 8 biți, deci intuitiv este suficient de justificat depășirea. Din punct de vedere al aplicării RDA obținem pe 8 biți 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 biți!) ce ar fi trebuit obținută ($100011000b = 118h$), iar trunchierea are loc tocmai datorită depășirii. Ca urmare, nu se poate obține un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depășire!). Se observă că o astfel de trunchiere înseamnă întotdeauna și apariția unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat și $CF=1$.

$10010110b = 96h = -106$ (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 depășirii). Tehnic, am văzut deja că $CF = 1$ și rezultă astfel clar că avem depășire în interpretarea fără semn.

Nu putem avea deci $-106 + (-126) = 24$! (pentru că $00011000_b = 18_h = 24$ în ambele interpretări) Acesta este sensul în care se aplică RDA aici. Un alt mod de justificare intuitivă a depășirii în acest tip de situație este:

În interpretarea cu semn avem $-106 + (-126) = -232 \notin [-128..127]$ deci OF=1.

Această ultimă motivație este mai intuitivă pentru justificarea depășirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne "cea mai rapid aplicabilă regulă practică din punct de vedere algoritmic" dacă ne putem exprima așa... (și iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnalala niciodată depășire. De asemenea, dacă adunăm două numere de același semn dar rezultatul are același semn cu operanții nu se va semnalala nici în acest caz depășire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeași dimensiune ca și cea a operanților). Se poate verifica ușor din punct de vedere matematic că în nici unul din aceste cazuri nu ieșim din intervalul de interpretare admis.

Pentru SCĂDERE: se interpretează operanții respectiv cu semn, se efectuează scăderea solicitată asupra configurațiilor corespunzătoare de biți și dacă rezultatul obținut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul $[-128..127]$ pentru octeți cu semn și respectiv $[-32768..32767]$ pentru cuvinte interpretate cu semn) atunci se semnalază depășire (*overflow*) și astfel OF=1. Această formulare o putem numi *regula depășirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depășirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea CF=1, pe care o putem interpreta semantic drept "depășire la scădere în interpretarea fără semn".

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum și impactul lor asupra modului de setare al flag-urilor.

Exemple:

- ```
mov ah,82h ;82h = 130 (interpretarea fără semn) = -126 (interpretarea cu semn)
 ;= 10000010b (bitul de semn fiind 1 cele două interpretări diferă)
mov bh,2ah ;2ah = 42 (atât în interpretarea cu semn cât și în cea fără semn)
 ;= 00101010b (bitul de semn fiind 0 cele două interpretări coincid)
cmp ah,bh ;se realizează scăderea fictivă ah-bh=10000010b - 00101010b = 01011000b
 ;= 58h = 88 (atât în interpretarea cu semn cât și în cea fără semn deoarece
 ;bitul de semn este 0)
Această scădere setează flag-urile astfel:
```

SF = 0 (deoarece bitul de semn pentru rezultatul 58h = 01011000b este 0)

CF = 0 (deoarece  $|82h| > |2ah|$  nu se pune problema unei scăderi cu împrumut de cifră; deci nu vom avea depășire în interpretarea fără semn care se efectuează:  $130 - 42 = 88$ ) OF = 1 (se efectuează scăderea în interpretarea cu semn, adică  $ah-bh = -126 - 42 = -168$  și cum  $-168 \notin [-128..127]$  se semnalizează *signed overflow* și ca urmare OF=1)

`cmp bh,ah` ;se realizează scăderea fictivă  $bh-ah = 00101010b-10000010b = 10101000b$ 
 ;= A8h = 168 (în interpretarea fără semn) = -88 (în interpretarea cu semn)
Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul A8h = 10101000b este 1)
CF = 1 (deoarece  $|2ah| < |82h|$  se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine  $42 - 130 = 168$  (!) provenită de fapt din necesitatea unei scăderi de tipul  $(256 + 42) - 130 = 168$  și ca urmare a necesității împrumutului se va semnalala depășire în interpretarea fără semn, înțeleasă aici ca "nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut")

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică  $bh-ah = 42 - (-126) = +168$  și cum  $+168 \notin [-128..127]$  se semnalizează *signed overflow* și ca urmare OF=1)

- ```
mov ah,126 ;echivalent cu mov ah,7eh deoarece 126 = 7Eh = 01111110b (bitul de
            ;semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este
            ;126 atât în interpretarea cu semn cât și în cea fără semn)
mov bh,2ah ;2ah = 42 (atât în interpretarea cu semn cât și în cea fără semn)
            ;= 00101010b (bitul de semn fiind 0 cele două interpretări coincid)
cmp ah,bh ;se realizează scăderea fictivă  $ah-bh = 01111110b-00101010b = 01010100b$ 
            ;= 54h = 84 = 126 - 42 (atât în interpretarea cu semn cât și în cea fără semn
            ;deoarece bitul de semn al rezultatului este 0)
```

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul 54h = 01010100b este 0)
CF = 0 (deoarece $|126| > |42|$ nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnalala depășire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $ah-bh = 126 - 42 = 84$ și cum $84 \in [-128..127]$ NU se semnalizează *signed overflow* și ca urmare OF=0)

`cmp bh,ah` ;se realizează scăderea fictivă $bh-ah = 00101010b-01111110b = 10101100b$
 ;= 42-126 = ACh = 172 (în interpretarea fără semn) = -84 (în interpretarea
 ;cu semn)
Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul ACh = 10101100b este 1)
CF = 1 (deoarece $|42| < |126|$ se pune problema unei scăderi cu împrumut de cifră ; în interpretarea fără semn scăderea devine $42 - 126 = 172$ (!) provenită de fapt din necesitatea unei

scăderi de tipul $(256 + 42) - 126 = 172$ și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn prin setarea flagului carry)

$OF = 0$ (se efectuează scăderea în interpretarea cu semn, adică $bh-ah = 42-126 = -84$ și cum $-84 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare $OF=0$)

Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii $a-b \in [-127..127]$ atunci și $b-a \in [-127..127]$ (situația particulară în care $a-b = -128$ o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului $[-32767..32767]$ cu discuție asupra cazului particular -32768 . Ca urmare se poate concluziona faptul că instrucțiunile **cmp a,b** și **cmp b,a** vor furniza întotdeauna aceeași valoare pentru OF.

iii). - discuție asupra cazurilor **cmp 80h,0** și **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 depășire în interpretarea fără semn)
 $OF = 0$ (se efectuează scăderea în interpretarea cu semn, adică $ah-bh = -128 - 0 = -128$ și cum $-128 \in [-128..127]$ NU se semnalează *signed overflow* și 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 $(256 + 0) - 128 = 128$ și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn prin setarea flagului carry)

$OF = 1$ (se efectuează scăderea în interpretarea cu semn, adică $bh-ah = 0 - (-128) = +128$ și cum $+128 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare $OF=1$)

$CF = 1$ în cazul **cmp 0,80h** deoarece se efectuează o scădere cu împrumut de tipul :

$$\begin{array}{rcl} 0 - 10000000b & = & 100000000 - \\ & & \underline{10000000} \\ & & 010000000 \end{array}$$

și cifra de împrumut se transferă în CF.

Să analizăm în acest context ce înseamnă și 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 și totodată să facă precizarea că numerele -128 și +128 nu pot coexista între limitele aceluiasi interval de reprezentare al aceluiasi tip de dată! (reamintim că în limbaj de asamblare *tip de dată = dimensiune de reprezentare*)

În acest sens este de observat și impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât **shortint** cât și **byte** în Turbo Pascal acceptă valoarea $80h$ (-128 ca **shortint** și +128 ca **byte**) însă $80h$ nu poate avea două interpretări distințe în cadrul aceluiași tip de dată! Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 și +128 ca fiind prezente în cadrul aceluiași 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 și a tipului de dată **shortint** din Turbo Pascal): deci +128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet!

Totuși, după cum putem verifica foarte ușor, instrucțiunile **mov ah, 128** și **mov ah,-128** sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în **ah** a configurației 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 configurație binară nu presupune și necesitatea interpretării respectivei configurații într-un anumit fel. Sarcina interpretării acelei configurații drept cu semn sau fără semn va cădea în sarcina instrucțiunilor ce urmează și care vor folosi ca operanți aceste valori. De exemplu, utilizarea lui **IMUL** în loc de **MUL** va provoca interpretarea configurației binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui **DIV** în loc de **IDIV** va provoca interpretarea aceluiași operand ca fără semn și.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 depășire în interpretarea fără semn și astfel $CF = 0$. În interpretarea cu semn a operanților și a rezultatului final avem $-128 - 0 = -128 \in [-128..127]$ deci nu avem depășire nici în interpretarea cu semn și astfel $OF = 0$.

Pe altă parte, avem evident în ambele cazuri SF=1. Justificarea *intuitivă*: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea *tehnică*: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

iv). Să analizăm în continuare modurile în care putem compara valorile 0 și 1 (și apoi 0 și -1) și ce efecte are asupra flagurilor instrucțiunea **CMP** în fiecare dintre situații.

Situația **CMP 1,0** (evidențiată la nivelul unui text sursă de exemplu prin **CMP AH,0** cu **AH=1**) va efectua scăderea fictivă $1-0 = 1 = 00000001b$. Efectul asupra flag-urilor va fi CF = SF = OF = ZF = PF = AF = 0. Justificările sunt evidente pe baza discuțiilor din exemplele anterioare.

Situația **CMP 0,1** (evidențiată la nivelul unui text sursă de exemplu prin **CMP AH,1** cu **AH=0**) va efectua scăderea fictivă $0-1 = -1 = 11111111b$:

$$\begin{array}{r} 0 - 00000001b = 10000000 - \\ \underline{00000001} \\ 01111111 \end{array}$$

Efectul asupra flag-urilor va fi CF = SF = PF = AF = 1 și ZF = OF = 0. Justificarea valorilor din CF și SF este și aici evidentă pe baza discuțiilor din exemplele anterioare iar OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$.

Situația **CMP -1,0** (evidențiată la nivelul unui text sursă de exemplu prin **CMP AH,0** cu **AH=-1**) va efectua scăderea fictivă $-1-0 = -1 = 11111111b$. Efectul asupra flag-urilor va fi SF = PF = 1 și CF = OF = ZF = AF = 0. SF=1 deoarece bitul de semn este 1. OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$. CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situația **CMP 0,-1** (evidențiată la nivelul unui text sursă de exemplu prin **CMP AH,-1** cu **AH=0**) va efectua scăderea fictivă $0 - (-1) = +1 = 00000001b$:

$$\begin{array}{r} 0 - 11111111b = 10000000 - \\ \underline{11111111} \\ 00000001 \end{array}$$

Efectul asupra flag-urilor va fi CF = AF = 1 și OF = SF = ZF = PF = 0. SF = 0 deoarece bitul de semn este 0. OF=0 deoarece $0 - (-1) = +1 \in [-128..127]$. CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica și așa: în interpretarea fără semn această scădere înseamnă de fapt $0 - 255 = 1$ (!), care trebuie justificată prin $(256+0) - 255 = 1$, deci e nevoie de cifră de împrumut și astfel se semnalează depășire în cazul interpretării fără semn, deci CF = 1.

v). Cazurile studiate anterior (i-iv) s-au referit la operații de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucțiunii **CMP**. Să analizăm în continuare și cazul unei depășiri furnizate de operația de adunare revenind astfel la discuția asupra aplicării regulii RDA:

mov ah,126 ; $126 = 0111110b = 7eh$ (aceeași valoare 126 în ambele interpretări)
add ah, 2 ; $2 = 2h = 00000010b$; AH := $0111110b + 00000010b = 7eh + 02h =$
 $; 10000000b = 80h$ (= 128 fără semn = -128 cu semn)
CF = 0 deoarece:
 $0111110 +$
 00000010
 10000000 - nu există transport în afara spațiului de reprezentare al rez.

SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operației efectuate este strict negativ = -128).

OF = 1 deoarece:

- justificare *tehnică* - conform RDA se adună două numere de același semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).
- justificare *intuitivă* - adunăm două numere fără semn a căror sumă este $126 + 2 = 128$. Însă numărul $+128 \notin [-128..127]$ deci se semnalează *signed overflow* și ca urmare OF=1.

vi). Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situația în care programatorul își inițializează operanții cu anumite valori inițiale dorite (cu semn sau fără semn, conform necesităților problemei în cauză) și se așteaptă la obținerea unor rezultate sau reacții în conformitate cu valorile furnizate. Atenție însă! De obicei aceste valori au o dublă interpretare posibilă și nu vor fi interpretate în orice situație sub forma furnizată la inițializare!

Utilizarea ulterioară a unor instrucțiuni care forțează prin modul de lor de acțiune interpretarea complementară (cu semn/fără semn) celei de la inițializare poate provoca apariția unor situații în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare... Aceasta se întâmplă dacă nu se ține cont în permanență de dubla interpretare posibilă a configurațiilor binare manipulate. Să luăm un exemplu:

mov al, 200 ; $al = 11001000b = 0C8h = 200$ (fără semn) = -56 (cu semn)
mov bl, -1 ; $bl = 11111111b = 0FFh = 255$ (fără semn) = -1 (cu semn)
cmp al, bl ; $al-bl = 11001001b = C9h = -55$ (cu semn) = 201 (fără semn)
(și se setează corespunzător OF=ZF=0 și CF=SF=1)

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 așa cum precizează valorile de la inițializare? Sau poate pe 200 cu 255? Sau pe -56 cu -1? Sau pe -56 cu 255?

Răspuns: comparăm întotdeauna pe 0C8h cu 0FFh sau în exprimare binară pe 11001000 cu 11111111. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparației efectuate în baza 10 nu este

dedus din acțiunea instrucțiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucțiuni ulterioare care vor avea rolul de a interpreta în unul din cele 4 moduri de mai sus comparația efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucțiunilor corespunzătoare de salt condiționat:

jl et1 ; evident că $200 < -1$ deci la prima vedere pare că nu este îndeplinită condiția necesară pentru efectuarea saltului... să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparației ca fiind cu semn (deci -55) aceasta însemnând implicit și faptul că scăderea este interpretată ca $(-56 - (-1))$ deci și operanții vor fi amândoi interpretați cu semn... cum $-56 < -1$ iată că și intuitiv condiția se verifică (pe lângă justificarea tehnică a îndeplinirii condiției de salt SF≠OF) și deci saltul se va efectua! Deci chiar dacă programatorul a furnizat la inițializare valorile 200 și -1, utilizarea instrucțiunii JL a provocat interpretarea comparației ca fiind între -55 și -1 și nu între 200 și -1! (explicația de aici și faptul că saltul se va efectua vă poate ajuta să "demonstrați" unor colegi cum 200 poate fi mai mic decât -1 !!!)

ja et2 ; deoarece $200 > -1$ în acest caz ne-am așteptă ca saltul să se efectueze... însă utilizarea instrucțiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparație corectă aici este comparația lui 200 cu 255 și cum $200 > 255$ condiția nu este îndeplinită și 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 condiția 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 $-56 > -1$, iar tehnic deși ZF=0 nu este îndeplinită și condiția SF = OF deci ; saltul nu se va efectua

Ca urmare din cele 4 situații teoretic posibile de mai sus, vom întâlni concret numai două:

- comparație fără semn (200 cu 255) - impusă de "above" sau "below"
- comparație cu semn (-56 cu -1) – impusă de "less than" sau "greater than"

Nu putem aşadar compara de fapt pe 200 cu -1 așa cum au fost specificate valorile la inițializare și nici pe -56 cu 255 deoarece **interpretarea este ori cu semn ori fără semn pentru ambele operanzi!**

vii). Am studiat în exemplele anterioare modalitatea de reacție (de interpretare) a procesorului 80x86 legată de noțiunea de depășire în cazul operațiilor de adunare și de scădere. Când și cum semnalează însă procesoarele din familia 80x86 depășirea la înmulțire și respectiv la împărțire?

"Depășirea" la înmulțire. Instrucțiunile MUL și IMUL setează CF=1 și 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 definiția noționii de "depășire la înmulțire" în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincție între MUL și IMUL și de aceea nici între CF și OF. Ori vor fi amândouă flag-

urile setate la valoarea 1 cu semnificația de "depășire la înmulțire" în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biți:

```
mov al, 5
mov bl, 170
mul bl      ;AX := AL * BL = 5 * 170 = 850 = 0352h și vom avea CF=1 și 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 și vom avea CF=1 și
              ;OF=1 deoarece octetul superior AH = 0feh ≠ 0.
```

În cazul unor operanzi pe 16 biți putem avea de exemplu:

```
val1 DW 2000h
val2 DW 0100h
...
mov ax, val1
mul val2    ;DX:AX = 00200000h și vom avea CF=1 și OF=1 deoarece jumătatea
              ;superioară a produsului DX:AX, adică registrul DX conține valoarea 0020h ≠ 0.
```

Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potențială pierdere de informație ca și în cazul celorlalte depășiri - adunare, scădere sau împărțire. Aceasta deoarece chiar dacă înmulțim valorile maximale posibile a fi reprezentate pe dimensiunea operanților ($255 * 255$ pentru octeti și respectiv $65535 * 65535$ pentru cuvinte) tot nu se depășește dublul dimensiunii de reprezentare a operanților, adică spațiul pe care îl avem oricum la dispoziție prin definiție, 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 înmulțirii cu semn (instrucțiunea 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).

Depășirea în cazul înmulțirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanți octeți (respectiv cuvinte) produsul nu începe 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 și capitolul 1, în care din punct de vedere matematic s-a specificat clar că înmulțirea nu provoacă de fapt depășire, tocmai din cauza alocării unui spațiu suficient pentru reprezentarea produsului. În concluzie, se poate spune că din punct de vedere matematic singura operație care nu provoacă depășire este înmulțirea, însă procesoarele 80x86 promovează totuși noțiunea de "depășire la înmulțire" pentru a diferenția între situațiile în care produsul începe într-un spațiu de dimensiunea operanților și în care nu.

Situatiile in care produsul incapsuleaza pe dimensiunea operanzilor vor fi caracterizate de setarile CF = OF = 0 (nu avem deci depasire la inmultire). Iata un exemplu:

```
mov al, 5
mov bl, 51
mul bl      ; AX := AL * BL = 5 * 51 = 255 = 00ffh si vom avea CF=0 si OF=0
               ; deoarece octetul superior AH = 0.
```

Depasirea la impartire. In cazul impartirii, specificarea acestei operații sub forma

(I)DIV operand

presupune că operandul specificat este împărțitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biți) iar deîmpărțitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărțitorul este cuvânt). Efectuarea operației are ca efect:

AX : operand pe 8 biți = cîtul în AL și restul în AH;
DX:AX / operand pe 16 biți = cîtul în AX și restul în DX;

În cazul împărțirii depasirea atunci când rezultatul împărțirii nu încapsează în spațiul rezervat conform definiției pentru reprezentare, mai exact, când cîtul nu încapsează în AL sau respectiv AX. Într-o astfel de situație, procesorul 80x86 emite o întrerupere 0, execuția terminându-se cu un mesaj furnizat de către rutina de tratare a întreruperii 0, de genul "Divide by zero", "Zero divide" sau "Divide overflow" (în funcție de tipul de procesor și/sau de SO instalat). Pare ciudat la prima vedere că o împărțire 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 împărțire care matematic se poate efectua. Secvența

```
mov ax,60000
mov bl,2
div bl
```

ar trebui să furnizeze din punct de vedere matematic cîtul 30000. Însă conform definiției împărțirii DIV acest cît 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 împărțirea de mai sus nu se poate nici ea efectua (similar cu o situație de tip *div 0*) și ca urmare înțelegem acum decizia proiectanților de a trata tot prin emiterea unei întreruperi 0 și o situație de genul celei de mai sus. Să remarcăm în acest sens și faptul că mesajul "Divide overflow" (depasire la împărțire) este acceptat în acest context ca similar unui "Divide by zero".

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentați este de a confunda exprimările "numere cu semn" și "numere fără semn" cu exprimările "numere negative" și respectiv "numere pozitive". Numere cu semn nu înseamnă automat numere negative! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.

Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunțul unei probleme care cere efectuarea unei anumite acțiuni "dacă numărul v este (strict) negativ"? În primul rînd vom 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 configurației binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra primului bit din reprezentarea numărului. Iată două alternative pentru realizarea unui astfel de test:

a). Realizăm o deplasare a primului bit în CF și testăm valoarea sa printr-o instrucție adecvată de salt condiționat. Secvența

mov al,v shl al,1 jc este_negativ	;pentru a nu afecta deosebitiv conținutul variabilei V ;shift stînga cu 1 poziție pentru ca primul bit să treacă în CF. ;dacă CF=1 atunci salt la eticheta este_negativ
---	---

asigură testarea faptului dacă variabila v este sau nu un număr negativ.

b). Utilizăm instrucția cmp pentru o comparație în raport cu 0:

cmp v,0 jl este_negativ	;scădere fictivă V-0 ;dacă V<0 atunci salt la eticheta este_negativ
----------------------------	--

sau alternativ

cmp 0,v jg este_negativ	; scădere fictivă 0-V ;dacă 0>V atunci salt la eticheta este_negativ
----------------------------	---

ix). Am văzut că la nivelul efectuării operațiilor de adunare sau scădere procesorul 80x86 nu diferențiază între adunări/scăderi cu semn sau fără semn (tehnic vorbind ele se efectuează drept operații binare cu rezultat interpretabil ulterior drept cu semn sau fără). Totuși, în momentul în care se pune problema exprimării în baza 10 a unei operații de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect operanții operației respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obținut? Mai concret:

00000101 + 11111110 (1) 00000011	(= 5 în ambele interpretări) (= 254 fără semn și -2 în interpretarea cu semn) (= 3 în ambele interpretări ale configurației pe 8 biți)
--	--

reprezintă $5 + 254 = 259 (= 1\ 00000011$ – configurație pe 9 biți !) sau reprezintă $5 + (-2) = 3$? După cum vom vedea și aici răspunsul este că putem interpreta în ambele moduri și să justificăm astfel ca două reacții separate modul de setare al flag-urilor CF și respectiv OF.

Datorită cifrei de transport vom avea CF=1 (independent de interpretarea operanzilor sau a rezultatului final drept cu semn sau fără semn, deoarece este vorba despre o consecință tehnică a

modului de efectuare a operației binare de adunare). Ca urmare în interpretarea fără semn avem depășire (evident, deoarece $259 > 255$, adică decât numărul maxim reprezentabil pe 1 octet).

Ce se întâmplă cu OF ? Rularea secvenței

```
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 situația de mai sus nu este considerată "depășire" în interpretarea cu semn! Din punct de vedere al justificării modului de setare a flag-ului OF secvența de mai sus ar fi mai corectă dacă ar fi scrisă:

```
mov al, 5
mov bl, -2
add al, bl ; deci 5 + (-2) = 3
```

și este evident că în această interpretare nu este vorba despre nici o depășire (și de aceea și OF = 0).

Să ne reamintim în acest context și exemplele date la prezentarea RDA și RDS de la paginile ??-??: adunarea $100 + 50 = 150$ va semnala depășire (*signed overflow* - conform RDA), iar scăderile $130 - 42$ (interpretată ca $-126 - 42 = -168 \notin [-128..127]$) și $42 - 130$ (interpretată ca scăderea $42 - (-126) = +168 \notin [-128..127]$) produc la rândul lor *signed overflow* și ca urmare OF=1.

4.3.3. Instrucțiuni de ciclare

Cu ajutorul instrucțiunilor de salt condiționat se pot construi cicluri (bucle). Un ciclu nu este altceva decât un bloc de instrucțiuni care se termină cu o instrucțiune de salt condiționat (în acest sens se cunosc foarte bine construcțiile de tip **for**, **while** și **repeat** din unele limbi de nivel înalt).

Limbajul de asamblare 80x86 prevede instrucțiuni speciale pentru realizarea ciclării. Ele sunt: **LOOP**, **LOOPE**, **LOOPNE** și **JCXZ**. Sintaxa lor este

<instrucție> etichetă

Instrucțiunea **LOOP** comandă reluarea execuției blocului de instrucțiuni ce începe la *etichetă*, atât timp cât valoarea din registrul CX este diferită de 0. Se efectuează întâi decrementarea registrului CX și apoi se face testul și eventual saltul. Saltul este "scurt" (max. 127 octeți - atenție deci la "distanța" dintre LOOP și etichetă!).

Ca exemplu să luăm tipărirea caracterelor unui sir dat:

data segment

```
Test DB 'Acesta este un exemplu'
SF LABEL BYTE
```

cod segment

```
mov cx, SF-Test
mov bx,OFFSET Test
```

Tiparește:

mov dl,[bx]	;preia următorul caracter
inc bx	;punetează spre următorul caracter
mov ah,2	;funcția de tipărire a unui caracter
int 21h	;activarea tipăririi
loop Tiparește	;decrementez CX și revine la eticheta Tiparește ;dacă mai sunt caractere

În cazul în care condițiile de terminare a ciclului sunt mai complexe se pot folosi instrucțiunile **LOOPE** și **LOOPNE**.

Instrucțiunea **LOOPE** (*LOOP while Equal*) diferă față de LOOP prin condiția de terminare, ciclul terminându-se fie dacă CX=0, fie dacă ZF=1. În cazul instrucțiunii **LOOPNE** (*LOOP while Not Equal*) ciclul se va termina fie dacă CX=0, fie dacă ZF=0. Chiar dacă ieșirea din ciclu se face pe baza valorii din ZF, decrementarea lui CX are oricum loc.

Să presupunem de exemplu că dorim să reținem într-un vector caracterele citite de la tastatură până când se apasă tasta *<Enter>* și atât timp cât numărul caracterelor citite nu depășește 128. Acest lucru se poate obține prin secvența:

```
data segment
Vector DB 128 DUP(?)
```

cod segment

```
mov cx,128
lea bx,Vector
```

Bucla:

mov ah,1	;funcția DOS pentru citire caracter
int 21h	;activarea funcției
mov [bx],al	;memorarea tastei apăsat
inc bx	;pregătirea vectorului pentru următorul caracter
cmp al,0dh	;a fost <ENTER>?
loopne Bucla	;dacă nu, predă controlul instrucțiunii de după ;Bucla

LOOPE mai este cunoscută și sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută și sub numele de **LOOPNZ**. Aceste instrucțiuni se folosesc de obicei precedate de o instrucțiune CMP sau SUB.

O altă instrucțiune folosită pentru controlul ciclării este **JCXZ** (*Jump if CX is Zero*). Această instrucțiune realizează saltul la eticheta operand numai dacă CX=0, fiind utilă în situația în care se dorește testarea valorii din CX înaintea intrării într-o buclă. Exemplul următor se referă la inițializarea cu 0 a unui sir de octeți, CX conținând lungimea acestui sir. Instrucțiunea JCXZ se folosește pentru a se evita intrarea în ciclu dacă CX=0:

```

jcxz MaiDeparté ;dacă CX=0 se sare peste buclă
Bucla:
    Mov  BYTE PTR [si],0 ;inițializarea octetului curent
    inc  si               ;trecere la octetul următor
    loop Bucla            ;reluare ciclu sau terminare
MaiDeparté:

```

Dar să vedem de ce se acordă totuși atenție sporită situației în care se începe execuția unei bucle cu CX=0. La întâlnirea instrucțiunii LOOP cu CX=0, CX este decrementat, obținându-se valoarea 0FFFFh (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în CX, adică de încă 65535 ori! Iată deci de ce este necesar să ne asigurăm că nu intrăm într-un ciclu cu CX=0, iar instrucțiunea JCXZ acționează rapid și eficient exact în acest sens.

Este important să precizăm aici faptul că nici una dintre instrucțiunile de ciclare prezentate nu afectează flag-urile.

```

loop Bucla     dec cx
                jnz Bucla

```

deși semantic echivalente, nu au exact același efect, deoarece spre deosebire de LOOP, instrucțiunea DEC afectează indicatorii OF, ZF, SF și PF.

4.3.4. Instrucțiunile CALL și RET

Apelul unei proceduri se face cu ajutorul instrucțiunii **CALL**, acesta putând fi *apel direct* sau *apel indirect*. Apelul direct are sintaxa

CALL operand

Asemănător instrucțiunii JMP și instrucțiunea **CALL** transferă controlul la adresa desemnată de operand. În plus față de aceasta, înainte de a face saltul, instrucțiunea CALL salvează în stivă adresa următoarei instrucțiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalență

CALL operand	↔	[push CS] (numai dacă este operand far) push offset A jmp operand
---------------------	---	---

Terminarea execuției secvenței apelate este marcată de întâlnirea unei instrucțiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucțiunea de la această adresă. Sintaxa instrucțiunii RET este

RET [n]

unde *n* este un parametru optional. El indică eliberarea din stivă a *n* octeți aflați sub adresa de revenire. Vom detalia acest mecanism în capitolul 8. Instrucțiunea RET poate fi far sau near. Ca efect, avem echivalențele

RET n (revenire near)	↔	B dw ? pop B add sp,n jmp B
RET n (revenire far)	↔	B dd ? pop word ptr B pop word ptr B+2 add sp,n jmp B

De cele mai multe ori, instrucțiunile CALL și RET apar în următorul context

```

nume PROC
    .
    .
    .
ret n
nume ENDP
CALL nume

```

Directivele PROC și ENDP au fost prezentate în 3.3.4. Apelul și revenirea sunt implicit far sau near, după cum procedura *nume* este declarată FAR sau respectiv NEAR.

Instrucțiunea CALL poate de asemenea prelua adresa de transfer dintr-un registru (pentru un apel intrasegment) sau dintr-o variabilă de memorie. Un asemenea gen de apel este denumit *apel indirect*. Exemple:

```

call bx ;adresă preluată din registru
call vptr ;adresă preluată din memorie

```

Rezumând, operandul destinație al unei instrucțiuni 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 și în cazul instrucțiunii JMP).

În principiu, un macro și un apel de macro poate fi înlocuit cu o procedură și respectiv apelul ei. Se pune întrebarea: ce este mai bine să alegem pentru îndeplinirea unei sarcini? Un macro sau o procedură? Dacă primordial este obținerea 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 instrucțiunilor CALL și RET. Mai mult, flexibilitatea utilizării macrourilor rezultă și din posibilitățile de "croire" a unui macro pentru a genera după dorință coduri diferite pentru apeluri diferite. Marele avantaj al macrourilor în limbajul de asamblare rezidă în faptul că sunt singurele structuri parametrizabile ale acestuia, permitând astfel adaptarea apelurilor unui același macro în funcție de un context definibil prin parametri (a se vedea în acest sens 3.3.7.)

Să facem observația (foarte importantă!) că spre deosebire de procedurile din limbajele de programare de nivel înalt, subroutinele 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 registri de dinainte de apel, valori pe care subrutina are libertatea să le modifice, dar care ar putea fi necesare după revenire. O soluție ar fi ca la intrarea în subrutină registrii să fie salvați în stivă, iar la ieșire să se refacă starea lor inițială. Această variantă consumă însă mult timp și cere un număr destul de mare de instrucțiuni scrise în acest scop. O altă variantă ar fi să acceptăm ideea că valorile inițiale pot fi oricând alterate și în consecință să nu ne mai bazăm pe posibilitatea de păstrare a valorilor inițiale. Aceasta ar fi o variantă restrictivă care ne constrângă să limităm folosirea eficientă a registriilor.

Se adoptă ca soluție introducerea de către utilizator în fiecare subrutină a unor linii de comentariu ce conțin suficiente informații pentru precizarea registriilor afectați, pentru a ști deci și care sunt cei pe care valoarea inițială ne mai putem baza. Aceste informații devin deosebit de utile pentru programator în scopul gestionării eficiente a resurselor programului scris.

4.4. INSTRUCȚIUNI PE ȘIRURI

4.4.1. Generalități privind șirurile și instrucțiunile pe siruri

Instrucțiunile pe șiruri sunt niște instrucțiuni puternice, ele având ca efect simultan (eventual repetat) atât accesarea memoriei cât și incrementarea sau decrementarea unor registri pointer. Așa cum le arată și numele, aceste instrucțiuni se folosesc pentru manipularea șirurilor de octeți sau

cuvinte, ele fiind mai scurte (ca și cod rezultat) și mai rapide în execuție decât combinațiile echivalente de instrucțiuni MOV, INC și LOOP.

Un șir în sensul 8086 este caracterizat de următoarele atribute:

- a). tipul elementelor (octeți sau cuvinte).
- b). adresa primului element din șir.
- c). direcția de parcursere (de la adrese mici spre mari sau de la adrese mari spre mici).
- d). numărul de elemente.

Din punct de vedere al instrucțiunilor pe șiruri, un șir poate fi șir sursă sau șir destinație. Instrucțiunile pe șiruri sunt în număr de 10 și se împart în trei categorii:

- instrucțiuni care folosesc un șir sursă și un șir destinație (MOVSB, MOVSW, CMPSB, CMPSW).
- instrucțiuni care folosesc numai un șir sursă (LODSB, LODSW).
- instrucțiuni care folosesc numai un șir destinație (STOSB, STOSW, SCASB, SCASW).

Fiecare dintre aceste instrucțiuni pretinde să-i fie pregătite în prealabil caracteristicile șirurilor cu care operează.

a). *Tipul* este indicat prin ultima literă a numelui instrucțiunii: **B** pentru elemente octeți și **W** pentru elemente cuvinte. Instrucțiunile ce folosesc două șiruri presupun că ambele șiruri sunt de același tip.

b). *Adresa primului element dintr-un șir* este o adresă far memorată în registri, astfel:

- în DS:SI pentru șirurile care sunt sursă;
- în ES:DI pentru șirurile care sunt destinație;

c). *Direcția de parcursere* este indicată de flagul DF, astfel:

- DF=0 impune ca ordinea de parcursere să fie de la adrese mici spre adrese mari. În acest caz, adresa primului element este cea mai mică din șir.
- DF=1 impune ca ordinea de parcursere să fie de la adrese mari spre adrese mici. În acest caz, adresa primului element este cea mai mare din șir.

Pozitionarea prealabilă a lui DF se face folosind instrucțiunile CLD sau STD. Instrucțiunile care folosesc două șiruri presupun că ambele șiruri sunt parcuse în aceeași direcție.

d). *Numărul de elemente*, atunci când se dorește a fi exploataț, trebuie trecut în registrul CX.

Vom prezenta instrucțiunile pe șiruri sub forma a două grupe funcționale:

- * instrucțiuni utilizate pentru transferul de date (**LODS, STOS și MOVS**)
- * instrucțiuni utilizate pentru consultarea și compararea datelor (**SCAS și CMPS**)

4.4.2. Instrucțiuni pe șiruri pentru transferul de date

Acste instrucțiuni sunt asemănătoare instrucțiunii MOV, dar ele realizează mai mult și operează mai rapid.

Instrucțiunea LODS se prezintă sub două forme: **LODSB** și **LODSW**. Ea încarcă un octet sau respectiv un cuvânt din memorie în registrul acumulator.

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}	—	—

Instrucțiunea **LODSB** încarcă octetul de adresă DS:SI în AL și apoi incrementează sau decrementează SI, aceasta depinzând de starea flagului DF (*Direction Flag*): dacă DF=0 (valoare ce se poate seta după cum am văzut prin instrucțiunea CLD) atunci SI este incrementat, iar dacă DF=1 (setarea acestuia cu 1 făcându-se cu instrucțiunea STD) SI este decrementat. Această regulă impusă de valoarea din DF este valabilă pentru toate instrucțiunile pe șiruri ce afectează registri pointer. De exemplu,

```
cld
mov si,0
lodsb
```

va încărca AL cu conținutul octetului de deplasament 0 din cadrul segmentului de date și apoi se va incrementa SI cu 1, acțiuni echivalente cu

```
mov si,0
mov al,[si]
inc si
```

Instrucțiunea LODSB este însă mai rapidă și cu 2 octeți mai scurtă decât echivalentul

```
mov al,[si]
inc si
```

Instrucțiunea **LODSW** încarcă în AX cuvântul adresat de DS:SI, incrementând sau decrementând apoi SI cu 2 (deoarece este vorba de valori reprezentate pe cuvânt). De exemplu,

```
std
mov si,10
lodsw
```

încarcă AX cu valoarea cuvântului de memorie de deplasament 10 din cadrul segmentului de date, decrementând apoi SI cu 2.

Instrucțiunea **STOS** este complementara instrucțiunii LODS, ea transferând valoarea octet sau cuvânt din acumulator la locația de memorie de adresă ES:DI, incrementând sau decrementând apoi corespunzător pe DI. și instrucțiunea STOS se prezintă sub două forme: **STOSB** și **STOSW**.

Instrucțiunea **STOSB** copiază octetul din AL la octetul de adresă ES:DI, incrementând sau decrementând DI în funcție de valoarea din DF. De exemplu,

```
std
mov di,0ffffh
mov al,55h
stosb
```

depune valoarea 55h la octetul de deplasament 0FFFFh din cadrul segmentului pointat de ES, decrementând apoi DI la valoarea 0FFFEh.

Instrucțiunea **STOSW** este asemănătoare, copiind valoarea cuvânt din AX în cuvântul adresat de ES:DI, incrementând sau decrementând apoi DI cu 2. De exemplu,

```
cld
mov di,0ffeh
mov ax,102h
stosw
```

copiază valoarea cuvânt 102h din AX la adresa ES:0ffeh, incrementând pe DI la valoarea 1000h.

Instrucțiunile LODS și STOS funcționează eficient împreună pentru copierea de șiruri. Subrutina COPIERE de mai jos realizează copierea șirului terminat cu 0 care începe la DS:SI în șirul ce începe la adresa ES:DI :

```
;Subrutină pentru copierea unui șir terminat cu 0 în altul
;Intrări:      adresa de început a șirului sursă    DS:SI
;                  adresa de început a șirului destinație ES:DI
;Ieșiri : -
;Regiștri afectați : AL, SI, DI
```

Copiere PROC

```
cld
```

;parcurgerea se va face crescător deci se va impune incrementarea

iar: lods	;preia caracterul sursă
stosb	;memorează la destinație
cmp al,0	;a fost 0 ?
jnz iar	;dacă nu, se continuă
ret	;dacă da, se revine din procedură
Copiere ENDP	

Dacă cunoaștem direct numărul de octeți (cuvinte) ce trebuie copiați (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 locație de memorie în alta este prin folosirea instrucțiunii **MOVS**.

Instrucțiunea **MOVS** poate fi privită ca o combinație a instrucțiunilor LODS și STOS, ea preluând octetul (**MOVSB**) sau cuvântul (**MOVSW**) de la DS:SI și 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. Instrucțiuni pe siruri pentru consultarea și compararea datelor

Instrucțiunea SCAS (care are și ea două variante: SCASB și SCASW) căută în memorie o anumită valoare particulară (octet sau respectiv cuvânt).

Instrucțiunea SCASB compară valoarea din AL cu valoarea octet adresată de ES:DI, setând flagurile în acord cu rezultatele comparării (la fel ca și o instrucțiune 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, secvența următoare căută prima literă 'a' din cadrul sirului *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
    mov al,'a'
    mov cx,LungimeSir
    cld
    CautareA:
        scasb
        je GasitA
    loop CautareA
    ;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 comparației efectuate, DI este incrementat după execuția instrucțiunii SCASB, ceea ce face necesară decrementarea sa dacă după găsirea lui 'a' dorim ca DI să conțină 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 diferență ca ordine de efectuare a acțiunilor apărând în faptul că SCASB realizează incrementarea lui DI înaintea execuției instrucțiunii JE. Dacă aici incrementarea o facem explicit, atunci trebuie să o efectuăm după instrucțiunea JE pentru a nu afecta flagurile setate de către CMP!

Legat de aceasta să facem aici precizarea că instrucțiunile pe șiruri NU setează flagurile în urma acțiunii asupra regiștrilor SI, DI sau CX. Instrucțiunile LODS, STOS și MOVS nu afectează nici un flag, iar SCAS și CMPS modifică flagurile doar ca rezultat al comparațiilor pe care le efectuează.

Instrucțiunea SCASW compară conținutul registrului AX cu cuvântul de adresă ES:DI, incrementând sau decrementând DI cu 2, în funcție de valoarea din DF. Secvența de mai jos utilizează instrucțiunea SCASW cu prefixul REPE (vezi 4.4.4.) pentru a căuta ultima valoare nenulă dintr-un vector de întregi:

```

mov ax, SEG Tablou
mov es, ax
mov di, OFFSET Tablou+((NrElem-1)*2)
          ;ES:DI punctează astfel spre ultimul element al tabloului
mov cx, NrElem
sub ax, ax      ;punе 0 în AX pentru a căuta un element nenul
std             ;căutarea începe de la sfârșit
repe scasw      ;se repetă căutarea până la primul element nenul sau
                ;până la epuizarea elementelor tabloului
jne AmGasit     ;dacă se ajunge aici, tabloul conține numai elemente nule
AmGasit: inc di
               ;se actualizează DI pentru a puncta spre elementul găsit
inc di

```

Instrucțiunea CMPS are ca rol efectuarea de comparații de șiruri de octeți sau cuvinte. Execuția unei instrucțiuni CMPS are ca efect comparația locațiilor de memorie de adrese DS:SI și respectiv ES:DI, urmată de incrementarea sau decrementarea regiștrilor SI și DI. Flagurile vor fi actualizate corespunzător pentru a reflecta rezultatul comparației.

Instrucțiunea CMPSB realizează comparația la nivel de octet, iar CMPSW la nivel de cuvânt, prima incrementând sau decrementând SI și DI cu 1 iar ultima cu 2. În exemplul următor se compară două tablouri ce conțin elemente reprezentate pe cuvânt pentru a se decide dacă primele 100 de elemente sunt sau nu identice:

```

mov si, OFFSET Tablou1
mov ax, SEG Tablou1
mov ds, ax
mov di, OFFSET Tablou2
mov ax, SEG Tablou2
mov es, ax

```

```

mov cx, 100
cld
repe cmpsw
jne TabDiferite

```

TabDiferite:

```

dec si      ;se actualizează regiștrii SI și DI pentru
dec si      ;a puncta spre elementul prin care diferă
dec di
dec di

```

4.4.4. Execuția repetată a unei instrucțiuni pe șiruri

Pentru execuția repetată a unei instrucțiuni pe șiruri, limbajul de asamblare dispune de variante echivalente instrucțiunii de ciclare LOOP. Acestea sunt oferite de-așa numitele *prefixe de instrucțiune*. Sintaxa utilizării lor este

prefix_de_instrucțiune instrucțiune_pe_șir

În principiu este vorba despre un singur prefix de instrucțiune, și anume REP. Vom vedea mai jos însă, că în cazul utilizării instrucțiunii SCAS sau CMPS apar două variante posibile de REP. Prefixul de instrucțiune REP impune execuția repetată a instrucțiunii pe care o prefixează, până când valoarea din CX devine 0. Dacă de la început avem CX=0 atunci instrucțiunea respectivă este inoperantă. Astfel, ciclul

Bucla: *instrucțiune_pe_șir*
loop Bucla

este echivalent cu instrucțiunea

rep *instrucțiune_pe_șir*

În cazul utilizării prefixelor cu instrucțiunile SCAS sau CMPS condiția verificată se completează, ținându-se cont de valoarea flagului ZF. Acest lucru face ca prefixul REP să fie prezent în două variante.

Forma REP (echivalentă cu formele REPE - REPeat while Equal și REPZ - REPeat while Zero) provoacă execuția repetată a instrucțiunilor SCAS sau CMPS până când CX devine 0 sau până când apare o nepotrivire (caz în care ZF va primi valoarea 0).

Asemănător, REPNE (*REPeat while Not Equal*, formă echivalentă cu REPNZ -*REPeat while Not Zero*) provoacă execuția repetată a instrucțiunii 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 instrucțiunilor SCAS și CMPS? Foarte simplu: pentru că sunt singurele care afectează vreun flag (și anume ZF), permitând rafinarea condițiilor pe baza valorii flagului afectat. Rezultă deci, că în cazul instrucțiunilor LODS, STOS și MOVS toate cele cinci mnemonici prezентate au același efect: repetarea cât timp $CX < 0$.

Față de variantele cu LOOP, avantajul evident al folosirii acestor prefixe este comprimarea scierii.

4.4.5. Utilizarea de operanzi pentru instrucțiuni pe șiruri

În exemplele date până acum am folosit numai formele explicite ale instrucțiunilor pe șiruri în funcție de dimensiunile de reprezentare a datelor. De exemplu, am folosit LODSB și LODSW dar nu am utilizat LODS.

Limbajul de asamblare 80x86 acceptă utilizarea instrucțiunilor neexplicite dacă se prevăd operanzi pentru specificarea dimensiunii de reprezentare a argumentelor, ceea ce va permite alegerea corectă a instrucțiunii de executat (de exemplu, LODSB sau LODSW).

Instrucțiunea 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 instrucțiune MOVS în funcție de dimensiunea de reprezentare a operanzilor, aici aceasta fiind octetul.

Să facem precizarea că această informație este folosită numai pentru determinarea variantei de instrucțiune necesară, situația fiind oarecum asemănătoare unei declarații. Codul corespunzător operanzilor nu este asamblat în codul executabil, cel asamblat aici fiind de fapt cel corespunzător instrucțiunii rep movsb.

Am precizat aceasta pentru a evidenția faptul că prezența operanzilor într-o instrucțiune pe șiruri nu scutește 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 secțiune vom prezenta un exemplu complet de program. În acest scop ne-am oprit asupra determinării numerelor prime (și impare) până la un n dat. Pentru economie de memorie, punem în șir doar numerele impare. Descrierea algoritmului preferăm să o facem prin programul Pascal care urmează, program care tipărește toate numerele prime și 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 facilitățile oferite de acesta din urmă. De aceea, în loc de a reține câte un octet pentru fiecare poziție din ciur, vom reține doar câte un bit. Variabila *i* din programul Pascal este înlocuită aici cu registrul **bx**, care va conține numărul bitului curent. Numărul cuvântului curent este conținut în registrul **di**. Variabila *j* este înlocuită cu registrul **dx**. Tipărirea este realizată cu ajutorul funcției 9 a intreruperii 21h, despre care vom vorbi în capitolul următor. Aici trebuie doar remarcat faptul că operația de conversie în sir ASCII a unui număr cade în sarcina programatorului.

Cu aceste precizări, programul ASM este următorul:

```
.model small

n    equ 65535      ;definire de constante
ni   equ (n-1)/2
ni2  equ ni/2
nc   equ (ni+15)/16

.data

rez  db  ',13,10,$'
pas  dw ?
zece dw 10
s    dw nc dup (?)

.code
Start:
    mov ax,@data
    mov ds,ax
    mov es,ax

    mov ax,0ffffh
    cld
    mov cx,nc
    lea di,S
    rep stosw      ;for i
    xor bx,bx

whilei:
    cmp bx,ni2
    jb peste        ;echivalent cu jnb exit dar eticheta exit
    jmp exit         ;este prea departe pentru un salt condiționat
pestei:
    mov di,bx

    .code
exit:
```

```
mov cl,4
shr di,cl
shl di,1
mov ax,[S+di]    ;ax cuvântul curent

mov cx,bx
and cx,0fh
shr ax,cl
shl ax,cl
cmp ax,0          ;în cuvântul curent au mai rămas biți 1
jnz AreBit1
mov cx,nc
add di,2 + offset S
repe scasw        ;s-a găsit un cuvânt nenul
sub i,2 + offset S
mov x,[S+di]

AreBit1:
    mov cl,3
    mov bx,di      ;bx = numărul de biți din cuvintele precedente
    shl bx,cl
    iar:
        shr ax,1
        jc gasit
        add bx,1      ; se adaugă până la primul bit 1
        jmp iar
    gasit:
        mov pas,bx
        add pas,bx
        add pas,3      ;pas := i + i + 3
        mov dx,bx
        add dx,pas      ;j := i + pas

    whilej:
        cmp dx,ni
        jae Alti
        mov di,dx
        mov cl,3
        shr di,cl
        mov al,byte ptr [S+di]
        mov cx,dx
```

```

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 conține 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 ;tipărește numărul din bx
;se folosește schema lui Horner
    push ax
    push cx
    push dx

    lea di,Rez
    mov cx,10
    mov al,''
    rep stosb ;umple cu spații
    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 funcția 09h a întreruperii 21h
    lea dx,Rez

    int 21h

    pop dx
    pop cx
    pop ax

    ret

conv endp

end Start

```

C A P I T O L U L 5

ÎNTRERUPERI

5.1. PROBLEME GENERALE PRIVIND ÎNTRERUPERILE

O întrerupere este o acțiune a microprocesorului prin care acesta anunță apariția unui eveniment. Mai concret, întreruperea este un semnal electric transmis sistemului de calcul (SC) prin care acesta este anunțat de apariția unui eveniment particular.

Acțiunile pe care le efectuează sistemul de calcul la apariția unei încreruperi sunt:

1. suspendarea programului în curs de desfășurare;
2. lansarea în execuție a unei rutine specializate, numită *Rutină de Tratare a Întreruperii (RTI)* sau *Handler de încrerupere*, care deservește încreruperea;
3. eventual, reluarea execuției programului suspendat (depinzând de tipul de încrerupere).

Cauzele apariției acestor evenimente pot fi de 2 tipuri:

- a). *externe* (apăsarea unor combinații de taste, inițierea sau terminarea unor operații de I/O);
- b). *interne* (împărțirea la 0, tentativa de adresare a unei zone de memorie inexistente, tentativa de execuție a unei instrucțiuni având un cod inexistență, depășirea capacitatei de reprezentare a unui rezultat).

De obicei, după tratarea unei încreruperi externe programul se reia, după o încrerupere internă nu!

La apariția unei încreruperi, SC trebuie, în ordine:

- 1) să determine tipul evenimentului care a generat încreruperea (intern, extern);
- 2) să afle care este cauza încreruperii;
- 3) să determine adresa RTI (rutinei de tratare a încreruperii);

Există 3 categorii de rutine de tratare a încreruperilor (RTI):

- furnizate odată cu sistemul de calcul;
- scrise de proiectanții sistemului de operare (SO);
- scrise de utilizatori;

CAPITOLUL 5

INTRERUPERI

5.1. PROBLEME GENERALE PRIVIND ÎNTRERUPERILE

O întrerupere este o acțiune a microprocesorului prin care acesta anunță apariția unui eveniment. Mai concret, întreruperea este un semnal electric transmis sistemului de calcul (SC) prin care acesta este anunțat de apariția unui eveniment particular.

Acțiunile pe care le efectuează sistemul de calcul la apariția unei încreruperi sunt:

1. suspendarea programului în curs de desfășurare;
2. lansarea în execuție a unei rutine specializate, numită *Rutină de Tratare a Întreruperii (RTI)* sau *Handler de întrerupere*, care deservește încreruperea;
3. eventual, reluarea execuției programului suspendat (depinzând de tipul de încrerupere).

Cauzele apariției acestor evenimente pot fi de 2 tipuri:

- a). *externe* (apăsarea unor combinații de taste, inițierea sau terminarea unor operații de I/O);
- b). *interne* (împărțirea la 0, tentativa de adresare a unei zone de memorie inexistente, tentativa de execuție a unei instrucții având un cod inexistent, depășirea capacitatei de reprezentare a unui rezultat).

De obicei, după tratarea unei încreruperi externe programul se reia, după o încrerupere internă nu!

La apariția unei încreruperi, SC trebuie, în ordine:

- 1) să determine tipul evenimentului care a generat încreruperea (intern, extern);
- 2) să afle care este cauza încreruperii;
- 3) să determine adresa RTI (rutinei de tratare a încreruperii);

Există 3 categorii de rutine de tratare a încreruperilor (RTI):

- furnizate odată cu sistemul de calcul;
- scrise de proiectanții sistemului de operare (SO);
- scrise de utilizatori;

Pentru localizarea rapidă a RTI se folosește vectorizarea intreruperilor: asocierea fiecărei intreruperi cu o locație de memorie dublucuvânt cu adresa fixă, unde se va memora adresa fară a RTI corespunzătoare intreruperii.

Tabela RTI (vectorul de intreruperi) se află în memorie la adresa 0000:0000. Primii 256 x 4 = 1024 octeți conțin aceste adrese (zona se numește *tabela vectorilor de intrerupere - TVI*). Pentru intreruperea k, adresa RTI(k) se găsește la adresa 0000 : k*4.

Acest tablou cu adrese se inițializează în momentul încărcării sistemului de operare. În timpul funcționării sistemului, este posibilă modificarea unor dintre adrese: Modificările pot apărea fie accidental, fie intentionat. Modificarea accidentală conduce de cele mai multe ori la blocarea sistemului, fiind necesară reinițializarea lui. Modificarea voită o vom numi *returnare a intreruperii*, și ne vom ocupa de aceasta în capitolul 6.

5.2. CLASIFICAREA ÎNTRERUPERILOR

La nivelul arhitecturii 80x86 apar trei tipuri de evenimente numite în documentații intreruperi:

a). intreruperi hardware – intreruperi generate în mod automat ca răspuns la apariția 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, ținând însă că de sistemul de intrare-iesire (I/O system) și fiind astfel intreruperi BIOS (Basic Input Output System). RTI corespunzătoare sunt încărcate în memorie la pornirea sistemului de calcul din fișierele ROM-BIOS.

b). excepții - intreruperi generate în mod automat, ca răspuns la apariția unor cauze de tip intern. Exemple de cauze: împărțirea prin 0, încercarea de execuție a unui cod de instrucție inexistent, accesarea unei zone de memorie interzise (*memory protection fault*).

c). intreruperi software (*software interrupts sau traps*) – acestea presupun un transfer de control al execuției, inițiat de programator către o rutină specială (handler). Mijlocul prin care programatorul inițiază o astfel de acțiune este instrucțiunea INT. Aceste intreruperi se numesc intreruperi software tocmai pentru că ele sunt invocate soft printr-o instrucțiune specificată explicit de către programator.

Datorită acestui specific putem da o definiție alternativă acestui tip de intreruperi: acele intreruperi ce pot fi inițiate numai de către programator prin INT sunt intreruperi soft.

Vom prezenta și analiza în continuare cele mai importante intreruperi din fiecare categorie.

5.2.1. Întreruperi hardware

În documentații recunoaștem intrerupurile hardware prin referirea la IRQ (*Interrupt Request*), aceasta fiind o cerere de intrerupere nativă la nivelul unui dispozitiv numit PIC (*Programmable Interrupt Controller* – Intel 8259A). Exemplu: IRQ 0 = INT 8 , IRQ 1 = INT 9 etc.

Un excelent tutorial oferind detalii despre funcționarea acestui dispozitiv precum și despre intrerupurile hard găsim la: <http://www.delorie.com/djgpp/doc/ug/interrupts/inthandlers1.html>.

Cele mai cunoscute și frecvent utilizate (emise) intreruperi hardware sunt prezentate în cele ce urmează.

INT 8 este intreruperea 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:

- reține nr. de cicluri de ceas (*timer ticks*) la adresa 0000:046Ch pentru a gestiona în bune condiții 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 și octetul de la adresa 0000:043fh este actualizat pentru a reflecta aceasta.
- generează INT 1Ch (intreruperea soft de ceas).

INT 9 este intreruperea de tastatură (*keyboard interrupt*). Este recunoscută prin IRQ 1. Această intrerupere este generată de tastatură, la fiecare apăsare și eliberare a unei taste. Acțiunile concrete ale RTI corespunzătoare sunt în esență: 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 documentații), urmată de memorarea acestei perechi de atribute în bufferul de tastatură, acesta fiind localizat la adresa 0000:041Ch. Începând de la această adresă și până la deplasamentul 043eh (adică 20h = 32 octeți) se pot reține 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 tasturi și poziția unui caracter în configurația tastaturii trebuie caracterizată prin atributul scan code. Acesta este însă că un cod asociat unei taste în funcție de poziția ei pe tastatură.

Dacă s-a apăsat una dintre tastele Ctrl, Alt sau Shift, se actualizează corespunzător octetii de la adresele 0000:0417h (Shift Status byte) și 0000:0418h (Extended Shift Status byte).

Atenție! – a nu se dezactiva intreruperea 9, deoarece atunci SC nu va mai răspunde nici la CTRL + ALT + DELETE !!!.

INT 0Bh și **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 informației comparativ cu cea paralelă? În transmisia serială există o singură linie de date iar informația este transmisă succesiv bit cu bit (un singur bit odată). Deși lentă în comparație cu o transmisie paralelă ce permite transmiterea simultană a mai multor biți (8,16,32 sau 64) această tehnică a transmisiei seriale este adecvată transmiterii informației la distanțe mari sau în cazul utilizării ca medii de transmisie a cablurilor telefonice, cablurilor coaxiale, undelor radio sau fibrei optice.

INT 0Bh (IRQ 3) se activează când este vorba despre o comunicare serială pe interfața (portul) COM2, iar INT 0Ch (IRQ 4) se activează când este vorba despre o comunicare serială pe interfața (portul) COM1.

Gestionarea comunicației seriale la nivelul arhitecturii 80x86 este asigurată de către dispozitivul UART (*Universal Asynchronous Receiver/Transmitter*) 8250 (sau compatibil cu acesta) care generează o întrerupere (IRQ 3 sau IRQ 4, depinzând de linia serială pe care are loc comunicarea) într-o din următoarele patru situații: un caracter se întâlnește pe o linie serială, dispozitivul UART a terminat transmisia unui caracter și solicită un altul, apariția unei erori sau solicitarea unei modificări de stare. RTI corespunzătoare va trebui să determine cauza exactă a întreruperii prin interogarea dispozitivului UART.

INT 0Dh și **INT 0Fh** - întreruperi ce deservesc porturile parallele. După cum aminteam mai sus interfețele parallele permit transmisia simultană a mai multor biți (8,16,32 sau 64). Principalul avantaj al interfețelor parallele este astfel viteza de transmisie. Aceasta este însă obținută prin costuri suplimentare reprezentate de necesitatea unor cablări adiționale pentru a asigura extinderea canalelor de date. Din cauza acestor costuri, interfețele parallele sunt de obicei limitate la conexiuni scurte ce au sub 1 m lungime.

Înțial, întreruperile INT 0Dh (IRQ 5) și INT 0Fh (IRQ 7) au fost proiectate pentru a deservi porturile parallele LPT1, LPT2 (*Line Printer*), însă imediat după aceasta, IBM a proiectat o interfață pentru imprimantă (*printer interface card*) care nu este compatibilă cu aceste întreruperi! Ca urmare, astăzi ele nu mai sunt utilizate pentru imprimante ci preponderent pentru plăci SCSI și plăci de sunet.

INT 0Eh → întreruperea de dischetă (IRQ 6 - *Diskette Drive interrupt*)

INT 76h → întreruperea de hard-disk (IRQ 14 - *Hard Disk Controller*)

Unitățile de dischetă și harddisk generează fiecare câte o întrerupere la încheierea unei operații de citire/scrisoare. Aceste întreruperi sunt utile pentru gestiunea aplicațiilor în sistemele de operare multitasking (OS/2, Linux, Windows): în timp ce are loc o operație I/O cu discheta sau hard-disk-ul, microprocesorul poate executa părți din alte procese; când un disc și-a încheiat operația

curentă de citire/scrisoare va întrerupe acțiunea 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 funcționări 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 situație de excepție de tip virgulă flotantă (*floating-point exception*).

5.2.2. Excepții

Datorită faptului că și acestea sunt întreruperi BIOS sunt clasificări care încadrează excepțiile la **întreruperi hard** (care sunt și ele BIOS). Nu este însă corect pentru că după tratarea unei întreruperi hard programul se reia întotdeauna, în timp ce după tratarea excepțiilor, de obicei programul NU se reia!

INT 0 - întreruperea împărțirii la zero (*Zero Divide interrupt*). Întreruperea 0 este generată de fiecare dată când apare o așa numită *condiție de împărțire la zero*. INT 0 poate fi emisă în trei situații distincte:

i). depășirea rezultatului (câțulu) la împărțire atunci când utilizăm DIV sau IDIV;

```
mov ax,600
mov bh,2
(i) div bh ; se efectuează ax/bh, cu câtul în AL și restul în AH
```

Câțul ar trebui să fie 300 și să fie obținut în AL, însă valoarea 300 nu începe pe 1 octet în AL. Ca urmare, se va emite INT 0 cu mesajul de eroare “*Divide by zero*”. Într-un astfel de caz se recomandă efectuarea unei conversii prin lărgire care să asigure efectuarea corectă a împărțirii indiferent de valorile considerate:

```
mov ax,600
mov dx,0 ; sau cwd dacă se face conversie cu semn
mov bx,2
div bx ; se efectuează dx:ax/bx, cu câtul în AX și restul în DX
```

Acum valoarea 300 începe în AX și nu se mai emite INT 0.

ii). încercarea de efectuare a unei împărțiri la zero:

```
mov ax, 600
mov bh,0
div bh ; se va emite INT 0 deoarece se încearcă împărțirea la BH=0!
```

iii). Emiterea explicită a acestei intreruperi 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 afișează "Divide by zero" și predă controlul SO MS-DOS.

Observatie: De fapt toate intreruperile, indiferent de categoria din care fac parte pot fi activate și SOFT prin specificarea explicită la nivelul codului sursă a instrucțiunii INT n, unde n este numărul intreruperii ce se dorește a fi emisă. Asta nu le face însă intreruperi software! A se vedea definiția alternativă pe care am dat-o acestora: intreruperile software sunt acele intreruperi ce pot fi inițiate numai de către programator prin INT! Excepțiile și intreruperile hardware pot fi invocate și software, însă caracteristica lor de bază este că ele sunt emise în mod automat la apariția evenimentului corespunzător asociat.

INT 1 (Single Step). Această intrerupere este inițiată de procesor după fiecare instrucțiune mașină, dacă flagul TF = 1. Se folosește la depanare, execuțiile pas cu pas ale programelor în cadrul depanatoarelor fiind posibile tocmai datorită emiterii INT 1 după fiecare linie de cod sursă.

INT 2 - intreruperea nemascabilă (*Non-Maskable Interrupt - NMI*). Intreruperile pot fi dezactivate (mascate) prin instrucțiunea CLI (*Clear Interrupts*). INT 2 este singura intrerupere ce nu poate fi mascată, ea fiind generată de fiecare dată când apare o condiție nemascabilă, ca de exemplu o eroare de paritate a memoriei (*memory parity error*).

INT 3 (Breakpoint interrupt). Această intrerupere este folosită de către depanatoare pentru stabilirea de puncte de intrerupere a execuției (*breakpoints*) în cursul depanării programelor.

INT 4 (Overflow). Această intrerupere se emite când are loc o depășire în cadrul unor operații aritmetice. Mai precis, se emite când se execută instrucțiunea INTO (*interrupt on overflow*) și 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 condițiilor de depășire aritmetică. Prin execuția unei instrucțiuni INTO după căte o operație aritmetică expusă depășirii, se poate asigura tratarea adecvată a situațiilor de depășire.

Instrucțiunea INTO are următorul efect:

```
INTO ↔ if (OF=1) PUSHF  
TF:=0  
IF:=0  
CALL FAR 0000:0010h
```

INT 6 – (invalid opcode) – cod de instrucțiune ilegal. Această intrerupere se emite la încercarea de execuție a unui cod de instrucțiune inexistent. De exemplu:

```
.....  
add ax,2      ; ok, se execută fără probleme  
a db 199      ;această linie de cod, deși la prima vedere ar trebui să aparțină numai  
unui segment de date, poate apărea și în segmentul de cod, semnificația ei fiind în acest caz  
interpretarea valorii atribuite (aici 199) drept cod de instrucțiune; am generat aici codul 199 care nu  
reprezintă un cod de instrucțiune valid și ca urmare se va emite la această linie INT 6, al cărei  
handler va afișa mesajul "Illegal instruction".
```

5.2.3. Întreruperi software

O intrerupere software este invocată numai de către programator prin apelul instrucțiunii INT (cu excepția intreruperilor 05h, 19h, 1Bh). Diferența între invocarea unei intreruperi prin instrucțiunea INT și apelul far al unei proceduri cu instrucțiunea CALL este faptul că instrucțiunea INT pune și flagurile în stivă înainte de adresa de revenire, astfel încât se va folosi instrucțiunea IRET pentru revenire din rutină (spre deosebire de instrucțiunea RET, aceasta scoate din stivă și registrul de flaguri).

Intreruperile soft se clasifică în:

- **întreruperi BIOS** (rutinele de tratare a acestor intreruperi sunt încărcate în memorie la pornire din fișierele ROMBIOS)
- **întreruperi DOS** (rutinele de tratare a acestor intreruperi sunt încărcate în memorie la pornire din fișierele BDOS)

În plus, unele intreruperi pot fi caracterizate prin faptul că nu au un scop inițial declarat sau utilizatorii își pot scrie propriile rutine de tratare a lor (intreruperi utilizator).

Principalele intreruperi BIOS sunt:

05h Se emite la apăsarea tastei PrintScreen în scopul trimiterii conținutului ecranului la imprimantă. Instrucțiunea BOUND (ce verifică dacă valoarea unui index de tablou se află între limitele specificate) apelează și ea această intrerupere dacă condițiile verificate nu sunt îndeplinite.

10h Servicii de lucru cu écranul, în mod text și în mod grafic. Funcțiile acestei intreruperi pot fi folosite pentru setarea modului video (funcția 00h), setarea dimensiunii și a poziției cursorului (funcțiile 01h, 02h), citirea și scrierea de caractere și atribute (funcțiile 08h, 09h) și.a.m.d.

11h Returnează lista echipamentelor BIOS instalate în sistem: numărul de porturi parallele, dacă modemul intern este instalat, dacă adaptorul de jocuri este instalat, numărul de porturi seriale, numărul unităților de dischetă, modul video inițial, dacă coprocesorul matematic este instalat, dacă sunt unități de dischetă instalate.

12h Returnează dimensiunea memoriei RAM. Apelul acestei intreruperi avea sens atunci când calculatoarele aveau o memorie de până la 64K. Acest lucru nu mai este valabil în zilele noastre, aşadar această intrerupere este oarecum depăşită.

13h Pune la dispoziție servicii de lucru cu harddisk-ul și cu discheta: resetarea unui disc (funcția 00h), obținerea stării unei operații făcute asupra discului (funcția 01h), citirea și scrierea de sectoare de pe un disc fix sau dischetă într-un (respectiv dintr-un) buffer din memoria internă (funcția 02h, respectiv 03h), verificarea sectoarelor (funcția 04h) și.a.m.d.

14h Permite accesul la porturile seriale. Această intrerupere pune la dispoziție funcții de inițializare a unui port serial (funcția 00h), transmitere sau primire a unui caracter (funcțiile 01h și 02h) și obținere a stării unui port serial (funcția 03h).

15h Pune la dispoziție funcții de acces la memoria extinsă, de citire a dispozitivelor de tip joystick și.a.m.d.

16h Această intrerupere oferă servicii de manipulare a tastaturii, dintre care: citirea unui caracter din buffer-ul tastaturii (funcția 00h), returnarea stării unor taste (Caps Lock, Scroll Lock, Num Lock, ... – funcția 02h).

17h Această intrerupere oferă servicii de manipulare a imprimantei: tipărirea unui caracter (funcția 00h), inițializarea imprimantei (funcția 01h) și returnarea stării imprimantei (funcția 02h).

18h Activează interpretorul ROM BASIC. Astăzi mai puțin folosită, deoarece calculatoarele compatibile IBM nu mai includ acest interpretor.

19h Oferă servicii de încărcare a sistemului de operare. Efectul apelului acestei intreruperi este echivalent cu cel al apăsării combinației de taste Ctrl-Alt-Del.

1Ah Servicii legate de ceasul sistem. Există două astfel de servicii: citirea ceasului (funcția 00h) și setarea ceasului (funcția 01h).

1Bh Această intrerupere este apelată la apăsarea combinației de taste <CTRL/BREAK>. Ca efect al RTI corespunzătoare, în bufferul tastaturii se va pune CTRL-C ca următorul caracter. La citirea acestuia se va invoca intreruperea 23h, care termină execuția programului curent și redă controlul sistemului de operare.

1Ch Această intrerupere este apelată de 18.2 ori / secundă de RTI 8. Rutina de tratare a acestei intreruperi nu face nici o acțiune, lăsând posibilitatea utilizatorului de a scrie propria rutină de tratare. Aceasta este o intrerupere utilizator.

Adrese ale unor structuri de date BIOS (vezi Norton Guide pentru detalii):

1Dh Adresa zonei de parametri video.

1Eh Adresa zonei de parametri a unităților de dischetă.

1Fh Parametrii adaptorului grafic.

41h Parametrii harddisk-ului.

50h Accesarea memoriei CMOS.

Principalele intreruperi DOS sunt:

Principala intrerupere DOS este **21h**. Ea înmagazinează practic întreaga componentă BDOS a sistemului de operare DOS. Secțiunea următoare va fi destinată exclusiv acestei intreruperi.

Alte intreruperi DOS:

20h Aceasta este unul dintre apelurile care pot termina execuția unui program. Memoria ocupată va fi eliberată ca efect al acestui apel.

25h Permite citirea fizică de pe disc de la o anumită locație de memorie, începând cu un anumit sector, într-o anumită locație de memorie.

26h Permite scrierea fizică pe disc dintr-o anumită locație de memorie, începând cu un anumit sector.

27h Termină execuția programului curent lăsând rezidentă în memorie o parte sau întreg programul, astfel încât această zonă de memorie să nu fie suprascrisă de un alt program.

28h Întrerupere DOS nedокументată pentru partajarea timpului. Asupra intreruperilor nedocumentate vom reveni ulterior cu explicații.

2Eh (nedокументată). Execută o comandă DOS ca și când ar fi dată de la prompter.

2Fh Funcțiile acestei intreruperi se ocupă cu: multiplexarea resurselor sistemului, gestiunea memoriei extinse (XMS) dacă aceasta există, controlul programelor TSR și.a.m.d.

67h Această intrerupere grupează toate serviciile necesare gestiunii memoriei expandate dacă SC este dotat cu o astfel de componentă hard.

33h Această intrerupere grupează toate funcțiile necesare lucrului cu mouse-ul.

Următoarele intreruperi furnizează o serie de adrese ale unor structuri DOS:

22h Adresa de terminare a programului.

23h Adresa handlerului care tratează tastarea <CTRL/BREAK>.

24h Adresa handlerului care tratează erorile critice apărute în execuție.

Principalele funcții DOS

Așa cum am mai arătat, principala intrerupere DOS este **21h**. Ea înmagazinează practic întreaga componentă BDOS a sistemului de operare DOS. Mai mult, sarcinile unora dintre intreruperile mai sus amintite au fost preluate și uneori extinse de către unele dintre funcțiile acestei intreruperi.

Vom exemplifica, pe categorii, unele dintre funcțiile DOS mai importante. O prezentare exhaustivă a lor ar necesita mult spațiu tipografic. Documentațiile DOS fac o astfel de prezentare, ca și o serie de programe de tip HELP, accesibile pe orice calculator.

Functii de gestiune a memoriei:

48h Alocă un bloc de memorie și returnează un pointer spre începutul acelei zone de memorie.

49h Elibereză o zonă de memorie pentru a o face disponibilă altor programe.

4ah Ajustează spațiul de memorie alocat.

Functii de gestiune a proceselor:

4Bh Încarcă un program pentru a fi executat sub controlul unui program existent. La terminarea execuției programului apelat, controlul se întoarce în programul apelant.

4Ch Termină execuția unui program întorcându-se în command.com sau în rutina apelantă, cu un anumit cod de return (error level) setat în AL.

31h Termină execuția unui program lăsându-l rezident în memorie. Prin intermediul lui AL se va transmite codul de return.

4Dh Această funcție este folosită pentru obținerea codului de return al unui proces lansat de către un program prin apelul funcției 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 Obține adresa de început a PSP-ului în memorie.

Functii 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 operații care folosesc FCB (File Control Block). Este responsabilitatea programatorului să se asigure că dimensiunea acestei zone este suficient de mare pentru operațiile ce urmează a fi făcute.

1Bh Furnizează informații despre tabela de alocare a fișierelor (FAT).

Functii specifice directoarelor și fișierelor:

39h Creează un nou director folosind discul și calea specificată.

3Ah Sterge un director specificând calea spre acesta.

3bh Schimbă directorul curent în cel specificat.

47h Obține un string ASCII reprezentând calea spre directorul curent.

56h Schimbarea numelui unui fișier.

4Eh Caută în directorul specificat sau în cel curent primul nume de fișier care se potrivește cu o specificare generică.

4Fh Caută în directorul specificat sau în cel curent următorul nume de fișier care se potrivește cu o specificare generică.

41h Sterge un fișier din directorul specificat sau din cel curent.

3Dh Deschide un fișier din directorul specificat sau din cel curent. Dacă deschiderea s-a făcut cu succes, returnează un identificator al fișierului (*handle*) pe 16 biți pentru accesarea fișierului de acum înainte.

3Eh Închide un fișier care a fost deschis cu succes. Identificatorul fișierului devine astfel liber pentru a fi folosit ca identificator al altui fișier. Dacă fișierul a fost modificat, data și ora ultimei modificări vor fi actualizate.

3Fh Citește un anumit număr de octeți dintr-un fișier deschis cu succes.

40h Scrie un anumit număr de octeți într-un fișier deschis cu succes.

Intrări / ieșiri cu periferice de tip caracter:

01h Citește un caracter de la intrarea standard și îl afișează la ieșirea standard.

02h Afisează un caracter la ieșirea standard.

09h Tipărește un sir de caractere la ieșirea standard. Acest sir de caractere trebuie să conțină ca și marcaj de sfârșit de sir caracterul '\$'. Acest caracter nu va fi tipărit.

0Ah Citește de la intrarea standard un sir de caractere până la tastarea lui *Enter*.

Informații despre sistem:

30h Furnizează versiunea sistemului de operare.

38h Tratează parametrii dependenți de regiunea geografică.

2Bh Setează data curentă reținută de ceasul sistem.

2Dh Setează ora curentă reținută de ceasul sistem.

Diverse alte funcții:

35h Obține adresa unui handler de intrerupere, sub forma adresă de segment:offset

25h Modifică adresa unui handler de intrerupere.

44h Ansamblu de funcții destinat lucrului la nivel fizic cu diverse tipuri de periferice.

34h Returnează numărul proceselor curente active (nedокументată).

52h Atribuie valori unor variabile DOS (nedocumentată).

5.3. CÂTEVA OBSERVAȚII ASUPRA ÎNTRERUPERILOR 8086

a) Înșiruirea principalelor intreruperi 8086, precum și a principalelor funcții DOS relevă următorul fapt. Numai o parte dintre ele deservesc apariția unui eveniment neobișnuit, intern sau extern. Acestea, evident că vor fi activate la apariția evenimentului-lui, fără a se putea prevedea momentul apariției. Vom spune despre handler-ele din această categorie că sunt activate prin evenimente.

În schimb, cealaltă parte a handler-elor oferă aceleasi servicii ca o bibliotecă de subprograme, apelabile de către programele utilizator. Activarea lor are loc atunci când programele o cer. Handler-ele din această categorie sunt activate prin instrucțiuni speciale de apel de intreruperi. În 5.4 vom trata aceste instrucțiuni.

b) Funcțiile DOS ale intreruperii 21h execută unele sarcini pe care le execută și alte intreruperi. De exemplu, funcția 4ch a intreruperii 21h - terminarea unui program, este realizată și de către intreruperea 20h. Din punct de vedere istoric, funcțiile DOS au apărut ultimele. Ele sunt mai performante și mai fiabile decât intreruperile similare, motiv pentru care în prezent sunt folosite aproape exclusiv numai acestea.

c) După cum s-a văzut deja, există așa-numitele *întreruperi nedocumentedate*. Acestea sunt rezervate spre folosire doar de către proiectanții DOS. Unele dintre ele sunt efectiv rezervate, altele au o serie de sarcini intermediare pentru alte intreruperi, iar altele au roluri mai mult sau mai puțin "obscure" pentru muritorii de rând. Aceste intreruperi sau funcții constituie ținta preocupărilor pentru mulți programatori pasionați. Nu trebuie uitat însă faptul că proiectanții sistemului de operare își rezervă dreptul de a folosi aceste intreruperi pentru dezvoltări ulterioare.

d) O serie de numere de intreruperi sunt în prezent neocupate. Ele pot fi ocupate de către utilizatori prin handlele proprii, după cum vom vedea în capitolul 6. Instrumentele de contaminare a programelor, cunoscute sub numele de "virusi", folosesc de multe ori aceste numere de intreruperi pentru scopurile lor distructive.

5.4. INSTRUCȚIUNI SPECIFICE LUCRULUI CU ÎNTRERUPERI

După cum am arătat în 5.2, un handler de intrerupere poate fi apelat și direct prin intermediul instrucțiunii INT. Instrucțiunea INT, cu sintaxa:

INT n

provoacă activarea handlerului corespunzător intreruperii cu numărul *n*.

Ea realizează patru acțiuni succesive:

- punе în stivă flagurile;
- punе în stivă adresa FAR de revenire;
- punе 0 în flagurile TF și IF;
- apelează, prin adresare indirectă, handlerul asociat intreruperii.

De multe ori, în aplicații este utilă simularea acestei instrucțiuni sau a unei părți din ea. Iată o secvență care realizează simularea ei.

AdRTI dd ... ;Presupunem că conține adresa FAR a handlerului

```
pushf          ;Depune flagurile în stivă
push cs        ;Segmentul adresei de revenire
lea  ax,REV    ;Offsetul adresei de revenire
push  ax
xor  ax,ax
```

```

push ax      ;Se anulează toate flagurile pentru a anula și TF și IF
popf
jmp AdRTI   ;Salt indirect la handler
REV: -----

```

Instrucțiunea INT0 (care nu are operanzi) se comportă în două moduri, în funcție de valoarea flagului OF:

- dacă OF = 1, atunci este echivalentă cu INT_4 (se apelează handlerul întreruperii de depășire aritmetică);
- dacă OF = 0, atunci este echivalentă cu NOP (instrucțiunea ineffectivă, nu a apărut depășire).

De aceea, orice instrucțiune care ar putea provoca depășire este indicat a se programa astfel:

```

add ax,b
into      ;dacă se bănuiește că operanții adunării
           ;precedente ar putea conduce la depășire.

```

Instrucțiunile STI și CLI (care nu au operanzi) acționează asupra flagului de întreruperi IF și prin acesta indică procesorului cum să se comporte la apariția unei întreruperi.

Instrucțiunea CLI (Clear Interrupt) interzice procesorului să recunoască apariția vreunei întreruperi. Apare de obicei la începutul unui handler pentru a interzice perturbarea activității acestuia.

Instrucțiunea STI (Set Interrupt) permite procesorului să recunoască apariția unei întreruperi. Are rolul de a anula efectul instrucțiunii CLI.

Întreruperile nemascabile nu țin cont de flagul IF!

Instrucțiunea IRET (care nu are operanzi), provoacă revenirea dintr-o întrepere. Ea este ultima instrucțiune executată în cadrul handlerului. Efectul ei este invers instrucțiunii INT, adică:

- refac flagurile din stivă;
- revine la instrucțiunea căreia adresa FAR se află în vârful stivei.

De multe ori, în aplicații este utilă simularea acestei instrucțiuni sau a unei părți din ea. Iată o secvență care realizează simularea ei.

```

AdRev dd ?      ;Păstrează temporar adresa de revenire.
-----
```

```

popf      ;Refac 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 și EXE

Sub sistemul de operare MS-DOS există trei tipuri de fișiere care sunt lansabile în execuție: fișiere de tip .BAT, de tip .EXE și de tip .COM. Fișierile de tip BAT sunt fișiere text care conțin comenzi DOS și eventual directive care controlează ordinea de execuție a comenzilor. La lansarea în execuție a unui fișier de tip BAT sunt executate una după alta toate comenziile trecute în fișier, respectându-se semnificațiile directivelor. Fișierile COM și EXE sunt fișiere binare ce conțin instrucțiuni în limbaj mașină. În această secțiune ne vom ocupa de acestea două din urmă.

Din punct de vedere istoric primele fișiere apărute sunt cele de tip COM. Fișierile de tip COM sunt în general fișiere mici având lungimea de maximum 64 octeți.

Fișierele EXE sunt fișiere mari. Cu ajutorul lor se pot descrie programe complexe, de aceea ele tend 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 octeți. Informațiile din PSP sunt utilizabile direct de către sistemul de operare DOS și indirect de către utilizator. În figura 5.1 este indicată structura tabeliei PSP. În utilitarul Norton Guide găsim informații 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 puțin folosite. Ele au fost introduse la prima versiune de DOS, pentru asigurarea compatibilității 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	Semnificație
00h	2	Codul instrucțiunii INT 20h - terminare program
02h	2	Adresa de sfârșit a memoriei ocupate de program
04h	1	rezervat
05h	1	Codul instrucțiunii INT 21h – funcții DOS
06h	2	Memoria disponibilă (număr octeți) 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 întreuperea 23h (<i>CTRL+Break</i>)

12h	4	Adresa FAR a RTI 24h - se utilizează pentru restaurarea vechiului handler dacă programul curent a deturnat cumva întreruperea 24h (<i>Critical error interrupt handler</i>)
16h	22	rezervat
2Ch	2	Adresa de segment a mediului DOS, unde se găsesc variabilele de sistem MS-DOS
2Eh	46	rezervat
5Ch	32	FCB1 și FCB2, câte 16 octeți pentru accesarea fișierelor standard de intrare și ieșire (se evită utilizarea lor la momentul actual – păstrați pentru compatibilitate)
7Ch	4	rezervat
80h	1	Lungimea cozii liniei de comandă
81h	127	Coada liniei de comandă - astfel se pot accesa din limbaj de asamblare parametrii transmiși în linia de comandă la lansarea în execuție a programului

Fig. 5.1. Structura unui PSP

Fiecare program, pe lângă codul lui propriu-zis mai are o zonă de memorie în care este descris contextul în care lucrează programul. Acest context include, printre altele, informații referitoare la:

- numele discului implicit;
- numele directorului implicit;
- calea spre interpretorul de comenzi COMMAND.COM etc.

Această zonă de context este un segment numit *mediu* (*environment*) și PSP-ul conține un pointer la începutul acestuia (la deplasament 2ch). Segmentul de mediu conține siruri de caractere ASCII de forma:

NumeParam	=	ValoareParam	0
-----------	---	--------------	---

NumeParam este un identificator reprezentând o variabilă de mediu (PATH, PROMPT, ABC etc). Semnul "=" separă identificatorul de sirul de caractere *Valoareparam*. Construcția se termină cu un octet conținând valoarea zero.

Se știe că o comandă DOS are forma:

...>numecomanda arg1,...,argn

Porțiunea *arg1,...,argn* se numește *coada liniei de comandă* și ea este memorată în jumătatea a două a tăbelei PSP.

5.5.2. Structura unui program EXE

Un program de tip EXE poate avea oricără segmente de tip cod, date sau stivă. Acestea pot fi prezente toate în memorie sau numai o parte dintre ele. Prezența optională a unor segmente se realizează prin 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 execuției unui program este activ un segment de cod, un segment de date și un segment de stivă. Toate segmentele sunt plasate de regulă după PSP, însă ordinea lor nu este importantă. Stiva nu are implicit 64 kocete 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 și sunt plasate în ordinea cod, date, stivă.

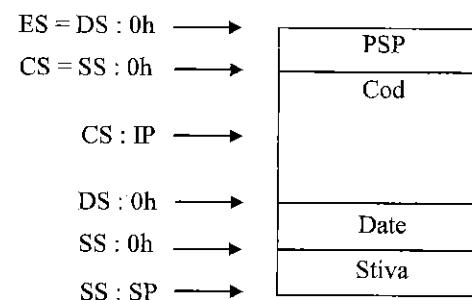


Fig. 5.2. Structura unui program EXE în memorie

În momentul încărcării în memorie a unui program EXE, se creează PSP, și se depun în memorie segmentele. Se încarcă CS cu adresa segmentului de cod ce trebuie să fie activ primul. Prima instrucție executabilă poate să apară oriunde în segmentul de cod punctat de CS, CS:IP va puncta spre această instrucție.

La încărcare ES și DS se inițializează și punctează la începutul PSP. Ulterior, programatorul are obligația să încarce (cel puțin) registrul DS cu adresa segmentului de date curent. De aceea în figura 5.2, DS:0h apare în două poziții.

Registrul SS se va încărca automat cu adresa de început a segmentului de stivă, dacă este declarat un astfel de segment. Dacă nu, atunci SS va primi aceeași valoare ca și CS, considerându-se o stivă de lungime 64 Ko. Registrul SP va puncta spre ultimul cuvânt al segmentului de stivă. Din această cauză, în figura 5.2 SS:0h apare în două poziții.

Dăm în continuare programul E.ASM, care tipărește numărul unităților de dischetă din sistem. El va avea declarată o stivă de 512 octeți (200h).

```

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 configurația lui AX precizează dacă există unități de
            ;dischetă instalate (valoarea 1) sau nu (valoarea 0)

    mov bx, ax ;reținem în bx valoarea lui ax, pentru a păstra configurația obținută
    and bx, 1 ;forțăm la zero valoarea bitilor 1-15 din configurația 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 afișarea numărului unităților de dischetă din sistem
    mov ah, 9
    int 21h ;tipărire pe ecran a mesajului t1
            ;în urma apelului întreruperii 11h, dacă bitul 0 din configurația lui
            ;AX este 1, bitii 6 și 7 din această configurație reprezintă numărul de
            ;unități de dischetă din sistem, mai puțin una
            ;păstrăm în al valoarea bitelor 6 și 7, iar valoarea celorlalți biti o
            ;forțăm la zero

    and al, 0C0h

    mov cl, 6
    shr al, cl ;shiftăm spre dreapta cu 6 poziții configurația din al pentru a obține
            ;în al numărul format din cei doi biți

    add al, 1+'0' ;se adună valoarea 1 la valoarea din al pentru a obține în al numărul
            ;de unități de dischetă din sistem; deoarece acest număr este format
            ;dintr-o singură cifră (e reprezentat pe 2 biți), îl vom transforma în
            ;caracterul corespunzător cifrei adunând la valoarea sa codul ASCII
            ;al caracterului 0

    mov dl, al ;vom tipări caracterul astfel obținut cu funcția 02h a întreruperii 21h
    mov ah, 02h
    int 21h ;se tipărește pe ecran numărul de unități de dischetă din sistem

```

Cap.5. Întreruperi.

```

        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 execuție cu TD a lui E.EXE, registrii au avut valorile:

DS = ES = 628B	CS = 62BE	IP = 0000
SS = 629B	SP = 01FE	

După execuția primelor două instrucțiuni, registrul DS a primit valoarea 62BB.

Un același program, în format EXE are dimensiuni mult mai mari decât echivalentul lui în format COM. Acest fapt se petrece deoarece fiecare fișier 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 implicate ale registrilor la intrare, a se defini segmentele curente la lansarea în execuție etc. Structura antetului EXE pe disc este dată în figura 5.3.

Deplasament	Lungime	Semnificație
00h	2	Semnatura EXE: 5A4Dh (codul 'MZ' Mark Zbirkowski)
02h	2	Lungime fișier mod 512
04h	2	Lungime fișier 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	Pozitia relativă a segmentului SS (paragrafe)
10h	2	Valoarea inițială a registrului SP
12h	2	Suma de control (Checksum) a fișierului
14h	2	Valoare inițială a registrului IP
16h	2	Pozitia relativă a segmentului CS (paragrafe)

18h	2	Adresa disc a tabelii de relocare (TR) (uzual 1Ch)
1Ah	2	Număr de suprapunere (pentru overlay) (2 octeți)
1Ch	?	Zonă rezervată, parte a antetului
TR		Tabela de relocare:
		Offset1 Segment1 ... OffsetNAR SegmentNAR

Fig. 5.3. Structura antetului EXE pe disc

Este util să detaliem puțin mecanismul de încărcare și lansare în execuție a unui program EXE. Ordinea segmentelor este fixată de către editorul de legături. Prima acțiune a încărcătorului este crearea PSP. Urmează apoi încărcarea segmentelor. Ele vor fi plasate în ordinea dată de către editorul de legături. De regulă, segmentele sunt plasate începând de la PSP+100h (deci imediat după PSP). Să notăm cu StartSeg adresa de unde începe încărcarea segmentelor.

Așa cum am văzut în capitolul 3, unele instrucțiuni citează ca operand un nume de segment. De exemplu, sevența:

```
mov ax, SegmentDeDate
mov ds, ax
```

Încarcă registrul DS cu adresa segmentului cu numele SegmentDeDate. Editorul de legături plasează în codul primei instrucțiuni **MOV** valoarea care localizează segmentul respectiv. În urma încărcării segmentelor în memorie, valorile prin care se identifică segmente trebuie mărite cu valoarea StartSeg.

Acest proces poartă numele de operatie de relocare. Aceasta este necesară deoarece deplasamentele sunt determinabile ca și constante la momentul asamblării, însă adresele de segment nu. Ca urmare, elementele relocabile sunt de fapt operanții din cadrul programului care reprezintă adrese de segment și care trebuie acum ajustate cu o valoare rezultată din plasamentul concret în memorie al programului EXE (valoarea StartSeg). De exemplu în cadrul instrucțiunilor:

```
mov ax, data sau mov bx, seg a
```

elementele relocabile sunt *data* și respectiv *seg a*.

Octeții 6 și 7 din antet conțin numărul total de astfel de citări ale numelor de segmente (număr notat NAR – Numărul Adreselor de Relocare - în figura 5.3). Octeții 24 și 25 (adresa 18h) din antet indică poziția în antet (notată TR în figura 5.3), identică cu poziția în fișierul disc, a începutului unei 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 și păstra integritatea antetului disc. Astfel, se consideră întregul fișier ca o succesiune de cuvinte. Conținuturile negate ale acestor cuvinte sunt însumate modulo 65536, rezultatul obținut fiind acest checksum. La încărcarea programului în memorie, sistemul de operare DOS repetă acest calcul și compară cu ceea ce este în checksum, refuzând încărcarea în caz de neconcordanță.

După acest control, se creează PSP. Apoi sunt încărcate segmentele și se actualizează adresele relocabile. În sfârșit, sunt încărcate din antet registrii CS, SS, IP și SP așa după cum am arătat mai sus. Ca efect al încărcării CS:IP programul este lansat în execuție.

Descriuind într-o manieră algoritmică procesul de încărcare în memorie a unui program EXE, putem identifica următorii pași:

1. Se creează în memorie o structură de date PSP (Program Segment Prefix) de lungime 256 de octeți.
2. Se alege o adresă de la care începând se va încărca programul (de obicei aceasta este sfârșitul PSP). Fie această adresa StartSeg:

$$\text{StartSeg} := \text{adresa PSP} + 100h$$

3. Se încarcă la StartSeg porțiunea din fișierul EXE de după antetul de pe disc.

4. Se efectuează operația de relocare: pentru fiecare adresă de relocare AR (adică pentru fiecare adresă unde se găsește un element ce trebuie relocat), la conținutul adresei AR+StartSeg se adună StartSeg; în notație C acest lucru se poate exprima

$$[\text{AR}+\text{StartSeg}] += \text{StartSeg}$$



5. Se alocă dacă este posibil memoria necesară în plus în funcție de valorile din antet (de obicei pentru heap).

6. Se inițializează registrii astfel:

$$\begin{aligned} \text{CS} &:= \text{CS relativ (valoarea din antetul EXE)} + \text{StartSeg} \\ \text{IP} &:= \text{IP inițial (valoarea din antetul EXE)} \\ \text{SS} &:= \text{SS relativ (valoarea din antetul EXE)} + \text{StartSeg} \\ \text{SP} &:= \text{SP inițial (valoarea din antetul EXE)} \\ \text{ES} &:= \text{DS} := \text{adresa de început a PSP} \end{aligned}$$

5.5.3. Structura unui program COM

Un fișier de tip COM are o structură simplă. El conține imaginea (binară) a conținutului ce va fi încărcat în memorie după PSP pentru a obține un program lansabil în execuție. În figura 5.4 este ilustrată imaginea din memorie a unui program COM, cu indicarea valorilor inițiale ale registriilor de segment și a registratorului IP. Un program COM nu are antet pe disc! Numai programele EXE au așa ceva.

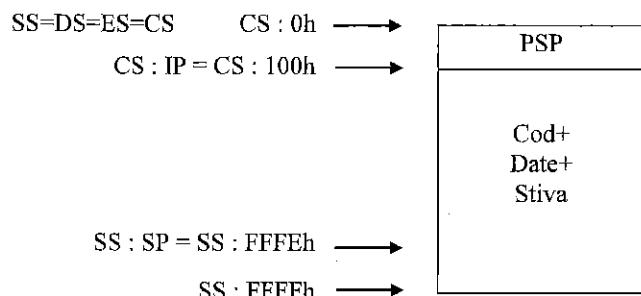


Fig. 5.4. Imaginea unui program COM în memorie

În momentul încărcării în memorie a unui program COM se creează PSP, se inițializează toți registrii de segment ca în figura 5.4 și se dă controlul instrucțiunii aflate la adresa 100h, adică primul octet care se află după PSP. Pointerul de stivă SP are ceea mai mare adresă posibilă, adică FFFEh, în ideea că programul are 64 kocete și stiva urcă spre adrese mici.

Pentru ca un program scris în limbaj de asamblare să fie de tip COM trebuie îndeplinite următoarele cerințe:

1. Programul trebuie să conțină un singur segment (toate referirile la cod, date și stivă se fac în cadrul aceluiași segment); în consecință directiva **assume** este de forma:

assume cs:<nume>,ds:<nume>

unde **<nume>** este același atât pentru CS cât și pentru DS.

2. Imediat după declarația de segment trebuie să apară o directivă:

ORG 100h

indicând faptul că instrucțiunile și datele încep de la adresa 100h. După ORG urmează o instrucțiune etichetată care trebuie să fie prima instrucțiune executabilă. Numele acestei etichete trebuie să apară în linia END a programului.

3. Datele pot fi plasate oriunde între instrucțiuni, singura condiție (dependentă numai de programator) este să nu interfereze datele cu instrucțiunile. Pentru aceasta programatorul trebuie să izoleze zonele de date prin instrucțiuni de salt corespunzătoare.

4. Registrii de segment sunt inițializați automat (ca în figura 5.4), deci utilizatorul nu mai trebuie să-i încarce cu valori inițiale (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 secțiunea anterioară, care tipărește numărul unităților 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 ;Așa se evita interferarea datelor cu codul
t1 db 'In sistem există $'
t2 db ' unități de disc $'
t3 db ' Nu există unități de discheta instalată în 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 condițiile de mai sus), cu ajutorul comenzi DOS EXE2BIN:

...>EXE2BIN C.EXE C.COM

Dacă programul EXE nu respectă condițiile impuse unuiu COM și are spre exemplu elemente relocabile, comanda Tlink va furniza eroarea:

"Fatal: Cannot generate COM file: segment relocatable items present".

5.5.4. Depanarea programelor .EXE și .COM

În ceea ce privește depanarea unui program .EXE, folosirea opțiunilor /zi pentru asamblare și /v pentru linkeditare,

```

>tasm /zi prb
>tlink /v prb

```

provoacă includerea de informații simbolice pentru depanare la nivelul codului obiect și respectiv la nivelul celui executabil. Includerea informației simbolice înseamnă reținerea asocierii adresă de memorie - nume simbolic pentru fiecare etichetă din program. Acest lucru înseamnă că în cadrul unei sesiuni de depanare putem identifica referirea la o etichetă prin numele său și nu doar prin adresa care o reprezintă. În mod implicit, fără specificarea opțiunilor de mai sus, asamblorul nu include informațiile simbolice în formatul obiect pe care îl generează.

De exemplu, dacă avem definită variabila

```
a dw 5
```

fișierele .obj și .exe vor conține informația simbolică ce face asocierea între numele a și deplasamentul la care se află această variabilă (de exemplu 0005h). Desigur că vom prefera ca în momentul depanării să vedem simbolul a mai degrabă decât deplasamentul acestuia. Pentru instrucțiunea:

```
mov ax, a
```

vom obține astfel la depanare:

```

mov ax, a
în loc de
    mov ax, [0005h]

```

Includerea acestor informații simbolice pentru depanare în fișierul .exe face ca dimensiunea acestuia să fie mai mare decât dacă nu ar conține aceste informații.

În ceea ce privește depanarea unui program .COM, care nu poate avea o dimensiune mai mare de 64 kocete (din care 256 octeți sunt folosiți pentru PSP), dimensiunea acestuia nu permite includerea informațiilor simbolice pentru depanare. Aceasta este motivul pentru care depanarea unui program .COM nu se poate face la nivel simbolic.

Prezentăm în continuare două programe echivalente semantic care efectuează suma a două numere. Pe de o parte avem fișierul pexe.asm din care vom obține un program .EXE și fișierul pcom.asm din care vom obține un program .COM. Informațiile obținute la depanare sunt prezentate comparativ.

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
```

Obținere 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
```

Obținere pcom.com:

```
tasm /zi pcom
tlink /t pcom
td pcom
```

Depanare pcom.com
(fără includerea informațiilor 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, proiectanții sistemului de operare MS-DOS au ales varianta memorării adreselor acestor rutine într-un vector aflat la o locație de memorie cunoscută (0000:0000). Acest mod de gestiune a întreruperilor permite un lucru important și anume posibilitatea de a redefini comportamentul unui handler în timpul funcționării 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 funcționalitate care lipsește în cadrul sistemului de operare, pentru a îmbunătăți sau optimiza unele rutine BIOS sau pentru a modifica comportamentul sistemului de operare în cazuri particulare. Pașii 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 (optional – 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 funcției DOS 25h;

Modificarea directă în tabela de vectori de întreruperi se poate face cu o secvență de cod similară cu cea descrisă mai jos. Această secvență de cod inițializează 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
```

Secvența 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 instrucțiuni MOV WORD PTR, transferul se face la noul deplasament, însă în cadrul vechiului segment. Dezactivarea întreruperilor (stă) 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 apariția unui eveniment (ceas, tastatură etc). Ele pot apărea în orice moment și de aceea se numesc *asincrone*. După actualizarea noii adrese a rutinei de tratare a întreruperii se reactivează întreruperile pentru a permite funcționarea corectă a sistemului. Având în vedere că adresa unui handler se reprezintă pe 4 octeți (un cuvânt pentru offset și un cuvânt pentru adresa de segment), rezultă că deplasamentul în cadrul tabelei pentru întreruperea 77h se află la adresa 4*77h. Conform convenției de reprezentare a datelor în memorie (octetul cel mai nesemnificativ se află la adresa mai mică) înseamnă ca începând cu deplasamentul 4*77h vom stoca offset-ul noului handler, iar la adresa 4*77h+2 adresa de segment a acestuia.

Modificarea adresei RTI cu ajutorul funcției DOS 25h se face printr-o secvență de cod similară cu cea prezentată mai jos. Vom modifica din nou RTI 77h:

```
push ds          ;salvează registrul DS întrucât îl vom modifica
mov ax, 2577h    ;AH – 25h, AL – numărul întreruperii
mov dx, seg NewHandler
mov ds, dx
mov dx, offset NewHandler
int 21h
pop ds          ;reface registrul DS (înapoi spre segmentul de date)
```

Secvența de cod de mai sus depune în registrul DS adresa de segment a noului handler, în DX deplasamentul noului handler, în AL numărul întreruperii și apelează funcția DOS 25h. În acest caz funcția DOS 25h dezactivează întreruperile în decursul modificării adresei RTI. Chiar dacă această secvență de cod pare mai complicată decât cea care accesează direct tabela vectorilor de întrerupere, ea este mai sigură. Aceasta deoarece multe dintre aplicațiile DOS monitorizează modificările aduse tabeliei vectorilor de întrerupere prin intermediul DOS (funcția 25h). Modificarea directă a tabeliei de vectori scurt-circuitează mecanismul de monitorizare și astfel aceste aplicații nu își mai pot da seama de modificările apărute, fapt ce poate duce la funcționarea lor incorectă.

Pentru a putea refa, 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 secvențe de cod ilustrează modul în care putem realiza acest lucru, atât prin accesul direct la tabela vectorilor de întrerupere cât și prin utilizarea funcției DOS 35h:

```
oldint77 dd ?      ;rezervare dublu cuvânt pt. memorarea adresei vechii RTI
...
...
```

```
mov ax, 0
mov es, ax
cli
mov ax, word ptr es:[4*77h]      ;depune offset-ul handler-ului original în AX
mov word ptr cs:oldint77, ax      ;salvează offset-ul handler-ului original
mov ax, word ptr es:[4*77h+2]
mov word ptr cs:[oldint77+2], ax      ;salvează adresa de segment a handler-ului original
st
```

Presupunând că am rezervat spațiu de stocare pentru adresa vechiului handler în segmentul de cod (oldint77), secvența de mai sus salvează în dublul cuvânt de la adresa oldint77, adresa rutinei de tratare a întreruperii 77h. Același lucru se poate face folosind funcția DOS 35h:

```
oldint77 dd ?
...
...
mov ax, 3577h
int 21h
mov word ptr cs:oldint77, bx      ;se obține în ES:BX adresa handler-ului curent
mov cs:[oldint77+2], es
```

În general există puține cazuri în care înlocuim complet funcționalitatea unei rutine de tratare de întrerupere. De cele mai multe ori modificăm doar comportamentul unei funcții a întreruperii sau modificăm comportamentul în anumite cazuri speciale (de exemplu rescriem handlerul întreruperii de tastatură 09h pentru a putea intercepta activarea unor anumite combinații de taste). În toate aceste cazuri vom trata prin codul rescris cazul care ne interesează, iar pentru restul scenariilor posibile vom apela handler-ul original al întreruperii. Acesta este un alt motiv pentru care este necesară salvarea adresei handler-ului original. Procesul duce la un mecanism de *înlănțuire* a apelurilor către handler-ele de întrerupere. Acest mecanism este extrem de util pentru buna funcționare a programelor *TSR* (*Terminate and Stay Resident*) după cum vom vedea în continuare. Această partajare/înlănțuire a vectorilor de 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
...
iret            ;cod executat după apelul handler-ului original
;terminarea execuției handler-ului curent (cel nou) și
;întoarcerea în contextul aplicației întrerupe
```

în cazul în care dorim să apelăm vechiul handler și apoi să continuăm execuția handler-ului nostru (folosind eventual efectul produs de handler-ul original), sau:

```
jmp dword ptr cs:oldint77 ;salt la handler-ul original al intreruperii
```

în cazul în care după apelul handler-ului original controlul se redă aplicației intrerupte fără a mai reveni în codul handler-ului nostru. În primul caz se poate remarcă simularea unei instrucțiuni *int* prin plasarea flag-urilor pe stivă și apelul far al handler-ului. Este necesară plasarea flag-urilor pe stivă întrucât handler-ul original se termină printr-o instrucțiune *iret* (echivalentă cu o secvență: *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 inițial de către sistem al handler-ului prin instrucțiunea *int* corespunzătoare sau la generarea intreruperii de către sistem.

Se poate observa de asemenea că atât la apelul de tip *call* cât și 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 registri CS:IP care pointează la instrucțiunea curentă, restul registriilor au valorile pe care le aveau în aplicația intreruptă (în cazul unui handler care gestionează un eveniment sistem sau o intrerupere hardware) sau cele pregătite special la apelul intreruperii (în cazul în care intreruperea a fost generată cu un apel *int*). Este greșit să presupunem la intrarea în handler-ului nostru că registrul DS este setat spre segmentul care memorează variabilele din cadrul codului nostru.

6.2. PROGRAME TSR

Aplicațiile MS-DOS obișnuite au o natură *transientă* (temporară). Ele sunt încărcate în memorie, executate iar apoi se termină urmând ca DOS să folosească memoria pentru următorul program care este lansat în execuție. Programele *rezidente* sunt aplicații care urmează în general aceeași regulă cu excepția 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. Aceste tip de aplicații sunt cunoscute și sub numele de aplicații *Terminate and Stay Resident (TSR)*, și sunt folosite în general pentru a introduce facilități de multitasking în cadrul unui sistem de operare monotask. Până la apariția sistemelor de operare din familia Windows, aplicațiile rezidente erau singura metodă prin care mai multe aplicații puteau să coexiste în memorie în același timp. Chiar dacă apariția sistemelor din familia Windows a diminuat sau înălțurat în unele cazuri necesitatea programelor rezidente și a procesării în fundal oferită de acestea, ele mai sunt încă folosite la aplicațiile native DOS, antiviri, drivere DOS, *patch-uri live*, etc.

6.3. HARTA DE MEMORIE DOS ȘI PROGRAMELE TSR

La demararea (procesul de boot) al sistemului de operare DOS (pe un calculator care rulează nativ sistemul MS-DOS) acesta initializează anumite zone de memorie predefinite și 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 folosește mecanismul de „mașină virtuală” pentru rularea aplicațiilor DOS și a interpretorului de comenzi. Sistemul de operare gazdă (Windows NT) simulează o mașină hardware virtuală în cadrul căreia rulează apoi sistemul de operare MS-DOS. Deși există unele diferențe din punct de vedere al accesării componentelor hardware, conceptele pe care le vom descrie în cadrul acestui capitol sunt valabile și pentru sistemele DOS emulare virtual. Acolo unde acest lucru nu este valabil vom diferenția explicit acest lucru.

Harta de memorie a unei mașini ce rulează sistemul de operare DOS arată, după încărcarea acestuia, similar cu cea prezentată în figura de mai jos:

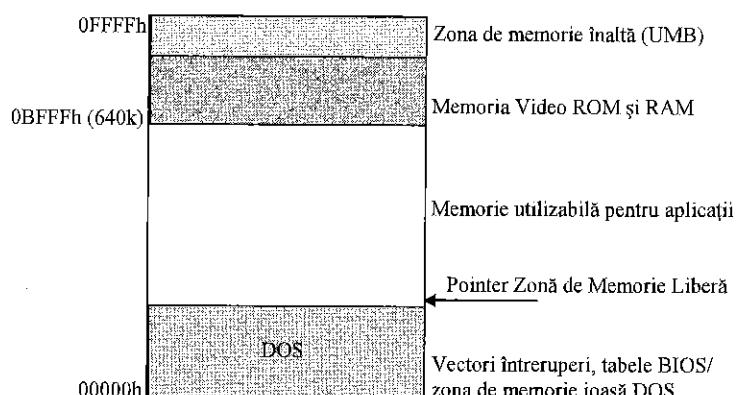


Fig. 6.1. Harta de memorie a sistemului de operare DOS după încărcare, fără aplicații active.

DOS gestionează un pointer care indică zona liberă de memorie după încărcarea sistemului de operare. Această zonă se află după spațiul de memorie alocat tabelor vectorilor de intrerupere, tabelelor BIOS și DOS și este limitată superior de granița celor 640K practic adresabili pe sistemele DOS.

Notă:

Procesoarele 8086, 80186 dispuneau la introducerea lor pe piață de bus-uri de adresă cu 20 canale (20 biți). Consecința directă a fost limitarea memoriei maxime adresabile la un spațiu de adresă cuprins între 0 și $2^{20} = 1024Kb = 1Mb$. În mod real adresarea se face folosind componentele segment și offset, ambele reprezentate pe 16 biți. Fiecare segment conține astfel 2^{16} octeți = 64K. Din cei 20 biți a unei adrese fizice mai rămân 4 pentru stabilirea adresei de segment din cei 16 biți a registrului de segment. Restul combinațiilor între registrii de segment și offset au ca rezultat valori de adrese fizice ce pot fi obținute deja cu politica descrisă mai sus. Mai precis din cele 2^{32} combinații posibile de valori segment+offset doar 2^{20} sunt unice. Din spațiul maxim de 1Mb adresabil, zona de memorie de peste 640Kb are în cadrul MS-DOS întrebunțări speciale.

Nativ MS-DOS nu poate rula mai multe aplicații simultan (este un sistem de operare monotask). MS-DOS întotdeauna încarcă în memorie programul de executat la adresa indicată de pointerul spre *Spațiul de Memorie Liberă* și îi alocă întreaga zonă de memorie liberă, până la adresa 0BFFFh. Practic întreaga memorie RAM disponibilă este alocată programului în curs de execuție. Harta de alocare a memoriei MS-DOS atunci când în sistem rulează o aplicație este prezentată în figura 6.2.

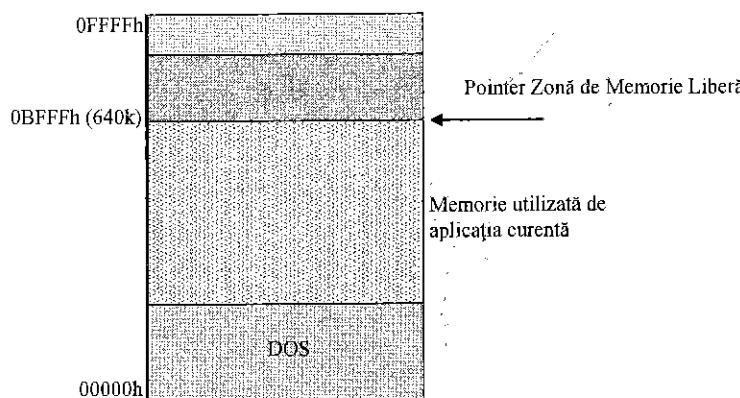


Fig. 6.2. Harta de memorie a sistemului de operare DOS cu o aplicație activă în sistem.

Atunci când o aplicație se termină prin invocarea funcției DOS 4Ch, sistemul de operare eliberează întreaga zonă de memorie alocată acesteia și resetează pointerul *Zonă de Memorie Liberă* imediat deasupra componentei MS-DOS din zona joasă de memorie. Funcția DOS 31h schimbă comportamentul la terminarea unei aplicații în modul ilustrat în figura 6.3.

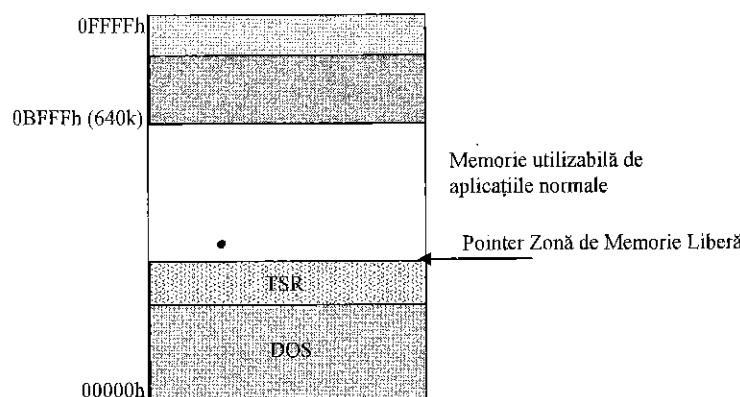


Fig. 6.3. Harta de memorie DOS după terminarea unei aplicații rezidente.

În acest caz DOS face o excepție: nu eliberează întreaga cantitate de memorie. Acest apel de tip *terminate and stay resident* menține alocat în memorie blocul de la începutul zonei alocate aplicației având dimensiunea stabilită în registrul DX. Dimensiunea acestui bloc de memorie este specificată în paragrafe (un paragraf = 16 octeți). La apelul funcției 31h, DOS setează de fapt pointerul *Zonă de Memorie Liberă* astfel încât să indice locația de memorie aflată la $DX * 16$ octeți „deasupra” PSP-ului programului. La execuția unei alte aplicații în sistem, DOS va aloca doar memoria care începe la noua poziție a pointerului spre *Zonă de Memorie Liberă* indicată în figura 6.3. Acest lucru permite protejarea spațiului de memorie în care se află încărcat programul TSR. Harta alocării memoriei după încărcarea unui program TSR și a unei aplicații normale DOS este ilustrată în figura următoare:

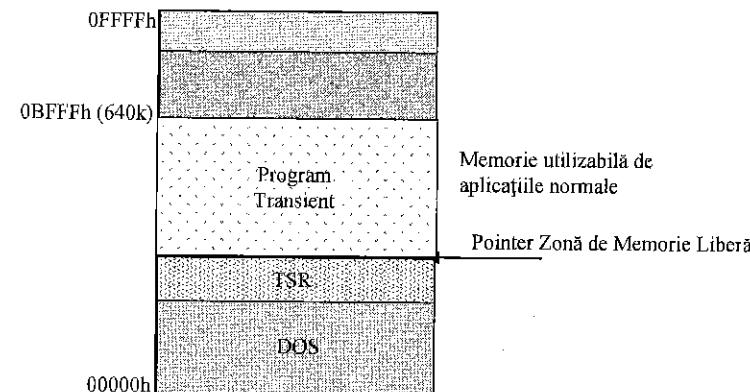


Fig. 6.4. Harta alocării memoriei DOS după încărcarea unui TSR și a unei aplicații tranziente.

La terminarea execuției aplicației tranziente (care este încărcată „peste” TSR), DOS va elibera doar memoria alocată acesteia, protejând încă o dată spațiul de memorie alocat TSR-ului.

Aspectul cel mai important la utilizarea funcției DOS 31h este calcularea corectă a numărului de paragrafe care trebuie să rămână rezidente. Cea mai mare parte a programelor TSR conțin două secțiuni: o secțiune *tranzientă* și o secțiune *rezidentă*. Partea tranzientă conține datele, programul principal și rutinele de suport care se execută atunci când lansăm programul în linie de comandă. Această porțiune de cod nu se mai execută în general niciodată după instalarea programului TSR în memorie. În consecință ea nu trebuie să rămână alocată în memorie la instalarea unui TSR este de fapt un octet mai puțin pentru restul programelor care vor rula în sistem.

Partea rezidentă este cea care rămâne în memorie și implementează funcționalitățile oferite de TSR. Întrucât PSP-ul este construit în memorie de către sistemul de operare imediat înaintea primului octet al programului, dimensiunea acestuia trebuie considerată atunci când calculăm numărul de paragrafe rezidente. Harta memoriei unui program TSR încărcat trebuie să fie una similară cu cea ilustrată în figura 6.5. Pentru ca un TSR să poată funcționa corect este necesar să organizăm codul

și datele de așa natură încât secțiunile rezidente ale programului să fie încărcate la adresele mici de memorie, iar cele tranziente la adresele înalte de memorie. În general toate asamblările oferă facilități care permit controlul ordinii de încărcare a segmentelor în memorie. Cea mai simplă soluție pentru ordinea de încărcare a segmentelor în memorie și cea care funcționează cu toate asamblările este aceea de a organiza atât datele cât și codul într-un singur segment care apare primul în fiecare fișier sursă al programului. În acest caz link-editorul va comoda definițiile din toate fișierele ale segmentului respectiv într-un singur segment care va fi primul încărcat în memorie.

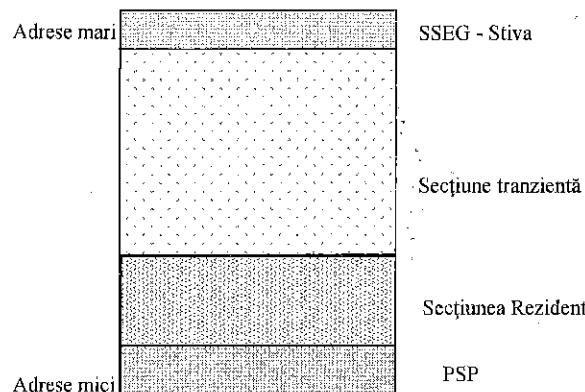


Fig. 6.5. Organizarea în memorie a unui program TSR.

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 secțiunea 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 intreruperi hardware. După intrarea în secțiunea rezidentă, rutina apelată poate presupune că anumiți registri conțin parametri cu anumite valori. Nu putem însă face nici o presupunere aici privitoare la valorile stocate în registrii de segment. Singurul registru de segment care conține o valoare semnificativă semantic este registrul de cod CS. Restul registriilor de segment vor trebui inițializați cu valori semnificative pentru aplicația TSR. La terminarea rutinei TSR aceasta va restaura valorile registriilor de segment pentru a permite execuția corectă a aplicației care a fost intreruptă.

6.4. TSR-URI ACTIVE ȘI TSR-URI PASIVE

Microsoft identifică două tipuri mari de programe TSR: active și pasive. Un TSR pasiv este activat printr-un apel explicit *call* din aplicația care rulează în sistem. Un TSR activ este acela care răspunde la intreruperile hardware generate de sistem pe parcursul funcționării acestuia.

TSR-urile sunt cu mici excepții rutine de tratare de intrerupere. TSR-urile active sunt în general RTI-uri ale unor intreruperi hardware generate de sistem, iar cele pasive sunt handle ale unor intreruperi 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 aplicație către imprimantă vom redirecta intreruperea 17h și vom intercepta toate datele destinate imprimantei. Putem să adăugăm funcționalități noi unei rutine BIOS prin înlățuirea unei RTI proprii în lanțul de handle ale vectorului de intrerupere corespunzător. De exemplu putem implementa o funcție BIOS nouă pentru interfața video prin redirectarea intreruperii 10h și gestiunea unui nou cod de funcție 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 concluziv în această direcție este implementarea serviciilor de gestiune a mouse-ului. Acestea sunt implementate de un driver de tip TSR folosind intreruperea 33h.

TSR-urile active sunt implementate pentru a defini direct serviciile unei intreruperi hardware, sau pentru a permite activarea periodică a TSR-ului prin înlățuirea în vectorul de intrerupere a unei intreruperi hardware. Așa cum am văzut deja în secțiunile anterioare există o multitudine de intreruperi hardware care sunt generate de către sistem la apariția unui eveniment. Dintre acestea amintim intreruperea de ceas, intreruperea de tastatură, etc. La fiecare eveniment specific aceste intreruperi sunt activate și permit TSR-ului care s-a înălțuit în vectorul acestora de intrerupere să fie activat. Un exemplu concluziv în acest sens sunt programele de tip *pop-up* care se lipesc în general de intreruperea de tastatură. Apăsarea unei taste apelează handler-ul intreruperii 9h și permite activarea TSR-ului. Acesta citește de obicei tasta sau combinația de taste apăsate și le compară cu o secvență care îi este cunoscută. În cazul în care combinația de taste este cea specifică TSR-ului acesta se activează. Virușii sunt alte exemple de programe TSR active. Pentru a putea să își ducă la îndeplinire activitățile ne-ortodoxe ei se înălțuie în vectorii unor intreruperi ale căror apeluri le permit să fie activați. În acest sens majoritatea virușilor se inserează în vectorii intreruperii 21h și 13h pentru a detecta orice încercare de scriere/citire dintr-un fișier sau pe/de pe disc. De fiecare dată când aceste operații sunt executate ei își inserează codul malefic în cadrul fișierului (mai ales dacă este vorba de un fișier executabil).

Spunem în general că un TSR este activ dacă el folosește cărora o componentă activă: cel puțin 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 și va afișa un mesaj corespunzător pe ecran. Desigur un adevărat economizor ecran va afișa în general o imagine animată sau va dezactiva ieșirea video pentru a nu utiliza inutil monitorul calculatorului.

Pentru a implementa funcționalitățile acestui program vom redirecta intreruperea 1Ch (intrerupere 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 acțiuni periodice. Spre deosebire de întreruperea 8h a cărei redirectare necesită *colaborarea* cu controller-ul de întreruperi al sistemului și programarea acestuia, întreruperea 1Ch nu necesită acest lucru. Folosim întreruperea 1Ch pentru a măsura intervalul de timp scurs între interacțiunile 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 și la eliberarea acesteia. Codul scan citit la eliberarea unei taste este codul scan al tastei respective 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 secțiunea de date rezidentă
```

```
msg      db 13,10,'Tastatura inactivă în 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 conține adresele originale ale întreruperilor 1Ch și 9h

```
oldInt1C dd ?
oldint9  dd ?
```

;Handler întrerupere 1Ch (timp).

;Acestă rutină incrementează un contor la fiecare apel. Atunci când valoarea contorului depășește ;o valoare prag se generează afișarea unui mesaj pe ecran. De fiecare dată este apelat handler-ul ;original al întreruperii 1Ch pentru a permite și celorlalte aplicații TSR potențiale să își desfășoare ;corect activitatea

```
int1c proc far
    cli
    inc word ptr cs:crt_timeout       ;incrementează contorul de apeluri
```

```
cmp cs:crt_timeout,timeout
```

;dacă contorul depășește numărul de apeluri
;echivalent unui interval de *t_out_sec* se cere afișarea
;mesajului pe ecran.

```
jb orig1c
```

;în caz contrar trece controlul la secțiunea de cod care
;apeleză handlerul original

;salvăm pe stivă regiștri pe care îi modificăm pentru a-i putea refa după afișarea mesajului.
;Handler-ele întreruperilor generate de sistem trebuie să păstreze în general nealterat mediul, adică
;regiștri și stiva pentru a permite reluarea corectă a execuției aplicațiilor întrerupte.

```
push ax
push bx
push cx
push bp
push es
```

```
push cs
pop es
mov ah, 03h
mov bh, 0
int 10h
```

```
mov ax,1301h
```

```
mov bl,07h
```

```
mov bh,0
mov cx, msg_len
mov bp, offset msg
int 10h
mov cs:crt_timeout, 0
pop es
pop bp
pop cx
pop bx
pop ax
```

orig1c:

```
sti
jmp dword ptr cs:oldint1c
```

int1c endp

;Funcția 03h a întreruperii 10h (video)
;numărul paginii video (mod de afișare text)

;Determină poziția actuală a cursorului pe ecran raportând-o
;în perechea de regiștri DH (linia) și DL (coloana)

;Funcția 13h a INT 10h afișează pe ecran la poziția
;determinată (linie,coloană) mesajul specificat în ES:BP.
;AL=1 – subserviciul 1 presupune actualizarea cursorului și
;utilizarea atributului (culoare font/fond=alb/negru)
;specificat în BL pentru tipărire caracterelor șirului.

;numărul paginii video (mod de afișare text)

;reinițializează contorul de apeluri la 0
;reface regiștri modificați

;apel la handler-ul original al întreruperii 1Ch (fără revenire
;la rutina curentă)

;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 execuției acestui handler. Nu vrem ca o nouă întrerupere 9h să fie generată pe parcursul execuției handler-ului, care să întrerupă chiar execuția 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
    push ax
    in al, 60h
    cmp al, 80h
    ;dezactivăm întreruperile
    ;salvăm pe stivă registrul AX pe care îl vom modifica
    ;citim din codul scan al tastei acționate
    ;verificăm că este vorba de apăsarea unei taste și nu
    ;eliberarea acesteia
    cmp al, 00h
    ;refacem conținutul registrului AX modificat la citirea tastei
    ;în cazul în care este vorba de o tastă eliberată trecem la
    ;apelul handler-ului original al întreruperii 9h
    ;s-a apăsat o tastă => resetăm contorul de timp
    mov cs:crt_timeout, 0
    orig9:
        ;reactivăm întreruperile. Handler-ul original al într 9h le va
        ;dezactiva singur dacă are nevoie
        sti
        jmp dword ptr cs:oldint9
    int9 endp
    ;salt fără revenire la RTI originală a întreruperii 9h
    ;marchează sfârșitul secțiunii rezidente de cod
RezEnd equ $

msg_success db 'IdleKeybMon - monitorizeaza timpii morți la tastatura', 13, 10, '$'

begin:
    push cs
    pop ds
    mov ax, 351Ch
    int 21h
    ;programul principal care instalează TSR-ul în memorie.
    ;Acesta face parte din secțiunea tranzientă de cod a TSR-ului.
    ;initializează regiștri de segment utilizati. DS conține inițial
    ;adresa PSP-ului în cazul unui program de tip EXE.

    ;salvăm adresa handler-ului original al întreruperii 1Ch
    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
    lea dx, msg_success
    ;afișăm mesajul corespunzător instalării cu succes.
    int 21h
    ;Calculăm dimensiunea secțiunii rezidente în paragrafe. Aceasta se face prin calcularea numărului
    ;de octeți (RezEnd-RezStart) din secțiunea rezidentă la care se adaugă dimensiunea PSP-ului (256
    ;octeți = 100h octeți) totul rotunjit în sus la un multiplu de 16. Prin împărțirea la 16 obținem
    ;numărul întreg corect de paragrafe ocupat de secțiunea 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 funcțiilor DOS non reentrant

O problemă spinoasă și care dă multă bătaie de cap programatorilor de aplicații 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 handlele sunt activate de întreruperi hardware (asincrone) este posibil ca sistemul să fie în mijlocul execuției a aproximativ oricărei secvențe de cod, fără ca noi să putem determina sau controla acest lucru. Problemele apar dacă programul TSR decide să apeleze cod „extern”, precum funcții DOS, rutine BIOS sau rutine ale unui alt TSR. De exemplu, aplicația 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 funcției DOS. Dacă în acest moment TSR-ul activat încearcă să apeleze o funcție DOS, acest lucru va duce la existența a două apeluri de funcții DOS simultan. Din păcate, DOS nu este *reentrant* (nu permite ca mai multe apeluri DOS să fie activate în același timp), ceea ce creează de obicei probleme care duc la blocarea sistemului. Acesta este doar un exemplu care ilustrează posibilitățile de a genera

apeluri *reentrant*. În realitate, de fiecare dată când un TSR apelează alte rutine decât cele implementate direct în TSR trebuie să simă conștienți de posibilitatea apariției problemelor legate de codul non reentrant.

În general TSR-urile pasive nu suferă de această problemă. Aceasta deoarece apelul la rutinele TSR în acest caz se face în contextul apelatorului. Dacă nu există posibilitatea ca TSR-ul să fie activat și printr-un eveniment asincron (generarea unei întreruperi hardware) atunci nu există probleme din punct de vedere al reentrantiei codului TSR-ului.

Așa cum am descris în exemplul de mai sus, DOS este probabil problema cea mai spinoasă din punct de vedere al programării de TSR-uri și reentrantiei codului. Aceasta deoarece DOS nu este reentrant, însă furnizează o multitudine de servicii de care un TSR are nevoie în mod normal. Atunci când au realizat acest lucru specialiștii de la Microsoft au adăugat suport DOS pentru TSR-uri astfel încât acestea să poată să determine atunci când DOS este activ (există apeluri DOS în curs) și când nu. Motivul care stă la baza rezolvării în acest mod al problemei: problema reentrantiei DOS se pune doar atunci când un TSR apelează DOS în timp ce există deja un apel DOS în curs. Dacă DOS nu este activ, atunci orice TSR poate apela funcțiile DOS fără probleme.

MS-DOS furnizează un flag special pe un octet (denumit flag InDOS) care conține valoarea zero dacă DOS nu este activ și o valoare diferită de zero dacă DOS se află în decursul procesării unei cereri. Prin testarea valorii flag-ului InDOS o aplicație TSR poate să își „de seama singură” dacă poate apela în siguranță funcții DOS. Dacă flag-ul InDOS conține valoarea zero TSR-ul poate face apel la funcții DOS. Dacă valoarea InDOS conține o altă valoare atunci există pericolul de a ne suprapune apelul cu o funcție DOS în curs. Există o funcție DOS (GetInDosFlagAddress) care returnează adresa flag-ului InDOS. Pentru utilizarea acestei funcții trebuie încărcat registrul AH cu valoarea 34h și apelat DOS, care va returna adresa flag-ului InDOS în perechea de regiștri ES:BX. Prin salvarea acestei adrese vom putea apoi în orice moment să verificăm valoarea flag-ului InDOS.

```
mov ah, 34h
int 21h
mov word ptr cs:InDOS, bx
mov word ptr cs:InDOS+2, es
```

De fapt există două flag-uri care trebuie testate: InDOS și CritError (flag-ul de eroare critică DOS). Ambele trebuie să conțină valoarea zero pentru a putea apela în siguranță funcții DOS. Flag-ul de eroare critică este setat atunci când un apel DOS se termină cu o eroare critică. Detaliile legate de eroare critică sunt stocate de DOS la o adresă fixă și sunt suprascrisă la fiecare apel al unei funcții DOS. Dacă alegem să facem un apel DOS atunci când acest flag este setat, cel mai probabil din cauza faptului că aplicația ce rulează în prim-plan a apelat o funcție DOS care nu s-a putut execuționa corect, vom suprascrie detaliile legate de eroarea critică și vom însela în acest fel aplicația întreruptă. Aceasta nu își va da seama că apelul DOS executat a eşuat, fapt care va duce cu siguranță la funcționarea ei incorectă în continuare. Flag-ul de eroare critică DOS se află stocat pentru versiunile DOS 3.1 și mai noi în octetul imediat precedent flag-ului InDOS.

Întrebarea care se pune este: ”Ce trebuie să facem atunci când cele două flag-uri au valori nenele?”. Răspunsul verbal este unul extrem de simplu: ”Întoarce-te mai târziu pentru a executa TSR-ul, atunci când funcția MS-DOS activă întoarce controlul la programul utilizator”. Cum putem însă implementa în practică acest lucru? Dacă TSR-ul nostru este activat de întreruperea de tastatură la apăsarea unei combinații de taste și redăm controlul handler-ului original din cauza faptului că DOS este activ, cum putem reactiva TSR-ul mai târziu când DOS nu mai este activ? Cea mai simplă soluție ar fi aceea de a obliga utilizatorul să activeze în mod repetat combinația de taste până când prinde un interval de timp între două apeluri DOS. Această soluție nu este însă aplicabilă în practică din motive pe care cititorul le poate bănuia cu ușurință. A doua soluție este aceea de a redirecta în cadrul TSR-ului întreruperea de ceas, pe lângă cea de tastatură. Atunci când la apăsarea combinației de taste descoperim în handler-ul întreruperii de tastatură că DOS este activ setăm un flag (CerereTSR) la nivelul TSR-ului care ne indică că am avut o cerere de activare a TSR-ului și redăm controlul handler-ului original de tastatură. În acest timp RTI de ceas va verifica periodic valoarea flag-ului din TSR. Dacă flag-ul nu este setat apelează handler-ul original al întreruperii de ceas. Dacă flag-ul este setat atunci verifică flag-ul InDOS și CritError. Dacă DOS este activ/ocupat RTI de ceas redă controlul handler-ului original. După ce DOS redă controlul aplicației utilizator prima 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 funcții DOS. Desigur, primul lucru pe care TSR-ul îl va face la activare este acela de a resetă flag-ul intern CerereTSR astfel încât următoarele întreruperi de ceas să nu restarteze TSR-ul.

Soluția de mai sus este valabilă în toate cazurile cu o singură excepție. Există unele funcții DOS care necesită un interval de timp nedefinit de execuție. Este vorba de exemplu de funcția de citire a unui caracter de la tastatură. Aceasta se poate termina într-o secundă dacă utilizatorul apasă o tastă, sau în 10 ore dacă nimenei nu apasă vreo tastă. Este cazul apelurilor blocante de funcții DOS. Implementarea DOS a acestor funcții se bazează pe o buclă internă DOS care așteaptă ca utilizatorul să apese o tastă. și până când utilizatorul face acest lucru flag-ul InDOS va rămâne nenu. Dacă TSR-ul pe care l-am implementat trebuie să scrie date într-un fișier la intervale regulate de timp, acest lucru nu va fi posibil dacă utilizatorul tocmai a plecat să ia masa în timp de DOS așteptă apăsarea unei taste sau introducerea unui sir de caractere de la tastatură.

Din fericire DOS furnizează o soluție și pentru această problemă prin implementarea unei întreruperi de *temp de inactivitate*. Cât timp DOS este într-un ciclu de așteptare a unei operații cu un dispozitiv I/O (intrare-iesire), apelează în mod continuu întreruperea 28h. Prin redirectarea vectorului întreruperii 28h, TSR-ul nostru poate determina cazurile în care DOS se află în bucle de așteptare. Atunci când DOS apelează int 28h putem face în siguranță apeluri DOS către funcții ale căror număr de identificare este mai mare decât 0Ch. Pentru funcțiile care au identificator mai mic de 0Ch DOS folosește stiva aflată la o adresă fixă. Pentru a nu distruge stiva unui apel în curs printr-un alt apel este interzisă apelarea acestor funcții DOS. De cele mai multe ori folosim întreruperea 21h pentru a afișa siruri de caractere pe ecran (funcția 09h). Conform remarcilor de mai sus, utilizarea acestei funcții nu este posibilă în timpul unei întreruperi 28h. Pentru a face posibil apelul oricărei funcții DOS atunci când DOS se află într-o buclă de așteptare putem salva conținutul anterior al stivei DOS într-o zonă de memorie din cadrul TSR-ului și să refacem conținutul stivei DOS după terminarea execuției curente a acestuia, sau atunci când nu mai facem

apeluri la funcții DOS. Un exemplu de redirectare a întreruperii 28h cu salvarea stivei pentru a putea utiliza orice funcție 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
  mov word ptr cs:save_ss, ss
  mov ss, cs:stiva_ss
  mov sp, cs:stiva_sp
  push cx
  push si
  push ds
  mov cx, 64
  mov ds, cs:save_ss
  mov si, cs:save_sp
rep_save_stiva:
  push word ptr [si]
  inc si
  inc si
  loop rep_save_stiva

  mov byte ptr cs:flag,0
  ...
; Secvență de cod care folosește funcții 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
  dec si
  pop word ptr [si]
  loop rep_restore_stiva

  pop ds
  pop si
  pop cx
  mov ss, cs:save_ss
  mov sp, cs:save_sp          ;reface adresa stivei curente

int28_end:
  sti
  iret
int28 endp

```

Vom furniza spre sfârșitul acestui capitol un exemplu complet de program TSR care utilizează o mare parte din tehniciile descrise mai sus pentru a putea apela funcții DOS.

6.4.2. Problema întreruperilor BIOS non reentrant

DOS nu este singura componentă non reentrantă a unui sistem. Există și rutine BIOS care se află în aceeași 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 rutinile BIOS nu sunt atât de sensibile la acest fenomen. Există unele funcții ale unor întreruperi BIOS non-reentrantă și există o multitudine de funcții/intreruperi BIOS care pot fi apelate fără probleme din TSR-uri. Pentru cele care sunt non-reentrantă singura soluție este implementarea de rutine *wrapper* în jurul acestora. O astfel de rutină rediectează 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 și la ieșirea din RTI resetează acest flag. Un exemplu de astfel de *wrapper* este descris în secvență 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. ÎNTRERUPERAȚIA MULTIPLEX (INT 2Fh)

Atunci când instalăm un TSR care conține componente pasive trebuie să alegem un vector de întrerupere pe care îl vom utiliza pentru a comunica cu rutinile pasive din cadrul TSR-ului. Pentru aceasta avem la îndemâna două soluții:

1. Alegerea la întâmplare a unui vector de întrerupere;
2. Alegerea unui vector de întrerupere deja definit în cadrul sistemului - care implementează o funcție specifică;

Prima soluție este nepotrivită din mai multe considerante. Dacă vectorul respectiv este deja folosit de către sistem, atunci rescrierea lui poate să ducă la disfuncționalitate în funcționarea sistemului de operare. Chiar dacă introducem doar o funcție suplimentară întreruperii alese, este posibil ca acea funcție să mai fie aleasă și de alți implementatori de TSR-uri sau este posibil ca funcția să fie aleasă de chiar implementatorii sistemului de operare în viitor pentru implementarea unui serviciu specific.

Pentru a doua soluție în unele cazuri alegerea este clară. Atunci când dorim să extindem serviciile de tastatură ale întreruperii 16h este clar că vom redirecta această întrerupere căreia îi vom redirecta/adăuga una sau mai multe funcții. Pe de altă parte, dacă implementăm un driver de dispozitiv sau un TSR care nu extinde servicii existente problema alegării vectorului de întrerupere prin care să comunicăm cu TSR-ul rămâne. Din fericire MS-DOS furnizează o soluție și în acest caz: întreruperea multiplex 2Fh. Această întrerupere a fost rezervată pentru a furniza un mecanism general de instalare, testarea prezenței și comunicare cu un TSR.

Pentru a utiliza întreruperea multiplex, aplicațiile stochează un număr de identificare în registrul AH și fac apel la INT 2Fh. Fiecare TSR din lanțul de 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 lanțul vectorilor întreruperii 2Fh.

Desigur, această metodă rezolvă doar o parte a problemei, aceea a alegării unui vector de întrerupere. Numărul de identificare trebuie să fie cunoscut atât de entitatea care verifică cât și de TSR-ul instalat. Cum nu există un organism de standardizare care să aloce numerele de identificare, desigur nu vor putea fi eliminate conflictele: alegerea același număr de identificare de către mai multe TSR-uri. Dacă numărul de identificare este ales dinamic atunci se pune întrebarea: „Cum poate fi el cunoscut de aplicația care testează prezența TSR-ului?”

Problema enunțată mai sus poate fi rezolvată printr-un mic artificiu. Folosim prin convenție funcția (comanda) zero pentru a testa dacă un TSR este instalat sau nu. Orice aplicație va executa întotdeauna această funcție pentru a determina dacă TSR-ul este instalat în memorie înainte de a executa alte funcții ale TSR-ului. În mod normal funcția zero va returna valoarea zero în registrul AL dacă TSR-ul nu este prezent în memorie sau 0FFh dacă TSR-ul este prezent. Desigur prezența unei valori 0FFh în AL ne indică doar prezența unui TSR instalat, dar nu ne garantează că TSR-ul

respectiv este cel pe care îl căutăm noi. Prin extinderea convenției făcute până acum însă, vom putea determina corect și exact prezența TSR-ului care ne interesează în memorie. Pentru aceasta să presupunem că funcția zero întoarce pe lângă valoarea din AL un pointer la string de identificare în registrul ES:DI. Prin compararea string-ului de la adresa ES:DI cu semnătura cunoscută de aplicație care testează prezența TSR-ului se poate detecta exact prezența unui TSR anume în memorie.

6.6. INSTALAREA UNUI TSR

Deși am văzut deja în primul exemplu de program TSR din acest capitol modalitatea de instalare a unui TSR în memorie, mai sunt câteva aspecte de discutat legate de această problemă. Întrebările care se pun în acest context sunt:

1. Ce se întâmplă dacă utilizatorul instalează un TSR care este deja instalat în memorie fără a dezinstala mai întâi copia existentă?
2. Cum putem afecta dinamic un număr de identificare unui TSR astfel încât să nu fie în conflict cu TSR-urile gata instalate?

Prima întrebare se referă la instalarea de mai multe ori a unui TSR în memorie. În general nu există aplicații TSR care să funcționeze cu mai multe copii simultan în memorie. Aceasta deoarece fiecare copie redirecțiază aceleași întreruperi și îndeplinește aceleași sarcini. Mai mult, instalarea de N ori, N>1 a unui TSR în memorie implică consumul unei cantități de N ori mai mari de memorie. În majoritatea cazurilor un astfel de scenariu nu poate decât să ducă la risipă de memorie și chiar la blocarea sistemului. În consecință fiecare TSR trebuie să verifice la instalare existența unei copii deja instalare anterior. În cazul în care se detectează o copie deja instalată de obicei se afișează un mesaj informativ și TSR-ul refuză instalarea.

Secvența de cod prezentată în continuare determină prezența unui TSR în memorie și furnizează numărul de identificare al acestuia. Această secvență de cod ne dă răspunsul și la a doua întrebare. Rutina determină în cazul în care TSR-ul nu este activ un număr de identificare liber și resetează CF la terminare.

; Scanează toate ID-urile posibile între 255 și 0. Dacă vreunul este instalat verifică dacă coincid ; semnăturile

```
...
Sign db 'Semnatura TSR test'
Sign_len EQU $-Sign
FuncID db 0
...
check_installed proc
    mov cx, 0FFh
```

;verifică ID-urile posibile de la 255 la 0

```

CiclulID:
    mov ah, cl          ;depune în AH ID-ul curent
    push cx             ;salvează CX pe stiva deoarece îl modificăm
    mov al, 0            ;codul comenzi (funcției) – “Are you There?”
    int 2Fh              ;verifică dacă există TSR pentru ID-ul din AH
    cmp al, 0            ;dacă nu încercăm următorul ID
    pop cx               ;dacă da comparăm semnătura Sign cu cea
    je NextID            ;stocată la adresa ES:DI (TSR)
    push cx
    mov si, offset Sign
    mov cx, Sign_len
    cld
    rep cmpsb
    pop cx
    jne NextID
    jmp Installed

NextID:
    mov FuncID, cl
    loop CiclulID
    jmp NotInstalled

Installed:
    Stc
    mov FuncID, cl
    ret

NotInstalled:
    clc
    mov cl, FuncID
    ret
check_installed endp

```

Rutina de mai sus verifică un ciclu toate numerele de identificare posibile între 255 și 0. În cazul în care pentru un ID se detectează prezența unui TSR (AL=0FFh) se testează semnătura specifică a TSR-ului. La fiecare iterație rutina stochează în FuncID ultimul număr de identificare liber găsit. La terminarea rutinii flag-ul CF este setat dacă TSR-ul a fost găsit în memorie și registrul CL conține numărul de identificare al TSR-ului. Dacă TSR-ul nu a fost găsit în memorie CF este resetat și 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 și să ia o decizie în consecință.

Desigur fiecare TSR care se instalează și folosește convențiile descrise mai sus trebuie să redirecteze intreruperea 2Fh pentru implementarea acestor funcționalități.

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 opreasă toate activitățile în curs ale programului TSR;
- Să refacă toți vectorii de intrerupere modificați de TSR;
- Să dealoce memoria alocată astfel încât aceasta să poată fi utilizată de alte aplicații;

Dificultatea majoră o constituie ultimele două puncte. Nu este întotdeauna posibil să refacem vectorii de intrerupere modificați și nu se poate dealoca întotdeauna memoria TSR-ului. Să vedem care sunt motivele care fac imposibile cele două activități.

În cazul în care codul TSR-ului încearcă doar să readucă la valorile originale vectorii de intrerupere poate să apară o problemă cu grave repercusiuni în funcționarea sistemului. Dacă utilizatorul încarcă în memorie și alte TSR-uri după TSR-ul nostru, este posibil ca aceste TSR-uri să redirecteze aceleși intreruperi, sau măcar unele dintre ele. Acest lucru duce la crearea unui lanț precum cel ilustrat în figura de mai jos:

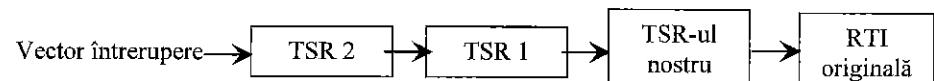


Fig. 6.6. Lanț de RTI pentru mai multe TSR-uri instalate pe aceeași intrerupere.

În figura 6.6 am reprezentat grafic două TSR-uri care au fost instalate după "TSR-ul nostru". Ambele au redirectat aceeași intrerupere. Dacă în situația prezentă TSR-ul nostru refac vectorul intreruperii la cel original, salvat la instalare, configurația finală va fi cea prezentată în figura 6.7.

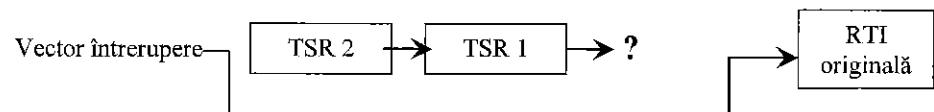


Fig. 6.7. Refacere RTI originală după instalarea mai multor TSR-uri pe aceeași intrerupere.

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 intrerupere generată trece prin întreg lanțul de RTI-uri. În momentul în care TSR-ul nostru readuce vectorul de intrerupere la valoarea salvată (adresa originală) de el, practic RTI din TSR 1 și TSR 2 sunt scoase din lanț și prin aceasta din uz. La generarea ulterioară a intreruperii, rutinele ei de tratare din TSR 1, respectiv TSR 2 nu vor mai fi apelate. Acest lucru duce în general la funcționarea defectuoasă a celor două TSR-uri implicate. Ceea ce este și mai rău este faptul că doar handler-ele intreruperilor comun redirectate de cele trei TSR-uri vor fi dezafectate, restul intreruperilor redirectate în TSR 1 și TSR 2 rămânând active. Este

greu de prezis care va fi comportamentul sistemului într-o astfel de situație. Depinde de funcționalitățile implementate de cele două TSR-uri rămase.

Cum putem rezolva această problemă? Soluția cea mai simplă și cea mai eficace atunci când detectăm un astfel de caz este să tipărim un mesaj de atenționare și să refuzăm dezinstalarea TSR-ului. Aceasta este o hibă bine cunoscută legată de TSR-uri și în general utilizatorii care instalează TSR-uri sunt conștienți (sau ar trebui să fie) de faptul că dezinstalarea lor trebuie făcută în ordine inversă instalării. Desigur se mai pune încă problema modalității prin care putem detecta dacă utilizatorul a mai instalat și alte TSR-uri care redirecțează 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 întrerupерii 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 și alte TSR-uri încărcate în memorie după el. Acest fenomen duce la o altă anomalie. Chiar dacă eliberăm memoria alocată TSR-ului nostru ea nu va putea fi folosită de către aplicațiile care vor rula mai târziu în sistem. Aceasta datorită schemei de alocare și gestiune a memoriei de către MS-DOS. Situația descrisă mai sus este prezentată în figura 6.8. După încărcarea TSR-ului nostru, pointerul spre zona de memorie utilizabilă este deplasat la adresa superioară imediat următoare acestuia. Încărcarea programului TSR1 va deplasa pointerul zonei de memorie liberă la adresa imediat superioară ultimului octet alocat de acesta. În consecință după dealocarea memoriei utilizate de către TSR-ul nostru, pointerul spre zona de memorie liberă (utilizabilă) va rămâne neschimbăt, pentru a proteja TSR1. Fenomenul descris mai sus se numește *fragmentarea memoriei* și poate să apară de fiecare dată când procesele de alocare și dealocare a memoriei alocate TSR-urilor nu sunt făcute în regim LIFO (*Last In First Out*).

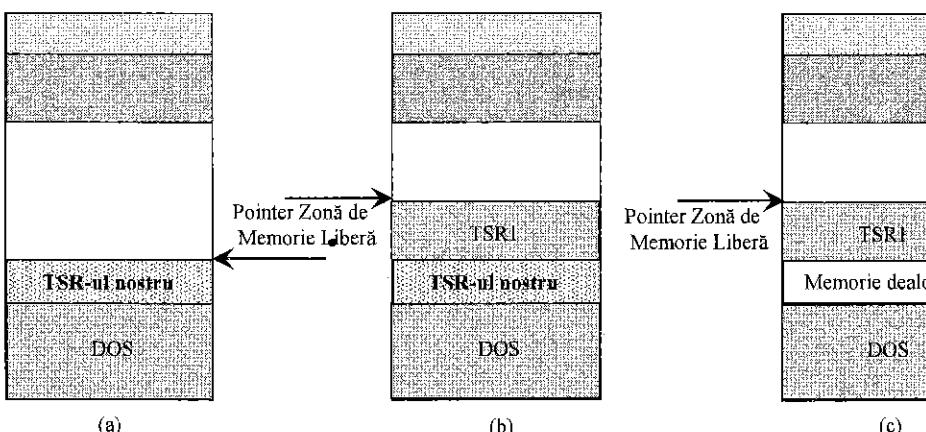


Fig. 6.8. Pointerul de Memorie Liberă în timpul alocării și dealocării TSR-urilor.

În figura 6.8 (a) este prezentată adresa pointerului de memorie liberă după instalarea TSR-ului nostru. În figura (b) este prezentată adresa de memorie a zonei libere după încărcarea în memorie a programului TSR1, iar în figura (c) adresa zonei de memorie libere după dezinstalarea TSR-ului nostru fără a dezinstala în prealabil TSR1. După cum se poate observa și în figură, zona de memorie devenită liberă după dezinstalarea TSR-ului nostru nu poate fi folosită până când nu se dezinstalează și TSR1.

Pentru dealocarea memoriei programului TSR avem nevoie de apeluri ale funcției DOS 49h. Acest apel preia din registrul ES adresa blocului de memorie care trebuie eliberat. Primul apel este necesar pentru eliberarea blocului de variabile de mediu alocat la crearea programului. Pointerul spre acest bloc de memorie se găsește la deplasamentul 2Ch în PSP. Acesta este de așa fel și unul dintre motivele pentru care orice program TSR dezinstalabil trebuie să salveze adresa sa PSP. Următorul bloc de memorie este cel al programului rezident înșăși. Acesta începe la adresa de început a PSP-ului în memorie. Secvența de cod care permite dealocarea memoriei unui TSR este prezentată în continuare:

; Secvența de cod de mai jos presupune că variabila PSP a fost inițializată cu adresa PSP-ului ; acestui program înainte de apelul care lasă TSR-ul rezident

mov es, PSP	;încarcă în ES adresa de segment a variabilelor de mediu
mov es, es:[2Ch]	
mov ah, 49h	;dealocă blocul variabilelor de mediu
int 21h	;încarcă în ES adresa de segment a PSP-ului (programul înșuși)
mov es, PSP	
mov ah, 49h	;dealocă programul TSR din memorie
int 21h	

La fel cum codul de instalare a unui TSR este înglobat în același cod sursă cu programul TSR, codul de dezinstalare face parte tot din partea tranzientă a aceluiași program. De obicei același program apelat cu parametrul /u în linia de comandă execută sarcina de dezinstalare a TSR-ului, atunci când acesta a fost instalat în prealabil.

6.8. TSR MONITOR TASTATURA

În cele ce urmează vom exemplifica noțiunile prezentate în paragrafele precedente printr-un program TSR complet. TSR-ul denumit KeybMon permite monitorizarea tastaturii și înregistrarea tuturor tastelor apăsate pe parcursul rulării sale. La apăsarea unei combinații de taste de activare (F9), KeybMon va afișa pe ecran toate tastele apăsate de la ultima activare. TSR-ul verifică la instalare dacă există deja o copie în memorie, caz în care refuză o nouă instalare. El poate fi dezinstalat prin lansarea aplicației cu parametrul /u în linia de comandă. TSR-ul verifică dacă procesul de dezinstalare este posibil și nu intră în conflict cu alte TSR-uri instalate după el.

```

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 înregistram

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$'
db '!!!!$', '!!!!$'

keys db maxh dup(0) ;tablou cu titlul tastelor indexat după codul SCAN al fiecarei
                     ;taste (1=ESC, etc)
                     ;tablou în care stocăm codurile SCAN ale tastelor apăsate

kindex dw -1 ;contorul curent de taste apăsate

msg db 'Taste înregistrate:',13,10,'$' ;mesajul afișat la apăsarea combinației de taste (F9) de
                                         ;activare a TSR-ului înainte de afișarea 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_ss dw cseg ;stiva în care vom salva 64 de cuvinte din stiva DOS
      dw 256 dup(0)
stiva_sp dw stiva_sp ;O vom folosi atunci când TSR-ul este activat

save_ss dw ? ;aici salvăm regiștrii 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 și vectorii
oldint2F dd ? ;originali ai intreruperilor redirectate
oldint28 dd ? ;originali ai intreruperilor redirectate
oldInt9 dd ? ;originali ai intreruperilor redirectate

```

```

;semnătura TSR-ului – utilizată de către copia tranzientă a KeybMon pentru a verifica dacă acesta
;este instalat în memorie
sign db 'Keyboard Monitor'
sign_len equ $-sign

```

; Int9 Noul handler al intreruperii 9h. Sistemul apelează această rutină de fiecare dată când este
; apăsată o tastă. Rutina culege non-destructiv codul scan al tastei apăsate din controller-ul
; de tastatură. Dacă s-a apăsat tasta F9 – se citește și șterge codul scan din controller-ul de
; tastatură. În continuare dacă DOS nu este activ lansează rutina de afișare 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 și predă controlul rutinei originale de tratare a intreruperii.
; Folosim porturile 60h și 61h pentru interacțiunea cu controller-ul de tastatură.

```

int9 proc far
    cli ;dezactivează intreruperile
    push ax ;culege nedestructiv codul scan din controller-ul de
    in al,60h ;tastatură

    cmp al, 43h ;dacă e tasta F9 ?
    je read_key ;codurile scan>80h semnifică eliberarea unei taste.
    cmp al,80h ;Nu este nevoie să le tratăm. Salt la rutina originală
    ja orig9 ;verificăm dacă mai avem spațiu de stocare
    inc cs:[kindex] ;dacă nu redăm controlul handler-ului int9 original
    push bx ;mai avem spațiu=> memorăm codul scan în keys
    push ds ;salvăm regiștrii BX, DS pe care îi modificăm
    push cs
    pop ds
    mov bx, offset keys ;salvăm codul scan în tabloul keys pe prima poziție
    add bx, kindex ;liberă
    mov byte ptr [bx],al ;refacem regiștrii modificați
    pop ds
    pop bx

orig9:
    sti ;apel rutină originală a int 9 fără revenire
    pop ax ;activăm intreruperile și apelăm oldint9
    jmp dword ptr cs:oldint9

```

;În general apăsarea unei taste generează apelul int 9h. Handler-ul original al int 9h citește codul
;scan al tastei apăsate din portul 60h și îl transformă în codul ASCII corespunzător. Apoi plasează

atât codul scan cât și codul ASCII în buffer-ul de tastatură. Acestea vor putea fi apoi citite din buffer-ul de tastatură cui ajutorul întreruperii 16h. Handler-ul întreruperii 9h execută următorii pași:

- ; 1) Citește codul scan din controller-ul de tastatură (portul 60h)
- ; 2) Setează bitul 7 în portul 61h
- ; 3) Resetează bitul 7 în portul 61h indicând astfel controller-ului de tastatură consumarea codului scan
- ; 4) Convertește codul scan la corespondentul ASCII și plasează ambele coduri în bufferul de tastatură
- ; 5) Trimite secvența EOI – 20h (End of Interrupt – Sfârșit de întrerupere) controller-ului programabil de întreruperi 9259 (portul 20h). Atât timp cât acesta nu primește secvența EOI nu mai generează alte apeluri int 9.

```

read_key:
    cli
    in al,61h
    mov ah,al
    or al,80h
    out 61h, al
    xchg ah,al
    out 61h, al
    mov al, 20h
    out 20h, al
    push ds
    push dx
    lds dx, dword ptr cs:inDos_OFFSET
    cmp byte ptr dx, 0
    jne later
    call write_keys
    jmp end_int9

later:
    mov byte ptr cs:flag,1

end_int9:
    pop ds
    pop dx
    pop ax
    sti
    iret
int9 endp

```

;dacă tastă apăsată este F9
;dezactivăm întreruperile
;consumăm codul scan din controller-ul de tastatură
;Simulăm acțiunea standard a int 9 = consumare
;a codului scan cu ștergerea acestuia din controller
;Controller-ul nu va mai semnaliza apăsarea acestei
;taste. Ea a fost citită deja dpdv al controller-ului de către sistemul de operare.
;trimitem secvența de sfârșit de întrerupere către controller-ul programabil de întreruperi 8259.

;verificăm dacă DOS este activ
;dacă DA setăm variabila flag și terminăm execuția
;handler-ului în ideea de a fi reactivați după.
;dacă NU, apelăm rutina de afișare a tastelor
;memorate și terminăm execuția handler-ului

;revenire din int 9 în cazurile în care handler-ul
;a gestionat întreruperea (tasta F9 apăsată)

```

int28 Noul handler al întreruperii 28h (DOS Idle).Sistemul apelează această rutină de fiecare dată când DOS este într-o buclă de așteptare I/O. Rutina apelează handler-ul original, cu revenire, pentru a permite și celorlalte aplicații să detecteze starea DOS și verifică apoi starea variabilei flag. În cazul în care aceasta este setată, salvăm stiva DOS pentru a permite revenirea corectă la apelul DOS întrerupt și apelăm rutina de afișare a tastelor memorate de la ultimul apel write_keys. Se resetează variabila flag (am tratat cererea de activare). După afișarea tastelor memorate refacem stiva DOS și redăm controlul aplicației întrerupe

int28 proc
    pushf
    call dword ptr cs:oldint28
    cli
    cmp byte ptr cs:flag,1
    jne int28_end

    mov word ptr cs:save_sp, sp
    mov word ptr cs:save_ss, ss
    mov ss, cs:stiva_ss
    mov sp, cs:stiva_sp
    push cx
    push si
    push ds
    mov cx, 64
    mov ds, cs:save_ss
    mov si, cs:save_sp
    rep_save_stiva:
        push word ptr [si]
        inc si
        inc si
    loop rep_save_stiva
    mov byte ptr cs:flag,0
    call write_keys

    mov cx,64
    mov ds, cs:save_ss
    mov si, cs:save_sp
    add si, 128
    rep_restore_stiva:
        dec si
        dec si
        pop word ptr [si]
    loop rep_restore_stiva

;simulare apel de tip call al int28 originală
;dezactivare întreruperi.
;verificăm dacă avem cerere de activare
;dacă NU redăm controlul aplicației întrerupe
;dacă DA – salvăm 64 de cuvinte din stiva DOS și
;adresa acesteia
;resetăm variabila flag
;apelăm rutina de afișare a tastelor memorate
;refacem primele 64 de cuvinte din stiva DOS

```

```

pop ds          ;refacem regisitrii modificarii
pop si
pop cx
mov ss, cs:save_ss
mov sp, cs:save_sp
int28_end:
    sti          ;activam intreruperile pentru a permite functionarea
    iret          ;corecta a sistemului si revenim din intrerupere
int28 endp

```

; Int2F Noul handler al intreruperii 2Fh (Multiplex). Noua rutina verifică dacă codul de funcție transmis este egal cu codul de identificare al TSR-ului. În caz afirmativ verifică codul subfuncției (comenzi) transmise în AL. Subfuncția acceptată este 00 – Verifică prezență. Aceasta întoarce în ES:DI adresa far a semnăturii TSR-ului și OFFh (prezent) în AL.

```

int2F proc
    cmp ah,cs:FuncID
    jne orig2F
    cmp al, 0
    jne orig2F
    push cs
    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 afișare a tastelor memorate. Această rutină este apelată la activarea TSR-ului și afișează pe ecran numele tastelor apăsate de la ultima activare. La fiecare apel se re-initializează tabloul codurilor scan memorate.

```

write_keys proc
    push cs          ;afisare preambul
    push offset msg
    call write_str

    push ax
    push cx
    push dx
    push si
    push ds          ;salvam regisitrii modificarii

```

```

push cs          ;initializam regisitru DS
pop ds
mov cx,0
cycle:
    cmp cx, kindex
    jg wk_end
    mov si, offset keys
    add si, cx
    lodsb
    mov dl,5
    xor ah,ah
    mul dl
    mov dx, offset key_codes
    add dx, ax
    push ds
    push dx
    call write_str
    inc cx
    jmp cycle
wk_end:
    mov kindex, -1
    pop ds
    pop si
    pop dx
    pop cx
    pop ax
    ret          ;revenire din rutina
write_keys endp

```

; Write_Str Rutina de afișare a unui sir de caractere terminat în \$ pe ecran. Rutina preia adresa de segment și deplasamentul sirului de afișat de pe stivă și folosește funcția DOS 09h pentru afișare. Rutina nu alterează regisitrii.

```

write_str proc near
    push bp
    mov bp,sp          ;pregateste stiva pentru recuperare parametri
    push ax
    push dx
    push ds
    mov dx, [bp+6]
    mov ds, dx          ;salveaza regisitrii modificarii
    mov dx, [bp+4]
    mov ah,09h          ;recupereaza deplasamentul sirului
                        ;recupereaza adresa de segment a sirului

```

```

int 21h          ;apelăm funcția DOS 09h de afișare sir caractere
pop ds          ;refacem regiștrii modificați
pop dx
pop ax
pop bp
ret 4           ;revenire cu dealocarea parametrilor din stivă
write_str endp

; sfârșitul zonei de cod rezident – se folosește la calculul dimensiunii zonei rezidente în paragrame
RezEnd equ $

; Aici începe secțiunea tranzientă a programului
;
```

Check_Installed

Rutina de verificare existență copie TSR în memorie. Această rutină apelează int 2F cu toate valorile de identificator TSR posibile de la 255 la 0 pentru a determina dacă TSR-ul este deja instalat în memorie. La un răspuns pozitiv de prezență (AL=0FFh la întoarcere) se verifică semnatura TSR-ului. Dacă semnaturile coincid flag-ul CF este setat la revenire și în variabila FuncID este depus numărul de identificare al TSR-ului. Dacă TSR-ul nu este instalat în memorie flag-ul CF este resetat iar în registrul CL și variabila FuncID este stocat un ID TSR nealocat (neutilizat). Folosim această funcție atât pentru determinarea existenței TSR-ului în memorie cât și pentru determinarea unui identificator TSR neocupat la instalare

check_installed proc

mov cx, 0FFh ;initializăm CX cu valoarea maximă a ID-ului

CiclulID:

mov ah, cl
 push cx ;salvăm conținutul CX deoarece îl vom altera
 mov al, 0 ;cod subfuncție 0
 int 2Fh ;apel int 2Fh
 cmp al, 0 ;verificăm prezență (AL = 0FFh) pozitiv
 ;refacem CX
 pop cx
 je NextID ;TSR-ul nu este instalat cu acest ID. Următorul...
 push cx
 mov si, offset Sign
 mov cx, Sign_len
 cld
 rep cmpsb
 pop cx
 jne NextID ;semnaturile nu coincid => Nu este TSR-ul nostru

jmp Installed
NextID:
 mov FuncID, cl
 loop CiclulID
 jmp NotInstalled

Installed:
 Stc
 mov FuncID, cl
 ret

NotInstalled:
 clc
 mov cl, FuncID
 ret

check_installed endp

Uninstall

Rutina de dezinstalare a TSR-ului din memorie. Această rutină verifică dacă TSR-ul este instalat apelând *check_installed*. Dacă TSR-ul nu este instalat afișează un mesaj de eroare și termină execuția programului. Dacă TSR-ul este instalat verifică dacă acesta poate fi dezinstalat fără a crea conflicte cu alte TSR-uri. În caz afirmativ se refac vectorii de intrerupere redirectați și se dealocă memoria TSR. Altfel se afișează un mesaj de eroare și se termină execuția programului. Vectorii originali ai intreruperilor redirectate de TSR (9h, 28h, 2Fh) sunt recuperăți de la adresele unde au fost stocați de către TSR. ES:DI după apelul int 2F indică adresa semnătării în TSR. Vectorii de intrerupere se află la ES:[DI-12], ES:[DI-8] și respectiv ES:[DI-4]. PSP-ul programului TSR este memorat la adresa ES:[DI-14].

uninstall proc

call check_installed
 jc try_uninstall

;verifică TSR instalat
 ;dacă DA încearcă dezinstalare

mov ah, 09h ;dacă NU afișează mesaj de eroare și...
 mov dx, offset msg_uninst_notinstalled
 int 21h
 jmp uninist_end

;termină execuția programului

try_uninstall:
 mov ah, FuncID
 mov al, 0
 int 2Fh
 push ds

;TSR prezent recuperăza în ES:DI adresa semnătării

```

mov ax,0
mov ds ,ax
mov ax, offset int2F
cmp ds:[4*2Fh], ax
jne CantRemove
mov ax, es
cmp ds:[4*2Fh+2], ax
jne CantRemove
mov ax, offset int28
cmp ds:[4*28h], ax
jne CantRemove
mov ax, es
cmp ds:[4*28h+2], ax
mov ax, offset int9
cmp ds:[4*9h], ax
jne CantRemove
mov ax, es
cmp ds:[4*9h+2], ax
jne CantRemove

cli
;putem dezinstala. Dezactivăm întreruperile
mov ax, 252Fh
lds dx, dword ptr es:[di-12]
int 21h
;refacem vectorii de întrerupere originali (cei
;stocați de TSR)
mov ax, 2528h
lds dx, dword ptr es:[di-8]
int 21h
mov ax, 2509h
lds dx, dword ptr es:[di-4]
int 21h
sti
;reactivăm întreruperile
;pregătim terenul pentru a dealoca memoria
;ES:[DI-14] – adresa PSP-ului din TSR
;salvăm adresa PSP pe stivă
;adresa de segment a variabilelor de mediu
mov es, es:[2Ch]
mov ah, 49h
int 21h
pop es
mov ah, 49h
int 21h
;dealocăm blocul variabilelor de mediu
;ES – adresa PSP-ului
;dealocăm memorie TSR
;afișăm mesaj succes dezinstalare TSR
mov ah, 09h
mov dx, offset msg_uninst_success
int 21h

```

;compară vectorii întreruperilor 9h, 28h, 2Fh cu cele ale rutinelor de tratare din TSR. Dacă cel puțin o adresă nu coincide nu putem dezinstala TSR-ul în ES având adresa segmentului de cod al handler-elor din TSR. Offset-ul e același cu cel al rutinelor din copia tranzientă a programului.

```

jmp uninst_end

CantRemove:
pop ds
mov ah,09h
mov dx, offset msg_uninst_fail
int 21h

uninst_end:
mov ax, 4c00h
int 21h
;terminăm execuția programului
uninstall endp

;Check_Uninst_Demand
;
;
;
;

Rutina de verificare a prezenței parametrului /u în linia de comandă.
Rutina verifică în PSP la adresa PSP:[81h] unde este stocată linia de comandă – prezența argumentului /u. Dacă acesta este prezent se setează flag-ul CF (cerere dezinstalare). Altfel CF este resetat la revenirea din rutină

check_uninst_demand proc
push cs:psp
pop es
;Depune în ES adresa PSP-ului copiei tranziente a TSR-ului
;deplasamentul în PSP a liniei de comandă
mov di, 81h
xor ch, ch
mov cl, es:[di-1]
;la adresa 80h în PSP este stocată pe un octet lungimea liniei de comandă. Sărîm peste spații.
mov al, ''
cld
rep scasb
jcxz no_uninst_demand;Nu sunt alte argumente decât spații. Nu s-a cerut dezinstalare.
inc cx
;corecție CX
push cs
pop ds
mov si, offset cmd_param
;comparăm valoarea parametrului din linia de comandă cu
;șirul /u
dec di
rep cmpsb
jne no_uninst_demand
;cele două șiruri nu sunt egale=> nu s-a cerut dezinstalare
;șirurile sunt identice => s-a cerut dezinstalare. Setăm CF
stc
ret

no_uninst_demand:
clc
ret
;nu s-a cerut dezinstalare. Resetăm CF.

check_uninst_demand endp

```

```

; -----
; Mesaje ce semnalizează eșecul sau succesul unor operații
;

msg_success      db  'KeybMon a fost instalat',13,10,'$'
msg_uninst_successdb  'KeybMon a fost dezinstalat',13,10,'$'
msg_uninst_fail    db  'KeybMon nu poate fi dezinstalat deoarece există un
alt TSR încărcat după KeybMon',13,10,'$'
msg_uninst_notinstalled db  'KeybMon nu este instalat',13,10,'$'
msg_already_installed db  'Eroare instalare. KeybMon este deja instalat',13,10,'$'
cmd_param        db  '/u'

; -----

```

;Programul principal.

```

;-----
begin:
  push cs
  pop ds

  mov ah, 62h          ;depune adresa PSP-ului în variabila PSP
  int 21h
  mov PSP, BX
  call check_uninst_demand
  ;verifică cerere dezinstalare (Parametrul /u)
  ;nu este cerere dezinstalare => încearcă instalare
  ;cerere dezinstalare.
  jnc install
  jmp uninstall

;Uninstall este de fapt o rutină nu o etichetă normală de cod. Întrucât nu dorim revenirea din rutina
;de dezinstalare facem un apel fără revenire. De remarcat aici apelul instrucțiunii jmp pe o etichetă
;ce desemnează numele unei rutine. Atât jmp cât și call pot fi utilizate pentru a transfera controlul la
;orice adresa din program (orice etichetă de orice tip). Diferența între cele două constă în faptul că
;instrucțiunea call depune pe stivă adresa de revenire pregătind astfel și permisând în același timp
;apelul instrucțiunii ret.

install:
  call check_installed
  ;verifică prezență TSR în memorie
  ;TSR-ul nu este prezent în memorie
  jnc continue_install

  mov ah, 09h          ;TSR deja prezent. Afisează mesaj de eroare și ...
  mov dx, offset msg_already_installed
  int 21h
  mov ax, 4C01h
  int 21h              ;termină execuția programului

```

;Uninstall este de fapt o rutină nu o etichetă normală de cod. Întrucât nu dorim revenirea din rutina
de dezinstalare facem un apel fără revenire. De remarcat aici apelul instrucțiunii jmp pe o etichetă
ce desemnează numele unei rutine. Atât jmp cât și call pot fi utilizate pentru a transfera controlul la
orice adresa din program (orice etichetă de orice tip). Diferența între cele două constă în faptul că
instrucțiunea call depune pe stivă adresa de revenire pregătind astfel și permisând în același timp
apelul instrucțiunii ret.

```

continue_install:
  mov ax, 3400h
  int 21h
  mov cs:inDos_OFFSET, bx
  mov cs:inDOS_SEG, es

  cli
  mov ax, 3509h
  int 21h
  mov word ptr [oldInt9],bx
  mov word ptr [oldInt9+2],es
  mov ax,2509h
  mov dx, offset int9
  int 21h

  ;Salvează vectorii de intrerupere și redirecțează
  ;înterupurile 9h, 28h, 2Fh

  mov ax,3528h
  int 21h
  mov word ptr [oldInt28],bx
  mov word ptr [oldInt28+2],es
  mov ax,2528h
  mov dx, offset int28
  int 21h

  mov ax,352Fh
  int 21h
  mov word ptr [oldInt2F],bx
  mov word ptr [oldInt2F+2],es
  mov ax,252Fh
  mov dx, offset int2F
  int 21h
  sti

  mov ah,09h          ;afisează mesaj – succes instalare
  mov dx, offset msg_success
  int 21h

  mov dx, (RezEnd - RezStart + 0Fh + 100h)/16      ;calculează
rezidentă
  mov ax, 3100h
  int 21h
  ;și rămâne rezident

cseg ends
end start

```

Programul TSR de mai sus ilustrează majoritatea conceptelor și tehniciilor descrise în cadrul acestui capitol. Scopul acestui program este pur educativ și el nu este destinat folosirii în scopul interceptării unor informații cu caracter privat: parole, nume utilizatori, monitorizarea activității angajaților la calculator, etc.

După lansarea în execuție și instalarea sa, KeybMon înregistrează codurile scan (coduri numerice care identifică unic fiecare tastă) ale tuturor tastelor apăsate. La apăsarea tastei F9 programul afișează mesaje de tipul:

G:\UBB\CARTE>Taste înregistrate:

d i r ENT

G:\UBB\CARTE>

Instalarea și dezinstalarea corectă se pot testa folosind programul *mem* furnizat cu sistemul de operare MS-DOS sau *tmem*, furnizat cu pachetul software al mediilor de dezvoltare Borland ce conțin și asamblorul *tasm*, editorul de legături *tlink* și depanatorul *td*. Instalarea corectă este confirmată de funcționarea programului și activarea TSR-ului la apăsarea tastei F9. Pentru a verifica faptul că memoria este corect dealocată vom folosi programul *mem*. Pentru aceasta vom apela *mem* înainte de instalarea TSR-ului în memorie și vom obține la consolă o ieșire similară cu cea de mai jos:

655360 bytes total conventional memory

655360 bytes available to MS-DOS

598000 largest executable program size

După instalarea programului TSR, apelul utilitarului *mem* ne va furniza următoarele rezultate:

655360 bytes total conventional memory

655360 bytes available to MS-DOS

593104 largest executable program size

După cum se poate observa, partea rezidentă a programului KeybMon ocupă

598000-593104 = 4896 octeți

în memoria RAM.

După dezinstalarea programului folosind opțiunea /u, rezultatele raportate de *mem* sunt următoarele:

655360 bytes total conventional memory

655360 bytes available to MS-DOS

598000 largest executable program size

adică exact cantitatea de memorie disponibilă raportată de DOS înainte de instalarea TSR-ului. Cifrele obținute sunt valabile pentru configurația de programe active pe calculatorul pe care au fost

făcute teste. În funcție de programele active cifrele obținute pot fi diferite. Cantitatea de memorie ocupată de programul TSR prezentat anterior însă va fi întotdeauna 4896 octeți.

Tabloul *key_codes* conține câte un sir de caractere pe 5 octeți ce descriu numele fiecărei taste după codul scan al acesteia. Spațiul de coduri scan nu este continuu alocat. Există unele coduri numerice în plaja de valori care nu sunt alocate, sau care sunt alocate unor combinații mai ciudate de taste. De exemplu, codul scan cu valoarea 2Ah nu este alocat. Pentru aceste coduri scan care lipsesc sau pentru combinațiile de taste neînțepătate am stocat în tabelul *key_codes* un sir de caractere de forma '!!!!S' (patru semne de exclamare). Nu am considerat toate codurile scan posibile. Combinațiile de taste cu CTRL, SHIFT, ALT, etc toate generează coduri scan distințe sau secvențe de coduri scan succesive. Nu am tratat în exemplul anterior aceste cazuri, ci doar apăsarea tastelor obișnuite, fără combinații multiple. Dacă se dorește recuperarea acestor combinații acest lucru este extrem de facil. Trebuie doar completat tabelul *key_codes* cu toate combinațiile care lipsesc pentru ca programul să le recunoască. Întrucât este un exemplu didactic nu am considerat acest lucru necesar.

La o inspectare atentă a codului se poate vedea că handler-ul rescris al întreruperii 9h activează TSR-ul doar dacă nu există nici un apel DOS în curs. Dacă în momentul cererii de activare există un apel DOS în curs setăm variabila *flag* pentru a notifica faptul că dorim activarea TSR-ului și înțoarcem controlul sistemului. În paragraful 6.4.1 am arătat că în general trebuie redirectată și întreruperea de ceas pentru a putea apoi prelua controlul și activa TSR-ul imediat ce apelul DOS în curs se termină. Acest lucru nu este implementat la nivelul aplicației KeybMon deoarece în acest context nu este necesar. De ce? Pentru că TSR-ul nostru se poate activa doar la apăsarea unei taste, fapt ce implică execuția unei funcții DOS (aceea de citire a unui caracter sau sir de caractere). Deci la fiecare cerere de activare a TSR-ului ne vom afla în decursul execuției unei funcții DOS. În consecință am folosit doar mecanismul întreruperii 28h (DOS Idle) care semnalizează faptul că MS-DOS se află în aşteptarea terminării unei operații de I/O. Cum DOS se află într-o stare la fiecare citire de tastă sau sir de caractere, rezultă că nu este necesară verificarea cu regularitate a valorii *flag*-ului. Ajunge să verificăm cererea de activare doar atunci când DOS așteaptă terminarea operației de I/O. Mai mult de atât, DOS se află aproape în permanență într-o stare de așteptare a terminării unei operații de I/O datorită faptului că însăși citirea unei comenzi de la tastatură de către interpretorul de comenzi induce această stare. În general orice program lansat în MS-DOS citește tastatura și acționează în conformitate cu tastele activate. În acest context redirectarea întreruperii de ceas pentru a da posibilitatea activării întârziate a TSR-ului nu este necesară.

Se mai poate observa că secvențele de cod care redirectă întreruperile folosind funcția DOS 25h se află în secțiuni de cod critice (întreruperile sunt dezactivate). Așa cum am spus la începutul acestui capitol acest lucru nu este în general necesar atunci când redirectăm o singură întrerupere, întrucât funcția DOS 25h asigură protejarea secvenței de cod care modifică adresa vectorului de intercalarea cu apeluri de întreruperi. De ce am dezactivat totuși secvențele de redirectare a întreruperilor la instalarea și dezinstalarea TSR-ului? Acest lucru este necesar mai ales atunci când TSR-ul funcționează corect doar având redirectate toate cele trei întreruperi folosite. Dacă întreruperile nu ar fi dezactivate în acest caz, ar fi posibilă generarea unei întreruperi într-o stare intermediară: când nu au fost redirectate încă toate handler-ele, lucru care poate să ducă la funcționarea incorrectă sau chiar să afecteze stabilitatea sistemului. Cel mai elovent exemplu în

această direcție este acela în care sunt redirectate două întreruperi asincrone: 9h (tastatură) și 13h (accesul la discul magnetic). Ambele sunt generate asincron în general întrucât operații de scriere pot fi generate nu doar de aplicația ce rulează în prim plan ci și de sistemul de operare. Acum dacă generarea unei întreruperi 9h (apăsare tastă) trebuie să apeleze întreruperea 13h pentru o activitate cu discul – acest lucru poate duce la blocarea sistemului întrucât întreruperea 13h cu implementarea presupusă de handler-ul int 9h nu este încă instalată.

6.9. DEPANAREA PROGRAMELOR TSR

O problemă importantă de tratat odată cu conceperea și implementarea programelor TSR este aceea a depanării acestora. Depanarea unui program se face în general folosind o unealtă de depanare. În cazul pachetului de dezvoltare Borland, această unealtă se numește Turbo Debugger și permite încărcarea programelor și execuția pas cu pas, urmărirea valorilor variabilelor, setarea unor puncte de întrerupere, etc. O sesiune clasică de depanare a unui program tranzient implică aşadar lansarea în execuție a depanatorului (Turbo Debugger) și încărcarea programului de depanat. Acest lucru este ilustrat în figura 6.9.

Caracterul asincron al execuției unui program TSR face imposibilă depanarea folosind mijlocele cunoscute pentru programele tranziente. Într-o sesiune obișnuită Turbo Debugger putem testa doar partea tranzientă a unui program TSR. Aceasta este secțiunea programului principal care se ocupă cu instalarea și dezinstalarea programului TSR. După execuția secțiunii tranziente a programului TSR, acesta se termină lăsând cod rezident în memorie și cu aceasta ia sfârșit și sesiunea de depanare a programului.

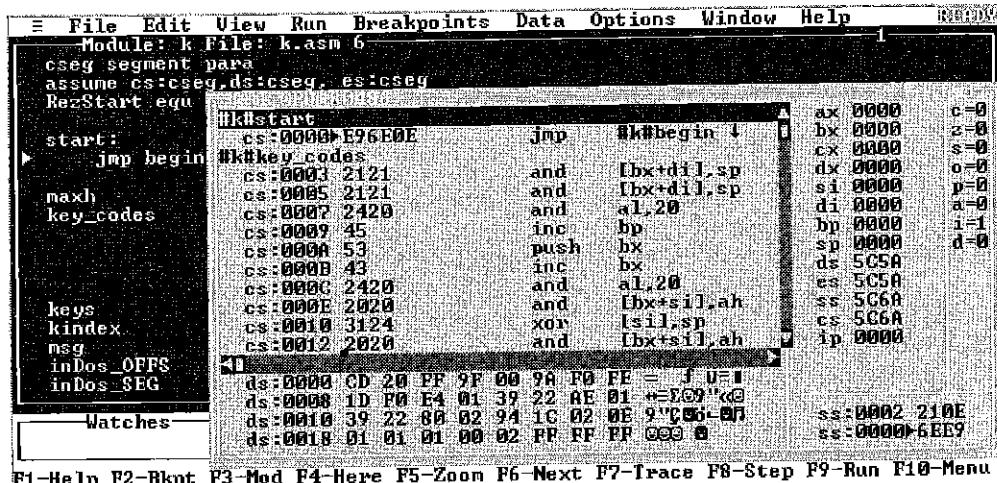


Fig. 6.9. Sesiune normală de depanare a unei aplicații tranziente cu Turbo Debugger.

Figura 6.10 ilustrează starea în care se ajunge cu depanatorul Turbo Debugger după terminarea execuției secțiunii tranziente. Se poate observa că după terminarea execuției programului singurele opțiuni pe care meniu Run ni le oferă sunt acelea de a relansa aplicația în depanare.

Dacă însă încercăm aceasta, singurul lucru pe care îl vom realiza va fi re-executarea secțiunii tranziente a TSR-ului încă o dată. Desigur nu dorim acest lucru deoarece TSR-ul este deja instalat în memorie. Execuția încă o dată a acesteia nu va face decât să încearcă instalarea lui a două oară. Nu există nici o opțiune în meniu Run care să ne permită interacțiunea cu secțiunea de cod care tocmai a rămas rezident.

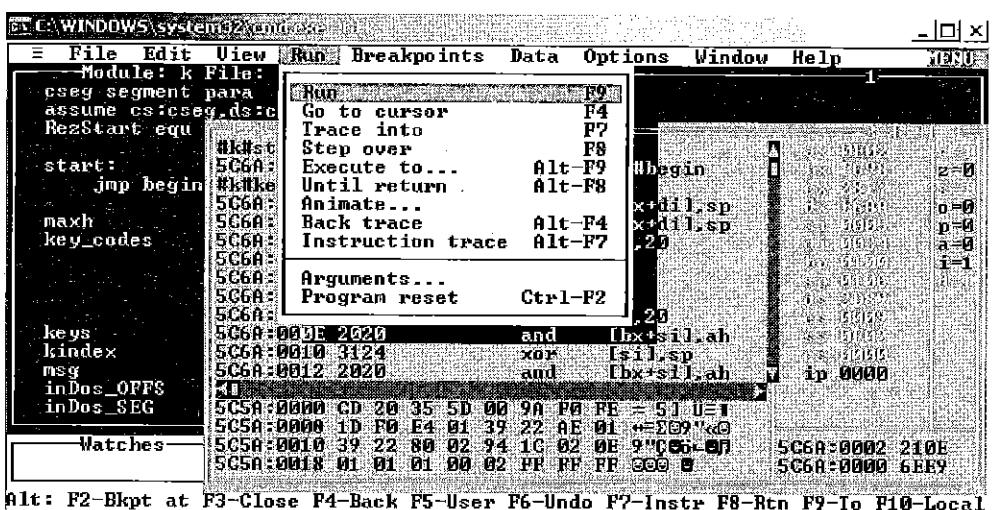


Fig. 6.10. Terminarea sesiunii de depanare a secțiunii tranziente a unui TSR.

Mai mult, dacă părăsim mediul Turbo Debugger după rularea și instalarea TSR-ului (din Turbo Debugger) putem observa că aplicația nu pare a fi instalată. Dacă luăm ca exemplu aplicația KeybMon vom observa că după instalarea acesteia în Turbo Debugger și părăsirea mediului Turbo Debugger orice încercare de activare a TSR-ului se va solda cu un eșec. Acest lucru se datorează faptului că Turbo Debugger face o copie a stării sistemului la lansarea sa în execuție și reduce sistemul la starea originală la părăsirea mediului integrat. Această implementare a utilitarului Turbo Debugger pare nefondată și neavenuță întrucât distrugе programele TSR pe care le-am încărcat în memorie pe parcursul sesiunii de depanare. La părăsirea mediului integrat starea memoriei sistemului, perifericelor, vectorii de întrerupere, etc sunt readuse la cele dinaintea lansării în execuție a mediului Turbo Debugger. Dacă stăm să ne gândim puțin la scopul pentru care a fost creat acest utilitar, alegerea acestei metode de salvare și refacere a mediului devine evidentă. Turbo Debugger a fost creat pentru a permite depanarea aplicațiilor. Acest lucru înseamnă că atunci când avem nevoie de utilizarea Turbo Debugger în cazul unei aplicații, aceasta conține încă erori de implementare sau funcționare. Unele dintre aceste erori pot fi chiar erori care duc la coruperea

memoriei sistemului (modificarea cu bună știință sau fără a unor zone de memorie care nu sunt direct alocate programului în execuție), erori induse de setarea incorectă a unor vectori de întrerupere sau de faptul că am uitat restaurarea acestora la terminarea aplicației. Majoritatea erorilor din categoriile de mai sus duc irevocabil la blocarea sistemului (atunci când vorbim de o mașină DOS nativă). Pentru a salva programatorii de la supliciul restartării sistemului la fiecare eroare de acest gen din aplicație, proiectanții Turbo Debugger au ales acest mecanism de protejare a sistemului de operare față de aplicațiile în curs de depanare. Desigur chiar și această politică are o limită. Dacă aplicația în cauză suprascrie din greșelă chiar zona de memorie alocată în sistem utilitarului Turbo Debugger, acest lucru va împiedica buna funcționare a acestuia și va duce din nou la blocarea sistemului.

Revenind înapoi la ideea de la care am pornit, nu am dat încă un răspuns la întrebarea: "Cum depanăm secțiunea rezidentă a unui program TSR?". Din fericire mediul Turbo Debugger este adaptat și pentru depanarea programelor rezidente. Pentru aceasta există opțiunea *Resident* din meniul *File* al Turbo Debugger.

Imediat după activarea opțiunii ne vom întoarce înapoi în linia de comandă a interpretorului DOS. Diferența față de cazul în care părăsim Turbo Debugger prin opțiunea *Quit* este aceea că în acest caz însăși mediul Turbo Debugger devine rezident – se transformă într-un TSR. Toate aplicațiile încărcate în sesiunea de depanare rămân de data aceasta alocate în memorie împreună cu mediul de depanare. Astfel, pentru demararea unei sesiuni de depanare a secțiunii rezidente a unui TSR vom încărca programul TSR în Turbo Debugger, executăm secțiunea tranzientă până la terminarea acesteia, moment în care Turbo Debugger afișează căsuță cu mesajul *Resident, exit code 0* și apoi activăm varianta rezidentă a Turbo Debugger. Pașii de mai sus sunt ilustrați în figura 6.11.

Înainte însă de a trece Turbo Debugger în mod rezident vom avea grijă să plasăm puncte de întrerupere (breakpoints) în rutinele rezidente care dorim să le depanăm. Pentru depanarea handler-ului întreruperii 9h din programul KeybMon vom plasa un punct de întrerupere pe secvența de cod care ne interesează din această rutină.

Pentru exemplul din figura 6.12, la fiecare apăsare de tastă diferită de F9 (în linia de comandă DOS sau din orice aplicație lansată între timp), mediul Turbo Debugger se va activa și va opri execuția TSR-ului la linia indicată. Vom avea astfel opțiunea de a executa pas cu pas rutina de tratare a întreruperii, să inspectăm valorile din registrii CPU, să inspectăm valorile variabilelor declarate în aplicație, etc. Într-un cuvânt, de îndată ce Turbo Debugger se activează la întâlnirea unui punct de întrerupere vom avea oportunitatea de a depana codul rezident în același fel în care am depana o secțiune tranzientă a unui program DOS normal.

Cât timp Turbo Debugger este rezident și reluăm controlul în linia de comandă a interpretorului DOS putem să îl activăm și în afara cazurilor în care se întâlnește un punct de întrerupere. Pentru aceasta vom folosi o combinație de taste standard care activează Turbo Debugger. Cu setările inițiale ale pachetului Borland această combinație de taste este CTRL+Break, dar ea poate fi ușor modificată schimbând setările mediului Turbo Debugger. Figura 6.13 prezintă mediul Turbo Debugger activat la întâlnirea punctului de întrerupere setat în figura 6.12.

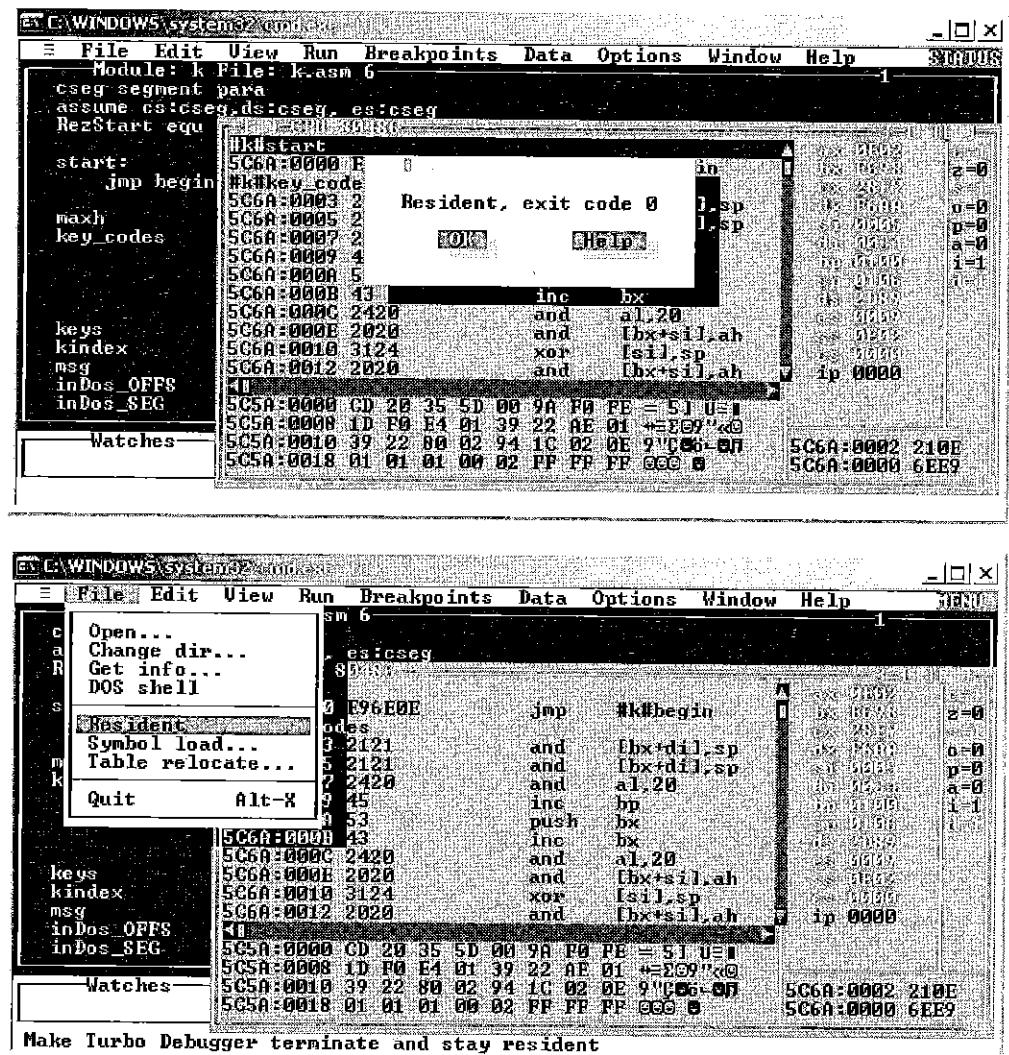


Fig. 6.11. Pași necesari pentru depanarea secțiunii rezidente a unui TSR.

În figura 6.13 se poate observa că mediul Turbo Debugger se află în depanarea instrucțiunii de la adresa la care a fost plasat punctul de întrerupere. Valorile registriilor sunt cele vizibile în binecunoscuta deja fereastră CPU. O astfel de sesiune de depanare durează până când utilizatorul părăsește mediul Turbo Debugger folosind opțiunea *File->Quit* care dezinstalează mediul de

depanare din memorie și reface starea sistemului la cea originală premergătoare lansării sale în execuție.

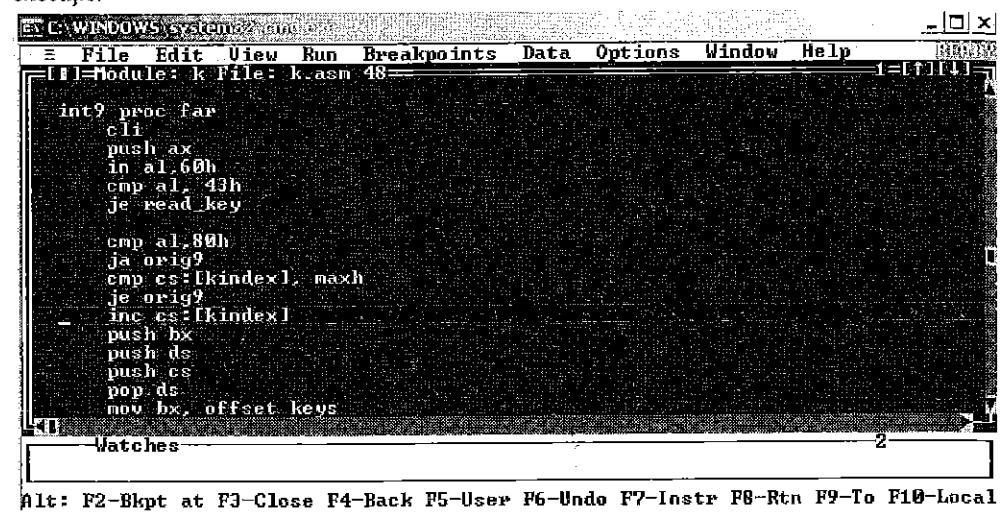


Fig. 6.12. Setarea punctelor de întrerupere în rutinele rezidente.

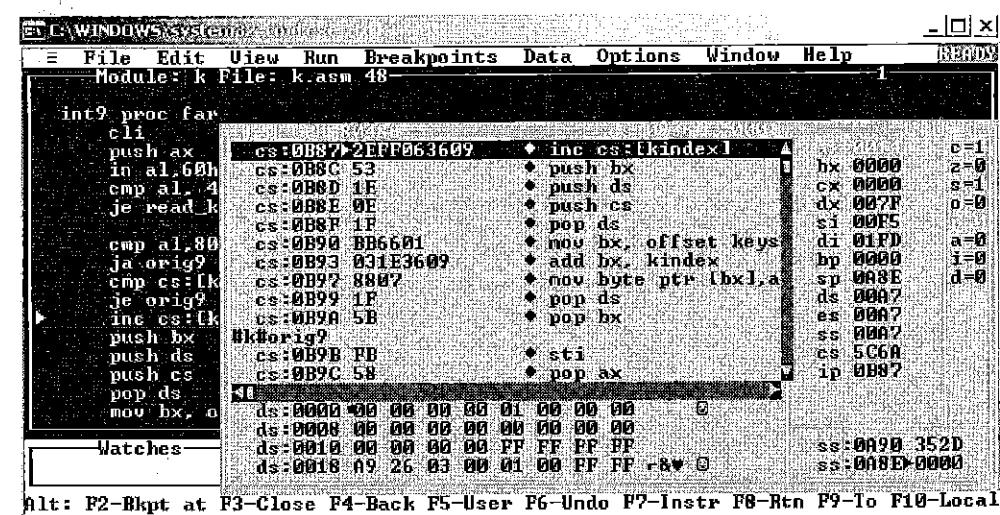


Fig. 6.13. Activarea mediului Turbo Debugger la întâlnirea punctului de întrerupere setat anterior.

Înainte de a încheia acest capitol vom mai da câteva sfaturi utile referitoare la depanarea aplicațiilor TSR. În general aplicațiile TSR se prezintă foarte bine la formatul COM de fișier executabil.

Aceasta deoarece majoritatea sunt mici ca dimensiuni și sunt construite într-un singur segment care înglobează atât codul cât și datele. Problema care apare la depanarea programelor COM este aceea că ele nu pot fi asamblate cu informații de depanare. Editorul de legături nu stochează informații de depanare în cazul formatului COM, ceea ce ne obligă ca atunci când am scris un astfel de program să sim nevoi să îl depanăm la nivel de cod mașină și nu la nivel simbolic (este accesibil doar codul mașină și adresele numerice). Acest lucru îngreunează foarte mult procesul de depanare întrucât codul mașină nu este atât de lizibil și nu mai avem acces la informațiile simbolice pe care ni le furnizează prezența numelor de etichetă și de variabile. Cu ajutorul unui simplu artificiu putem repara însă acest lucru. Dacă construim un astfel de program într-un mod similar cu programul KeybMon descris anterior, putem să îl transformăm simplu pe durata depanării programului într-un program EXE pe care să îl asamblăm cu informații complete de depanare. Să vedem modul în care este alcătuit programul și cum ne poate ajuta acest lucru:

```
cseg segment para
assume cs:cseg,ds:cseg, es:cseg
;org 100h
RezStart equ $
```

```
start:
    jmp begin
```

```
maxh    EQU 2000
```

```
...
Cseg ends
End start
```

Un program COM poate fi transformat foarte simplu într-unul EXE dacă prima instrucție din program se află la eticheta *start* iar programul se termină cu construcția specifică: marcator sfărșit segment urmat de eticheta *end start*. Prin simpla comentare a directivei *org 100h* specifică programelor COM vom obține la asamblare un program EXE. Acesta poate fi asamblat cu informații complete de depanare simbolică și depanat ca atare. După depanare se refac directiva *org 100h* pentru a obține un program COM.

6.10. TSR-URI ȘI REDIRECTARE ÎNTRERUPERI ÎN CADRUL SO WINDOWS

Întrucât în prezent nu mai există multe sisteme care să ruleze nativ sistemul de operare MS-DOS este bine de știut faptul că nu orice program DOS nativ va putea rula corect pe un sistem Windows. Sistemele de operare din familia Windows nu dispun de un mediu nativ DOS ci emulează o arhitectură hardware virtuală în cadrul căreia rulează apoi un sistem de operare MS-DOS (în general îmbunătățit). Fiecare *Command Prompt* reprezintă în realitate o instanță a mașinii virtuale în care se rulează sistemul de operare DOS. Având în vedere că este vorba de o mașină virtuală înseamnă că și dispozitivele periferice, de stocare, de I/O, etc sunt virtuale. Sistemul de operare Windows emulează aceste periferice pentru fiecare mașină virtuală în parte.

Acest lucru are avantaje și dezavantaje din punct de vedere DOS. Putem rula practic oricâte sisteme de operare DOS în tot atâtea mașini virtuale pe aceeași mașină fizică. Pe de altă parte accesul la nivel jos la unele dispozitive nu poate fi emulat întotdeauna la perfecție. Astfel există unele întreruperi pe care nu le putem redirecționa cu succes într-o mașină virtuală. Întreruperea 13h de lucru cu discul este un exemplu în acest sens. Modul în care este concepută interfața Windows de acces la discul magnetic face ineficiență și practic imposibilă emularea accesului fizic la discul magnetic într-o mașină virtuală. În general întreruperile care pun probleme sunt cele care sunt legate direct la dispozitive fizice. Lucrul cu moduri grafice în mod fereastră Windows este sortit eșecului – astfel încât o multitudine de funcții ale întreruperii 10h nu vor funcționa corect.

Ca o concluzie este bine să reținem că diferențele între o mașină virtuală și una reală fac imposibilă uneori funcționarea programelor DOS native pe un sistem Windows – fără ca acestea să fie modificate apriori. În unele cazuri acest lucru nu este posibil deloc.

CAPITOLUL 7

IMPLEMENTAREA APELULUI DE SUBPROGRAME

7.1. COD DE APEL, COD DE INTRARE, COD DE IEȘIRE

Regulile de implementare ale apelurilor de subrutine în cadrul limbajelor de programare presupun parcurserea următoarelor etape:

1. efectuarea apelului de către unitatea de program apelantă
2. intrarea în procedură
3. ieșirea din procedură

Pentru fiecare dintre aceste 3 etape se va genera corespunzător un cod de apel, cod de intrare și respectiv cod de ieșire.

7.1.1. Cod de apel

Codul de apel constă din efectuarea de către apelator a următoarelor acțiuni (în această ordine):

- (i) dacă subprogramul apelat este o funcție care întoarce un string, se va pune în stivă adresa stringului rezultat (segment:offset)
- (ii) punerea parametrilor în stivă
- (iii) generarea unei instrucțiuni CALL pentru apelarea propriu-zisă a subprogramului, care are ca și efect salvarea în stivă a adresei de revenire, care poate să fie far (segment:offset) sau near (specificată doar prin offset), în funcție de tipul de apel al subprogramului

Aceste acțiuni constituie ceea ce numim **codul de apel** al subprogramului respectiv. Dacă apelul subprogramului se face dintr-un limbaj de nivel înalt, codul de apel este generat automat. Dacă apelul se face din limbaj de asamblare, nu este obligatoriu (insă este recomandabil) ca aceste acțiuni să fie scrise explicit de către programator (este obligatoriu atunci când subprogramul apelat din limbaj de asamblare este definit într-un limbaj de nivel înalt). Ca o traducere în limbaj de asamblare a codului de apel generat de către unitatea apelantă la apelul unui subprogram, putem scrie:

- | | | |
|---------------------------|--------------------------|---|
| (i) push seg (rez) | push offset (rez) | ;punerea în stivă a adresei stringului rezultat (numai
;pentru funcțiile care întorc string) |
|---------------------------|--------------------------|---|

- (ii) sub sp, parametri ;valoarea lui sp (adresa primului cuvânt liber din stivă) se modifică, scăzându-se atâtia octeți căi sunt necesari pentru depunerea parametrilor în stivă
- (iii) push seg (adr_revenire) ;memorarea în stivă a adresei de revenire (far sau push offset (adr_revenire);near)

Așadar, ca efect al apelului de tip far al unui subprogram care întoarce un string ca și rezultat, stiva va avea următorul conținut:

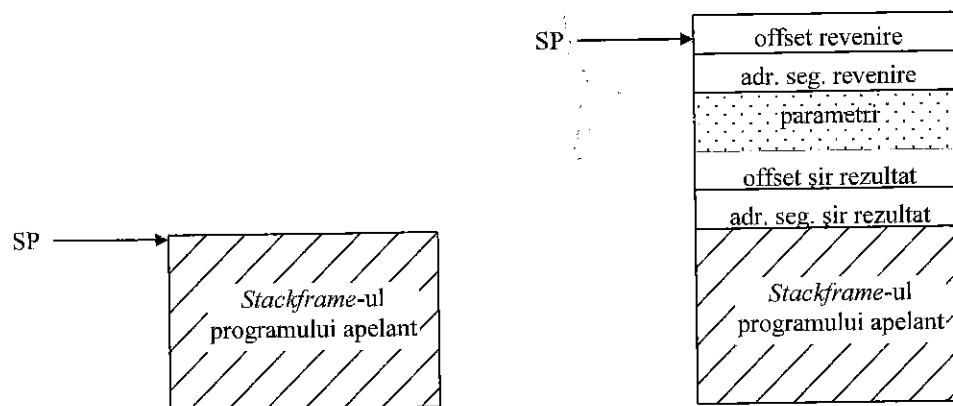


Fig. 7.1.a. Organizarea stivei înainte de apelul funcției

Fig. 7.1.b. Organizarea stivei după apelul funcției

7.1.2. Cod de intrare

Intrarea într-un subprogram presupune înainte de execuția primei instrucțiuni efectuarea, în această ordine, a următoarelor acțiuni:

- (i) izolarea stivei, adică definirea bazei zonei de stivă (*stackframe*) utilizate pentru desfășurarea execuției subruteinei curente
- (ii) alocarea de spațiu în stivă pentru eventualul rezultat de tip diferit de string (care în final se va transmite programului apelant prin intermediul regiștrilor) întors de acest subprogram
- (iii) alocarea de spațiu în stivă și realizarea copiilor locale ale eventualilor parametri de dimensiuni mari (>4 octeți) transmiși prin valoare și pentru care nu s-a realizat copierea pe stivă în cadrul codului de apel

- (iv) alocarea de spațiu pentru datele locale
- (v) alocarea de spațiu în stivă pentru rezultatul de tip string întors de eventualele funcții care sunt apelate în subprogramul respectiv.

Acste acțiuni constituie ceea ce numim **codul de intrare** în subprogramul respectiv. Dacă acest subprogram este scris într-un limbaj de nivel înalt, codul de intrare este generat automat (de exemplu în Pascal la întâlnirea liniei *begin*). Dacă subprogramul este scris în limbaj de asamblare, codul de intrare trebuie scris explicit de către programator dacă subprogramul va fi apelat dintr-un limbaj de nivel înalt. Ca o traducere în limbaj de asamblare a codului de intrare generat la intrarea într-un subprogram, putem scrie:

- (i) push bp ;bp va primi valoarea lui sp, astfel încât acesta să refere vârful stivei; mai întâi însă salvăm valoarea lui bp, urmând să o restaurăm înainte de ieșirea din subrutină, pentru a nu-i altera valoarea pe care o avea la intrarea în subrutină
- mov bp, sp ;bp definește acum baza zonei de stivă (*stack frame*) utilizată pentru desfășurarea execuției subruteinei curente
- (ii) sub sp, n ;alocarea de spațiu în stivă pentru eventualul rezultat de tip diferit de string întors de acest subprogram; n reprezintă numărul de octeți necesari pentru reprezentarea acestui rezultat
- (iii) sub sp, 100h ;alocarea de spațiu în stivă pentru realizarea copiilor locale ale eventualilor parametri de tip string transmiși prin valoare (aici s-a alocat spațiu pentru un string, 100h)
- ;efectuarea copiei locale pentru care s-a alocat spațiu prin instrucțiunea anterioară (ca exemplu, vezi 7.2.4.2.)
- (iv) sub sp, locale ;alocarea de spațiu în stivă pentru datele locale; locale reprezintă numărul de octeți necesari pentru reprezentarea datelor locale
- (v) sub sp, 100h ;alocarea de spațiu în stivă pentru rezultatul de tip string întors de eventualele funcții care vor fi apelate în acest subprogram (aici s-a alocat spațiu pentru un string, 100h)

Componența stivei după intrarea în subrutină, ca efect al codului de intrare, este ilustrată în figura 7.2.

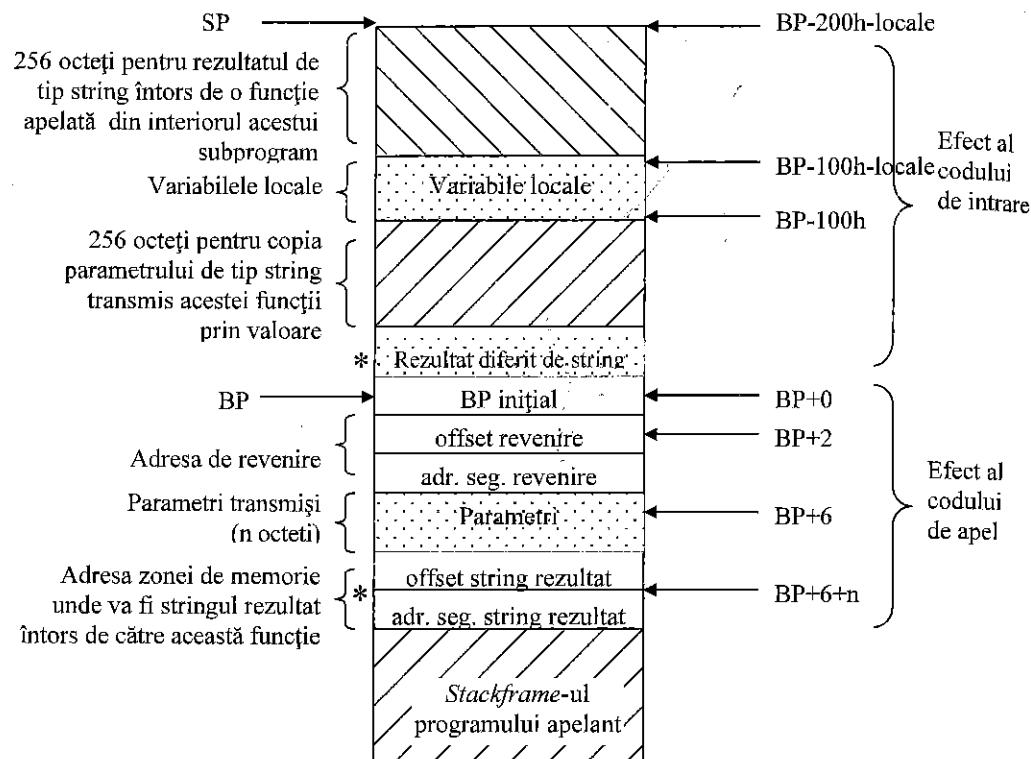


Fig. 7.2. Organizarea stivei după intrarea în subrutină

Observație:

Conținutul celor două locații din stivă marcate cu semnul * în figura 7.2. nu pot să apară concomitent după intrarea într-o subrutină:

- i) fie subrutina întoarce un rezultat de tip string, și în cazul acesta adresa řirului rezultat va fi pusă în stivă de către programul apelant, înainte de transmiterea parametrilor (făcând astfel parte din codul de apel);
- ii) fie subrutina întoarce un rezultat diferit de string, și pentru acesta se va aloca spațiu în stivă după salvarea vechiului BP (făcând astfel parte din codul de intrare), urmând ca la ieșirea din subrutină acest rezultat să fie returnat programului apelant prin intermediul regiștrilor;
- iii) fie subrutina nu întoarce nici un rezultat, caz în care conținutul ambelor locații lipsește.

7.1.3. Cod de ieșire

La ieșirea dintr-o subrutină au loc următoarele acțiuni (în această ordine):

- (i) refacerea stivei, astfel încât regiștrii care o caracterizează (SP, BP, SS) să conțină valorile pe care le aveau la intrarea în subrutină (refacerea valorilor acestor regiștri va duce la eliberarea spațiului ocupat de: eventualul rezultat de tip diferit de string întors de subrutina respectivă, copia eventualilor parametri de tip string transmiși prin valoare, variabilele locale, rezultatul de tip string returnat de eventualele funcții apelate din acest subprogram)
- (ii) revenirea din subprogram cu scoaterea din stivă a parametrilor

Aceste acțiuni constituie **codul de ieșire** dintr-un subprogram. Dacă acest subprogram este scris într-un limbaj de nivel înalt, codul de ieșire este generat automat la întâlnirea liniei *end*. Dacă subprogramul este scris în limbaj de asamblare, aceste acțiuni trebuie scrise explicit de către programator dacă acest subprogram va fi apelat dintr-un limbaj de nivel înalt. Ca o traducere în limbaj de asamblare a codului de ieșire generat la ieșirea dintr-un subprogram, putem scrie:

- | | |
|---------------------------|---|
| (i) mov sp, bp | refacem valorile lui bp și sp |
| | ;ou această ocazie se eliberează stiva de eventualul rezultat |
| | ;de tip diferit de string întors de subrutina respectivă, copia |
| | ;eventualilor parametri de tip string transmiși prin valoare, |
| | ;variabilele locale, rezultatul de tip string returnat de |
| | ;eventualele funcții apelate din acest subprogram |
| (ii) ret Parametri | revenirea din subprogram cu scoaterea din stivă a |
| | ;parametrilor; Parametri reprezintă numărul de octeți care |
| | ;au fost alocati în stivă pentru transmiterea parametrilor |

După cum se poate observa, după ieșirea din subprogram, în stivă se mai află încă adresa řirului rezultat (dacă subrutina respectivă întoarce un string ca și rezultat). Ea nu a fost încă dealocată deoarece în urma apelului unei funcții (utilizată de obicei într-o instrucțiune de atribuire de forma *E:=f(...)*) este necesară efectuarea corectă a atribuirii stringului rezultat unei expresii.

Aceasta adresă trebuie să fie în final și ea scoasă din stivă. De aceasta este răspunzător însă apelatorul, acesta fiind efectul instrucțiunii:

add sp, 004

Care va fi generată după instrucțiunea **CALL**.

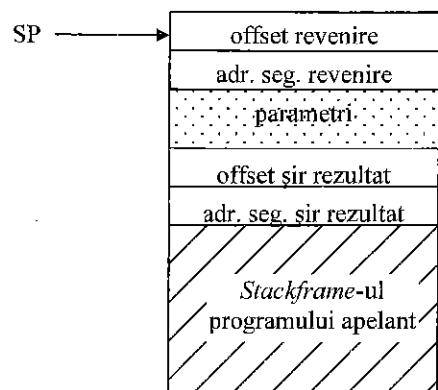


Fig. 7.3.a. Organizarea stivei după refacerea acesteia

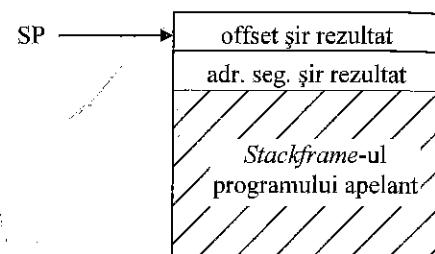


Fig. 7.3.b. Organizarea stivei după revenirea din subprogram

Se poate observa după ieșirea din subprogramul apelat și scoaterea din stivă a adresei șirului rezultat, că stiva are aceeași organizare la fel ca înainte de apelul subprogramului (figura 7.4. coincide cu figura 7.1.a.)

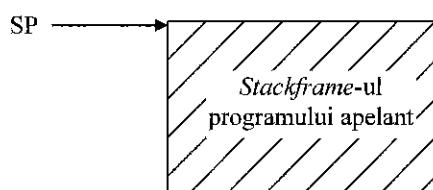


Fig. 7.4. Organizarea stivei după revenirea din subprogram și scoaterea adresei șirului rezultat

7.2. IMPLEMENTAREA SUBPROGRAMELOR ÎN TURBO PASCAL

În această secțiune vom prezenta modul în care sunt implementate subprogramele în Turbo Pascal. În acest sens vom preciza cum generează compilatorul Pascal *cod de apel*, *cod de intrare* și *cod de ieșire*.

7.2.1. Modele de memorie

Programele sunt formate dintr-unul sau mai multe segmente, fiecare constituind o parte fizică distinctă de cod sau de date (sau de amândouă) accesibilă prin intermediul unui registru de segment.

Astfel, este posibil un mare număr de combinații. S-au stabilit însă câteva modele de memorie standard. Acestea li se conformează și majoritatea limbajelor de programare de nivel înalt.

Modelele de memorie standard au segmente specifice pentru cod, date inițializate, date neinițializate, date inițializate cu acces far, date neinițializate cu acces far, constante și stivă.

Segmentul de cod conține de obicei codul unui modul (eventual și date). Datele cu acces far pot fi accesate doar modificând valoarea unui registru de segment.

Printre modelele de memorie acceptate de Turbo Assembler versiunea 2.0 sunt următoarele: tiny, small, medium, compact, large, huge, tpascal și altele.

În modelul de memorie *tiny* programul este format dintr-un singur segment care conține atât date cât și cod. Datele sunt accesate cu adrese near, iar instrucțiunile de salt, având destinația în același segment, sunt near. Acest model de memorie este folosit pentru programele *com*. Datele, codul și stiva se află într-un grup numit *dgroup*.

Modelul de memorie *small* presupune un segment de cod și un segment de date. Este cel mai utilizat model de memorie pentru programele scrise în întregime în limbaj de asamblare. Segmentul de date și segmentul de stivă sunt reunite împreună într-un grup numit *dgroup*.

În cazul modelului de memorie *medium* există mai multe segmente de cod, iar datele și stiva se află într-un grup numit *dgroup*.

Modelul de memorie *compact* presupune codul ca aflându-se într-un singur segment. Datele și stiva se află într-un grup numit *dgroup*. Datele sunt accesate folosind adrese far.

Modelul de memorie *large* se caracterizează prin aceea că există mai multe segmente de cod, iar datele și stiva sunt într-un grup numit *dgroup*. Datele sunt accesate folosind adrese far.

Modelul de memorie *huge* este similar modelului de memorie *large* în ceea ce privește limbajul de asamblare. La C, însă, există diferențe care provin din dimensiunea datelor care pot să fie declarate în segmentul de date.

7.2.2. Harta memoriei Turbo Pascal

Pentru a clarifica modelul de memorie al unei aplicații Turbo Pascal și deci care sunt tipurile segmentelor unei astfel de aplicații, considerăm că este interesant să prezentăm harta memoriei unei aplicații Turbo Pascal. Aceasta este prezentată în continuare:

Zona PSP a fost prezentată în capitolul 5. Adresa din memorie la care începe aceasta este furnizată de identificatorul global Pascal *PrefixSeg*.

Adresa de segment la care se află constantele cu tip poate fi aflată cu ajutorul identificatorului *DSeg*. Acestea încep la deplasamentul 0000 în acest segment. În continuare, în același segment se află variabilele globale ale programului Pascal.

Segmentul care cuprinde constantele cu tip și variabilele globale este segmentul de date global.

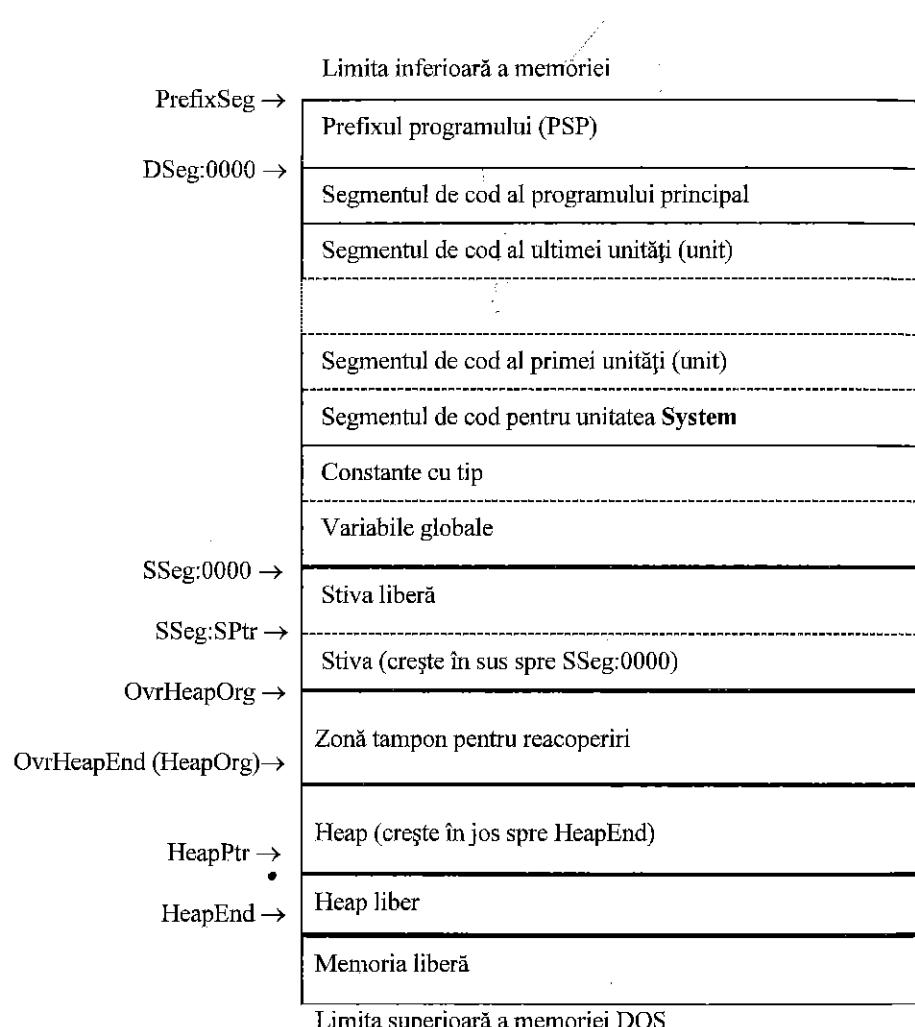


Fig. 7.5. Harta memoriei Turbo Pascal.

În continuare se află stiva. Adresa de început a zonei de stivă liberă este *SSeg:0000*, iar adresa de sfârșit a acestei zone este *SSeg:SPtr*, care este totodată și adresa vârfului stivei.

Zona tampon pentru reacoperire este utilizată de unitatea *Overlay*. Adresa ei de început este *OvrHeapOrg*, iar adresa de sfârșit este *OvrHeapEnd* (coincide cu *HeapOrg*).

La sfârșitul memoriei DOS se află heap-ul. Limitele acestei zone sunt *HeapOrg* și *HeapEnd*. Acesta este alocat începând cu adrese mici, zona ocupată fiind cuprinsă între *HeapOrg* și *HeapPtr*, iar zona liberă între *HeapPtr* și *HeapEnd*.

Identifierii *PrefixSeg*, *DSeg*, *SSeg* sunt adrese de segment (valori de tip Word), iar identifierii *HeapOrg*, *HeapPtr* și *HeapEnd* sunt adrese far (valori de tip Pointer).

7.2.3. Cod de apel al subprogramelor Pascal

Pentru limbajul Pascal, codul de apel constă din acțiunile descrise în secțiunea 7.1.1., adică:

- punerea parametrilor în stivă;
- generarea unei instrucțiuni CALL pentru apelarea propriu-zisă a subprogramului.

Tipul instrucțiunii CALL generate este conform cu tipul de apel al subprogramului: dacă subprogramul este apelat far, tipul instrucțiunii CALL este far, respectiv dacă subprogramul se apelează near, instrucțiunea CALL generată este near. În continuare vom prezenta mai amănunțit modul în care parametrii sunt puși în stivă precum și tipuri de apel al subprogramelor în Pascal.

De asemenea, în cazul subprogramelor imbricate, codul de apel generat de compilatorul Pascal este puțin deosebit. Acesta va fi prezentat în paragraful 7.2.6.

7.2.3.1. Transmiterea parametrilor

Turbo Pascal transmite parametrii utilizând stiva. Dacă există coprocesor numeric, parametrii valoare *Single*, *Double*, *Extended* și *Comp* sunt transmiși prin intermediul stivei acestuia. Parametrii sunt evaluati și puși în stivă în ordinea în care aceștia apar în declarația subprogramului.

Parametri de tip valoare

Aceștia se transmit prin valoare sau referință, în funcție de tipul și dimensiunea lor. În general, dacă lungimea valorii unui parametru este de 1, 2 sau 4 octeți, valoarea respectivă este pusă în stivă. În caz contrar, în stivă se pune un pointer la o locație de memorie care conține valoarea parametrului (deci se transmit ca și cum ar fi declarati ca parametri de tip referință).

Parametrii de lungime 1 octet sunt convertiți la tipul WORD, partea *low* a rezultatului conținând valoarea respectivă, iar partea *high* având o valoare nedefinită. Pentru parametrii a căror valoare

este formată din două cuvinte (DWORD) în stivă se pune mai întâi partea **high**, pe urmă partea **low**.

În continuare prezentăm ce se pune în stivă corespunzător unui parametru valoare de un tip dintre cele precizate:

TIP	CE SE PUNE ÎN STIVĂ
<i>Char</i>	- octet fără semn
<i>Boolean</i>	- octet (valoare 0 sau 1)
tip enumerare	- octet fără semn, dacă enumerarea are cel mult 256 de valori - cuvânt fără semn, în caz contrar
<i>Real</i>	- 6 octeți în stivă (exceptie !)
tip reprezentat în virgulă flotantă	- 4, 6, 8, 10 octeți în stivă coprocesorului numeric
<i>Pointer</i>	- 2 cuvinte în stivă
<i>String</i>	- pointer (far) la valoare
tip mulțime	- adresa unei mulțimi care ocupă 32 de octeți
<i>Array, Record</i>	- valoarea în stivă, dacă lungimea este de 1, 2 sau 4 octeți - pointer la valoare, în caz contrar

Parametri referință

Pentru aceștia se pune în stivă un pointer far la locațiile lor de memorie.

7.2.3.2. Tipuri de apel al subprogramelor

În ceea ce privește apelul, implementarea Turbo Pascal prezintă următoarele convenții de utilizare a subprogramelor:

Se consideră că se apelează FAR:

- subprogramele declarate în secțiunea **interface** a unei unități;
- subprogramele a căror declarație este în domeniul de acțiune al unei directive de compilare {\$F+} sau sunt declarate folosind directiva **far**;
- toate subprogramele declarate la nivelul cel mai exterior al unui program Pascal, dacă compilarea se face cu opțiunea de apel FAR implicit.

Se consideră că se apelează NEAR:

- toate subprogramele declarate la nivelul cel mai exterior al unui program Pascal, dacă compilarea se face fără opțiunea de apel FAR implicit și al căror antet nu este în domeniul de acțiune al unei opțiuni de compilare {\$F+};
- toate subprogramele declarate folosind directiva **near**;
- subprogramele declarate în secțiunea **implementation** a unei unități.

7.2.3.3. Exemplu

Presupunem că avem următorul program Pascal:

```
program Exemplu1;
var AY: Byte;
    AS, S: String;
Procedure A (X: Integer; var Y: Byte; S: String);
1 begin
2     Y := Lo(X);
3 end;

Function B(N: Integer): String;
var T: Real;
4 begin
5     B[0] := Chr(N);
6 end;

7 begin
8     A(5, AY, AS);
9     S := B(7);
10 end.
```

Codul generat de către compilatorul Pascal la întâlnirea apelului procedurii A, în linia numerotată în exemplu cu 8, este:

```
mov ax, 0005h
push ax
mov di, 0050h
push ds
push di
mov di, 0052h
push ds
push di
call EXEMPLU1.A
```

; punе în stivă valoarea parametrul X
; di <- deplasamentul lui AY în segmentul de date
; punе în stivă adresa de segment a lui AY
; punе în stivă offset-ul lui AY
; di <- offset-ul lui AS
; punе în stivă adresa de segment a lui AS
; punе în stivă offset-ul lui AS

} vezi 7.1.1. (ii)

; vezi 7.1.1. (iii)

7.2.4. Cod de intrare în subprogramele Pascal

Înainte de a preciza care este codul de intrare generat de către compilatorul Pascal, vom prezenta modul în care funcțiile Pascal întorc rezultatul modulului apelant.

7.2.4.1. Întoarcerea rezultatului de către funcții

Funcțiile Turbo Pascal întorc rezultatul în diferite moduri, depinzând de tipul acestuia. Astfel,

- rezultat scalar :
 - 1 octet → în AL
 - 2 octeți → în AX
 - 4 octeți → în DX:AX (DX conține partea high)
- rezultat real : în DX:BX:AX
- rezultat reprezentat în virgulă flotantă: în regiștrii coprocesorului numeric;
- rezultat string : într-o zonă temporară alocată de Turbo Pascal în momentul compilării programului care conține un apel de astfel de funcție; un pointer la această zonă este pus în stivă înapoi de a începe să pună parametrii. Acest pointer nu face parte din lista de parametri, deci nu afectează numărul de octeți ce trebuie scoși din stivă la revenirea din procedură (vezi codul de ieșire);
- rezultat pointer: în DX se pune adresa de segment, iar în AX se pune deplasamentul.

Codul de intrare generat la intrarea într-un subprogram Pascal efectuează toate acțiunile descrise în secțiunea 7.1.2. În particular, și programul principal Pascal este tratat ca un subprogram.

7.2.4.2. Exemplu

Pentru exemplul prezentat mai sus, codul de intrare generat la instrucțiunea **begin** corespunzătoare programului principal (linia 7 din exemplu) este următorul:

```
call 689Ch:0000h
push bp
mov bp, sp
mov ax, 0100h
call 689Ch:02CDh
sub sp, 0100h } generarea cadrului de stivă
```

Efectul primei instrucțiuni este apelarea părții de inițializare a unității System. Dacă ar mai fi folosite unități care au cod de inițializare, în continuare s-ar apela codurile de inițializare ale acestora. Secvența

```
push bp
mov bp, sp
```

Cap.7. Implementarea apelului de subprograme.

are ca și efect "izolarea stivei" astfel încât programul să nu coboare în zona care a fost ocupată până acum - vezi 7.1.2. (i).

Următoarele trei instrucțiuni urmăresc alocarea de spațiu în stivă pentru rezultatul de tip string întors de către funcția B care este apelată în cadrul programului principal – vezi 7.1.2. (v). Valoarea 0100h reprezintă cantitatea necesară pentru alocare. Se apelează o rutină care verifică dacă această cantitate este disponibilă (Stack Overflow Check). Dacă se revine din această rutină atunci spațiul respectiv poate fi alocat cu instrucțiunea **sub sp, 0100h**. Dacă spațiul cerut ar fi provocat o depășire a stivei, programul s-ar fi oprit înapoi de revenirea din rutina apelată cu o eroare de execuție care cred că vă este cunoscută : Stack Overflow. Aceste trei instrucțiuni generate în mod automat de către compilator la intrarea într-un subprogram le vom numi generarea cadrului de stivă. Mai specificăm că verificarea stivei este făcută numai în condițiile în care este activă opțiunea de compilare *Stack Checking*.

Codul de intrare generat la intrarea în procedura A (linia 1 din exemplu) este următorul:

```
push bp
mov bp, sp
mov ax, 0100h
call 689C:02CDh
sub sp, 0100h
mov bx, ss
mov es, bx
mov bx, ds
clc
lea di, [bp-100]
lds si, [bp+4]
lodsb
stosb
xchg cx, ax
xor ch, ch
rep movsb
mov ds, bx
```

Efectul acestui cod este următorul:

- izolarea stivei și pregătirea accesului la parametri (primele două instrucțiuni) - vezi 7.1.2. (v);
- alocarea de spațiu în stivă pentru copia locală a parametrului de tip string transmis prin valoare (următoarele trei instrucțiuni) - vezi 7.1.2. (iii);
- efectuarea copiei (restul instrucțiunilor).

Se poate observa modul în care se face accesul la parametrul S în instrucțiunea lds si, [bp+4]. În continuare, orice acces la S se va face cunoscându-i adresa de segment care este aceeași cu a segmentului de stivă (SS) precum și deplasamentul în cadrul acestui segment (bp-100h). Codul de intrare generat la intrarea în funcția B (linia 4 din exemplu) este următorul:

```
push bp
mov bp, sp
mov ax, 0006
call 689Ch:02CDh
sub sp, 0006
```

Ce se poate observa aici nou față de exemplele prezentate anterior este alocarea de spațiu pentru datele locale (o variabilă de tip Real, care va ocupa deci 6 octeți) – vezi 7.1.2. (iv).

Ce ni se pare mai interesant în ceea ce privește funcția B este codul de apel generat (pentru linia numerotată în exemplu cu 9):

```
lea di, [bp-0100h]
push ss
push di
mov ax, 0007
push ax
call EXEMPLU1.B
add sp, 0004
```

După cum am anticipat, codul de apel în acest caz mai conține punerea în stivă, înainte de parametri, a adresei locației alocate anterior pentru întoarcerea rezultatului de tip string de către funcția B – vezi 7.1.1. (i). Acest lucru îl realizează primele trei instrucțiuni.

La apelarea funcției B s-a pus în stivă adresa zonei în care această funcție va întoarce rezultatul. Aceasta adresa trebuie să fie și ea scoasă din stivă. De aceasta este răspunzător însă apelatorul, acesta fiind efectul instrucțiunii

```
add sp, 004
```

Care va fi generată după instrucțiunea CALL din cadrul codului de apel.

7.2.5. Cod de ieșire din subprogramele Pascal

Din codul de ieșire generat de către compilator la ieșirea dintr-un subprogram Pascal fac parte toate acțiunile descrise în secțiunea 7.1.3.

De exemplu, codul de ieșire din procedura A din exemplu (codul generat pentru linia 3) este:

```
mov sp, bp
pop bp } ;vezi 7.1.3. (i)
ret 000Ah ;vezi 7.1.3. (ii)
```

parametrii ocupând în stivă 10 octeți, iar codul de ieșire din funcția B (corespunzător liniei 6) este următorul:

```
mov sp, bp
pop bp } ;vezi 7.1.3. (i)
ret 0002 ;vezi 7.1.3. (ii)
```

Codul de ieșire generat pentru instrucțiunea care marchează sfârșitul programului (linia 10) este următorul:

```
mov sp, bp
pop bp
xor ax,ax
call 689Ch:0116h
```

Efectul ultimei instrucțiuni este apelarea procedurii de ieșire (variabila ExitProc conține adresa acesteia), căreia î se transmite în AX codul de return al programului (în cazul nostru valoarea 0).

7.2.6. Proceduri și funcții imbricate în Turbo Pascal

O procedură sau funcție se spune că este imbricată atunci când este declarată în cadrul altor funcții sau proceduri. Implicit, procedurile și funcțiile imbricate folosesc întotdeauna modelul de apel near, deoarece ele sunt vizibile doar în proceduri sau funcții aflate în același segment de cod. Totuși, se poate utiliza directiva {\$F+} pentru a forța ca toate procedurile și funcțiile care urmează să fie far, inclusiv cele imbricate (sau se poate utiliza alternativ directiva *far*).

Când se apelează o procedură sau funcție imbricată, compilatorul generează o instrucțiune PUSH BP înainte de instrucțiunea CALL, având ca efect transmiterea BP-ului apelantului ca un parametru suplimentar.

După ce procedura apelată și-a setat propriul BP, BP-ul apelantului este accesibil ca un cuvânt memorat la [BP+4] sau la [BP+6], după cum procedura este near sau far. Astfel, pot fi accesate variabilele locale din porțiunea de stivă care este a apelantului.

Luăm ca exemplu următorul program Pascal:

```

program Exemplu2;

procedure PA; near;
var
    IntA: Integer;

procedure B; far;
var
    IntB: Integer;
Procedure C; near;
var
    IntC: Integer;
begin
    IntC:=3;
    IntB:=4;
    IntA:=5;
end;

begin
    C;
end;

begin
    B;
end;

begin
    PA;
end.

```

Codul generat de către compilator la apelarea procedurii B este următorul:

```

push bp
push cs          ;asta deoarece procedura B este far
call EXEMPLU2.PA.B •

```

Codul generat la apelarea procedurii C este:

```

push bp
call EXEMPLU2.PA.B.C

```

Codul generat pentru cele trei instrucțiuni de atribuire care constituie corpul lui C este:

```

; IntC:=3
mov word ptr [bp-02], 0003
; IntB:=4
mov di, [bp+04]
mov ss:word ptr [di-02], 0004
; IntC:=5
mov di, [bp+04]
mov di, ss:[di+06]
mov ss:word ptr [di-02], 0005

```

Deci, Pascal "știe" exact unde se află zona alocată pentru o variabilă locală. În capitolul 9 vor fi prezentate cazuri în care programatorul este cel care trebuie să aibă grijă de accesarea corectă a unei variabile locale: includerea de cod mașină și de instrucțiuni în limbaj de asamblare în sursa unui program Pascal.

7.3. IMPLEMENTAREA SUBPROGRAMELOR ÎN BORLAND C

7.3.1. Pointeri far și near

Pentru a înțelege modul de apel al subprogramelor și modul de transmitere al parametrilor spre subprograme în Borland C, trebuie spus mai întâi că, în funcție de modelul de memorie folosit, compilatorul Borland C tratează fie far, fie near, adresele variabilelor și funcțiilor [Borland98].

Astfel, dacă modelul de memorie folosit este *tiny* sau *small* atât pointerii de date, cât și adresele funcțiilor sunt considerați *near*. Dacă modelul de memorie folosit este *compact* (un segment de cod și mai multe segmente de date) adresele funcțiilor sunt considerate *near* iar adresele variabilelor sunt considerate *far*. În cazul modelului de memorie *medium* (mai multe segmente de cod și un singur segment de date) adresele funcțiilor sunt considerate a fi *far*, iar adresele variabilelor *near*. Pentru modelele de memorie *large* sau *huge*, toți pointerii sunt adrese *far*.

Modelul de memorie folosit se răsfrângă astfel în modul în care sunt transmiși parametrii prin intermediul stivei și a modului în care se generează instrucțiunile *call* (near sau far) pentru apelul subprogramelor.

Variabilelor alocate dinamic li se rezervă spațiu de către compilator în *heap*. În cazul modelelor de memorie cu un singur segment de date (*tiny*, *small* și *medium*) *heap*-ul face parte integrantă ca și stiva din segmentul de date. El începe imediat după octetii generați pentru variabilele declarate global (care se alocă în segmentul de date). Stiva ocupă același segment, însă ea folosește segmentul de date de la sfârșitul acestuia spre început. În cazul acestor modele de memorie, pointerii care indică spre *heap* sunt *near*. În cazul modelelor de memorie *compact*, *huge* și *large*, *heap*-ul se identifică într-un segment de memorie separat care este alocat de către DOS la execuție. Pointerii care indică în acest *heap* sunt considerați *far*.

7.3.2. Cod de apel al subprogramelor C

Codul de apel al subprogramelor C constă în execuția acelorași pași ca cei prezentați în secțiunea 7.1.1.:

- punerea parametrilor pe stivă;
- generarea unei instrucțiuni CALL pentru apelarea propriu-zisă a subprogramului.

Spre deosebire de limbajul Pascal, în limbajul C parametrii se transmit numai prin valoare. Astfel pe stivă se pune un cuvânt pentru tipurile de date reprezentate pe unu sau doi octeți: *char*, *enum*, *int*, precum și pentru pointerii *near* și două cuvinte pentru tipurile de date reprezentate pe patru octeți ca *long* sau pointerii *far*. Un pointer este *far* sau *near* în funcție de modelul de memorie folosit și de tipul adresei indicate de pointer – adresa unei variabile din segmentul de date sau stivă sau adresa unei funcții dintr-un segment de cod (vezi discuția de la 7.3.1.). Parametrii reali, indiferent că sunt reprezentați pe 4, 8 sau 10 octeți (*float*, *double*, respectiv *long double*) sunt transmiși prin valoare tot prin intermediul stivei. Orică alt parametru, indiferent de dimensiune (spre exemplu un *struct*) este copiat în întregime pe stivă. Pentru a evita copierea unui număr mare de octeți la fiecare apel pe stivă, se recomandă folosirea ca parametru actual a adresei parametrului dorit (acest lucru duce la punerea pe stivă a doi, sau a cel mult patru octeți în cazul în care adresa parametrului trebuie specificată *far*) și folosirea modifierului *const* (în C++) pentru a împiedica modificări nedorite în cadrul subruteinei.

O altă deosebire față de limbajul Pascal este că parametrii sunt transmiși pe stivă de la dreapta la stânga. Ultimul parametru este pus primul pe stivă, iar primul parametru este pus ultimul astfel încât acesta se găsește mai sus pe stivă, la adresa mai mică.

Instrucțiunea CALL care se generează pentru apelul propriu-zis al subprogramului este *near* sau *far* în funcție de modelul de memorie folosit. Dacă se folosește un model de memorie cu mai multe segmente de cod, atunci se generează un CALL *far*, altfel unul *near* (vezi de asemenea discuția de la 7.3.1.);

Fie următorul exemplu pe care dorim să îl compilăm ca *medium*. Funcția *numara_caractere* primește ca parametru un sir (de fapt un pointer care are ca valoare adresa sirului) și un caracter al căruia număr de apariții trebuie să îl determine în sirul dat.

```
#include <stdio.h>

int numara_caractere(char *s, char c) {
    int k = 0;
    for ( ; *s != 0; s++)
        if (*s == c)
            k++;
    return k;
}
```

Cap.7. Implementarea apelului de subprograme.

```
void main() {
    char r[] = "vacanta de vara";
    int n;
    n = numara_caractere(r, 'V');
    printf("%d\n", n);
}
```

Modelul de memorie *medium* acceptând mai multe segmente de cod, apelul call *numara_caractere* va fi generat far. Parametrii sunt puși pe stivă de la dreapta la stânga, mai întâi un cuvânt pentru parametrul *c* (caracterul *c* va ocupa octetul mai puțin semnificativ al acestui cuvânt), iar ulterior un pointer *near* cu valoare adresa sirului *r* dat ca prim parametru (pointer *near* deoarece modelul *medium* presupune un singur segment de date/stivă și nu există astfel dubii în localizarea adresei indicată de acest offset).

7.3.3. Cod de intrare în subprogramele C

Codul de intrare al unui subprogram C este mult mai simplu decât codul de intrare generat la intrarea într-o subrutină Pascal. Limbajul C acceptând doar transmiterea parametrilor prin valoare, compilatorul este scutit de generarea de cod care să copieze local pe stivă parametrii transmiși prin valoare cu dimensiunea mai mare de patru octeți pe care limbajul Pascal îi transmitea prin referință (adică copierea are loc oricum la codul de apel, indiferent de dimensiune). De asemenea, compilatorul Pascal trebuia să genereze cod special în cazul în care funcția returnă un sir de caractere. În limbajul C, returnarea unui sir de caractere se reduce la simpla returnare a unui pointer.

Astfel codul de intrare în cazul unui subprogram C se reduce la:

- izolare stivei (vezi 7.1.2. (i));
- alocarea de spațiu pentru variabilele locale (vezi 7.1.2. (iv)).

Funcțiile C întorc rezultat tot prin intermediul regiștrilor AL, AX, DX:AX. Astfel, un rezultat de tip *char* este returnat în registratorul AL, un rezultat de tip *enum*, *int*, pointer *near* în registratorul AX, iar un *long* sau un pointer *far* în regiștrii DX:AX.

Fie următorul exemplu compilat în model de memorie *small* (toți pointerii sunt *near*):

```
#include <stdio.h>

void some_function(int x, int y) {
    int k = _BP;
    printf("Address of k is %p\n", &k);
    printf("Value of BP is %x\n", k);
    printf("Address of x is %p\n", &x);
    printf("Address of y is %p\n", &y); }
```

```
void main() {
    int p = 1, r = 2;
    some_function(p, r); }
```

Programul nu face altceva decât să afișeze starea stivei din interiorul funcției `some_function`. La o rulare a programului am obținut următoarele valori:

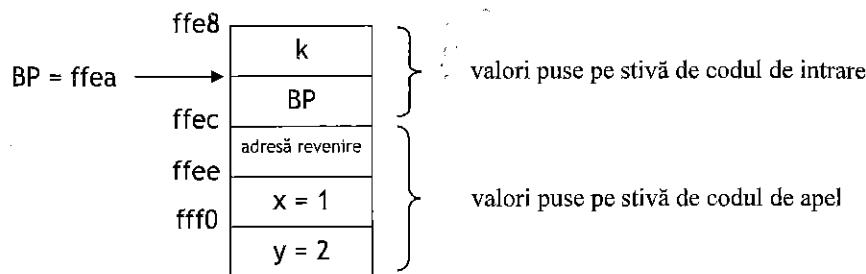
Address of k is FFE8

Value of BP is ffea

Address of x is FFEE

Address of y is FFF0

După generarea atât a codului de apel cât și a codului de intrare stiva arată astfel:



7.3.4. Cod de ieșire din subprogramele C

Codul de ieșire al unui subprogram C diferă față de codul de ieșire al unui subprogram Pascal. Ambele limbaje de programare refac stiva (vezi 7.1.3 (i)), însă la revenirea din subprogram în cazul limbajului C, acesta nu scoate de pe stivă parametrii. Scoaterea de pe stivă a parametrilor revine apelantului, cel care de altfel î-a și pus pe stivă. Astfel, codul de ieșire în cazul limbajului C constă în:

```
mov sp, bp
pop bp
ret
```

`:ret` simplu!, fără a specifica un anumit număr de octeți care să fie scoși de pe stivă. Se scoate de pe stivă doar adresa de revenire (`far` sau `near` în funcție de tipul de call cu care s-a apelat subprogramul).

CAPITOLUL 8

PROGRAMAREA MULTIMODUL

La un moment dat apare necesitatea ca un program să aibă codul sursă descris în mai multe fișiere.

Apare de asemenea necesitatea apelării, din programe scrise în limbaje de nivel înalt, de rutine scrise în limbaj de asamblare, descrise într-un fișier sursă separat. De asemenea, din module scrise în limbaj de asamblare, există posibilitatea apelării de rutine definite în module scrise în limbaj de nivel înalt.

Prin *programare multimodul* vom înțelege acel stil de descriere a codului sursă care utilizează mai multe fișiere, acestea fiind traduse în format obiect separat și linkeditate împreună.

8.1. DIRECTIVA MODEL. DIRECTIVE DE SEGMENT SIMPLIFICATE

Directiva .MODEL

Directiva `.MODEL` specifică modelul de memorie pentru un modul asamblare care utilizează pentru definirea segmentelor directive simplificate. Este cunoscut faptul că salturile `near` se fac încărcând doar registrul IP, pe când salturile `far` se realizează încărcând atât CS cât și IP. Analog, datele `near` sunt accesate doar cu ajutorul unui offset, pe când cele `far` trebuie să fie accesate prin intermediul unei adrese complete formată din segment și offset.

O sintaxă (mult simplificată) a directivei `.MODEL` este:

.MODEL model_de_memorie [,limbaj]

Parametrul `model_de_memorie` specifică modelul de segmentare utilizat de către program. Acesta este singurul parametru obligatoriu al directivei `.MODEL`. Valorile posibile pentru `model_de_memorie` sunt: **TINY**, **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, **HUGE**, **TPASCAL** etc. Aceste modele de memorie sunt cunoscute din capitolul anterior, mai puțin modelul **TPASCAL** care însă va fi prezentat în detaliu în paragraful 8.3.2.6.

Directivea `.MODEL` este necesară dacă se utilizează directive de segment simplificate, deoarece altfel Turbo Assembler nu știe cum să construiască segmentele definite de `.CODE` sau `.DATA`.

Parametrul `limbaj` specifică convențiile implicate de apel de procedură, codul de intrare și codul de ieșire generat în mod automat precum și modul în care sunt exportate (importate) simbolurile

globale (vom vedea ceva mai târziu care este rolul acestora). Valorile posibile pentru acest parametru sunt: **PASCAL**, **C**, **CPP**, **BASIC**, **FORTRAN**, **PROLOG** și **NOLANGUAGE**. Aceste convenții pot să fie suprascrise local în momentul definirii unei proceduri sau al declarării unui simbol global.

Directive de segment simplificate

Directivele de segment simplificate realizează un control relativ simplu al segmentelor și sunt potrivite pentru a lega module scrise în limbaj de asamblare măduile scrise în limbaje de nivel înalt. În schimb, acestea suportă doar câteva din facilitățile relative la segmente oferite de Turbo Assembler.

Vom prezenta pe scurt următoarele directive de segment simplificate: **.STACK**, **.CODE**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.CONST**. Acestea pot fi folosite doar dacă în prealabil a fost folosită directiva **.MODEL** pentru a defini un model de memorie.

.STACK [n]

Această directivă controlează dimensiunea stivei. Efectul ei este definirea unei stive de *n* octeți. Dacă parametrul *n* lipsește se rezervă pentru stivă 1024 octeți (1 Ko). De exemplu,

.STACK 200h

definește o stivă de 512 octeți.

Pentru a asigura funcționarea corectă a unui program care utilizează stiva, acesteia trebuie să-i fie rezervată zonă de memorie, cel puțin cantitatea estimată în cazul cel mai "rău" (din punct de vedere al umplerii stivei).

.CODE [nume]

Această directivă marchează începutul sau continuarea unui segment de cod al programului. Dacă modelul de memorie este *medium* sau *large*, atunci cu directiva **.CODE** se poate specifica și numele segmentului de cod definit. Este posibil să se genereze mai multe segmente de cod într-un singur modul.

.DATA

Această directivă marchează începutul sau continuarea segmentului de date inițializate al modulului. Înainte de a accesa locații de memorie din segmentul astfel definit, registrul de segment DS trebuie încărcat explicit cu simbolul **@data**, bineîntele prin intermediul unui registru general. Ca și efect, registrul DS va puncta spre segmentul de date care începe cu **.DATA**. De exemplu,

```
MOV AX,@data      ; @data=adresa de segment a segmentului de date
MOV DS,AX
```

.DATA?

Această directivă marchează începutul unui segment de date (ca și **.DATA**), doar că acest segment trebuie să conțină numai date neinițializate (listele de expresii din directivele de definire de date au valorile "?"). Utilizarea ei este necesară uneori când se face legarea unui modul asamblare cu un modul scris în limbaj de nivel înalt.

.CONST

Această directivă definește porțiunea din segmentul de date care conține date constante. Este utilă doar în cazul legării de module asamblare cu module scrise în limbaje de nivel înalt.

.FARDATA [nume]

Această directivă marchează începutul sau continuarea unui segment având numele specificat, care conține date inițializate FAR. Dacă numele nu este precizat, Turbo Assembler va folosi numele **FAR_DATA**. Pot exista mai multe segmente de date inițializate FAR într-un modul.

.FARDATA? [nume]

Această directivă marchează începutul sau continuarea unui segment de date neinițializate FAR. Dacă numele segmentului nu este precizat, Turbo Assembler va utiliza implicit numele **FAR_BSS**. Pot să existe mai multe segmente de date FAR neinițializate într-un modul.

Când se utilizează directive de segment simplificate pot fi accesate câteva etichete predefinite:

@FileName	= numele fișierului care este asamblat
@curseg	= numele segmentului curent care se asamblează
@code	= segmentul de cod (valoarea lui CS)
@data	= segmentul de date
@fardata	= numele segmentului de date FAR activ
@fardata?	= numele segmentului de date FAR neinițializate activ
@stack	= segmentul de stivă
@CodeSize	= 0, dacă segmentul de cod este near 1, dacă segmentul de cod este far
@DataSize	= 0, dacă segmentul de date este near 1, dacă modelul de memorie este compact sau large 2, dacă modelul de memorie este huge

Observație:

Pentru a scrie un program în limbaj de asamblare sunt suficiente directivele .DATA, .CODE și .FARDATA. Unele limbi de nivel înalt (vom vedea la limbajul C) cer ca datele neinitializate să fie definite într-un segment .DATA? sau .FARDATA?, iar datele constante să fie definite într-un segment .CONST.

Care tip de directive de segment se utilizează?

Când se leagă module asamblare cu module scrise în limbaj de programare de nivel înalt e utilă folosirea directivelor de segment simplificate, deoarece acestea gestionează singure detaliile legate de numele segmentelor și modelul de memorie, detalii care sunt asociate regulilor de legare cu limbajul de nivel înalt.

Când se scriu în limbaj de asamblare programe de tip **small** sau **medium** se preferă directivele de segment simplificate deoarece acestea sunt mai ușor de utilizat și fac programele mai lizibile.

Dacă se scriu în limbaj de asamblare programe cu mai multe segmente și care conțin cod atât far cât și **near** sau/și date atât **far** cât și **near**, atunci se folosesc directive de segment standard deoarece sunt singurile care permit controlul deplin asupra tipului, alinierii, numelor segmentelor și asupra modulului în care acestea sunt combinate.

8.2. CERINȚELE UNUI MODUL ASAMBLARE LA LEGAREA CU UN ALT MODUL

Pentru ca un modul asamblare să partajeze date sau cod cu un alt modul (scris în limbaj de asamblare sau limbaj de programare de nivel înalt) există directivele **PUBLIC** și **EXTRN**, pe care le prezentăm în continuare.

8.2.1. Directiva PUBLIC

Directivea **PUBLIC** este utilizată pentru a exporta simboluri definite în modulul asamblare în alte module. Sintaxa acesta este următoarea:

PUBLIC [limbaj] simbol[, [limbaj] simbol]

Efectul acestei directive este că simbolurile declarate, care trebuie să fie definite în modulul curent, pot să fie accesate din alte module. Parametrul *limbaj* poate să ia una din valorile **PASCAL**, **C**, **BASIC**, **ASSEMBLER**, **FORTRAN**, **PROLOG** sau **NOLANGUAGE**. Prezența acestuia impune ca simbolul aferent să fie făcut public după de acestuia și se aplică câteva reguli impuse de limbajul de programare respectiv.

De exemplu,

PUBLIC C ProcA

impune ca simbolul *ProcA* să fie exportat în celelalte module ca *_ProcA*, conform regulilor limbajului C.

Utilizarea unui specificator de limbaj cu directiva **PUBLIC** suprascrie pentru simbolurile respective setarea de limbaj curentă (implicită sau stabilită de directiva **MODEL**).

Simbolurile care pot fi făcute disponibile în alte module utilizând directiva **PUBLIC** sunt:

- nume de proceduri
- nume de variabile de memorie
- etichete definite cu ajutorul directivelor **EQU** sau =, ale căror valori se reprezintă pe 1 sau 2 octeți.

8.2.2. Directiva EXTRN

Directivea **EXTRN** este folosită pentru a face vizibile în modulul curent simboluri definite în alte module. Sintaxa acesta este prezentată în continuare:

EXTRN definiție {,definiție}

Fiecare definiție descrie un simbol, o sintaxă simplificată pentru acesta fiind următoarea:

[limbaj] nume : tip

Parametrul opțional *limbaj* indică limbajul ale cărui convenții se aplică simbolului. Valorile pe care le poate lua sunt acelea prezentate la sintaxa directivei **PUBLIC**. Parametrul *nume* reprezintă numele simbolului care este definit în alt modul. Parametrul *tip* trebuie să corespundă tipului acestui simbol, așa cum este el definit. Valorile permise sunt:

ABS	O valoare absolută (adică o etichetă definită în modulul original cu EQU sau =)
BYTE	O dată variabilă de lungime 1 octet
DATAPTR	Un pointer far sau near , depinzând de modelul de memorie curent
DWORD	O dată variabilă de lungime 4 octeți
NEAR	O etichetă în cod near
far	O etichetă în cod far
FWORD	O dată variabilă de 6 octeți
PROC	O etichetă de procedură far sau near , depinzând de modelul de memorie curent
QWORD	O dată variabilă de 8 octeți
TBYTE	O dată variabilă de 10 octeți
UNKNOWN	Tip necunoscut
WORD	O dată variabilă de 2 octeți

8.2.3. Directiva GLOBAL. Legarea de module asamblare

Pe lângă cele două directive prezentate anterior, Turbo Assembler oferă directiva **GLOBAL**, care întrunește funcțiile directivelor **PUBLIC** și **EXTRN**. Astfel, dacă într-un anumit modul este declarată o etichetă globală (cu ajutorul directivei **GLOBAL**) care apoi este definită (utilizând directivele **DB**, **DW** etc.) atunci eticheta respectivă este făcută vizibilă în alte module (ca și cum s-ar fi folosit directiva **PUBLIC**). Dacă se declară o etichetă globală, utilizând directiva **GLOBAL** într-un modul și nu este definită în acel modul, atunci aceasta este considerată ca o etichetă externă (ca și cum s-ar fi folosit directiva **EXTRN**).

Sintaxa directivei **GLOBAL** este următoarea:

GLOBAL definiție {, definiție}

Fiecare definiție descrie un simbol global și are aceeași sintaxă ca și în cazul directivei **EXTRN**.

În mod normal, asamblorul ignoră diferența dintre litere mari și litere mici, *convertind toate simbolurile publice* (adică cele declarate **PUBLIC**, **EXTRN** sau **GLOBAL**) la litere mari. Dacă se dorește să se facă distincție între litere mari și litere mici se va cere aceasta asamblorului, prin intermediul opțiunii de asamblare **/mx** sau **/ml**. În acest caz, trebuie să se utilizeze simbolurile publice în mod corespunzător.

Legarea a două module scrise în limbaj de asamblare se realizează foarte simplu, utilizând fie perechea de directive **PUBLIC - EXTRN**, fie doar directiva **GLOBAL**.

Exemplu: Următorul program este format din două module. Modulul principal are ca date locale definite trei șiruri, primele două fiind inițializate iar cel de-al treilea (numit *SirFinal*) va primi ca valoare (în final) șirul obținut prin concatenarea șirurilor inițiale. Concatenarea este făcută de o procedură apelată în primul modul și definită în cel de-al doilea modul. Deci, numele procedurii este declarat în modulul principal ca o etichetă externă, iar în modulul secundar ca o etichetă publică. Procedura de concatenare are ca "parametri" adresele celor două șiruri, adrese care vor fi pregătite în AX respectiv BX de către apelator. Rezultatul (șirul obținut prin concatenarea celor două șiruri) va fi întors în *SirFinal*. Deci, el trebuie declarat în segmentul de date al modulului secundar ca o etichetă externă de tip **BYTE** iar în modulul principal trebuie să fie declarat ca o etichetă publică.

Programul principal main.asm:

```
.MODEL SMALL
.STACK 200h
.DATA
    Sir1 DB 'Buna ', 0
    Sir2 DB 'dimineata!', '$', 0
    SirFinal DB 50 DUP (?)
```

PUBLIC SirFinal ; se poate înlocui cu **GLOBAL SirFinal:BYTE**

.CODE

EXTRN Concatenare:PROC

Start:

```
    mov ax, @data
    mov ds, ax ; încărcarea registrului ds
    mov ax, OFFSET Sir1
    mov bx, OFFSET Sir2
    call Concatenare ; SirFinal:=Sir1+Sir2
    mov ah, 9
    mov dx, OFFSET SirFinal
    int 21h ; tipărire șirului obținut
    mov ah, 4ch
    int 21h ; terminarea programului
```

END Start

Modulul secundar sub.asm:

.MODEL SMALL

.DATA

EXTRN SirFinal:BYTE ; se poate înlocui cu **GLOBAL SirFinal:BYTE**

.CODE

PUBLIC Concatenare
Concatenare PROC

```
    cld
    mov di, SEG SirFinal
    mov es, di
    mov di, OFFSET SirFinal ; es:di <- adresa șirului final
    mov si, ax ; ds:si <- adresa primului șir
    Sir1Loop:
        lodsb ; al <- caracterul curent
        and al, al ; verifică dacă e zeroul final
        jz cont
        stosb ; dacă nu, îl pune în destinație
    jmp Sir1Loop ; reia operațiile
    cont:
        mov si, bx ; ds:si <- adresa celuilalt șir
    Sir2Loop:
        lodsb
        stosb ; încarcă și zeroul final
        and al, al
```

```

jnz Sir2Loop
ret          ; revenirea din procedură
Concatenare ENDP
END

```

Cele două module vor fi asamblate separat:

```

TASM MAIN
TASM SUB

```

Editarea legăturilor se va face:

```
TLINK MAIN+SUB
```

Va rezulta un program executabil *main.exe* care, lansat în execuție, va tipări mesajul "Buna dimineata!".

8.3. LEGAREA DE MODULE ASAMBLARE CU MODULE SCRISE ÎN LIMBAJE DE NIVEL ÎNALT

8.3.1. Etapele legării unui modul asamblare cu un modul scris în limbaj de nivel înalt

Legătura dintre două module, indiferent de limbajele în care sunt scrise, presupune că dintr-un modul se apelează un subprogram descris în celălalt modul. În continuare vom numi "program apelator" modulul în care se realizează apelul și "procedură" subprogramul apelat. Vom presupune că unul dintre module este descris în limbaj de asamblare, iar celălalt într-un limbaj de programare de nivel înalt. Pentru a realiza o astfel de legătură trebuie avute în vedere următoarele probleme:

1. Cerințe ale editorului de legături.
2. Intrarea în procedură.
3. Nealterarea valorilor unor registri.
4. Transmiterea și accesarea parametrilor.
5. Alocarea de spațiu pentru datele locale (optional).
6. Întoarcerea unui rezultat (optional).
7. Revenirea din procedură.

8.3.1.1. Cerințe ale editorului de legături

În acest context editorul de legături are sarcina de a lega două module obiect care inițial au fost scrise unul în limbaj de asamblare, iar celălalt în limbaj de programare de nivel înalt. Pentru ca legarea lor să poată fi făcută, aceste module trebuie să respecte câteva condiții:

- numele segmentelor sunt impuse de limbajele de nivel înalt cu care se face legătura;

- orice simbol care este definit în modulul scris în limbaj de asamblare și se dorește a fi folosit în modulul scris în limbaj de nivel înalt trebuie făcut vizibil în acesta din urmă, utilizând directiva PUBLIC;
- orice simbol care este definit în modulul scris în limbaj de nivel înalt și care va fi utilizat în modulul scris în limbaj de asamblare trebuie declarat ca extern în acesta din urmă, utilizând directiva EXTRN;

Dacă se utilizează directive de segment simplificate, este necesar ca directiva .MODEL utilizată să fie corespunzătoare modelului de memorie al modulului în limbaj de programare de nivel înalt. Aceasta asigură ca numele de segment din modulul asamblare să corespundă celor utilizate de limbajele de nivel înalt cu care se face legarea și că etichetele de tip PROC, care sunt utilizate pentru a denumi subroutines, proceduri, funcții au tipul (near sau far) utilizat de limbajul de nivel înalt.

8.3.1.2. Intrarea în procedură

În momentul în care se face un apel de procedură sau funcție, apelatorul pune mai întâi în stivă adresa de revenire, pe urmă dă controlul subprogramului apelat folosind pentru aceasta o instrucțiune CALL. Această adresă de revenire poate să fie o adresă FAR sau o adresă NEAR, în funcție de modelul de memorie în care este compilat sau asamblat modulul.

Este foarte important ca instrucțiunea de revenire din subprogram să execute o revenire care să corespundă apelului. Dacă se dorește apelarea dintr-un modul scris în limbaj de nivel înalt (respectiv limbaj de asamblare) a unui subprogram scris în limbaj de asamblare (respectiv limbaj de nivel înalt), editorul de legături care leagă cele două module nu verifică dacă tipul apelului (far sau near) corespunde cu tipul revenirii. Această potrivire cade în sarcina programatorului.

8.3.1.3. Nealterarea valorilor unor registri

Limbajele de nivel înalt impun ca anumii registri să-și păstreze la ieșire valoarea cu care intră într-o procedură. În acest scop, dacă subprogramul, definit în limbaj de asamblare, modifică unii dintre aceștia, atunci valorile lor de intrare trebuie salvate (eventual în stivă). Ele vor fi restaurate înainte de revenirea din procedură.

8.3.1.4. Transmiterea și accesarea parametrilor

De regulă, transmiterea parametrilor se face prin intermediul stivei. Ordinea de transfer a parametrilor de la programul apelator la procedură depinde de limbajul de programare de nivel înalt care este implicat. În ceea ce privește metoda de transfer, pentru cazul limbajelor Pascal și C luate în discuție aici, ea este una din următoarele:

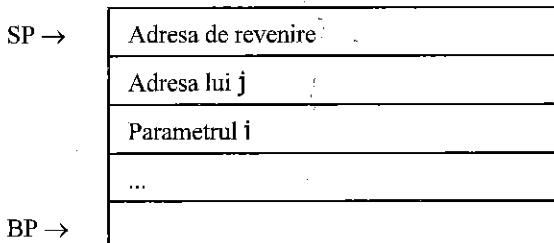
- prin referință NEAR: în stivă se pune, pe un cuvânt, offset-ul adresei;
- prin referință FAR: în stivă se pun două cuvinte; mai întâi se pune segmentul, pe urmă se pune offset-ul;

- prin valoare: în stivă se pune valoarea parametrului.

De exemplu, este apelată o procedură ProcA cu doi parametri: i, un întreg transmis prin valoare și j, un parametru transmis prin referință far. Apelatorul va efectua "secvența":

```
PUSH i
PUSH SEG j
PUSH OFFSET j
CALL ProcA
```

Când procedura primește controlul, vârful stivei conține adresa de revenire. Sub adresa de revenire se află parametrii. Stiva va arăta astfel:

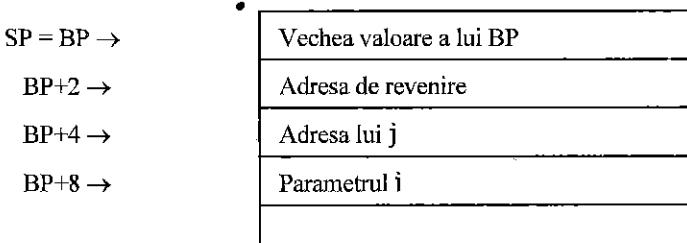


Deoarece SP se modifică odată cu introducerile în stivă, cel mai indicat mod de a accesa valorile lui i și j este prin intermediul unui registru de bază (BX, BP) sau index (SI, DI). Dintre aceste variante registrul BP este cel mai indicat scopului nostru deoarece orice referire la BP se face relativ la segmentul de stivă. Secvența care pregătește accesul la stivă este:

```
PUSH BP
MOV BP, SP
```

Secvența care trebuie specificată de programator sau este executată automat în unele situații, după cum se va vedea în continuare.

În continuare, presupunând că apelul este near, stiva se prezintă astfel:



Deci, bp conține adresa vârfului stivei furnizată de apelator. În continuare, expresii de forma [bp+n] cu n pozitiv puntează la conținutul stivei înainte de apel. Valoarea lui i este word ptr [bp+4], iar valoarea lui j este word ptr [bp+6]. Dacă valoarea lui BP nu se modifică, aceasta este valabil tot timpul, indiferent de alte operații cu stivă.

O metodă mai elegantă de accesare a parametrilor este utilizarea directivei EQU pentru "botezarea" locațiilor de memorie (zonelor din stivă) unde aceștia se găsesc. Pentru exemplul de mai sus, se poate scrie:

```
ProcA PROC NEAR
PUBLIC ProcA
j EQU DWORD PTR [BP+4]
i EQU WORD PTR [BP+8]
PUSH BP
MOV BP, SP
...
MOV AX, i ; încarcă în AX valoarea parametrului i
LES DI, j
MOV ES:[DI], AX ; j:=i
... ; revenirea din procedură
```

Se observă avantajul clar al utilizării directivei EQU. După cum s-a precizat și în capitolul 3, această utilizare are și un dezavantaj: simbolurile i și j nu pot fi redefinite în același fișier sursă .asm.

8.3.1.5. Alocarea de spațiu de memorie pentru datele locale

Uneori este necesar ca procedura să aibă date locale. Dacă valoarea lor nu trebuie să se păstreze între două apeluri consecutive ale procedurii atunci acestea vor fi alocate în segmentul de stivă și le vom numi date volatile. În caz contrar spunem că este vorba despre date statice și spațiul pentru ele va fi alocat într-un segment diferit de segmentul de stivă, de exemplu în segmentul de date.

Alocarea de spațiu pentru datele statice se face folosind directivele cunoscute DB, DW, DD și a. Cel mai practic mod de alocare de spațiu pentru datele locale volatile este cel utilizat de limbajele de programare de nivel înalt și anume alocarea în stivă. Alocarea unui număr de n octeți (n fiind număr par) pentru datele locale se poate face cu următoarea instrucție:

```
sub sp, n
```

Accesarea acestora se poate face relativ la bp sau, mai elegant, utilizând directiva EQU.

De exemplu, dacă într-o rutină avem nevoie de două variabile locale, maxim și minim, de câte un cuvânt fiecare, secvența care pregătește accesarea acestora ar putea fi:

```

push bp
mov bp, sp
sub sp, 4
minim EQU [bp-2]
maxim EQU [bp-4]

...
mov ax, 1
mov minim, ax
mov maxim, ax
...

```

Astfel, simbolurile *minim* și *maxim* pot fi utilizate ca orice variabilă în instrucțiunile asamblorului scrise în modulul respectiv.

8.3.1.6. Întoarcerea unui rezultat

Subprogramele de tip FUNCTION din Pascal precum și toate funcțiile C (care nu sunt declarate void) întorc o valoare. Dacă valoarea întoarsă este simplă (adică nu este de tip tablou sau alt tip structurat) și nu are lungime mai mare decât 4 octeți, atunci valoarea este întoarsă în registri. De exemplu, în cazul compilatoarelor produse de Borland, dacă lungimea rezultatului este de 1 octet, atunci rezultatul este întors în registrul AL; dacă este de 2 octeți, rezultatul este întors în AX; dacă este de 4 octeți, atunci rezultatul este întors în DX:AX, DX conținând partea **high** iar AX partea **low**. Dacă valoarea întoarsă este mai lungă există alte metode de transmitere a rezultatului către apelator. O excepție de la această regulă există, totuși: valorile de tip real, având lungimea de 6 octeți, sunt returnate tot prin intermediul regiștrilor (în DX:BX:AX).

8.3.1.7. Revenirea din procedură

La revenirea din subprogram trebuie să se execute operațiile:

- refacerea valorilor regiștrilor despre care am vorbit în 8.3.1.3.;
- refacerea stivei astfel încât în vârf să conțină adresa de revenire. De exemplu, dacă primele instrucțiuni din subprogram au fost

```

PUSH BP
MOV BP, SP

```

atunci secvența

```

MOV SP, BP
POP BP

```

va avea acest efect;

- dacă limbajul de nivel înalt pretinde apelatorului eliberarea spațiului ocupat în stivă de parametri, aceasta se face cu instrucțiunea *ret n*, unde *n* este numărul de octeți care sunt ocupati de către parametri. Pentru exemplul anterior, vom pune *ret 6*, dacă limbajul de nivel înalt pretinde ca procedura să scoată parametrii din stivă, respectiv *ret*, în caz contrar;

Pentru lămuriri suplimentare vedeti paragrafele următoare care se ocupă de interfața dintre limbajul de asamblare și limbajele de nivel înalt Pascal, respectiv C (implementările Borland ale acestor compilatoare). Veți vedea că fiecare dintre acestea are propriile reguli care trebuie urmate privind ordinea de introducere a parametrilor în stivă și revenirea din procedură.

8.3.1.8. Directivele ARG și LOCAL

Directive ARG

Utilizarea directivei ARG este o alternativă oferită de Turbo Assembler pentru accesul la parametri. Ea calculează automat deplasamentele parametrilor și eventual a rezultatului întors în stivă. De asemenea atribuie nume locațiilor din stivă corespunzătoare acestora, fără folosirea directivei EQU.

O sintaxă simplificată a acestei directive este:

ARG nume₁:tip₁, nume₂:tip₂, ..., nume_n:tip_n = lungime

Aici *nume₁*, *nume₂*, ..., *nume_n* sunt niște identificatori cu ajutorul căroror vor fi referiți parametrii. *tip₁*, ..., *tip_n* au ca valori tipuri recunoscute de Turbo Assembler (BYTE,WORD,DWORD etc.). Primul parametru din listă este cel aflat imediat sub adresa de revenire.

După ce sunt înșiruiti parametrii poate să apară construcția = *lungime*, ceea ce atribuie identificatorului *lungime* numărul de octeți pe care îi ocupă parametrii în stivă. Dacă procedura trebuie să scoată parametrii din stivă, atunci acest identificator este utilizat la terminarea procedurii, de către instrucțiunea *ret*.

Pentru exemplul anterior, utilizarea directivei ARG se poate face astfel:

```

ProcA PROC NEAR
PUBLIC ProcA
ARG j:DWORD, i:WORD = OctRet
PUSH BP
MOV BP, SP
...
MOV AX, i
LES DI, j
MOV ES:[DI], AX
...
POP BP

```

RET OctRet ;aceasta trebuie executată numai dacă procedura trebuie să scoată parametrii din stivă, altfel trebuie numai RET
ProcA ENDP

Ca rezultat al utilizării directivei **ARG** remarcăm aici atribuirea de nume (*j* și respectiv *i*) locațiilor de memorie (de dimensiuni deduse din specificarea tipului parametrilor) aflate imediat sub adresa de revenire, precum și faptul că dimensiunea totală a spațiului ocupat de parametri în stivă este calculată automat (6 octeți în cazul de față) și este atribuită simbolului *OcrRet*. Ca programatori nu mai suntem obligați ca și mai înainte să specificăm explicit deplasamentele variabilelor.

DIRECTIVA **ARG** calculează deplasamentele parametrilor față de cuvântul aflat în stivă deasupra adresei de revenire. Deci, pentru o funcționare corectă, BP (care este folosit în acest caz ca și registru de bază) trebuie să punteze spre acel cuvânt. Secvența

```
PUSH BP
MOV BP, SP
```

asigură acest lucru. Programatorul este cel care trebuie să o includă în textul procedurii.

Directive LOCAL

O alternativă la utilizarea directivei EQU și calcularea deplasamentelor în stivă a datelor locale este folosirea directivei **LOCAL**. Această directivă definește variabile locale pentru macrouri și proceduri. În proceduri, sintaxa ei este:

```
LOCAL nume:tip [ :contor ] {,nume:tip [ :contor ] } [ =lungime ]
```

Folosirea acestei directive în cadrul unei proceduri are ca efect declararea de variabile locale ca și nume care accesează locații în stivă (considerate deplasamente negative față de BP). Prima variabilă locală începe la BP-2. Dacă apare =*lungime*, simbolul respectiv va fi echivalat cu numărul de octeți ocupati în stivă de blocul de simboluri locale.

Urmărind sintaxa, *nume* este numele care va fi folosit pentru referirea la respectivul simbol local în cadrul procedurii; *tip* reprezintă tipul datei și poate fi unul din următoarele: WORD, DATAPTR, CODEPTR, DWORD, QWORD, TBYTE, etc. Dacă tipul nu este specificat, se consideră că este vorba despre WORD. *count* reprezintă numărul de elemente (de tipul specificat) care trebuie alocate în stivă. Directiva LOCAL poate să apară oriunde în cadrul procedurii, dar trebuie să precedă prima utilizare a simbolurilor pe care le definește.

Am precizat că directiva LOCAL doar declară simbolurile care vor fi utilizate pentru variabilele locale. Alocarea efectivă de spațiu trebuie să o facă utilizatorul prin actualizarea valorii din SP (de exemplu *SUB SP, lungime*).

Ca și la directiva ARG, funcționarea corectă a directivei LOCAL este condiționată de executarea instrucțiunilor

```
push BP
mov BP, SP
```

înainte de orice operație de punere sau scoatere din stivă și înainte de orice referire la simbolurile locale.

De exemplu,

```
P1 PROC NEAR
LOCAL minim:word, maxim:word = Locale
push BP           ; pregătirea lui BP pentru accesarea
mov BP, SP        ; corectă a variabilelor locale
sub SP, Locale   ; alocarea de spațiu pentru variabilele locale
...
mov minim, 0      ; minim = [BP-2]
mov maxim, 0FFFFh ; maxim = [BP-4]
...
add SP, Locale   ; eliberarea zonei din stivă ocupată de datele locale
...
; revenirea din procedură
```

P1 endp

8.3.2. Interfața dintre Turbo Assembler și Turbo Pascal

8.3.2.1. Directiva de compilare \$L și subprogramele external

Directiva de compilare \$L are sintaxa:

```
{$L nume[.obj]}
```

Parantezele drepte indică faptul că extensia este opțională. Dacă lipsește, se consideră că este *.obj*. Prezența acestei directive de compilare indică mediului Turbo Pascal să caute în directorul curent și în directoarele specificate pentru fișierele *.obj*, fișierul *nume.obj* și să lege acest modul la programul scris în limbajul Pascal. Fișierul *nume.obj* trebuie să fie în format obiect, linkeditabil.

Pentru a putea face această legătură trebuie ca fișierul *.obj* să îndeplinească condițiile:

- toate procedurile și funcțiile trebuie să se afle într-un segment numit **CODE** sau **CSEG**, sau într-un segment al cărui nume se termină cu **_TEXT**;
- toate datele inițializate trebuie să fie plasate într-un segment numit **CONST** sau într-un segment al cărui nume se termină cu **_DATA**;
- toate datele neinițializate trebuie să fie plasate într-un segment numit **DATA** sau **DSEG**, sau într-un segment al cărui nume se termină cu **_BSS**;

Fiecare funcție sau procedură care este utilizată în programul Pascal dar este definită într-un alt modul (eventual scris în limbaj de asamblare) trebuie să fie declarată în programul Pascal cu ajutorul directivei **external**. De exemplu:

```
Procedure AsmProc (a:Integer; b:Real); external;
Function AsmFunc (c:Word; d:Byte): Integer; external;
```

O declarație **external** trebuie să fie făcută numai la nivelul cel mai exterior al programului sau unității. Directiva **{\$I}** poate să apară oriunde în textul sursă Pascal.

Sigurele obiecte care pot fi exportate dintr-un modul scris în limbaj de asamblare într-un program sau unitate Pascal sunt etichetele de instrucțuni sau nume de proceduri declarate **PUBLIC**. Fiecare etichetă care este făcută **PUBLIC** trebuie să aibă o declarație (**external**) corespunzătoare de procedură sau funcție în programul Pascal.

De exemplu, declarațiilor de mai sus le corespunde, în fișierul **.asm**, următoarele secvențe:

```
CODE SEGMENT
AsmProc PROC NEAR
    PUBLIC AsmProc
    ...
AsmProc ENDP
AsmFunc PROC NEAR
    PUBLIC AsmFunc
    ...
AsmFunc ENDP
    •
CODE ENDS
END
```

Un modul TASM poate accesa orice procedură, funcție, variabilă sau constantă cu tip declarată la nivelul cel mai exterior al unui program sau al unei unități Pascal, inclusiv rutinele din biblioteci. Acest lucru se face cu ajutorul declarației **EXTRN**. Etichetele Pascal și constantele fără tip nu pot fi făcute vizibile în modulul scris în limbaj de asamblare.

Să presupunem de exemplu că în programul Pascal există următoarele declarații de variabile globale și declarații de subprograme:

```
...
{variabile globale}
Var a: Byte;
    b: Word;
    c: ShortInt;
...
Procedure ProcA;
...
{$F+}
Function FuncA:Integer;
...
```

Presupunând că nu este forțat apelul FAR acestea pot fi accesate din modulul **.asm** după ce sunt făcute următoarele declarații:

```
DATA SEGMENT
    ASSUME DS:DATA
    EXTRN A: BYTE
    EXTRN B: WORD
    EXTRN C: BYTE
    ...
DATA ENDS
CODE SEGMENT
    EXTRN ProcA:NEAR
    EXTRN FuncA:FAR
    ...
; aici pot fi utilizate variabilele a, b, c și pot fi apelate subprogramele ProcA, FuncA
    ...
CODE ENDS
END
```

Sintaxa de *identificator calificat*, care utilizează numele unei unități urmat de caracterul **.** și de numele unui obiect care aparține acelei unități nu este compatibil cu regulile sintactice TASM, deci nu este permisă în modulul TASM.

Referirea la proceduri și funcții nu poate utiliza aritmetică de adrese. De exemplu:

```
EXTRN PublicProc:FAR
...
CALL PublicProc+42      ; este eronat !
```

Operatorii care selectează BYTE din WORD (ex: HIGH, LOW) nu pot fi aplicăți la obiecte EXTRN.

Declarațiilor de tip standard din Pascal le corespund, în limbaj de asamblare, următoarele tipuri:

Integer	WORD
Real	FWORD
Single	DWORD
Pointer	DWORD

8.3.2.2. Reguli de utilizare a registrilor

Când este făcut un apel la o funcție sau procedură trebuie păstrată nealterată valoarea a patru registri: SS, DS, BP, SP. În momentul apelului, SS punctează spre segmentul de stivă, DS punctează spre segmentul de date global (numit DATA), BP și SP punctează spre baza, respectiv vârful stivei.

În cazul în care se face legătură între un modul scris în limbaj de asamblare și un program scris în Pascal (în particular în Turbo Pascal), sarcina scoaterii parametrilor din stivă revine unității apelate (procedurii)!

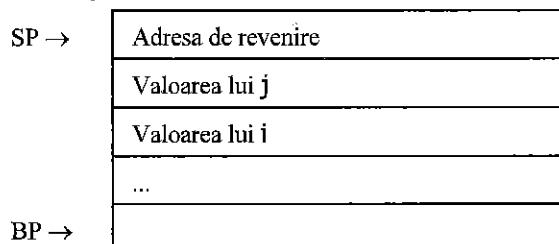
8.3.2.3. Transmiterea și accesarea parametrilor

Regulile după care Turbo Pascal transmite parametrii subprogramului sunt prezentate în capitolul 7. Accesarea parametrilor în subprogramul scris în limbaj de asamblare se face în oricare din modurile prezentate în 8.3.1.4. și 8.3.1.8.

De exemplu, presupunem că programul Pascal conține o declarație de forma:

Procedure ProcA(i, j:Integer); external;

Presupunem că apelul este de tip NEAR. În această ipoteză, stiva va arăta astfel la intrarea în procedură:



Utilizarea directivei ARG (de exemplu) se poate face astfel:

```
CODE SEGMENT
ASSUME CS:CODE
```

```
ProcA PROC NEAR
PUBLIC ProcA
```

```
ARG j:WORD, i:WORD = OctRet
```

```
PUSH BP
MOV BP, SP
```

```
...
```

```
MOV AX, i
...
POP BP
RET OctRet
```

```
ProcA ENDP
CODE ENDS
```

În lista parametrilor, aceștia trebuie să fie dați în ordine inversă decât sunt dați în declarația Pascal.

8.3.2.4. Întoarcerea rezultatului de către functii

Modul în care funcțiile Turbo Pascal întorc rezultatul este prezentat în capitolul 7.

În cazul în care o funcție întoarce rezultat de tip string, directiva ARG oferă opțiunea RETURNS, care permite definirea unei variabile care va avea ca valoare adresa zonei de memorie rezervată pentru întoarcerea rezultatului. De exemplu, dacă avem o declarație de forma:

```
Function FuncA(i:Integer; j:Byte):String;external;
```

directiva ARG din modulul *asm* care definește respectiva funcție ar putea arăta astfel:

```
ARG j:BYTE, i:WORD = OctRet RETURNS resultat:DWORD
```

În continuare, *rezultat* este adresa zonei de memorie din care apelantul va lua rezultatul funcției.

8.3.2.5. Alocarea de spațiu pentru datele locale

Amintim faptul că datele locale unei proceduri pot fi de două tipuri: statice sau volatile.

Datele statice se alocă în segmentul de date global, utilizând directivele DB, DW etc. Aceste date, însă, nu pot fi făcute vizibile în programul Pascal. De asemenea, ele nu pot fi inițializate (chiar dacă se încearcă o inițializare, aceasta este ignorată). Rezolvarea acestor probleme se poate face declarând aceste date ca variabile globale sau constante cu tip în programul Pascal și utilizarea directivei EXTRN pentru a le face vizibile în modulul *asm*.

Datele volatile se alocă în stivă prin oricare din metodele prezentate în 8.3.1.5. și 8.3.1.8.

8.3.2.6. Utilizarea directivelor de segment simplificate

Modelul de memorie al lui Turbo Pascal înglobează aspecte a două modele de memorie acceptate de Turbo Assembler: **medium** și **large**.

Utilizarea directivei .MODEL cu parametrul *TPASCAL* ca și model de memorie simplifică mult realizarea interfeței dintre Turbo Assembler și Turbo Pascal. Au astfel loc câteva acțiuni automate:

- inițierea segmentării simplificate
- definirea modelului de memorie corespunzător
- inițializarea convențiilor de apel Turbo Pascal
- definirea numelor segmentelor
- la începutul procedurilor se execută automat secvența

```
PUSH BP
MOV BP, SP
```

- la revenirea din procedură, când se execută instrucțiunea *ret*, se execută automat POP BP și se scot din stivă parametrii (deci, programatorul nu trebuie să specifice vreun parametru pentru instrucțiunea RET)

Să presupunem că într-un fișier sursă Turbo Pascal există următoarea declarație:

```
...
{$F+}
Function ConstrSir(c:Char; n:Integer; var s:String):String; external;
{$F-}
...
```

Funcția *ConstrSir* este definită în limbaj de asamblare, într-un alt fișier, după cum urmează:

```
.MODEL TPASCAL
.CODE
ConstrSir PROC FAR c:BYTE, n:WORD, s:DWORD RETURNS rez:DWORD
PUBLIC ConstrSir
mov cx, n          ; dimensiunea stringului întors ca rezultat
...
les di, rez
```

```
mov es:[di], cx      ; completarea poziției de indice 0 din stringul rezultat
ret
ConstrSir ENDP
END
```

Observații:

- pregătirea accesului la parametri se face scriind în linia de definiție a procedurii numele care corespund acestora, urmate fiecare de un tip
- parametrii se trec în lista de după cuvântul PROC în ordinea în care apar și în declarația din fișierul Pascal
- dacă subprogramul este o funcție care întoarce ca valoare un **string**, opțiunea RETURNS permite asocierea unui nume (rez) cu locul din memorie unde se va depune valoarea acestui sir de caractere
- instrucțiunea de revenire este RET simplu, fără nici un parametru
- la începutul procedurii NU trebuie să se specifice instrucțiunile PUSH BP și MOV BP, SP acestea fiind executate automat (le inserează asamblorul)
- analog, înainte de revenire se va executa POP BP și se va elibera spațiul din stivă ocupat de către parametri

8.3.2.7. Exemplul 1

Vom prezenta ca și exemplu o unitate în care sunt definite două rutine de gestionare a sirurilor de caractere. Acestea sunt descrise în limbaj de asamblare, într-un fișier separat. Funcția *Majuscule* convertește toate caracterele sirului dat ca parametru în majuscule, iar funcția *LantDe* întoarce ca valoare un sir de caractere de o lungime dată, format prin repetarea unui anumit caracter.

```
unit Lant;
interface
function Majuscule(S: String): String;
function LantDe(Car: Char; Contor: Byte): String;

implementation
{$L LANTASM}

function Majuscule; external;
function LantDe; external;

end.
```

Fișierul de asamblare care urmează definește cele două rutine. Acesta trebuie asamblat pentru a crea un fișier *lantasm.obj*, care este necesar la compilarea unității *lant*. Se poate observa că cele două rutine utilizează modelul de apel FAR, deoarece ele sunt declarate în secțiunea *interface* a unității.

```
CODE SEGMENT
ASSUME CS:CODE
PUBLIC Majuscule, LantDe
```

Majuscule PROC FAR

```
MajResult EQU DWORD PTR [BP+10]
ParamS EQU DWORD PTR [BP+ 6]
PUSH BP           ; salvare BP
MOV BP, SP        ; stabilirea stivei
PUSH DS           ; salvare DS
LDS SI, ParamS   ; încărcare adresă sir
LES DI, MajResult ; încărcare adresă rezultat
CLD
LODSB             ; încărcare lungime sir
STOSB             ; copiere în rezultat
MOV CL, AL        ; lungime sir în CX
XOR CH, CH
JCXZ ET3          ; salt dacă sirul e vid
ET1: LODSB         ; încărcare caracter
      CMP AL, 'a'    ; salt dacă nu e literă mică
      JB ET2
      CMP AL, 'z'
      JA ET2
      SUB AL, 'a'-'A' ; conversie în majusculă
ET2: STOSB         ; încărcare în rezultat
      LOOP ET1        ; ciclare pentru toate caracterele
ET3: POP DS          ; restaurare DS
      POP BP          ; restaurare BP
      RET 4            ; scoaterea parametrilor din stivă
Majuscule ENDP
```

LantDe PROC FAR

```
LantDeResult EQU DWORD PTR [BP+10]
ParamCar EQU BYTE PTR [BP+ 8]
ParamContor EQU BYTE PTR [BP+ 6]
PUSH BP           ; salvare BP
MOV BP, SP        ; stabilirea stivei
LES DI, LantDeResult ; încărcarea adresei rezultatului
MOV AL, ParamContor ; încărcare contor
```

```
CLD
STOSB             ; încărcare lungime sir
MOV CL, AL        ; lungime sir în CX
XOR CH, CH
MOV AL, ParamCar   ; încărcare caracter în AL
REP STOSB         ; încărcare caracter în rezultat
POP BP             ; restaurare BP
RET 4              ; scoatere parametri din stivă
```

LantDe ENDP

```
CODE ENDS
END
```

Asamblarea programului și compilarea unității se pot face utilizând comenziile următoare:

```
TASM LANTASM
TPC LANT
```

8.3.2.8. Exemplul 2

Se cere un program Turbo Pascal care apelează funcția *Asmf* scrisă în limbaj de asamblare. Această funcție primește ca și parametru un sir de caractere citit în programul Turbo Pascal, citește un sir de caractere apelând pentru aceasta funcția Pascal *CitSir* și mai acționează pe acest sir de caractere ce este variabilă globală Pascal (numită *glob*). Funcția *Asmf* construiește și întoarce ca rezultat sirul obținut prin concatenarea primelor 10 caractere ale celor 3 siruri. Acest sir va fi afișat pe ecran.

Modulul M1 (Turbo Pascal)

<u>Declarări variabile</u>
var glob:string; s:string;
<u>Definiții și declarații subroutines</u>
function Asmf (s:string):string; far; external;
function CitSir:string;far;
<u>Apeluri subroutines</u>
s:=CitSir; s := Asmf(s);

Modulul M2 (asamblare)

<u>Declarări variabile</u>
<u>Definiții și declarații subroutines</u>
Asmf proc (s:string):string; far; public; extrn CitSir: far
<u>Apeluri subroutines</u>
CitSir;

Fig. 8.1. Declarații variabile, definiții și apeluri subroutines pentru exemplul 2.

P.pas

```

program TPandASM;

var glob: string; s: string;
    {declararea variabilelor globale glob și s}

{$L asmf.obj}
    {directive de compilare $L indică mediul Turbo Pascal să caute fișierul asmf.obj
     în directorul curent și în directoarele specificate pentru fișierele .obj și să lege acest
     modul la programul scris în limbajul Pascal. Fișierul asmf.obj trebuie să fie în
     format obiect, linkeditabil}

function Asmf (s: string): string; far; external;
    {declararea funcției Asmf ca fiind externă; această funcție este definită în modulul
     scris în limbaj de asamblare dar va fi folosită în acest modul; fiecare funcție sau
     procedură care este utilizată în programul Pascal dar este definită într-un alt modul
     (în cazul nostru în modulul scris în limbaj de asamblare) trebuie să fie declarată în
     programul Turbo Pascal cu ajutorul directivei external; funcția primește ca și
     parametru un string și returnează un string; prezența directivei far indică modul
     de apel al acestui subprogram, și anume prin specificarea atât a adresei de segment
     cât și a deplasamentului în cadrul acestui segment}

function CitSir: string; far;
    {definirea funcției CitSir care nu primește nici un parametru, citește un string de
     la tastatură și returnează acest string ca și rezultat; tipul acestei funcții este far și
     va fi apelată atât din modulul curent cât și din modulul scris în limbaj de
     asamblare; nu este nevoie de prezența unei directive Turbo Pascal pentru a face
     această funcție vizibilă în modulul scris în limbaj de asamblare, deoarece un
     modul scris în limbaj de asamblare poate accesa orice procedură, funcție, variabilă
     sau constantă cu tip declarată la nivelul cel mai exterior al unui program sau al unei
     unități Pascal, inclusiv rutinele din biblioteci, prin folosirea declarației extrn în
     modulul asamblare}

var Strn: string;           {declararea variabilei locale Strn de tip string}
begin
    write ('Sirul: ');
    readln (Strn);          {citirea de la tastatură a stringului Strn}
    CitSir := Strn;          {rezultatul întors de această funcție este stringul citit Strn}
end;

begin
    s := CitSir;             {se apelează funcția CitSir pentru citirea unui string de la tastatură;
                             stringul citit și returnat de către funcția CitSir se atribuie variabilei
                             s}

```

```

glob := 'abc123';      {atribuirea unei valori variabilei globale glob (textul problemei nu
                        cere citirea acesteia de la tastatură)}
s := Asmf(s);          {apelul funcției Asmf (această funcție va concatena primele 10
                        caractere din fiecare dintre următoarele trei stringuri: (1) stringul
                        transmis ca parametru, (2) valoarea variabilei globale glob și (3)
                        stringul care se va citi în interiorul funcției Asmf; funcția Asmf va
                        întoarce ca rezultat sirul obținut în urma concatenării); la acest apel
                        se creează cadrul de stivă pentru subrutina Asmf}
writeln(s);            {afișarea pe ecran a rezultatului întors de către funcția de
                        concatenare Asmf}
readln;
end.

```

Asmf.asm

```

assume cs:_TEXT, ds:_DATA
;pentru a se putea face legătura între acest modul și modulul scris în Turbo Pascal, modulul scris
;în limbaj de asamblare trebuie să îndeplinească următoarele condiții:
; - toate procedurile și funcțiile trebuie să se afle într-un segment numit CODE sau CSEG, sau
;   într-un segment al cărui nume se termină cu _TEXT;
; - toate datele declarate trebuie să fie plasate într-un segment numit CONST sau într-un
;   segment al cărui nume se termină cu _DATA;
_DATA segment
    extrn glob:byte      ;declarația variabilei glob, definită în modulul Turbo Pascal
_DATA ends

_TEXT segment
    extrn CitSir: far    ;declarația funcției CitSir, care a fost definită în modulul Turbo
                           ;Pascal, dar va fi folosită în modulul curent; tipul subrutei este
                           ;far
Asmf proc far          ;definirea funcției Asmf, de tip far; această funcție primește ca și
public Asmf             ;parametru un string și construiește și returnează un alt string
                         ;funcția Asmf este definită în acest modul dar va fi folosită în
                         ;modulul Turbo Pascal, motiv pentru care o declarăm public
                         ;având în vedere faptul că această funcție va fi apelată în cadrul modulului Turbo Pascal,
                         ;codul de apel al funcției Asmf va fi generat în mod automat de către apelator. Ca
                         ;urmăre, punerea în stivă a adresei sirului rezultat, a parametrului și a adresei de revenire
                         ;sunt acțiuni efectuate automat la apel de către Turbo Pascal și evidențiate ca efect în
                         ;fig.8.2. În ceea ce privește rezervarea spațiului pentru sirul rezultat, aceasta este o acțiune
                         ;ce are loc în cadrul codului de intrare în rutina apelantă (programul principal în acest
                         ;caz).

```

în continuare este descris **codul de intrare** corespunzător subrutinei Asmf; dacă această funcție ar fi fost definită în modulul Turbo Pascal, codul de intrare ar fi fost generat automat; fiind scrisă însă în limbaj de asamblare, codul de intrare trebuie scris de către programator

în cazul acestei subrutine, codul de intrare trebuie să asigure izolarea stivei, realizarea copiei locale a parametrului de tip string transmis prin valoare și alocarea de spațiu în stivă pentru rezultatul de tip string întors de funcția CitSir, care va fi apelată în acest subprogram (vezi capitolul 7, paragraful 7.1.2.)

<code>push bp</code>	;aceste două instrucțiuni au ca și efect "izolarea stivei", astfel încât programul să nu coboare în zona care a fost ocupată până acum; bp definește baza zonei de stivă (<i>stack frame</i>) utilizată pentru desfășurarea execuției subrutinei curente
<code>mov bp, sp</code>	
<code>sub sp, 100h</code>	;se alocă în stivă 100h octeți pentru copia parametrului de tip string transmis prin valoare
<code>sub sp, 100h</code>	;se alocă în stivă 100h octeți pentru rezultatul de tip string întors de către funcția CitSir, care va fi apelată în acest subprogram

;desigur că puteam scrie o singură instrucție care să aibă același efect ca și cele două scrisă mai sus; este vorba despre `sub sp, 200h`; singurul motiv pentru care s-au scris cele două instrucțiuni a fost evidențierea celor două etape de alocare de memorie în stivă; componența stivei în acest moment (ca rezultat al codului de apel generat de Turbo Pascal la apelul funcției Asmf și al codului de intrare scris explicit în modulul curent la intrarea în această funcție) este descrisă în figura 8.2.

<code>rez equ dword ptr [bp+10]</code>	;în continuare sunt definite trei constante simbolice: rez , sloc și copieSir
	;asociem cu simbolul rez dublucuvântul din stivă de la deplasamentul <code>[bp+10]</code> , care reprezintă adresa de început a șirului în care se va întoarce rezultatul
<code>copieSir equ byte ptr [bp-100h]</code>	;asociem cu simbolul copieSir primul octet (octetul de la deplasamentul <code>[bp-100h]</code>) din spațiul rezervat în stivă pentru copia parametrului de tip string transmis prin valoare funcției Asmf
<code>sloc equ byte ptr [bp-200h]</code>	;asociem cu simbolul sloc primul octet (octetul de la deplasamentul <code>[bp-200h]</code>) din spațiul rezervat în stivă pentru rezultatul de tip string întors de către funcția CitSir, care va fi apelată în această funcție

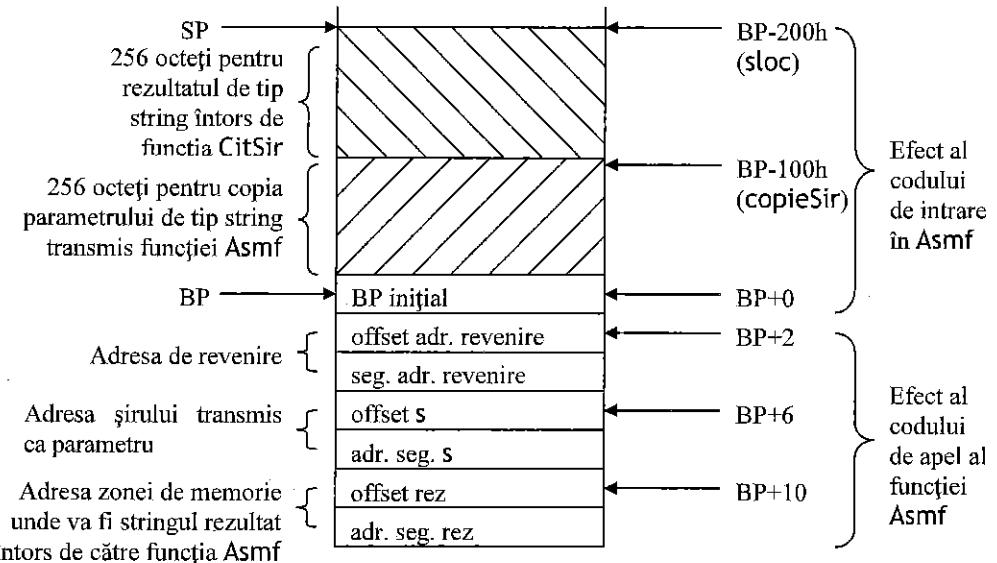


Fig. 8.2. Organizarea stivei după apelul și intrarea în funcția Asmf.

;ca efect al codului de intrare în Asmf pe care trebuie să îl construim efectuăm o copie locală pe stivă a parametrului de tip string transmis prin valoare; șirul sursă se găsește la adresa (far) depusă în stivă la ss:[bp+6], iar șirul destinație începe de la adresa ss:[bp+100h]

<code>push ds</code>	;dorim să utilizăm în continuare registrul ds , motiv pentru care vom salva în stivă valoarea acestuia pentru a o putea restaura înainte de ieșirea din funcție
<code>lds si, [bp+6]</code>	;în ds:si copiem adresa far a șirului sursă (parametru al funcției)
<code>mov bx, ss</code>	;copiem în es valoarea din registrul ss care reprezintă adresa de segment a spațiului alocat în stivă pentru copia parametrului de tip string; folosim bx ca intermediar
<code>mov es, bx</code>	;încarcăm în di deplasamentul șirului destinație, adică offset-ul primului octet din acest șir
<code>lea di, copieSir</code>	;DF = 0 – vom parurge ascendent șirurile sursă și destinație
<code>cld</code>	;stîm că șirul este declarat în modulul Turbo Pascal ca având lungime 255 octeți => încarcăm în CX lungimea maximă a șirului
<code>mov cx, 00FFh</code>	;de CX ori se copiază căte un octet din șirul sursă în șirul destinație; această instrucție face copierea locală pe stivă a parametrului de tip string transmis prin valoare
<code>rep movsb</code>	

;pregătim apelul funcției CitSir punând în stivă adresa spațiului unde funcția CitSir va returna stringul citit; aceasta face parte din codul de apel al funcției CitSir, care ar fi fost generat automat dacă apelul funcției s-ar fi făcut într-un limbaj de nivel înalt, dar trebuie scris explicit de către programator dacă funcția este apelată în limbaj de asamblare

push ss	;spațiul pentru sirul rezultat este alocat în stivă, aşadar adresa de segment a sirului rezultat este ss; salvăm în stivă această adresă
lea ax, sloc	;deplasamentul sirului rezultat este offset-ul primului octet din acest sir, adică sloc
push ax	;salvăm în stivă acest deplasament
call CitSir	;apelul funcției CitSir, care citește un sir și îl depune la adresa ss:sloc

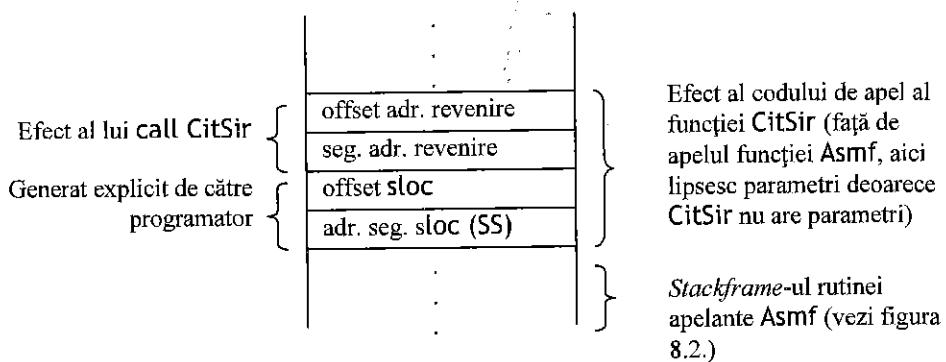


Fig. 8.3. Organizarea stivei după apelul rutinei CitSir.

pop ax ;eliberarea zonei din stivă în care s-a pus adresa sirului rezultat este
pop ax ;responsabilitatea apelatorului; aceste două instrucțiuni pop scoț din stivă adresa sirului rezultat (adică ss:sloc)

;o modalitate alternativă de a scoate din stivă adresa sirului rezultat este prin adunarea valorii 4 la valoarea lui sp, deoarece am avut 2 octeți pentru adresa de segment și 2 octeți pentru offsetul sirului rezultat:
; add sp, 4

;incepem construirea sirului rezultat ce trebuie returnat de către funcția Asmf, pentru care vom utiliza instrucțiuni pe siruri

les di, rez ;încarcăm în es:di adresa de început a sirului pe care subrutina curentă îl va întoarce ca rezultat
cld ;stabilim direcția de parcurgere de la stânga spre dreapta (ascendent) a sirurilor sursă și destinație

;mai întâi vom copia în sirul rezultat primele 10 caractere din sirul transmis ca parametru

mov bx, ss ;copiem în ds valoarea din registrul ss, folosind bx ca intermediu
mov ds, bx
lea si, copieSir ;încarcăm în si deplasamentul sirului sursă, adică offset-ul primului octet din acest sir

;în cx vom pune numărul de elemente al sirului transmis ca parametru; acest număr se află în primul octet al sirului (byte ptr [si], deoarece și reprezintă adresa de început a acestui sir)
mov ch, 0
mov cl, byte ptr [si] ;cl:=s[0] (lungimea sirului transmis ca parametru)

inc si ;si va referi primul element din sirul transmis ca parametru, adică al doilea octet (pe primul octet fiind dimensiunea sirului)
push di ;salvăm offset-ul stringului destinație (pentru ca după construirea sirului să completăm primul octet din sir cu dimensiunea acestuia)
inc di ;di va referi primul element din sirul rezultat, adică al doilea octet (pe primul octet fiind dimensiunea sirului)
cmp cx, 10 ;dacă sirul transmis ca parametru are mai puțin de 10 elemente,
jb et1 ;atunci se va face salt la eticheta et1 unde se vor copia în sirul destinație toate elementele acestui sir, adică cx elemente
mov cx, 10 ;dacă sirul transmis ca parametru are mai mult de 10 caractere,
;vom copia în sirul destinație doar primele 10; în acest scop îl setăm pe cx la valoarea 10

et1:
rep movsb ;instrucțiunea movsb se va executa de cx ori; această instrucțiune copiază la adresa es:di (unde se află sirul rezultat) un octet de la adresa ds:si (unde se află copia sirului transmis ca parametru)

;în continuare vom copia în sirul rezultat primele 10 caractere din sirul returnat de către funcția CitSir (sirul sloc)
;în acest scop, încarcăm în ds:si adresa de început a sirului sloc

push ss ;adresa de segment a sirului sloc este ss, deoarece acesta este
pop ds ;alocat în stivă; încarcăm această adresă în ds
lea si, sloc ;si va conține deplasamentul sirului sloc

mov cl, byte ptr [si] ;în cx reținem numărul de elemente al sirului sloc (în ch
;avem deja valoarea 0, iar în cl reținem valoarea primului octet din sirul sloc, adică byte ptr [si])

```

inc si           ;îi va referi primul element din şirul sloc, adică al doilea octet (pe
;primul octet fiind dimensiunea şirului)

cmp cx, 10      ;dacă şirul sloc are mai puțin de 10 elemente, atunci se va face salt
jb et2          ;la eticheta et2 unde se vor copia în şirul destinație toate
;elementele acestui şir, adică CX elemente

mov cx, 10      ;dacă şirul sloc are mai mult de 10 caractere, vom copia în şirul
;destinație doar primele 10; în acest scop îl setăm pe CX la valoarea
;10

et2:
rep movsb       ;instrucțiunea movsb se va executa de CX ori; această instrucțiune
;copiază la adresa es:di (unde se află poziția curentă în şirul
;rezultat) un octet de la adresa ds:si (unde se află şirul sloc)

;în continuare vom copia în şirul rezultat primele 10 caractere din variabila globală glob
;în acest scop, încarcăm în ds:si adresa de început a şirului glob

lea si, glob     ;îi va conține deplasamentul şirului sloc
mov ax, _DATA    ;ds va conține adresa de segment a şirului sloc (variabilele globale
mov ds, ax        ;trebuie referite relativ la ds – regulă Turbo Pascal)

mov cl, byte ptr [si]   ;în CX reținem numărul de elemente al şirului glob (în ch
;avem deja valoarea 0, iar în CL reținem valoarea primului
;octet din şirul glob, adică byte ptr [si])

inc si           ;îi va referi primul element din şirul glob, adică al doilea octet (pe
;primul octet fiind dimensiunea şirului)

cmp cx, 10      ;dacă şirul glob are mai puțin de 10 elemente, atunci se va face salt
jb et3          ;la eticheta et3 unde se vor copia în şirul destinație toate
;elementele acestui şir, adică CX elemente

mov cx, 10      ;dacă şirul glob are mai mult de 10 caractere, vom copia în şirul
;destinație doar primele 10; în acest scop îl setăm pe CX la valoarea
;10

et3:
rep movsb       ;instrucțiunea movsb se va executa de CX ori; această instrucțiune
;copiază la adresa es:di (unde se află şirul rezultat) un octet de la
;adresa ds:si (unde se află şirul glob)

pop ax           ;valoarea inițială a lui di (offset-ul primului octet din şirul rezultat)
;se află în vârful stivei; scoatem din stivă această valoare și o
;reținem în ax

mov bx, di      ;calculăm în bx lungimea şirului rezultat (bx=actual di - fost di)
sub bx, ax        ;scăzând valoarea inițială a lui di din valoarea actuală a lui di

```

```

dec bx           ;din valoarea lui bx se mai scade un octet (deoarece în şirul
;destinație elementele încep doar de pe al doilea octet, primul octet
;fiind rezervat pentru a reține dimensiunea şirului)
;încarcăm în es:di adresa de început a şirului rezultat
;în primul octet al şirului rezultat salvăm numărul de elemente ale
;acestui şir (rez[0]=length(rez))

pop ds           ;restaurăm valoarea pe care ds o avea la intrarea în subrutină
mov sp, bp       ;refacem valorile lui sp și bp (partea a codului de ieșire din
;subrutina Asmf – cu această ocazie se elibereză cei 200h octeți
;alocați pentru stringul întors de CitSir și pentru copia
parametrului ;transmis către Asmf)

ret 4            ;ieșire din funcție cu scoaterea din stivă a 4 octeți (este vorba
;despre scoaterea din stivă a parametrului – 2 octeți pentru adresa
;de segment și 2 octeți pentru deplasament – locațiile identificate
prin [BP+6] în figura 8.2.); eliberarea spațiului din stivă ocupat de
;parametri face parte tot din codul de ieșire din subrutină

;dacă această funcție ar fi fost definită într-un limbaj de nivel înalt, codul de ieșire – în
;acest caz responsabil de refacerea valorilor lui sp și bp și de scoaterea parametrilor din
;stivă – ar fi fost generat automat; fiind scrisă însă în limbaj de asamblare, codul de ieșire
;trebuie scris explicit de către programator

asmf endp
_TEXT ends
end

```

8.3.3. Interfața dintre Turbo Assembler și Borland C++

8.3.3.1. Cerințe ale editorului de legături privind definirea modulului asamblare

De obicei, modulele în limbaj de asamblare care urmează să fie legate cu Borland C++ constau din trei secțiuni: cod, date inițializate și date neinițializate. Utilizarea directivei .MODEL va permite definirea foarte simplă a acestor segmente, utilizând directivele de segment simplificate .CODE, .DATA și .DATA?. Borland C++ suportă următoarele șase modele de memorie: TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE (vezi 7.2.1 și 7.3.1). De exemplu, dacă alegem modelul de memorie small, modulul asamblare poate fi organizat astfel:

```

.CODE
... segmentul de cod ...
.DATA
... segmentul de date ...
.DATA?
... segmentul de date neinițializate ...

```

În unele cazuri se dorește utilizarea directivelor de segment standard. Formatul fișierului scris în limbaj de asamblare este, în acest caz, următorul:

```

code SEGMENT BYTE PUBLIC 'CODE'
    ASSUME CS:code, DS:dseg
    ... segmentul de cod ...
code ENDS

dseg GROUP _DATA, _BSS
data SEGMENT WORD PUBLIC 'DATA'
    ... segmentul de date inițializate ...
data ENDS

_BSS SEGMENT WORD PUBLIC 'BSS'
    ... segmentul de date neinițializate ...
_BSS ENDS
END

```

Identifierii *code*, *data* și *dseg* vor fi înlocuiți (de către programator!) cu valori specifice, dependente de modelul de memorie. Aceste valori sunt prezentate în tabelul următor (*filename* reprezintă numele modulului):

MODEL	VALORI IDENTIFICATORI	POINTERI LA COD ȘI DATE
Tiny, Small	<i>code</i> = <u>TEXT</u> <i>data</i> = <u>DATA</u> <i>dseg</i> = <u>DGROUP</u>	cod : DW <u>TEXT</u> :xxxx date: DW <u>DGROUP</u> :xxxx
Compact	<i>code</i> = <u>TEXT</u> <i>data</i> = <u>DATA</u> <i>dseg</i> = <u>DGROUP</u>	cod : DW <u>TEXT</u> :xxxx date: DD <u>DGROUP</u> :xxxx
Medium	<i>code</i> = <i>filename</i> <u>TEXT</u> <i>data</i> = <u>DATA</u> <i>dseg</i> = <u>DGROUP</u>	cod : DD xxxx date: DW <u>DGROUP</u> :xxxx
Large	<i>code</i> = <i>filename</i> <u>TEXT</u> <i>data</i> = <u>DATA</u> <i>dseg</i> = <u>DGROUP</u>	cod : DD xxxx date: DD <u>DGROUP</u> :xxxx
Huge	<i>code</i> = <i>filename</i> <u>TEXT</u> <i>data</i> = <i>filename</i> <u>DATA</u>	cod : DD xxxx date: DD xxxx

Cea de-a treia coloană a tabelului arată cum se definesc datele constante care sunt pointeri la cod, respectiv la date (xxxx este adresa referită). Datele constante numerice și siruri de caractere se

definesc în maniera ușuală. Variabilele se definesc la fel ca și datele constante. Variabilele neinițializate se declară în segmentul _BSS, punând simbolul ? în locul în care la definirea constantelor se pune o valoare.

Când se declară un identificator extern, în fișierul sursă C, compilatorul Borland C++ adaugă un caracter "_" la începutul identificatorului, înainte de a-l salva în modulul obiect. Aceasta se poate evita, cu ajutorul opțiunii de compilare **-u**. Dacă se utilizează această opțiune de compilare vor apărea probleme la utilizarea bibliotecilor standard C.

Dacă în modulul asamblare se utilizează identificatori C (date sau funcții), aceștia trebuie să înceapă cu simbolul "_" (în paragraful următor vom aduce lămuriri suplimentare în legătură cu aceasta).

Turbo Assembler nu face distincție între litere mari și litere mici. Când este asamblat un modul, toți identificatorii sunt scriși cu majuscule. Opțiunea /mx face ca TASM să facă distincție între litere mari și mici pentru simbolurile publice și externe. Editorul de legături Borland C++ salvează identificatorii **extern** cu litere mari.

Pentru a face identificatori vizibili în afara modulului asamblare, aceștia trebuie declarați **PUBLIC**. De exemplu,

```

PUBLIC _max, _min
    _max DW 0
    _min DW 0

```

Turbo Assembler oferă directiva **.MODEL** cu specificatorul de limbaj C al cărei efect este că toate numele de identificatori vor fi salvate în modulul obiect precedate de simbolul _. Același efect îl are precizarea specificatorului de limbaj C împreună cu directiva **PUBLIC** (pentru identificatorii declarați cu directiva PUBLIC respectivă). Secvența următoare este echivalentă cu cea anterioară:

```

PUBLIC C max, C min
    max DW 0
    min DW 0

```

sau cu:

```

.MODEL small, C
PUBLIC max, min
    max DW 0
    min DW 0

```

Pentru a putea apela o funcție C dintr-o rutină scrisă în limbaj de asamblare și pentru a referi variabile se folosește directiva EXTRN. De exemplu, dacă programul C are variabilele globale:

```

int i, j[10];
char c;
long rez;

```

acestea pot fi făcute vizibile în modulul asamblare:

```
EXTRN _i:WORD, _j:WORD, _c:BYTE, _rez:DWORD
```

sau, echivalent

```
EXTRN C i:WORD, C j:WORD, C c:BYTE, C rez:DWORD
```

Dacă se utilizează modelul de memorie **huge**, declarațiile **extrn** atât pentru funcții cât și pentru variabile trebuie să apară în afara oricărui segment.

8.3.3.2. Transmiterea parametrilor

Borland C++ suportă două metode de transmitere a parametrilor unei funcții. Una este metoda standard C (convenția de apel C) iar cealaltă este metoda Pascal (convenția de apel Pascal).

Convenția de apel C

Parametrii sunt depuși în stivă în ordinea inversă aripii lor în lista de parametri ai funcției, fiind urmări de adresa de revenire. Să luăm ca exemplu funcția care are următoarea declarație:

```
void funca(int p1, int p2, long p3);
```

și care este apelată astfel:

```
funca(5, 7, 0x1407AA);
```

La apel, înainte de a pune în stivă adresa de revenire, stiva va arăta astfel:

sp →	0005	p1
sp+2 →	0007	p2
sp+4 →	07AA	p3
	0014	

Rutina apelată nu trebuie să știe cu exactitate câți parametri au fost puși în stivă: este sigur că primul parametru actual se află imediat sub adresa de revenire, următorul în continuare și.m.d. În consecință, convenția de apel C permite apelul unei funcții cu un număr de parametri mai mic decât numărul de parametri aflați în antetul funcției (numărul de parametri formali). De asemenea, rutina apelată nu trebuie să scoată parametrii din stivă. Aceasta este sarcina apelatorului.

Identifierii externi trebuie să aibă numele precedat de simbolul **_**.

Convenția de apel Pascal

Funcția considerată ca exemplu mai înainte se declară astfel:

```
void pascal funca(int p1, int p2, long p3);
```

Ca și efect, parametrii vor fi puși în stivă în ordinea aripii lor în declarația funcției (de la stânga spre dreapta), urmări de adresa de revenire. În consecință, rutina apelată (**funca**) trebuie să știe câți parametri i-au fost transmiși și trebuie să ajusteze stiva corespunzător înainte de revenire. Stiva va arăta după cum urmează (înainte de a pune adresa de revenire):

sp →	07AA	p3
sp+2 →	0014	
sp+4 →	0007	p2
sp+6 →	0005	p1

Identifierii externi nu trebuie precedați de **_**, ei se salvează în modulul obiect scriși cu majuscule.

În mod implicit, toate funcțiile scrise în Borland C++ utilizează convenția de apel C. Pentru a specifica faptul că o anumită funcție utilizează metoda Pascal se utilizează modifierul **pascal**.

Dacă se folosește opțiunea de compilare **-p** (sau se selectează butonul **Pascal** din fereastra de dialog **Code Generation**), toate funcțiile folosesc convenția de apel Pascal. În acest caz, se poate forța ca o anumită funcție C să utilizeze metoda C, utilizând modifierul **cdecl**:

```
void cdecl funca(int p1, int p2, long p3);
```

E utilă folosirea convențiilor de apel Pascal în două cazuri:

- Se apelează rutine în limbaj de asamblare deja scrise, care utilizează această metodă de transmitere a parametrilor.
- Codul apelator produs este mai mic, deoarece el nu trebuie să elibereze stiva de parametri după revenire.

Utilizarea convenției de apel Pascal poate să provoace anumite probleme:

- Nu poate fi transmis un număr variabil de parametri, aşa cum se poate face folosind convenția de apel C. Asta se întâmplă deoarece apelatorul trebuie să scoată din stivă parametrii, deci trebuie să știe exact câți sunt.

- Dacă se utilizează opțiunea de compilare **-p**, trebuie incluse fișierele antet ce conțin declarațiile funcțiilor C apelate standard. Fișierele antet au extensia **.h** și se includ cu directiva de preprocesare **#include** (de exemplu **#include <stdio.h>**). În caz contrar, Borland C++ va utiliza convențiile Pascal de apel și de denumire pentru aceste funcții iar editorul de legături nu va face legarea lor. Fișierele antet folosesc pentru declararea acestor funcții modifierul **cdecl** deci compilatorul C va să folosească pentru apelarea lor convențiile C.

Accesarea parametrilor se face conform metodelor generale prezentate în paragraful 8.3.1.4. și înănd seama de convențiile de apel utilizate.

8.3.3.3. Întoarcerea de valori

Funcțiile întorc implicit valori întregi. Pentru valorile pe 8 și pe 16 biți (**char, short, int, enum** și pointeri **near**) se utilizează registrul AX; pentru valorile pe 32 de biți (pointeri **far** și **huge**) se utilizează regiștrii DX (partea cea mai semnificativă, adică segmentul adresei) și AX (partea cea mai puțin semnificativă, respectiv deplasamentul). Valorile **float, double** și **long double** sunt returnate în registrul ST(0) al coprocesorului numeric 80x87, dacă acesta există sau al emulatorului 80x87, dacă se utilizează acesta.

Structurile de 1 octet se întorc în AL, cele de 2 octeți în AX iar cele de 4 octeți în DX:AX. Valorile structurilor de 3 octeți sau mai mari de 4 octeți se returnază punând valoarea într-o locație de date statică și returnând un pointer la acea locație (în AX în modelele de memorie cu date **near**, respectiv în DX:AX în modelele de memorie cu date **far**). Rutina apelată trebuie să copieze valoarea returnată pentru structură în locația de memorie la care punctează pointerul respectiv.

8.3.3.4. Convenții privind utilizarea regiștrilor

Funcțiile apelate dintr-un program C și care sunt scrise în limbaj de asamblare trebuie să conserve valoarea regiștrilor BP, SP, DS și SS. Dacă se dorește modificarea lor în cadrul funcției, înainte de revenirea din funcție valoarea lor inițială trebuie restaurată.

Regiștri SI și DI sunt utilizati de Borland C++ pentru variabilele **register**. Dacă variabilele **register** sunt permise în modulul care apelează funcția descrisă în asamblare, atunci și valoarea acestor regiștri trebuie salvată după intrarea în funcție și restaurată înainte de revenire, dacă valoarea cu care au intrat în funcție a fost modificată. În cazul în care nu sunt permise variabilele **register**, regiștri SI și DI pot să fie modificați oricum.

8.3.3.5. Exemplu

În exemplul care este prezentat în continuare, din programul C scris în fișierul APELASM.CPP se apelează o funcție NUMARALINII scrisă în limbaj de asamblare în fișierul NUMARA.ASM. Această funcție returnează numărul de linii dintr-un sir primit ca parametru, și prin intermediul unui parametru de tip adresă (pointer) returnează numărul de caractere din același sir. De asemenea, din

limbaj de asamblare se apelează funcția AFISARESIR scrisă în modulul C. Fișierul APELASM.CPP este dat în continuare:

```
#include <stdio.h>

char *sir="linia 1 \nlinia 2\nlinia 3";

extern unsigned int NUMARALINII(char * near SirDeNumarat,
                                unsigned int * near CiteCaractere);

void AFISARESIR(char * near Sir) {
    printf(Sir);
}

void main() {
    unsigned int NumarLinii, NumarCaractere;

    AFISARESIR(sir);
    printf("\nOK\n");
    NumarLinii=NUMARALINII(sir, &NumarCaractere);
    printf("Linii: %d\nCaractere: %d\n", NumarLinii, NumarCaractere);
}
```

În continuare este prezentat fișierul NUMARA.ASM:

```
LINIENOUA EQU 0ah ; codul ASCII al caracterului new line '\n'

.MODEL SMALL
.CODE
    EXTRN _AfisareSir:PROC ; declarearea funcției externe definită în APELASM.CPP
    PUBLIC _NumaraLinii ; exportă funcția definită în modulul curent

_NumaraLinii PROC
    PUSH BP ; codul de intrare în funcția NumaraLinii
    MOV BP, SP
    MOV AX, [BP+4]
    PUSH AX
    CALL _AfisareSir ; s-a pus pe stivă adresa sirului pe care dorim să-l tipărim
    ADD SP, 2 ; codul de apel al funcției AfisaSir
    ; am scos cei doi octeți ai parametrului de pe stivă.
    ; În C, parametrii sunt scoși de pe stivă de apelant
```

```
PUSH SI      ; salvăm registrul SI pe stivă
MOV SI, AX   ; punem adresa de offset a șirului în SI. Adresa de segment este deja în DS
SUB CX, CX
MOV DX, CX
; s-a pus 0 atât în CX, cât și în DX. În CX sunt calculate numărul de caractere, iar
; în DX numărul de linii.
```

ET1:

```
LODSB        ; încarcăm un caracter din șir în AL
AND AL, AL
JZ ET2
; dacă este caracterul cu codul ASCII 0 (orice șir C valid se termină cu acest
; caracter) am terminat de parcurs șirul
INC CX        ; altfel, contorizăm și acest caracter
CMP AL, LINIENOUA
; dacă este caracterul cu codul ASCII 0ah, contorizăm noua linie în DX
JNZ ET1
INC DX
JMP ET1
```

ET2:

```
INC DX
MOV BX, [BP+6] ; adresa unde trebuie salvat numărul de caractere se găsește pe stivă
MOV [BX], CX   ; punem la această adresă numărul de caractere
MOV AX, DX     ; returnăm în AX numărul de linii
```

```
POP SI         ; refacem valoarea registrului SI
MOV SP, BP     ; codul de ieșire
POP BP
RET
; revenim fără a scoate parametrii. Apelantul trebuie să facă asta.
```

```
_NumaraLinii ENDP
END
```

Pentru a crea fișierul executabil se asamblează fișierul NUMARA.ASM, apoi se creează un proiect care va conține APELASM.CPP și NUMARA.OBJ. O altă posibilitate este cu următoarea linie de comandă:

```
BCC -ms APELASM.CPP NUMARA.ASM
```

CAPITOLUL 9

PROGRAMAREA LOW LEVEL ÎN LIMBAJELE PASCAL ȘI C

Unele limbi de nivel înalt oferă facilități de inserare în textul sursă a unor instrucțiuni scrise în cod mașină sau în limbaj de asamblare. În acest context, capitolul de față prezintă modalitățile de inserare de cod mașină în sursa unui program Borland Pascal. Vor fi descrise de asemenea, posibilitățile de inserare într-un program Pascal de instrucțiuni scrise în limbaj de asamblare (așa numitul asamblor inline al lui mediului Borland Pascal 6.0). La finalul capitolului, se vor prezenta asamblorul inline al lui mediului Borland C 3.1 care permite inserarea de instrucțiuni asamblare în textul sursă al unui program C.

9.1. INSERAREA DE COD MAȘINĂ ÎN TEXTUL SURSA PASCAL

Borland Pascal oferă programatorului posibilitatea de a insera în codul sursă al unui program instrucțiuni scrise în cod mașină. În acest scop se utilizează instrucțiunea și directiva **inline**. Acestea permit generarea de octeți care vor fi interpretați ca și instrucțiuni mașină. Se utilizează în situațiile în care o anumită acțiune nu poate fi efectuată folosind instrucțiuni Pascal.

9.1.1. Instrucțiunea **inline**

O instrucțiune **inline** constă din cuvântul rezervat **inline** urmat de unul sau mai multe *elemente inline*, separate prin caracterul slash (/) și închise între paranteze:

inline (element_inline { / element_inline }),

unde sintaxa unui *element_inline* este:

$$\left[\begin{matrix} < \\ > \end{matrix} \right] \left\{ \begin{matrix} \text{constanta} \\ \text{identificator_variabilă} \end{matrix} \right\} \left[\begin{matrix} + \\ - \end{matrix} \right] \left\{ \begin{matrix} \text{constanta} \\ \text{offset} \end{matrix} \right\}$$

Fiecare *element inline* constă dintr-un specificator de dimensiune "<" sau ">" (optional), urmat de o constantă sau un identificator de variabilă și de eventual unul sau mai mulți specificatori de offset. Un specificator de offset constă dintr-un caracter "+" sau "-" urmat de o constantă.

Fiecare *element inline* generează un octet sau un cuvânt de cod. Valoarea este calculată din prima constantă sau din offset-ul variabilei, la care se adună sau se scade valoarea fiecărei dintre constantele care urmează. Un *element inline* generează un octet de cod dacă constă numai din constante și dacă valoarea lui este situată între 0 și 255. Dacă valoarea este în afara acestui interval sau *elementul inline* face referire la o variabilă, atunci se generează un cuvânt de cod (octetul mai puțin semnificativ mai întâi).

Dacă un *element inline* începe cu "<" atunci se generează un octet de cod, indiferent de valoarea determinată pentru acel element. Dacă valoarea este reprezentată pe 16 biți, se ia în considerare doar octetul mai puțin semnificativ. Dacă un *element inline* începe cu ">" atunci se generează un cuvânt de cod, eventual completându-se cu 0 partea cea mai semnificativă (dacă e necesar).

De exemplu, instrucțiunea

```
inline (<$1234/>$44)
```

generează trei octeți de cod: \$34, \$44, \$00.

Valoarea unui identificator de variabilă dintr-un *element inline* este offset-ul adresei variabilei respective în cadrul segmentului ei. Segmentul din care fac parte variabilele globale (cele declarate la nivelul cel mai exterior al unui program sau al unei unități Pascal, sau cele declarate în afara oricărui funcție în limbajul C) și constantele cu tip este segmentul de date global, care poate fi accesat prin intermediul lui DS. Segmentul din care fac parte variabilele locale (cele declarate în cadrul subprogramului curent) este segmentul de stivă. În acest caz, offset-ul se calculează relativ la valoarea registrului BP.

Următorul exemplu de instrucțiune **inline** generează cod mașină pentru a copia un anumit număr de cuvinte la o adresă specificată. Procedura descrisă **UmplereCuvinte** va memora **Contor** cu valoarea **Data** în memorie, începând cu primul octet de la adresa conținută în variabila **Dest**.

```
Procedure UmplereCuvinte (var Dest; Contor, Data:Word);
```

```
begin
```

```
    inline(
        $C4/$BE/Dest/ { LES DI, Dest[BP] }
        $8B/$8E/Contor/ { MOV CX, Contor[BP] }
        $8B/$86/Data/ { MOV AX, Data[BP] }
        $FC/ { CLD }
        $F3/$AB); { REP STOSW }
```

```
end;
```

Instrucțiunea **inline** poate să apară oriunde în partea de instrucțiuni a unui bloc.

9.1.2. Directiva **inline**

Directiva **inline** permite scrierea de proceduri și funcții care în momentul invocării sunt expandate într-o secvență dată de instrucțiuni în cod mașină. Acestea pot fi comparate cu macrourile în limbaj de asamblare (vezi 3.3.7.). Le vom numi în continuare proceduri, respectiv funcții inline.

Directivele inline au aceeași sintaxă ca și instrucțiunile **inline**.

Când este apelată o procedură sau o funcție "normală" (care poate conține și instrucțiuni **inline**), compilatorul generează codul de apel: pune pe stivă parametrii (dacă există) și generează o instrucțiune **call** pentru a apela procedură sau funcția respectivă. Pe de altă parte, când se invocă o procedură sau funcție inline, compilatorul generează codul dictat de directiva **inline** în loc de a genera cod de apel. Eventualii parametri sunt puși în stivă.

De exemplu,

```
Procedure DisableInterrupts; inline($FA); { CLI }
```

Când este apelată **DisableInterrupts** se generează 1 octet de cod - o instrucțiune **cli**.

Procedurile și funcțiile declarate cu directiva **inline** pot să aibă parametri; acești parametri nu pot fi referiți cu ajutorul numelor lor în directiva **inline** (alte variabile, însă, pot). Ei pot fi accesati numai de pe stivă. De asemenea, deoarece astfel de proceduri și funcții sunt, de fapt, macrouri, nu se generează în mod automat cod de intrare și cod de ieșire.

Funcția următoare înmulțește două valori **Integer** producând un rezultat **LongInt**:

```
Function LongMul(x,y:Integer):LongInt;
```

```
inline(
    $5A/ { POP DX ; DX := y }
    $58/ { POP AX ; AX := x }
    $F7/$EA); { IMUL DX ; DX:AX := x*y }
```

Observați faptul că nu este necesar cod de intrare sau de ieșire. Acestea nu se cer, deoarece cei 4 octeți generați sunt inserați în fluxul de instrucțiuni în momentul în care este apelată funcția **LongMul**.

Datorită lizibilității reduse și a modului în care sunt tratate, directivele **inline** sunt utile doar pentru proceduri și funcții foarte scurte. Procedurile și funcțiile **inline** nu pot fi folosite ca și argumente ale operatorului **@** și nici ale funcțiilor **Addr**, **Ofs** și **Seg**.

9.2. ASAMBLOARE INLINE

9.2.1. Asamblorul inline al lui Borland Pascal 6.0

Asamblorul inline al lui Borland Pascal 6.0 permite inserarea de cod asamblare direct în sursa programelor Pascal. El implementează o submulțime mare din sintaxa suportată de Turbo Assembler (oferit de Borland) și Macro Assembler (oferit de Microsoft). De asemenea, permite o scriere mai comodă decât inserarea de cod mașină folosind instrucțiunea **inline**.

9.2.1.1. Instrucțiunea **asm**

Sintaxa instrucțiunii **asm** este:

```
asm Instrucțiune_Asm { Separator Instrucțiune_Asm } end
```

unde *Instrucțiune_Asm* este o instrucțiune a asamblorului iar *Separator* este caracterul ";" , caracterul de început de linie sau un comentariu Pascal.

Exemplu:

```
asm
    mov ax, A; xchg ax, B; mov A, ax
end;
```

O instrucțiune **asm** trebuie să asigure integritatea registrilor BP, SP, SS și DS, dar poate utiliza cum dorește registrii AX, BX, CX, DX, SI, DI, ES și registrul de flaguri.

9.2.1.2. Instrucțiunile asamblorului

Sintaxa unei instrucțiuni a asamblorului inline este prezentată în continuare:

```
[Etichetă] { Prefix } [Mnemonica [ Operand { , Operand } ]]
```

În continuare vom considera pe rând elementele precizate.

Eticheta

Eticheta este un identificator de etichetă. Asamblorul inline al Borland Pascal 6.0 suportă două tipuri de etichete: etichete Pascal și etichete locale. Etichetele Pascal sunt cele definite în secțiunea de declarații **label** a programului Pascal.

Etichetele locale sunt etichete vizibile doar în cadrul instrucțiunii **asm** care le definește. Ca și sintaxă, ele încep cu simbolul "@" care este urmat de oricără litere, cifre, simboluri "@" sau simboluri "_". Aceste etichete nu trebuie să fie declarate utilizând declarația **label**.

În continuare vom da un exemplu de utilizare a celor două tipuri de etichete.

```
label Start, Stop;
...
begin
  asm
    Start:
    ...
    jz Stop
    @1:
    ...
    loop @1
  end;
  asm
    @1:
    ...
    jc @2
    ...
    jmp @1
    @2:
  end;
  goto Start;
Stop:
end;
```

Se poate observa că o etichetă Pascal standard, spre deosebire de o etichetă locală, poate fi definită în cadrul unei instrucțiuni **asm** și referită în afara ei și reciproc. Același nume de etichetă locală poate fi utilizat în diferite instrucțiuni **asm**.

Prefix

Dintre prefixele de instrucțiuni definite de Turbo Assembler, cele suportate de asamblorul inline sunt:

REP REPE/REPZ REPNE/REPNZ - SEGCS SEGDS SEGES SEGSS

prefixe pentru repetarea unor instrucțiuni pe șiruri
prefixe de specificare a segmentului; acestea se folosesc pentru a preciza care este segmentul din care se iau operanții instrucțiunii prefixate

Spre exemplu,

```
asm
  rep movsb      { copiază CX octeți de la adresa DS:SI la adresa ES:DI }
  SEGES lodsw    { încarcă în AX un cuvânt de la ES:SI și nu de la DS:SI }
```

```

SEGCS mov ax,[bx]      { echivalentă cu mov ax, cs:[bx] }
SEGES      { se referă la instrucțiunea în limbaj de asamblare care urmează }
mov WORD PTR [DI],0    { devine mov WORD PTR ES:[DI], 0 }
end;

```

Pot exista mai multe prefixe pentru o singură instrucție. În cazul în care sunt utilizate mai multe prefixe, ordinea lor este foarte importantă, nu toate combinațiile interpretându-se corect.

Mnemonica

Asamblorul inline suportă toate codurile de operații ale instrucțiunilor 8086/8087 și 80826/80287 și câteva directive de asamblare. Codurile instrucțiunilor 8087 sunt disponibile doar în starea {\$N+}, codurile 80286 sunt disponibile doar în starea {\$G+}, iar codurile 80287 sunt disponibile doar în starea {\$G+, \$N+}.

Sigurele directive de asamblare permise sunt DB, DW și DD. Acestea au ca și efect generarea unor secvențe de octeți (DB), cuvinte (DW) sau dublucuvinte (DD). Aceste date vor fi generate în segmentul de cod. Pentru a genera date în segmentul de date se folosesc declarațiile var și const din Pascal. Pentru celelalte directive ale limbajului de asamblare, limbajul Pascal oferă construcții cu care acestea pot fi înlocuite. Astfel, majoritatea directivelor EQU corespund declarațiilor const, var și type din Borland Pascal iar directiva PROC corespunde declarațiilor procedure și function.

Asamblorul inline nu permite declarații de variabile folosind sintaxa din Turbo Assembler (vezi 3.3.2.). Toate variabilele trebuie declarate utilizând sintaxa Pascal. Sigurele simboluri care pot fi definite sunt etichetele.

De exemplu, Turbo Assembler permite următoarea secvență:

```

VarByte   DB   ?
VarWord   DW   ?
...
mov al, VarByte
mov bx, VarWord
...

```

În schimb, asamblorul inline al lui Borland Pascal 6.0 nu suportă astfel de declarații de variabile. Construcția precedentă poate fi descrisă astfel:

```

var
  VarByte: Byte;
  VarWord: Word;
...

```

```

asm
...
  mov al, VarByte
  mov bx, VarWord
...
end;

```

Operand

Operanții unei instrucții a asamblorului inline sunt, în general, expresii - combinații de constante, registri, simboluri și operatori. În expresii intră și anumite cuvinte rezervate. Acestea sunt: AH, CL, FAR, SEG, AL, CS, HIGH, SHL, AND, CX, LOW, SHR, AX, DH, MOD, SI, BH, DI, NEAR, SP, BL, DL, NOT, SS, BP, DS, OFFSET, ST, BX, DWORD, OR, TBYTE, BYTE, DX, PTR, TYPE, CH, ES, QWORD, WORD și XOR.

Aceste cuvinte rezervate sunt prioritare identificatorilor cu același nume definiti de utilizatori. Pentru a preciza că se face referire la simbolul utilizator și nu la cel predefinit, identificatorul respectiv va fi precedat de caracterul "&".

Exemplu:

```

Var ch:Char;
...
asm
  mov ch,1; { se referă la registrul CH }
  mov &ch,1; { se referă la variabila ch }
end;

```

9.2.1.3. Expresii

Asamblorul inline evaluează toate expresiile ca valori întregi pe 32 de biți; nu suportă valori reale și nici șir de caractere (exceptând constantele șir de caractere). Expresiile coincid cu cele descrise în 3.2. Constantele hexazecimale pot să fie scrise și în sintaxa Pascal (precedate de simbolul "\$"). Referirea la o variabilă înseamnă deplasamentul acestei variabile în segmentul ei de bază (și nu conținutul ei, cum este în Pascal). Spre exemplu,

```

Var x: Integer;
...
asm
  mov ax, x+4;
  { memorează în registrul AX valoarea unui cuvânt memorat la 4 octeti de x și nu valoarea lui x plus 4! }
  mov bx,x; { memorează în registrul BX valoarea lui x }
end;

```

Fiecare expresie are asociată o *clăsă* și un *tip*.

Identifierii recunoscuți de către asamblorul inline sunt: etichete Pascal, constante Pascal, nume de tipuri Pascal, variabile Pascal, proceduri și funcții Pascal și simbolurile speciale: @Code, @Data și @Result.

Semnificațiile celor trei simboluri speciale enumerate anterior sunt:

- @Code - segmentul de cod curent
- @Data - segmentul de date curent
- @Result - o variabilă care conține rezultatul unei funcții

Primele două dintre aceste simboluri pot fi utilizate doar împreună cu operatorul SEG. De exemplu,

```
asm
  mov ax, SEG @Data
  mov ds, ax {memorează în registrul DS adresa de segment a segmentului curent de date}
end;
```

Exemplu:

```
Function Suma(x,y:Integer):Integer;
begin
  asm
    mov ax, x
    add ax, y
    mov @Result, ax
    {pone valoarea din AX în locul de unde apelantul va lua rezultatul întors de funcție}
  end;
end;
```

În expresiile destinate asamblorului inline NU pot să apară:

- proceduri și funcții standard;
- numele de tablouri Mem, MemW, MemL, Port, PortW;
- constante sir de caractere mai lungi decât 4 octeți;
- constante reale și mulțimi;
- proceduri și funcții inline;
- etichete care nu au fost declarate în blocul curent;
- simbolul @Result, în afara unei funcții

În continuare vom prezenta, într-un tabel, modul în care se determină valorile simbolurilor care pot să apară în expresii:

Simbol	Valoare
Eticheta	adresa etichetei
Constantă	valoarea constantei
Nume de tip	0
Câmp	deplasament câmp
Variabilă	adresa variabilei
Procedură	adresa procedurii
Funcție	adresa funcției
Nume unitate	0
@Code	adresa segmentului de cod
@Data	adresa segmentului de date
@Result	offset variabilă rezultat

Variabilele locale se alocă pe stivă. Valoarea unui simbol variabilă locală este deplasamentul (cu semn) al acestuia față de SS:BP. De exemplu,

```
Procedure X;
var c: Integer;
...
asm
  mov ax,c { se generează codul mov ax, [BP-2] }
end;
...
```

Asamblorul inline tratează întotdeauna un parametru var ca un pointer far (4 octeți). În Pascal, sintaxa pentru a accesa un parametru valoare și cea pentru accesarea unui parametru var este aceeași. În asamblorul inline, sintaxele pentru a accesa parametri valoare respectiv parametri var sunt diferite. Astfel, pentru a accesa conținutul unui parametru var, mai întâi trebuie încărcată adresa acestuia, apoi trebuie accesată locația de memorie la care punctează. De exemplu, dacă funcția Suma definită anterior ar avea parametrii referință, codul ar deveni:

```
Function Suma(var x,y:Integer):Integer;
begin
  asm
    les bx, x
    { consideră că x este o adresă far, pe care o încarcă în ES (segmentul) și în BX (offset-ul)}
```

```

mov ax, es:[bx]
{ memorează în AX valoarea găsită la adresa ES:[BX], adică valoarea parametrului X }
les bx, y
{ consideră că y este o adresă far, pe care o încarcă în ES (segmentul) și în BX (offset-ul)}
add ax, es:[bx]
{ adună la AX valoarea găsită la adresa ES:[BX], adică valoarea parametrului y }
mov @Result, ax
{ pune valoarea din AX în locul de unde apelantul va lua rezultatul întors de funcție }
end;
end;

```

Tipul unei expresii

Prin tipul unei expresii vom înțelege dimensiunea locației de memorie în care se află valoarea ei (de exemplu, tipul unei variabile **Integer** este 2).

Asamblorul inline verifică corectitudinea tipului operanților ori de câte ori este posibil. Uneori, o referință la memorie nu are un tip asociat, acesta deducându-se din tipul celorlalți operanți. De exemplu,

```

asm
  mov al,[100h]
  { pune în registrul AL un octet de la adresa ds:[100H]; tipul asociat - se deduce din
    dimensiunea registrului AL - este 1}

  mov bx,[100h]
  { se pune în BX un cuvânt de la adresa ds:[100H]; tipul asociat - se deduce din
    dimensiunea lui BX - este 2 }

end;

```

Dacă totuși tipul nu se poate deduce, asamblorul cere o conversie de tip explicită. Tipul unei referințe la memorie poate fi modificat cu ajutorul operatorului de conversie PTR. De exemplu,

```

asm
  inc WORD PTR [100h]
  imul BYTE PTR [100h]
end;

```

Tabelul următor conține tipurile predefinite pe care asamblorul inline le adaugă tipurilor Pascal, predefinite sau definite de către utilizator.

Simbol	Tip
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFE H
FAR	0FFE H

Operatori

În continuare prezentăm, într-un tabel, operatorii care pot să apară în constituirea expresiilor asamblorului inline. Aceștia sunt prezenți în ordine descrescătoare a precedenței:

&
0
[]
.
HIGH LOW
+ - (Operatori unari)
:
OFFSET SEG TYPE PTR
* / MOD SHL SHR
+ - (Operatori binari)
NOT
AND OR XOR

Majoritatea operatorilor prezenți anterior sunt cunoscute din capitolul 3. Semnificația unora dintre operatori o descriem în continuare:

& - Suprascrierea identificatorului. Identificatorul imediat următor ampersandului este tratat ca un simbol utilizator.

() - *Subexpresie*. Expresia dintre paranteze este evaluată complet înainte de a fi tratată ca element al unei expresii. Expresia dintre paranteze poate să fie precedată de o altă expresie, caz în care rezultatul este suma valorilor celor două expresii cu tipul dat de tipul primei expresii.

[] - *Referință la memorie*. Expresia dintre paranteze este evaluată complet înainte de a fi tratată ca element al unei expresii. Expresia dintre paranteze poate fi combinată cu regiștri BX, BP, SI sau DI folosind operatorul + pentru a indica procesorului indexarea cu regiștri. Expresia dintre paranteze poate să fie precedată de o altă expresie, caz în care rezultatul este suma valorilor celor două expresii cu tipul dat de tipul primei expresii. Rezultatul este întotdeauna o referință la memorie.

. - *Selector de membru al unei structuri*. Rezultatul este suma dintre expresia situată înaintea operatorului . și expresia situată după operatorul ., având același tip cu cea de-a doua expresie (situată după .).

TYPE - Întoarce tipul expresiei care urmează (dimensiunea în octeți). Tipul unei valori imediate este 0.

PTR - *Operatorul de conversie de tip*. Rezultatul este o referință la memorie având valoarea expresiei ce urmează operatorului și tipul dat de expresia dinaintea operatorului.

- - *Scădere*. Prima expresie poate avea orice clasă dar a doua trebuie să fie o valoare absolută imediată. Rezultatul are aceeași clasă cu prima expresie.

Pentru tipuri înregistrare și variabile de acest tip se poate folosi operatorul de selectare ". ". Ex :

```

type
  Punct = Record
    x, y: Integer;
  End;
  Rect = Record
    A, B:Punct;
  End;
var P:Punct;
  R:Rect;
...
asm
  mov ax, P.x
  mov bx, P.y
  mov cx, R.A.x
  mov dx, R.B.y
end;
...
asm
  mov ax, (Rect PTR ES:[DI]).B.x      { toate }
  mov ax, Rect (ES:[DI]).B.x          { acestea }
  mov ax, ES:Rect[DI].B.x            { sunt }
  mov ax, ES:[DI].Rect.B.x          { echivalente }
end;
```

Ultimele patru instrucțiuni sunt toate echivalente între ele, generând la asamblare aceeași instrucțiune `mov AX, ES:[DI+4]`.

În mod analog, se poate utiliza aceeași sintaxă a calificării cu punct pentru a accesa un simbol definit într-o unitate.

9.2.1.4. Proceduri și funcții asembler

Procedurile (funcțiile) asembler sunt proceduri (funcții) scrise complet în asamblare inline, fără a fi necesară partea `begin ... end`. Ele se definesc cu ajutorul directivei `assembler`. Spre exemplu,

```
Function LongMul(x,y:Integer):LongInt; assembler;
asm
  mov ax,x
  imul y
end;
```

Utilizarea directivei `assembler` are ca efect realizarea de către compilator a câtorva optimizări la generarea codului de intrare în subrutină:

- compilatorul NU generează cod pentru copierea parametrilor valoare în variabile locale. Aceasta afectează toți parametrii valoare de tip string și alți parametri valoare a căror dimensiune este diferită de 1, 2 sau 4 octeți. În cadrul acestor proceduri sau funcții, astfel de parametri trebuie să fie tratați ca și cum ar fi parametri var.
- compilatorul nu alocă o variabilă pentru întoarcerea rezultatului unei funcții; deci, referirea la `@Result` este o eroare. Fac excepție de la aceasta funcțiile care întorc tipul `String`. Pentru acestea există o variabilă `@Result` situată pe stivă sub parametrii care va conține adresa fară locului de unde apelantul va lua rezultatul funcției. Această adresă este pusă pe stivă de către apelant în cadrul codului său de apel;
- compilatorul nu generează cadru de stivă pentru procedurile și funcțiile care nu au parametri și nici variabile locale;

Deci, presupunând că `Locals` este numărul de octeți ocupați de datele locale, iar `Params` este numărul de octeți ocupați de parametri, se generează automat cod de intrare și cod de ieșire, după cum urmează:

codul de intrare (la întâlnirea liniei `asm`):

<code>push bp</code>	; dacă Locals > 0 sau Params > 0
<code>mov bp, sp</code>	; dacă Locals > 0 sau Params > 0
<code>sub sp, Locals</code>	; dacă Locals > 0

codul de ieșire (la întâlnirea liniei *end*):

```
mov sp, bp ; dacă Locals < 0
pop bp ; dacă Locals < 0 sau Params < 0
ret Params ; întotdeauna
```

Funcțiile întorc rezultatul după cum urmează:

Integer, Char, Boolean, enumerare:

1 octet - în AL ; 2 octeți - în AX ; 4 octeți - în DX:AX

Real - în DX:BX:AX

Single, Double, Extended, Comp - în ST(0)

Pointer - în DX:AX

String - în locația temporară punctată de @Result

9.2.1.5. Exemple

Mai întâi vom da un exemplu de funcție care operează asupra stringurilor. Apoi vom implementa un unit Borland Pascal care definește un obiect numit **matrice de biți**.

Exemplul 1: Funcția ce operează asupra stringurilor este realizată cu ajutorul instrucțiunilor asamblorului inline, o variantă descrisă fără directiva **assembler** și o variantă descrisă cu directiva **assembler**. Funcția întoarce ca valoare un șir de caractere care reprezintă scrierea cu majuscule a șirului de caractere care este parametrul funcției. Parametrul funcției este transmis prin valoare.

Varianta 1

```
Function Uppercase( Str:String): String;
begin
  asm
    cld
    lea si, Str
    { Str este un parametru valoare de tip string; deplasamentul lui este încărcat în registrul SI.
    Astfel, se pregătește în SS:SI adresa șirului sursă pentru instrucțiunea lodsb. Str, fiind
    parametru valoare, se transmite punând în stivă valoarea lui, deci segmentul în care se află Str
    este segmentul de stivă. }

    les di, @Result
    { Se încarcă în ES:DI adresa șirului destinație pentru instrucțiunea stosb }
    SEGSS lodsb
    { Se încarcă în registrul AL un octet de la adresa SS:[SI]. Prefixul SEGSS este folosit pentru a
    preciza că segmentul din care se ia operandul sursă al lui lodsb este segmentul de stivă și nu
    segmentul de date }
```

```
stosb { se încarcă la adresa ES:[DI] valoarea din registrul AL, adică lungimea șirului Str }
xor ah, ah { ah = 0 }
xchg ax, cx { se memorează în registrul CX lungimea șirului Str }
jcxz @3
@1:
SEGSS lodsb { AL <- SS:[SI] }
cmp al, 'a'
jb @2
cmp al, 'z'
ja @2
add al, 'A'-'a' { dacă e literă mică, se transformă în literă mare corespunzătoare }
@2:
stosb
loop @1
@3:
end;
end;
```

Varianta 2

```
Function Uppercase( Str:String): String; assembler;
asm
  push ds { Se salvează în stivă conținutul registrului ds deoarece acesta va fi modificat }
  cld
  lds si, Str
  { Str este un parametru valoare care are dimensiunea diferită de 1, 2 sau 4 octeți. Utilizarea
  directivei assembler are ca efect tratarea acestui parametru valoare ca un parametru referință.
  Deci Str este adresa locului din memorie în care se află valoarea șirului. Această adresă este
  încărcată în DS:SI }

  les di, @Result
  lodsb
  stosb
  xor ah, ah
  xchg ax, cx
  jcxz @3
@1:
lodsb
cmp al, 'a'
jb @2
cmp al, 'z'
ja @2
add al, 'A'-'a'
```

```

@2:
  stosb
  loop @1
@3:
  pop ds {restaurăm registrul DS cu valoarea avută la intrarea în funcție}
end;

```

Observații:

- Să vedem ce s-ar întâmpla dacă în locul instrucțiunii **lea si, Str** din varianta 1 am folosi instrucțiunea **mov si, offset Str**, cu care stim că este echivalentă. Știm de asemenea că operatorul **OFFSET** impune evaluarea expresiei în care apare în momentul asamblării, respectiv a compilării. Deci, compilatorul întâlnește instrucțiunea **mov si, offset s**. El trebuie să înlocuiască expresia **offset s** cu valoarea ei din momentul respectiv. Mai întâi va înlocui pe **s** cu **[BP-100h]** (știe că, după apelarea funcției, la adresa BP-100h, pe stivă, va găsi copia valorii parametrului **s**), pe urmă va evalua **offset s**, obținând ca rezultat BP-100h, dar cu valoarea lui registrului BP din momentul compilării. Valoarea memorată în registrul SI este diferită de adresa efectiva a lui **s** în momentul execuției! Acesta este un caz în care instrucțiunile **mov si, offset s** și **lea si, s** nu sunt echivalente.
- Directiva **assembler** este asemănătoare cu directiva **external**, iar procedurile și funcțiile **assembler** trebuie să urmeze aceleasi reguli ca și procedurile și funcțiile **external** (pentru alte precizări, revedeți 8.3.2.).

Exemplul 2: Obiectul matrice de biți conține patru metode:

- constructorul rezervă numărul necesar de cuvinte pentru memorarea matricei de biți, reține dimensiunile și pune la zero toate elementele matricei;
- destructorul eliberează spațiul de memorie ocupat de matrice;
- funcția **dabit** întoarce valoarea bitului din linia **i** și coloana **j** a matricei;
- procedura **punebit** atribuie valoarea **v** (0 sau 1) elementului din linia **i** și coloana **j** a matricei.

Textul programului este:

```
unit MatriceB;
```

Interface

```
type TMatriceB = Object
  m, n: integer;
  p: pointer;
```

```

Constructor Init(vm, vn:integer);
Destructor Done;
Function DaBit(i, j:integer):boolean;
Procedure PuneBit(i, j:integer; v:word);
end;

```

Implementation

```

Constructor TMatriceB.Init(vm, vn:integer);
var q:pointer;
  k:integer;
begin
  m := abs(vm);
  n := abs(vn);
  p := nil;
  if (m = 0) or (n = 0) then exit;
  k := (m*n+15) div 16;
  GetMem(p, 2*k);
  if (p = nil) then exit;
  q := p;
  asm
    cld
    mov cx, k
    mov ax, 0
    les di, q
    rep stosw
  end
end; { Init }

```

```

Destructor TMatriceB.Done;
var k:integer;
begin
  if p = nil then exit;
  k := (m*n+15) div 16;
  FreeMem(p, 2*k);
end;

```

```

Function TMatriceB.DaBit(i, j:integer):boolean;
var cuvint,bit:integer;
  q:pointer;
  k:word;
begin
  k := 0;

```

```

if (i<1) or (i>m) or (j<1) or (j>n) then exit;
bit := (i-1)*n + j - 1;
cuvint := bit div 16;
bit := bit mod 16;
q := p;
asm
    les di, q
    add di, cuvint
    add di, cuvint
    mov ax, es:[di]
    mov cx, bit
    shr ax, cl
    and ax, $0001
    mov k, ax
end;
DaBit := (k=1);
end;

```

```

Procedure TMatriceB.PuneBit(i, j:integer; v:word);
var cuvint, bit:integer;
    q:pointer;
begin
    if (i<1) or (i>m) or (j<1) or (j>n) then exit;
    bit := (i-1)*n + j - 1;
    cuvint := bit div 16;
    bit := bit mod 16;
    q := p;
    asm
        es di, q
        add di, cuvint
        add di, cuvint
        mov ax, es:[di]
        mov cx, bit
        ror ax, cl
        and ax, $ffff
        or ax, v
        rol ax, cl
        mov es:[di], ax
    end
end;
end.

```

9.2.2. Asamblorul inline al Borland C++

Pentru a utiliza asamblorul inline al lui Borland C++, trebuie ca una dintre următoarele condiții să fie îndeplinită:

- fișierul sursă C să fie compilat cu opțiunea **-B**;
- printre instrucțiunile C, înainte de prima instrucțiune **asm**, să apară linia **#pragma inline**.

Acestea au același efect, și anume compilarea urmată de invocarea lui Turbo Assembler pentru a prelucra codul asamblorului inline. Compilatorul generează mai întâi un fișier asamblare, pe urmă apelează TASM pentru a crea fișierul **.obj**.

Linia **#pragma inline** nu este obligatorie. Dacă nu este prezentă, în momentul în care compilatorul Borland C++ întâlnește pentru prima oară instrucțiunea **asm**, reia compilarea de la început, deci se pierde ceva timp. Din același motiv, este indicat ca **#pragma inline** să fie pusă la începutul fișierului, în caz contrar compilarea reluându-se la întâlnirea ei.

9.2.2.1. Instrucțiunea **asm**

Sintaxa instrucțiunii **asm** este:

asm Cod_Operație [Operand {, Operand }] Delimitator

unde *Cod_Operație* este o instrucțiune 8086 validă, în continuare fiind trecuți operanții acelei instrucțiuni. *Delimitator* poate fi caracterul ";", linie nouă sau un comentariu C.

Se poate observa că sintaxa instrucțiunii **asm** nu specifică prezența unei etichete. Într-adevăr, instrucțiunile de salt (jmp, jc și.a.) pot să aibă ca și argumente doar etichete C. Vom detalia mai târziu această problemă.

Există posibilitatea ca o instrucțiune **asm** să conțină mai multe instrucțiuni ale asamblorului inline, dacă acestea sunt cuprinse între paranteze **{ ... }**. De exemplu,

```

asm {
    pop ax; pop ds
    iret
}

```

Spre deosebire de Pascal, în C nu există un delimitator final pentru instrucțiunea **asm** (în Pascal acest lucru se realiza cu ajutorul cuvântului *end*). Astfel, în C o instrucțiune **asm** specifică sau o singură instrucțiune asamblare sau mai multe, cuprinse între accolade.

Pentru a comenta instrucțiuni **asm** trebuie să se utilizeze sintaxa C. De exemplu,

```

asm mov ax, ds;          /* acest comentariu este permis */
asm { pop ax; pop ds; iret } // un alt comentariu legal
asm push ds;             ; acest comentariu nu este permis

```

Instrucțiunile **asm** sunt instrucțiuni C obișnuite, supunându-se acelorași reguli ca și orice alte instrucțiuni C, doar că ele nu se încheie în mod obligatoriu cu caracterul ";" (ele pot să se termine și cu linie nouă).

O instrucțiune asamblare poate fi utilizată ca o *instrucțiune executabilă*, în interiorul unei funcții (acestea sunt plasate în segmentul de cod), sau ca o *declarație externă* în afara unei funcții (acestea sunt plasate în segmentul de date).

Prezentăm în continuare un program C complet. El conține un apel și o descriere a unei funcții C care calculează minimul dintre cele două argumente ale sale. Unele elemente probabil că nu sunt încă suficient de clare, dar paragrafele următoare vor aduce toate lămuririle necesare.

```

#include <stdio.h>
#pragma inline

asm A dw 5      /* definirea unui cuvânt A, în segmentul de date */
                 /* acest simbol poate fi folosit doar în instrucțiuni ale asamblorului, în
                  instrucțiuni C nefiind recunoscut */

int min (int v1, int v2)
{
    asm {
        MOV ax, v1      // AX <- v1
        cmp ax, v2      // AX <= v2 ?
        jle minexit     // dacă da, salt la eticheta C minexit
        mov AX, v2      // dacă nu, AX <- v2
    }
    minexit:
    return (_AX);    // registrul AX conține min(v1,v2)
                     // vezi 9.4.2.
}

int main()
{
    printf ("\n%d", min(4,3));
}

```

Se poate observa că, deși limbajul C este *case-sensitive*, atât instrucțiunile asamblorului inline cât și operanții lor care nu sunt simboluri C pot fi scrisi atât cu litere mari cât și cu litere mici.

De asemenea, atragem atenția asupra situației parantezei {} care semnifică începutul unui bloc de instrucțiuni în limbaj de asamblare: aceasta trebuie pusă neapărat pe aceeași linie cu cuvântul **asm**.

Compilatorul Borland C++ suportă instrucțiunile cunoscute din Turbo Assembler și directive de asamblare:

Instrucțiunile de salt sunt tratate special. Deoarece o etichetă nu poate fi inclusă în instrucțiune, salturile trebuie făcute la etichete C. Salturile sunt permise doar în cadrul unei aceleiași funcții. Nu pot fi generate salturi fară directe. Salturile indirecte pot fi făcute, utilizând pentru aceasta ca și operand numele unui registru.

În instrucțiunile asamblorului inline al lui Borland C++ sunt permise următoarele directive: db, dd, dw, extrn. După cum s-a putut observa și în exemplul prezentat, aceste directive pot fi folosite în exteriorul oricărei funcții C, efectul fiind definirea unor valori în segmentul de date. Aceste valori, însă, nu pot fi folosite decât în instrucțiuni în limbaj de asamblare. Instrucțiunile C nu recunosc aceste etichete. Spre deosebire de asamblorul inline al lui Borland Pascal 6.0, directivele db, dd, dw pot să fie folosite pentru a declara variabile într-o instrucțiune **asm**.

În instrucțiunile **asm** se pot utiliza toate simbolurile C, inclusiv variabile locale (automatice), variabile **register** și parametrii de funcție; Borland C++ le convertește automat la operanții corespunzători din limbaj de asamblare. Programatorul nu trebuie să se preocupe în a calcula deplasamentele variabilelor locale (în stivă), acestea vor fi determinate în momentul utilizării numelor variabilelor respective.

Instrucțiunile **asm** pot face referiri la membrii unei structuri în maniera *uzuală* (adică *variabila.membru*). Numele unui membru al unei structuri poate fi referit și fără ajutorul unui nume de variabilă. În acest caz, numele membrului este înlocuit cu deplasamentul acestuia față de începutul structurii care conține acel membru. Membrul de structură trebuie precedat de simbolul ":" pentru a specifica faptul că nu este vorba despre un simbol C obișnuit. Dacă există două structuri care au un nume de membru comun, pentru utilizarea acestui nume de membru trebuie specificat în mod clar despre ce structură este vorba. De exemplu:

```

struct Student {
    char nume[30];
    int vîrstă;
} UnStudent;

struct Profesor {
    int vîrstă;
    int an_debut;
} UnProfesor;

...
asm mov ax, UnStudent.(struct Student) vîrstă
...

```

9.2.2.2. Exemplu

Prezentăm în continuare un program C care conține o funcție pentru convertirea unui sir de caractere într-un alt sir de caractere format prin înlocuirea fiecărei litere mici cu majuscula corespunzătoare. Funcția este descrisă folosind instrucțiuni ale asamblorului inline.

```
#pragma inline
#include <stdio.h>

// Prototipul funcției StringToUpper
unsigned int StringToUpper(unsigned char far *DestFarString,
                           unsigned char far *SourceFarString);

#define MAX_STRING_LENGTH 100

char *TestString="Acesta este sirul care va fi modificat";
char UpperCaseString[MAX_STRING_LENGTH];

void main() {
    unsigned int StringLength;

    // Apelarea funcției de conversie
    StringLength=StringToUpper(UpperCaseString,TestString);

    // Tipărirea rezultatelor
    printf("Sirul original: %s\n", TestString);
    printf("Sirul obținut: %s\n", UpperCaseString);
    printf("Numarul de caractere: %d\n\n", StringLength);
}

/* Funcția care realizează conversia la majuscule a unui sir far
Parametrii:
    DestFarString - sir care va conține rezultatul conversiei;
    SourceFarString - sir care conține sirul care trebuie convertit; se termină cu octetul 0
Valoarea returnată:
    Lungimea sirului obținut, mai puțin zero-ul final
*/
unsigned int StringToUpper(unsigned char far *DestFarString,
                           unsigned char far *SourceFarString) {

    unsigned int CharacterCount;
```

```
#define LOWER_CASE_A 'a'
#define LOWER_CASE_Z 'z'

asm ADJUST_VALUE EQU 20h           // valoarea ce va fi scăzută din literele mici
asm cld
asm push ds                         // salvăm pe stivă valoarea registrului DS
asm lds si, SourceFarString         // DS:SI <- adresa sirului săsă
asm les di, DestFarString           // ES:DI <- adresa destinației
CharacterCount=0;                   // numărul de caractere

StringToUpperLoop:
    asm lodsb                         // următorul caracter
    asm cmp al, LOWER_CASE_A          // dacă AL<'a', caracterul din AL nu este literă mică
    asm jb SaveCharacter
    asm cmp al, LOWER_CASE_Z          // dacă AL>'z', caracterul din AL nu este literă mica
    asm ja SaveCharacter
    asm sub al, ADJUST_VALUE         // e literă mică, se transformă în literă mare

SaveCharacter:
    asm stosb                         // salvează caracterul
    CharacterCount++;                // numără-1
    asm and al,al
    asm jnz StringToUpperLoop        // dacă nu, prelucrează acest nou caracter
    CharacterCount--;                // dacă da, nu se numără zero-ul terminal
    asm pop ds                         // refacem valoarea registrului DS salvată pe stivă

return CharacterCount;
```

9.3. PROCEDURI ȘI FUNCȚII IMBRICATE ÎN BORLAND PASCAL

Acum suntem măsură să aprofundăm modul în care sunt implementate în Pascal procedurile și funcțiile imbricate. În acest sens prezentăm un exemplu despre modul în care sunt accesate variabilele locale într-o instrucțiune **inline** aflată într-o procedură imbricată.

- 1 program Exemplu;
- 2
- 3 procedure PA; near;
- 4 var
- 5 IntA: Integer;
- 6

```

7 procedure B; far;
8 var
9   IntB: Integer;
10
11 procedure C; near;
12 var
13   IntC: Integer;
14 begin
15   inline(
16     $8B/$46/<IntC/      { mov ax, [bp+IntC] }
17     $8B/$5E/$04/        { mov bx, [bp+04] }
18     $36/$8B/$47/<IntB/ { mov ax, ss:[bx+IntB] }
19     $8B/$5E/$04/        { mov bx, [bp+04] }
20     $36/$8B/$5F/$06/  { mov bx, ss:[bx+IntA] }
21     $36/$8B/$47/<IntA); { mov ax, ss:[bx+IntA] }
22 end;
23
24 begin
25   C;
26 end;
27
28 begin
29   B;
30 end;
31
32 begin
33   PA;
34 end.

```

În continuare vom prezenta codul obținut în urma compilării acestui program:

```

EXEMPLU.PA.B.C : begin
  cs:0000 55          push bp
  cs:0001 89E5        mov bp, sp
  cs:0003 B80200      mov ax, 0002
  cs:0006 9A7C02F25F call 5FF2:027C
  cs:000B 83EC02      sub sp, 0002
EXEMPLU.15 : inline(
  cs:000E 8B46FE      mov ax, [bp-2]
  cs:0011 8B5E04      mov bx, [bp+4]
  cs:0014 368B47FE    mov ax, ss:[bx-02]
  cs:0018 8B5E04      mov bx, [bp+4]
  cs:001B 368B5F06    mov bx, ss:[bx+6]
  cs:001F 368B47FE    mov ax, ss:[bx-2]

```

```

EXEMPLU.22 : end;
  cs:0023 89EC        mov sp, bp
  cs:0025 5D          pop bp
  cs:0026 C20200      ret 0002
EXEMPLU.PA.B : begin
  cs:0029 55          push bp
  cs:002A 89E5        mov bp, sp
  cs:002C B80200      mov ax, 0002
  cs:002F 9A7C02F25F call 5FF2:027C
  cs:0034 82EC02      sub sp, 0002
EXEMPLU.25 : C;
  cs:0037 55          push bp
  cs:0038 E8C5FF      call EXEMPLU.PA.B.C
EXEMPLU.26 : end;
  cs:003B 89EC        mov sp, bp
  cs:003D 5D          pop bp
  cs:003E CA0200      retf 0002
EXEMPLU.PA : begin
  cs:0041 55          push bp
  cs:0042 89E5        mov bp, sp
  cs:0044 B80200      mov ax, 0002
  cs:0047 9A7C02F25F call 5FF2:027C
  cs:004C 83EC02      sub sp, 0002
EXEMPLU.29 : B;
  cs:004F 55          push bp
  cs:0050 0E          push cs
  cs:0051 E8D5FF      call EXEMPLU.PA.B
EXEMPLU.30 : end;
  cs:0054 89EC        mov sp, bp
  cs:0056 5D          pop bp
  cs:0057 C3          ret
EXEMPLU.32 : begin
  cs:0058 9A0000F257 call 5FF2:0000
  cs:005D 55          push bp
  cs:005E 89E5        mov bp, sp
  cs:0060 31C0        xor ax, ax
  cs:0062 9A7C02F25F call 5FF2:027C
EXEMPLU.33 : PA;
  cs:0067 E8D7FF      call EXEMPLU.PA
EXEMPLU.34 : end.
  cs:006A 5D          pop bp
  cs:006B 31C0        xor ax, ax
  cs:006D 9AE900F25F call 5FF2:00E9

```

Dacă nu este pregătit corect conținutul registrului care constituie registrul de bază pentru accesarea variabilelor locale unei proceduri sau funcții imbricate, atunci rezultatele nu vor fi cele scontate. De exemplu, următorul program:

```
program Exemplu2;
procedure PA; near;
var IntA: Integer;
procedure B; far;
var IntB: Integer;
Procedure C; near;
var IntC: Integer;
begin
  IntC := 3;
  asm
    mov cx, IntC { pune în registrul CX valoarea 3 }
    mov bx, IntB { pune în registrul BX valoarea [BP-2], adică 3 }
    mov ax, IntA { pune în registrul AX valoarea [BP-2], adică 3 }
  end;
end;
begin
  IntB := 2;
  C;
end;
begin
  IntA := 1;
  B;
end;
begin
  PA;
end.
```

Procedurile și funcțiile imbricate nu pot fi declarate cu directiva **external** și nu pot fi parametri procedurali. Se remarcă faptul că, spre deosebire de directiva **external** cu care este asemănătoare, directiva **assembler** poate să fie folosită la proceduri sau funcții imbricate.

9.4. ACCESAREA REGIȘTRILOR ȘI APELAREA DE ÎNTRERUPERI

9.4.1. Borland Pascal 6.0

Pentru accesarea regiștrilor unității centrale, Borland Pascal 6.0 oferă tipul de date **Registers**, care este definit în unitul **dos** astfel:

```
type Registers = record
  case Integer of
    0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : Word);
    1: (AL, AH, BL, BH, CL, CH, DL, DH : Byte);
  end;
```

O variabilă de tipul **Registers** conține valorile regiștrilor. Acestea se folosesc pentru a apela intreruperi dintr-un program Pascal.

Pentru apelarea de intreruperi, sunt folosite proceduri și funcții oferite de unitul **dos**. Ne vom opri doar asupra a două dintre ele, și anume **Intr** și **MsDos**.

Procedura **Intr** are doi parametri: un parametru valoare de tip **Byte**, care va reprezenta numărul intreruperii apelate și un parametru referință la o variabilă de tip **Registers**, care va conține la intrare valorile regiștrilor înainte de apelarea intreruperii iar la ieșire valorile regiștrilor după revenirea din rutina de deservire a intreruperii.

Să luăm ca exemplu un program Pascal care afișează pe ecran un text, folosind pentru aceasta funcția DOS 9h:

```
uses dos;
const mesaj: String= 'Hello, everybody ! $';
var
  reg: Registers;
begin
  reg.AH:= 9;           { încarc în AH valoarea 9 }
  reg.DS:= Seg(mesaj); { încarc în DS:DX adresa far a șiру lui care va fi tipărit }
  reg.DX:= Ofs(mesaj[1]);
  Intr($21,reg);        { apelarea intreruperii 21h }
end;
```

Procedura **MsDos** este folosită pentru apelarea de funcții DOS (deci de funcții ale intreruperii 21h). Are un singur parametru de tip **Registers** care are aceeași semnificație cu cel de-al doilea parametru al procedurii **Intr**. Programul descris anterior face același lucru dacă instrucțiunea **Intr(\$21,reg)** este înlocuită cu instrucțiunea **MsDos(reg)**.

9.4.2. Borland C++

C++ introduce noțiunea de pseudovariabile. O pseudovariabilă este un identificator care corespunde unui registru dat. Aceasta poate fi utilizată ca și cum ar fi o variabilă de tipul `unsigned int` pentru regisztrii de 16 biți sau `unsigned char` pentru regisztrii pe 8 biți. Pseudovariabilele din Borland C++ sunt: `_AX, _AL, _AH, _BX, _BL, _BH, _CX, _CL, _CH, _DX, _DL, _DH, _CS, _DS, _SS, _ES, _SP, _BP, _DI, _SI și _FLAGS`. Dacă de exemplu se dorește apelarea unor rutine prin executarea unor instrucțiuni INT, mai întâi este necesară încărcarea unor regisztri cu anumite informații.

Pseudovariabilele servesc la:

- încărcarea de valori în regisztrii unității centrale (eventual înainte de a apela o rutină sistem);
- obținerea valorilor aflate la un moment dat în regisztri.

Pentru utilizarea corectă a pseudovariabilelor trebuie avute în vedere câteva reguli:

- atribuirea unei variabile simple unei pseudovariabile și reciproc nu va afecta alți regisztri dacă nu este făcută o conversie de tip;
- atribuirea de constante unor pseudovariabile, de asemenea, nu va afecta alți regisztri (exceptând încărcarea regisztrilor de segment, care folosesc ca și intermediar regiszrul AX);
- indirectarea simplă prin intermediul unei variabile pointer va distruge în general datele din unii din regisztrii BX, SI, DI sau ES;
- dacă se dorește încărcarea unui număr de regisztrii (de exemplu pentru a apela o rutină de intrerupere) este indicat ca regiszrul AX să fie încărcat ultimul, acesta putând fi eventual afectat de alte instrucțiuni;
- operatorul de adresă & nu poate fi folosit cu o pseudovariabilă, deoarece aceasta nu are adresă;
- încărcarea cu valori a regisztrilor trebuie făcută exact înainte de a fi utilizati, deoarece compilatorul folosește regisztrii și ca locații temporare;
- valorile unor pseudovariabile nu rămân în general aceleași după un apel de funcție; singurii regiszri care au aceleași valori înainte și după un apel de funcție sunt DS, BP, SI și DI;
- anumiți regiszri trebuie modificați cu atenție, aceasta putând avea efecte nedorite. De exemplu, încărcarea directă a regisztrilor CS, DS, SS, SP sau BP poate genera execuție defectuoasă.

Cum am mai amintit, utilizarea pseudovariabilelor se impune în cazul apelării unei intreruperi. Prezentăm în continuare sintaxa unui astfel de apel. Fișierul dos.h conține antetele unor funcții prin intermediul cărora se pot accesa serviciile DOS, BIOS și alte capabilități specifice masinii. Amintim dintre acestea bdos, bdosptr, bioscom, int86, intdos, enable, disable și multe altele. Noi vom prezenta doar funcția geninterrupt a cărei sintaxă este:

```
void geninterrupt(int intr_num);
```

Are ca efect generarea intreruperii soft precizată de parametrul său. Starea tuturor regisztrilor după apel este dependentă de intreruperea apelată.

Următoarea funcție C apelează INT 10h pentru a citi de la poziția cursorului un caracter și atribuile sale:

```
void readchar(unsigned char page, unsigned char *ch, unsigned char *attr) {
    _AH = 8;           // funcția 8 a intreruperii 10h
    _BH = page;        // specificarea paginii, în mod text
    geninterrupt(0x10); // apelarea serviciilor intreruperii 10h
    *ch=_AL;          // obținerea codului ASCII al caracterului
    *attr=_AH;         // obținerea atributelor caracterului
}
```

9.5. SCRIREA DE RUTINE DE TRATARE A ÎNTRERUPERILOR ÎN LIMBAJELA PASCAL ȘI C

Reamintim că primii 1024 octeți de memorie sunt rezervați de 8086 pentru o mulțime de 256 de pointeri far la rutine sistem speciale, numite handleri de intrerupere. Cei 256 de pointeri mai sunt numiți vectori de intrerupere. Aceste rutine sunt apelate executând instrucțiunea 8086 INT cu parametrul numărul intreruperii (acesta ia valori între 0h și OFFh). Când se execută o astfel de instrucțiune au loc următoarele operații:

- salvarea lui CS și a lui IP în stivă;
- salvarea în stivă a regisztrului de flag-uri;
- interzicerea apariției altor intreruperi;
- salt far la locația punctată de vectorul de intrerupere corespunzător.

Mulți dintre vectorii de intrerupere sunt, însă nefolosiți. Aceasta înseamnă că utilizatorul își poate scrie propriul handler de intrerupere, punând un pointer far la acesta în unul din vectorii de intrerupere nefolosiți.

Majoritatea rutinelor rezidente se instalează în memorie ca handlerile de intrerupere. De câte ori are loc o acțiune specială (cum ar fi semnal de ceas, apăsarea unei taste sau altele) aceste rutine rezidente pot intercepta semnalul care le activează și în consecință să aleagă acțiunile care trebuie să aibă loc. După aceasta, ele pot să dea controlul vechilor rutine de tratare a intreruperilor.

9.5.1. Proceduri interrupt în Pascal

O procedură ce este proiectată ca handler de intrerupere trebuie să conțină directiva interrupt înainte de blocul de instrucțiuni. O procedură interrupt nu poate să fie apelată dintr-o altă procedură și ea trebuie să specifice o listă de parametri (sau doar o submulțime) după cum urmează:

```
procedure MyInt(rFlags, rCS, rIP, rAX, rBX, rCX, rDX, rSI, rDS, rES, rBP:Word);
    interrupt;
begin
    ...
end;
```

De asemenea, o procedură **interrupt** trebuie să fie declarată far. Parametrii corespund regiștrilor procesorului. Regiștrii care sunt transmiși ca parametri pot fi folosiți și modificăți în codul procedurii de deservire a întreruperii (noua valoare a lor va rămâne după ieșirea din rutină). Din lista de parametri prezentată anterior pot fi omisi toți regiștrii sau doar o parte (incepând cu Flags). Dacă, de exemplu, se dorește utilizarea și modificarea în rutina de întrerupere a registrului CX, vor fi declarați ca și parametrii cel puțin rCX, rDX, rSI, rDI, rDS, rES și rBP.

Indiferent de lista parametrilor unei proceduri **interrupt**, compilatorul produce (automat) cod la intrarea în rutina pentru salvarea tuturor regiștrilor pe stivă. Corespondent, la ieșirea din rutină se restaurează automat acești regiștri și se generează (tot automat) o instrucțiune **IRET**. Codul de intrare într-o procedură interrupt este:

```
push ax
push bx
push cx
push dx
push si
push di
push ds
push es
push bp
mov bp, sp
mov ax, seg @Data
mov ds, ax
```

Corespondent, la ieșire se generează automat codul:

```
pop bp
pop es
pop ds
pop di
pop si
pop dx
pop cx
pop bx
pop ax
iret
```

Pentru a seta un anumit vector de întrerupere la o adresă specificată se folosește procedura **SetIntVec**, care este definită în cadrul unit-ului DOS. Sintaxa ei este:

SetIntVec (NrInt, Vector)

unde **NrInt** este un parametru valoare de tip **byte** putând lua valori între 0 și 255 și reprezentând numărul întreruperii (intrarea în tabloul vectorilor de întreruperi) iar **Vector** este un parametru valoare de tip **pointer** reprezentând adresa la care se va seta vectorul de întrerupere corespondent la **NrInt** (vezi funcția DOS 25h).

Uneori se dorește rescrierea unei rutine de tratare a unei întreruperi și, eventual restaurarea vechii adrese. Pentru aceasta este utilă o procedură care întoarce adresa memorată într-un anumit vector de întrerupere. Esta vorba despre procedura **GetIntVec**, definită tot în unitul DOS (vezi funcția DOS 35h):

GetIntVec (NrInt, Vector)

unde primul parametru are aceeași semnificație ca și pentru procedura prezentată anterior, iar al doilea este un parametru transmis prin referință, care va conține, la întoarcerea din procedură, adresa rutinei de întrerupere care deservește întreruperea **NrInt**.

În descrierea de rutine de tratare de întreruperi în limbajul Pascal se poate dori ca o astfel de rutină să fie rezidentă în memorie. Pentru aceasta se folosește procedura **Keep**, care termină un program și îl face rezident în memorie (*Terminate and Stay Resident*). Această procedură are ca și parametru un cod de revenire (vezi funcția DOS 31h).

Prezentăm în continuare un program care modifică rutina de tratare a întreruperii 9, afișând la fiecare apăsare a tastei 'A' mesajul 'Ați apăsat tastă A'.

```
{$M $800,0,0} { 2K stack, no heap }

uses Crt, Dos;
var c:char;
    OldHand : Procedure;

{$F+}
procedure MyHand; interrupt;
var i:Byte;

begin
    i := Port[$60];      { se citește un octet din portul $60 al controlerului tastaturii }
    inline ($9C);        { PUSHF - salvăm flagurile pe stivă }
    OldHand;
    if (i=65) then Writeln('Ați apăsat tastă A')
end;
{$F-}
```

```

begin
    GetIntVec($9,@OldHand);
    SetIntVec($9,Addr(MyHand));
    Keep(0);           { Terminate, stay resident }
end.

```

9.5.2. Funcții interrupt în C

Pentru a scrie o rutină de tratare a unei întreruperi în C trebuie să definiți o funcție de tip interrupt. Mai exact, funcția respectivă trebuie să fie definită astfel:

```
void interrupt MyHand(rbp, rdi, rsi, rds, res, rdx, rcx, rbx, rax, rip, rcs, rflags);
```

Parametrii corespund regeștrilor: rbp lui BP, rdi lui DI, etc. După cum se poate observa, toți regeștrii pot fi transmiși ca și parametri, astfel putând fi utilizati și modificați fără a utiliza pseudovariabilele prezentate în paragraful anterior. O funcție de tip **interrupt** va salva automat, pe lângă regeștrii SI, DI și BP și regeștrii AX, BX, CX, DX, ES și DS. Aceștia vor fi restaurați la ieșirea din procedură. O funcție **interrupt** se termină cu instrucțunea **iret** (aceasta nu trebuie să fie specificată de către programator ci va fi inclusă în cod de către compilator).

O funcție **interrupt** poate să își modifice parametrii: modificarea unui parametru dintre cei declarați va schimba conținutul registrului corespunzător la revenirea din rutina de întrerupere.

Pentru obținerea adresei rutinei de deservire a unei anumite întreruperi se poate utiliza funcția **getvect**, iar pentru modificarea vectorului unei anumite întreruperi se poate folosi funcția **setvect**.

CAPITOLUL 10

EXTENSIИ x86

Odată cu evoluția arhitecturii x86, au apărut o serie de mecanisme noi în programarea de sistem, mecanisme necesare pentru a fructifica noile facilități aduse de această linie de procesoare aflată în continuă dezvoltare. Printre aceste noi facilități apărute amintim:

- posibilitatea de accesare a unei cantități mai mari de memorie, pe lângă megaoctetul memoriei standard;
- apariția nivelerelor de protecție;
- creșterea dimensiunii cuvântului de memorie de la 16 la 32 biți (și mai nou la 64 biți).

Capitolul de față se concentrează asupra acestor noi extensii ale arhitecturii x86 precum și asupra modului în care ele pot fi fructificate. Vom începe cu prezentarea mecanismelor folosite pentru accesarea unei cantități suplimentare de memorie, peste megaoctetul standard.

Procesorul 8086 putea accesa maxim 1 megaoctect ($=2^{20}$ octeți) de memorie fizică – megaoctet numit și memorie standard. Acest lucru era datorat magistralei de adresare (*address bus*) pe 20 biți. Din acești 1024 kiloocteji, primii 640 de kiloocteji, numiți memorie convențională, erau folosiți de către DOS pentru a încărca o parte din rutinele proprii și pentru rularea aplicațiilor utilizator, în timp ce octeji din spațiul de adrese 640-1024 Kilo erau folosiți de sistem pentru încărcarea diverselor drivere pentru periferice (mouse, adaptor de rețea), încărcarea rutinelor BIOS, memoria în mod text și în mod grafic al adaptorului video. Deși la începutul anilor '80, odată cu vânzarea primului IBM-PC rulând DOS 2.0, Bill Gates prezicea că "640 de kiloocteji ar trebui să fie suficienți pentru toată lumea", evoluția ulterioară a industriei soft l-a contrazis.

Câteva din factorii care au făcut ca megaoctetul memoriei standard să devină înainte de mijlocul anilor '80 insuficient sunt:

- dezvoltarea de aplicații soft din ce în ce mai complexe;
- apariția sistemelor de operare multiutilizator și multitasking și pe microcalculatoarele din familia x86;
- apariția suprafeteelor de operare și ulterior a sistemelor de operare cu interfață grafică;
- apariția de noi periferice ale căror drivere erau și ele consumatoare de memorie;
- dezvoltarea protocolelor de rețea (TCP, IPX, NetBEUI) consumatoare și ele de memorie.

Soluția pentru accesarea unei cantități mai mari de memorie fizică a fost creșterea dimensiunii magistralei de adresare. Astfel, magistrala de adresare a crescut la 24 biți la procesorul 286, care putea accesa astfel 2^{24} octeți = 16 megaocteji de memorie fizică, la 32 biți la procesoarele din

familia Pentium sau echivalente, care puteau accesa un maxim de 2^{32} octeți = 4 gigaoceteți, și la 36 de biți la procesorul Pentium II care putea accesa astfel un maxim de 64 gigaoceteți de RAM. Procesoarele de ultimă generație, AMD Athlon64 prezintă o magistrală de adresare pe 40 de biți, iar Intel Itanium pe 44 de biți. Aceste procesoare pot accesa 1 teraoctet, respectiv 16 teraocteți de memorie fizică.

Cantitatea de memorie, peste megaoctetul de bază al unui sistem x86 se numește memorie extinsă. Folosind adrese de segment și offset pe 16 biți, memoria extinsă rămâne însă inaccesibilă (cu o mică excepție) aplicațiilor DOS care folosesc acest mecanism clasic de adresare. Pentru a avea acces la memoria extinsă, în cadrul sistemului de operare DOS există următoarele două posibilități:

- folosirea unui manager de memorie extinsă (*Extended Memory Manager - XMM*). Un manager de memorie extinsă este dependent de mașină, însă oferă într-o manieră independentă de aceasta acces la memoria extinsă. Cel mai cunoscut manager de memorie extinsă este *himmem.sys*, oferit de Microsoft. Asupra funcționalității și facilităților oferite de acest manager de memorie extinsă ne vom referi mai târziu în acest capitol;
- trecerea procesorului din mod real în mod protejat. Mai multe despre modul protejat vom vorbi mai târziu de asemenea în acest capitol. Spunem acum, doar că, în mod protejat, mecanismul de adresare al memoriei diferă de cel bazat pe adrese de segment și offset pe 16 biți folosite de modul real.

10.1. MEMORIA ÎNALȚĂ (HIGH MEMORY) ȘI MEMORIA EXTINSĂ

Folosind mecanismul de adresare bazat pe adresa de segment și adresa de offset, se pot accesa însă, nu 1 Megaoctet de memorie, ci un pic mai mult. Spre exemplu, locația de memorie având adresa de segment 0FFFFh și adresa de offset 1234h se găsește la adresa fizică $0FFFFh * 16 + 1234h = 0FFFh * 10h + 1234h = 101224h$, adresa fizică situată deasupra adresei 0FFFFh (adresa fizică a ultimei locații de memorie din megaoctetul memoriei de bază). Adresa 101224h se scrie în binar: 1 0000 0001 0010 0010 0100b și sunt necesari pentru reprezentarea ei 21 biți. Un procesor 8086 sau 8088, având magistrala de adresare doar pe 20 biți, ar fi ignorat datorită depășirii bitul cel mai din stânga având valoarea 1, obținând adresa fizică: 0000 0001 0010 0010 0100b = 1224h, corespunzătoare adresei 0000:1224h în reprezentarea segment:offset (stîm că reprezentarea unei adrese fizice sub forma segment:offset nu este unică). Ultima locație de memorie accesibilă de procesoarele 8086 și 8088 sub forma segment:offset este 0FFFFh:000Fh. Următoarea adresa, 0FFFFh:0010h, corespunde adresei fizice $0FFFFh * 10h + 10h = 100000h$. Această adresa este interpretată, datorită depășirii, de către aceste procesoare ca 0000h:0000h – adresa primei locații de memorie. Procesorul 80286, având magistrala de adresare pe 24 biți putea interpreta fără probleme această adresa ca adresa primei locații de memorie situată deasupra megaoctetului ce alcătuia memoria de bază. Mai mult chiar, acest procesor putea accesa ca memorie distinctă tot spațiul de adrese de la 0FFFFh:0010h la 0FFFFh:FFFFh, adică 64 de kilooceteți fără 16 octeți. Această zonă de memorie poartă denumirea de memorie înaltă (*high*

memory). Unii au privit această facilitate oferită de procesorul 80286 ca pe un bug, procesorul ne fiind astfel compatibil cu predecesoarele liniei, deși se dorea acest lucru (multe aplicații se baza pe depășirea amintită mai sus). Alții, au privit cei aproape 64 de kilooceteți ca pe o mană cerească, deoarece o parte a memoriei convenționale (cei 640 de kilooceteți de la începutul memoriei fizice) se elibera astfel de rutinele sistemului de operare DOS care puteau fi încărcate în memoria înaltă. Pentru a împăca spiritele, IBM a lăsat posibilitatea dezactivării celei de a 20-a linii a magistralei de adresare (de fapt a 20-a linie este a 21 – numerotarea acestora începând de la 0). Cu această linie dezactivată, un procesor 80286 producea depășire la calculul de adresa și se comporta identic cu predecesorii săi. Această linie se poate dezactiva prin intermediul driverului de memorie extinsă (*himmem.sys* de exemplu), iar la unele calculatoare prin intermediul unei opțiuni disponibile în BIOS-ul calculatorului.

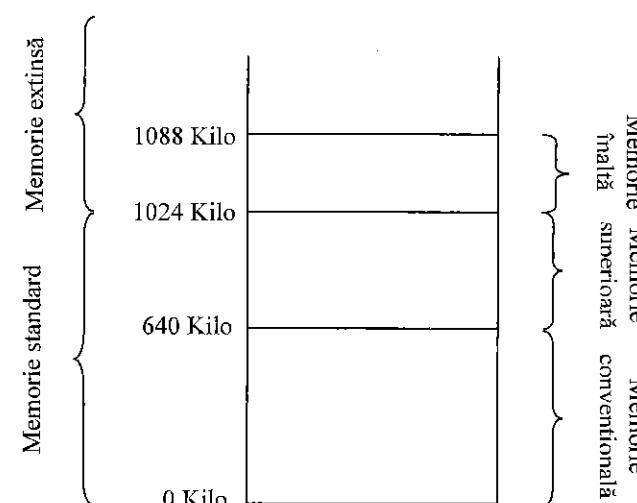


Fig. 10.1. Harta memoriei DOS

La mijlocul anilor '80 memoria înaltă a avut un rol special. Era singura zona din memoria extinsă a unui calculator (memoria situată peste megaoctetul standard) care putea fi accesată cu procesorul rulând în mod real.

Pentru a face memoria extinsă disponibilă în mod real (inclusiv memoria înaltă), după cum am amintit mai sus, este nevoie de un manager de memorie extinsă. Încărcarea lui *himmem.sys* se face prin intermediul fișierului de boot *config.sys*. Managerul de memorie extinsă trebuie specificat la începutul fișierului *config.sys*, înaintea oricăror altor drivere care ar putea folosi memoria extinsă.

În fișierul *config.sys*:

```
DEVICE=C:\DOS\HIMEM.SYS
```

Dacă se dorește ca și rutinele sistemului de operare DOS să fie încărcate în memoria înaltă, eliberându-se astfel o parte a celor 640 Kilocteji convenționali, atunci în fișierul config.sys trebuie adăugată următoarea linie (după linia care încarcă managerul de memorie extinsă):

```
DOS=HIGH
```

Managerul de memorie extinsă poate fi folosit și pentru a activa/dezactiva cea de a 20-a linie a magistralei de adresare (pentru a activa/dezactiva memoria înaltă). Prezentăm în continuare, un exemplu mai puțin corect, dar didactic, pentru a verifica dacă memoria înaltă poate fi accesată, sau dacă procesorul produce depășirea la calculul de adresă și memoria înaltă este asimilată începutului memoriei standard.

Exemplul următor copiază un scurt text, stringul **message**, la începutul memoriei extinse (adresa 0FFFFh:0010h). Dacă se produce depășire, începutul memoriei extinse este asimilat începutului memoriei standard. Sirul scris, de lungime patru octeți, va suprascrie primii patru octeți ai memoriei standard – octeți ce conțin adresa fară rutinei de tratare a intreruperii 0h. Pentru a testa efectul modificării, vom afișa folosind funcția 9h a intreruperii 21h, sirul aflat în memorie la adresa 0000h:0000h. Dacă memoria înaltă nu există și s-a produs depășire la calculul de adresă, la adresa 0000h:0000h se va afla sirul **message** (ura! - tocmai ne-am distrus vectorul de intreruperi). Apelul rutinei de tratare a intreruperii 0h, va duce la blocarea calculatorului (primii patru octeți de la începutul memoriei nu vor mai conține adresa fară unei rutine valide de tratare a unei intreruperi). Dacă memoria înaltă există, și nu se produce depășire la calculul de adresă, atunci sirul **message** va fi copiat la începutul memoriei înalte, zonă distinctă de începutul memoriei standard. Tentativa de a afișa sirul aflat în memorie la adresa 0000h:0000h, va duce la afișarea unor caractere ale căror coduri ASCII se găsesc în memorie la această adresă (unele caractere, având codul ASCII mai mic decât 32, sunt netipăribile – lucru materializat de obicei printr-o serie de *beep*-uri). Funcția 9h, a intreruperii 21h va termina afișarea la întâlnirea primului caracter '\$' din memorie. Vestea bună însă, este că nu am distrus vectorul de intreruperi. Apelând rutina de tratare a intreruperii 0h, vom provoca afișarea cunoscutului mesaj de eroare produs de această intrerupere și la revenirea sistemului la prompterul DOS.

```
assume cs:code, ds:data
```

```
data segment
    message db 'Hi!$'
    l equ $ - offset message
data ends
```

```
code segment
```

```
start:
```

```
        mov ax, data
        mov ds, ax
        mov si, offset message ; DS:SI – adresa sirului sursă

        mov ax, 0ffffh
        mov es, ax
        mov di, 0010h ; ES:DI – adresa sirului destinație

        mov cx, l
        rep movsb ; copiem cei l octeți din sursă în destinație

        mov ax, 0h
        mov ds, ax
        mov dx, 0h
        mov ah, 09h
        int 21h ; afișăm conținutul de la DS:DX

        int 0h ; sistemul ar trebui să se blocheze dacă memoria înaltă nu există

        mov ax, 4c00h
        int 21h

code ends

end start
```

După cum am spus, acesta este un exemplu didactic și nu corect. În cazul fericit al existenței memoriei înalte, când calculatorul nu se blochează, suprascriem, fără să avem voie, primii patru octeți ai acestei memorii – este posibil ca sistemul să folosească acești octeți "la ceva" – fapt ce va duce mai devreme sau mai târziu la un comportament nedefinit al sistemului.

Observații:

1. Rularea acestui exemplu (și a următoarelor) sub sistemul de operare Windows nu este recomandată. Sistemul de operare Windows emulează pentru aplicațiile DOS o mașină virtuală pe 16 biți, dotată cu manager de memorie extinsă. Pentru rularea corectă a exemplului este necesar DOS standard (cel mult versiunea 6.22) sau modul DOS al sistemelor de operare Windows (cel mult Windows 98);
2. În mod DOS (Command Prompt Only) sistemul de operare Windows 98 încarcă managerul de memorie extinsă chiar dacă în fișierul config.sys nu este specificat acest lucru. Pentru rularea exemplului fără manager de memorie extinsă instalat, se va alege la boot-are opțiunea „Safe Mode Command Prompt Only”.

Deși memoria înaltă poate fi accesată direct, gestiunea ei și a memoriei extinse se face prin managerul de memorie extinsă. Întreruperea 2Fh, prin intermediul funcției 43h, permite atât verificarea existenței în memorie a unui manager de memorie extinsă, cât și obținerea adresei funcției de control oferită de acest manager.

Cele două moduri de apel ale funcției 43h a întreruperii 2Fh sunt:

- pentru verificarea existenței în memorie a unui manager de memorie extinsă:

La intrare: al = 00h

La ieșire: al = 80h, dacă în memorie este încărcat un manager de memorie extinsă

- pentru obținerea adresei funcției de control oferită de managerul de memorie extinsă:

La intrare: al = 10h

La ieșire: es:bx = adresa FAR a funcției de control

Prezentăm în continuare un scurt exemplu care verifică dacă un manager de memorie extinsă este încărcat în sistem. Dacă da, se obține adresa funcției sale de control, și apelând mai departe această funcție de control se obține versiunea driverului de memorie extinsă încărcat. În cele ce urmează, vom insista mai mult asupra funcției de control oferită de managerul de memorie extinsă, nu numai datorită funcționalității oferite ci și datorită exercițiului de folosire și de apel a acestei funcții.

```

assume cs:code, ds:data

data segment
neinstalat db 'Nu este instalat un manager de memorie extinsa$'
mesaj db 'Versiunea specificațiilor XMS implementate de driverul instalat este: '
versiune db '','','','','','$'

emm_function dd ?
data ends

code segment

start:
    mov ax, data
    mov ds, ax

    ; verifică dacă este instalat un manager de memorie extinsă (vezi descrierea funcției 43h
    ; a întreruperii 2fh de mai sus)
    mov ah, 43h
    mov al, 00h
    int 2fh

```

```

; dacă nu e instalat afișăm mesajul de eroare și terminăm programul
cmp al, 80h
jne not_installed

```

```

; e instalat; obținem adresa funcției de control implementate de către driver
mov ah, 43h
mov al, 10h
int 2fh

```

```

; adresa funcției de control este o adresă far; salvăm adresa de offset în partea mai puțin
; semnificativă a unui dublu cuvânt, iar adresa de segment în cuvântul mai semnificativ
; al aceluiași dublu cuvânt

```

```

mov word ptr [emm_function], bx
mov word ptr [emm_function + 2], es

```

```

; apelează această funcție cu subfuncția 00h (obține versiunea driverului instalat)

```

```

mov ah, 00h
call emm_function

```

```

; subfuncția returnează în registrul AX versiunea specificațiilor XMS implementate de
; către driver prin intermediul funcției sale de control; Dacă driverul implementează
; specificațiile XMS 3.00, în registrul AX se va memora valoarea 0300h.

```

```

; convertește această versiune la sir tipăribil
mov bx, ax
and bl, 00001111b
add bl, '0'
mov versiune[4], bl

```

```

mov bx, ax
and bl, 11110000b
mov cl, 4
shr bl, cl
add bl, '0'
mov versiune[3], bl

```

```

; la versiune[2] am memorat caracterul !
mov bx, ax
and bh, 00001111b
add bh, '0'
mov versiune[1], bh

```

```

mov bx, ax
and bh, 11110000b
mov cl, 4
shr bh, cl
add bh, '0'
mov versiune[0], bh

```

```

mov ah, 09h
mov dx, offset mesaj
int 21h

```

```
jmp sfarsit
```

```
not_installed:
```

```

mov ah, 09h
mov dx, offset neinstalat
int 21h

```

```
sfarsit:
```

```

mov ax, 4c00h
int 21h

```

```
code ends
end start
```

Funcția de control oferită de managerul de memorie extinsă oferă o serie de subfuncții. Modul de specificare a subfuncției dorite se face prin intermediul registrului AH (asemănător modului de apel al unei funcții oferită de o intrerupere). Prezentăm cele mai importante subfuncții oferite de un manager de memorie extinsă. Pentru lista completă a acestora și informații detaliate despre modul lor de apel, cititorul este rugat să parcurgă [XMS30].

Subfuncția (se specifică în registrul AH)	Utilizarea funcției
0h	Obține versiunea managerului de memorie extinsă
1h	Alocă memorie în zona de memorie înaltă
2h	Deallocă memorie din zona de memorie înaltă
3h	Activează cea de a 20-a linie a magistralei de adresare
4h	Dezactivează cea de a 20-a linie a magistralei de adresare
7h	Obține informații despre starea celei de a 20-a linii a magistralei de adresare
8h	Obține informații despre cantitatea de memorie extinsă disponibilă, cât și informații despre cel mai mare bloc contigu de memorie extinsă

9h	Alocă un bloc de memorie extinsă
0ah	Deallocă un bloc de memorie extinsă
0bh	Mută o zonă de memorie din (în) memoria extinsă în (din) memoria convențională
0ch	Blochează un bloc de memorie extinsă
0dh	Deblochează un bloc de memorie extinsă

Dăm în continuare un exemplu mai complex care folosește memoria extinsă. Programul declară un sir de octeți pe care dorim să-l afișăm. În prealabil însă, acest sir va fi copiat din memoria convențională în memoria extinsă, adus din memoria extinsă înapoi în memoria convențională și abia acum afișat. Dacă programul afișează sirul dorit, înseamnă că operațiile de alocare a memoriei extinse, copiere în și din aceasta s-au desfășurat cu succes. La sfârșit, dealocăm blocul de memorie extinsă alocat.

```
assume cs:code, ds:data
```

```
data segment
```

```
mesaj1 db 'Nu există încarcat un manager de memorie extinsă$'
```

```
mesaj2 db 'Nu pot aloca memorie extinsă$'
```

```
mesaj3 db 'Eroare la copierea din memoria convențională în memoria extinsă$'
```

```
mesaj4 db 'Eroare la copierea din memoria extinsă în memoria convențională$'
```

```
mesaj5 db 'Nu pot dealoca blocul de memorie extinsă$'
```

```
zona db 2048 dup(?)
```

```
afisare db 'Mesaj trecut prin memoria extinsă!$'
```

```
l equ $ - offset afisare
```

```
id dw ? ; aici vom salva descriptorul zonei de memorie extinsă alocate
```

```
xms dd ? ; vom memora adresa far a managerului de memorie extinsă
```

```
label struct_xms ; structură folosită pentru copierea în și din memoria extinsă
```

```
len dd 2048
```

```
source_handle dw ?
```

```
source_address dd ?
```

```
destination_handle dw ?
```

```
destination_address dd ?
```

```
data ends
```

```
code segment
```

```
start:
```

```

mov ax, data
mov ds, ax

; verificăm mai întâi dacă este instalat manager de memorie extinsă
mov ah, 43h
mov al, 00h
int 2fh

cmp al, 80h
je este_driver

; eroare - nu este driver de memorie extinsă instalat
mov ah, 09h
mov dx, offset mesaj1
int 21h

jmp sfarsit_cu_eroare

este_driver:

; dacă da, obținem adresa funcției de control
mov ah, 43h
mov al, 10h
int 2Fh

; și depozităm această adresa în cuvântul xms
mov word ptr [xms], bx
mov word ptr [xms+2], es

; funcția 09h al managerului de memorie extinsă - alocă o zona de memorie extinsă
mov ah, 09h
mov dx, 2h
; în registrul DX se specifică în kiloocteți dimensiunea zonei de memorie extinsă care
; se dorește a fi alocată
call xms

cmp ax, 1
; dacă registrul AX conține valoarea 1, alocarea s-a făcut cu succes iar registrul DX
; conține descriptorul zonei de memorie extinsă

je alocare_ok

```

```

; eroare la alocarea zonei de memorie extinsă de 2 kiloocteți
mov ah, 09h
mov dx, offset mesaj2
int 21h
jmp sfarsit_cu_eroare

alocare_ok:

mov id, dx
; salvăm descriptorul zonei de memorie extinsă în cuvântul id, s-ar putea să
; suprascriem conținutul registrului DX

; funcția 0bh a managerului de memorie extinsă - mută o zonă de memorie în (din)
; memoria extinsă din (în) memoria convențională
; dorim să copiem sirul afisare în zona de memorie extinsă
mov ah, 0bh
mov si, offset struct_xms
; la adresa DS:SI trebuie să se găsească o structură care să conțină adresele far sau
; descriptorii zonelor de memorie convențională sau extinsă cu care se lucrează

mov word ptr len, 1
mov word ptr len+2, 0
; primul dublu cuvânt al structurii conține lungimea zonei de memorie copiate

mov source_handle, 0
; dacă descriptorul zonei sursă este 0, înseamnă că nu dorim să copiem o zonă de
; memorie extinsă ci o zonă de memorie convențională care se specifică prin adresa far
; memorată la dublu cuvântul următor

mov word ptr source_address, offset afisare
mov word ptr source_address+2, seg afisare

mov word ptr destination_address, 0
mov word ptr destination_address+2, 0
; unde dorim să copiem - cuvânt ce conține descriptorul zonei de memorie extinsă
; destinație
mov destination_handle, dx

; facem copierea
call xms

```

```
cmp ax, 1 ; registrul AX este 1 dacă totul s-a desfășurat cu succes
```

```
je copiere_in_xms_ok
```

```
mov ah, 09h  
mov dx, offset mesaj3  
int 21h
```

```
jmp sfarsit_cu_eroare
```

```
copiere_in_xms_ok:
```

; dorim să copiem sirul înapoi din memoria extinsă în memoria convențională
; folosim aceeași funcție
mov ah, 0bh

```
mov si, offset struct_xms ; și aceeași structură
```

; presupunem că nu cunoaștem lungimea sirul pe care dorim să îl copiem
; copiem toată zona de memorie extinsă (2 kiloocteți) în memoria convențională

```
mov word ptr len, 2048  
mov word ptr len+2, 0
```

; în acest cuvânt se specifică descriptorul zonei de memorie extinsă de unde dorim să
; copiem
mov source_handle, dx
; dublu cuvântul următor având valoarea 0, indicăm că sursa copiată va fi o zona
; de memorie extinsă (cea cu descriptorul specificat în registrul DX) și nu o zona de
; memorie convențională

```
mov word ptr source_address, 0  
mov word ptr source_address+2, 0
```

; dând descriptorului destinație valoarea 0, indicăm că destinația va fi o zonă de memorie
; convențională
mov destination_handle, 0
; adresa far a zonei de memorie unde copiem zona de memorie extinsă
; în cazul nostru destinația este sirul zona, suficient de lung - am alocat 2 kiloocteți
; pentru acesta
mov word ptr destination_address, offset zona
mov word ptr destination_address+2, seg zona
call xms

```
cmp ax, 1  
; în registrul AX se returnează modul de terminare a operației de copiere: 1 - ok, diferit  
; de 1 - eroare
```

```
je copiere_din_xms_ok
```

```
mov ah, 09h  
mov dx, offset mesaj4  
int 21h
```

```
jmp sfarsit_cu_eroare
```

```
copiere_din_xms_ok:
```

; dacă copierea sa desfășurat cu succes și din sens invers afișăm sirul dorit
mov ah, 09h
mov dx, offset zona
int 21h

```
sfarsit:
```

; eliberez blocul de memorie extinsă
mov ah, 0ah
mov dx, id
; în registrul DX trebuie specificat descriptorul segmentului de memorie pe care dorim
; să-l dealocăm
call xms

```
cmp ax, 1  
je dealocare_ok  
; dacă nu s-a putut dealoca afișăm mesajul de eroare  
mov ah, 09h  
mov dx, offset mesaj5  
int 21h
```

```
jmp sfarsit_cu_eroare
```

```
deallocare_ok:
```

; terminăm programul după un periplu prin zona de memorie extinsă
mov ax, 4c00h
int 21h

```
sfarsit_cu_eroare:
```

```
; terminăm programul cu cod de eroare 1 dacă s-au întâlnit erori în timpul execuției  
; programului
```

```
mov ax, 4c01h  
int 21h
```

```
code ends  
end start
```

Cea mai importantă funcție oferită de managerul de memorie extinsă este funcția 0bh. Insistăm asupra ei datorită importanței dar și complexității sale. Această funcție este folosită pentru a muta un bloc de memorie din memoria extinsă în memoria convențională, sau invers, dar poate fi folosită pentru a muta o zonă de memorie în cadrul memoriei convenționale, sau din memoria extinsă în memoria extinsă. Funcția 0bh a managerului de memorie extinsă trebuie să primească ca parametru în regiștrii DS:SI adresa far a unei structuri care trebuie să arate în modul următor:

```
label struct_xms  
len dd ?  
source_handle dw ?  
source_address dd ?  
destination_handle dw ?  
destination_address dd ?  
label struct_xms_ends
```

Prima dată membră a acestei structuri, dublu cuvântul **len** reprezintă lungimea în octeți a zonei de memorie transferate. Dacă în procesul de transfer al memoriei este implicată o zonă de memorie convențională, de obicei această dată membră are o valoare reprezentată pe cel mult un cuvânt, cuvântul mai puțin semnificativ al acestui dublu cuvânt (de obicei transferăm din memoria convențională în memoria extinsă sau invers un segment sau o parte a unui segment). Cuvântul mai semnificativ al dublului cuvântului **len** este folosit atunci când în procesul de transfer sunt implicate doar blocuri de memorie din memoria extinsă de lungime mai mare de 64 kiloocteți.

Dacă zona de memorie sursă se găsește în memoria extinsă, descriptorul acestei zone trebuie memorat în data membră **source_handle**. Dacă **source_handle** e 0 (0 nu este un descriptor valid al unui bloc de memorie extinsă), atunci zona de memorie sursă se găsește în memoria convențională. Adresa acesteia se specifică sub forma segment:offset în dublu cuvânt **source_address** (atenție! – adresa de offset se memorează la adresa mai mică, pe cuvântul mai puțin semnificativ al acestui dublu cuvânt).

Zona de memorie destinație trebuie specificată fie în **destination_handle**, dacă zona de memorie destinație se găsește în memoria extinsă, fie în **destination_address**, sub forma segment:offset dacă zona de memorie se găsește în memoria convențională. În acest din urmă caz, **destination_handle** trebuie să fie 0.

În exemplul de mai sus s-au tratat eventualele erori și s-a terminat programul cu cod de eroare 1 în asemenea cazuri.

Memoria înaltă și memoria extinsă au dus la revoluționarea aplicațiilor soft la sfârșitul anilor '80 și începutul anilor '90. Totuși, memoria extinsă era greu de folosit. Programatorul trebuia să-și gestioneze singur blocurile de memorie extinsă, iar imposibilitatea accesării directe a acestora, ci doar după aducerea lor în prealabil în memoria convențională, a descurajat mulți programatori. Memoria extinsă a devenit depășită odată cu trecerea la modul de lucru protejat și a sistemelor de operare care folosesc acest mod de lucru.

10.2. PROCESORUL 80386 ȘI MODUL DE LUCRU PROTEJAT

Elementul esențial adus de procesorul 80386 este lucrul pe 32 de biți. Din acest motiv, majoritatea regiștrilor pe 16 biți ai procesorului 8086, au fost extinși la 32 biți. Pentru a fi identificați ușor, prin convenție, numele lor începe cu litera "E".

Astfel, regiștrii de uz general au fost extinși apărând noi regiștri EAX, EBX, ECX și EDX, fiecare putând memora un dublu cuvânt. Cuvântul mai puțin semnificativ al registrului EAX este registrul pe 16 biți AX pe care îl cunoaștem. Această regulă este valabilă pentru toți regiștrii de uz general. La fel, regiștrii de index au fost extinși și ei la 32 de biți, apărând astfel noi regiștri: ESI, EDI, EBP, ESP, EIP. Cuvântul cel mai puțin semnificativ al acestor regiștri poate fi accesat de asemenea cu numele SI, DI, BP, SP și respectiv IP. Regiștrii de segment însă nu au fost extinși, ei rămnând în continuare la 16 biți. Au fost introdusi însă doi noi regiștri de segment: FS și GS. Pe lângă acești regiștri, există o serie de alți regiștri noi necesari operării procesorului în mod protejat. Vom vorbi despre acești regiștri la momentul oportun. De asemenea, registrul de flaguri este extins și el la 32 de biți; vom sublinia pe parcurs biții mai importanți nou introdusi ai registrului de flaguri. Despre instrucțiunile nou aduse de procesorul 386 vom vorbi mai târziu într-o secțiune separată dedicată acestora.

Fragmentul de cod următor exemplifică modul de folosire al regiștrilor și a instrucțiunilor pe 32 de biți. Problema rezolvată dorește să adune două dublu cuvinte, folosindu-se pentru aceasta de noii regiștri pe 32 de biți introdusi.

```
assume cs:code, ds:data
```

```
data segment
```

```
 a dd 7
```

```
 b dd 8
```

```
 rez dd ? ; dorim să calculăm rez = a + b
```

```
data ends
```

```
code segment
```

start:

```
mov ax, data
mov ds, ax
```

.386 ; specificăm asamblorului să genereze cod pe 32 de biți pentru procesorul 386
mov eax, a ; mov pe 32 de biți (operanții au această dimensiune)

```
add eax, b ; add tot pe 32 de biți
mov rez, eax
```

; folosind registri doar pe 16 biți este nevoie de un număr dublu de instrucții
; mov ax, word ptr a
; add ax, word ptr b

```
; mov dx, word ptr a + 2
; adc dx, word ptr b + 2

; mov word ptr rez, ax
; mov word ptr rez + 2, dx
```

.8086

```
mov ax, 4c00h
int 21h
```

code ends
end start

Se observă în exemplul de mai sus numărul mai mic de instrucții pe 32 de biți necesare adunării a două dublu cuvinte. În cazul de față, număr mai mic de instrucții, nu înseamnă neapărat o viteză mai mare pentru rezolvarea problemei propuse (adunarea a două dublu cuvinte). De ce acest lucru? Instrucția **mov ax, word ptr a** din exemplul de față (pone în registrul AX cuvântul din segmentul de date de la offset **a = 0**) este codificată în limbaj mașină ca A10000h. La introducerea registriilor pe 32 de biți, pentru traducerea în limbaj mașină a instrucției **mov eax, a** (pone în registrul EAX dublu cuvântul din segmentul de date de la offset **a = 0**) s-a ales aceeași codificare. Pentru a se deosebi însă cele două instrucții **mov** (una pe 16 biți, și una pe 32), ceea din urmă este codificată în 66A10000h (codificarea instrucției pe 16 biți prefixată cu octetul 66h). Practic acest prefix indică faptul că operanții instrucției **mov** vor fi pe 32 de biți, și nu pe 16. Timpul de execuție a unei instrucții, măsurat în perioade de ceas, este funcție și de lungimea codificării în limbaj mașină a instrucției respective. Cu cât instrucția este codificată pe un număr mai mare de octeți crește și numărul de perioade de ceas necesar execuției ei. Astfel, instrucția **mov eax, a** necesită o perioadă de ceas în plus față de execuția instrucției **mov** pe 16 biți. Acesta este scenariul de codificare al

instrucțiunilor pentru execuția lor în mod real. La execuția în mod protejat (vom vorbi despre acest mod nativ al procesoarelor pe 32 de biți mai târziu în acest capitol), instrucția codificată în A10000h este interpretată ca fiind **mov eax, a** (în ipoteza că **offset a = 0**), iar codificarea 66A10000h este a instrucției **mov ax, word ptr a** (în aceeași ipoteză că **offset a = 0**). Pentru execuția în mod protejat, instrucția **mov ax, word ptr a** fiind codificată pe un număr mai mare de octeți va fi executată într-un timp mai mare decât instrucția **mov** pe 32 de biți (').

Apariția procesoarelor pe 32 de biți și a modului de lucru protejat, a însemnat sfârșitul modului de lucru real. Deși procesoarele rulând în mod real, pot executa instrucții pe 32 de biți, această facilitate a lor nu a fost exploatață niciodată pe scară largă, poate și datorită slabului suport oferit de compilatoare, care nu generau cod pe 32 de biți destinat a fi rulat în mod real. Rularea de cod pe 32 de biți în mod real poate fi comparată cu mersul unui Ferrari pe un drum de țară.

Metodele descrise în prima parte a acestui capitol pentru accesarea unei cantități suplimentare de memorie au fost în scurt timp surclasate de facilitățile oferite de noile procesoare ale seriei x86. Deși procesorul 286 dispunea de o magistrală de adresare de 24 biți, putând accesa astfel până la 16 megaobiți, iar procesoarele 386 și 486 de 32 biți, putând accesa până la 4 gigaoobiți de RAM, adresarea acestui spațiu de memorie a devenit efectivă odată cu trecerea la modul de lucru protejat. Acest mod de lucru al procesorului nu s-a născut însă din nevoie de a accesa un spațiu mai mare de memorie, ci mai degrabă din nevoie de a porta și crea și pe platforma x86 sisteme de operare multiutilizator și *multitasking* - sistemele de operare din familia UNIX făceau valuri la mijlocul anilor '80 și deși calculatorul personal se bucura și el de mare succes, aceste sisteme de operare nu puteau fi portate pe platforma PC din lipsa suportului oferit de procesor. Procesorul 386 a fost primul procesor pe 32 de biți, dar nu și primul al seriei x86 care a oferit modul de lucru protejat. Deși procesorul 286 putea opera și el în mod protejat, fiind un procesor pe 16 biți, facilitățile sale de lucru în mod protejat nu au fost exploatață niciodată pe scară largă (nici unul dintre sistemele de operare dezvoltate ulterior - Windows 9x sau Linux nu pot fi rulate folosind acest procesor).

Modul de lucru protejat presupune existența unor privilegii diferite sub care se pot accesa datele unor segmente de date/stivă, sau se pot executa instrucțiunile anumitor segmente de cod. În funcție de aceste privilegii, instrucțiunile unui segment de date pot modifica, accesa doar în citire, sau nu pot accesa deloc, conținutul altor segmente de date/stivă.

Acest mod de lucru a apărut din nevoie de a proteja sistemul de operare, codul și datele acestuia de aplicațiile utilizator, în cadrul sistemelor de operare *multitasking* și *multiuser*. În modul de lucru protejat, fiecare program în execuție și fiecare segment de memorie își se atribuie ca nivel de protecție (privilegii) un număr între 0 și 3. Ca regulă generală valabilă, un proces poate accesa orice segment cu nivelul de protecție mai mare sau egal cu al său. Rolul celor patru nivele de protecție este următorul:

- nivelul de protecție 0 este atribuit nucleului sistemului de operare. La acest nivel se execută operațiile de intrare/ieșire la nivel fizic, acțiunile legate de gestiunea memoriei și a proceselor;

- nivelul de protecție 1 este nivelul atribuit apelurilor sistem. Procesele utilizator pot cere serviciile acestui nivel apelând funcții dintr-o listă predefinită de către sistem. Apelurile sistem în cadrul sistemelor de operare moderne sunt echivalentul funcțiilor DOS oferite de întreruperea 21h;
- nivelul de protecție 2 este atribuit bibliotecilor de funcții care pot fi partajate între mai multe programe aflate în execuție. Aceste funcții pot fi apelate de către utilizatori, dar codul acestor instrucțiuni nu poate fi modificat. Astfel de biblioteci sunt fișierele *dll* în Windows și fișierele *so* (*shared objects*) în Linux;
- cel mai slab nivel de protecție, 3, este atribuit proceselor utilizator.

În realitate, majoritatea sistemelor de operare moderne folosesc doar nivelul de protecție 0 pentru datele și codul nucleului sistemului de operare (așa numitul *kernel space*) și nivelul de protecție 3 pentru aplicațiile utilizator (așa numitul *user space*).

Una dintre cele mai importante modificări adusă de modul de lucru protejat este modalitatea de realizare a calculului de adresă, total diferită de cea a calculului de adresă în mod real. Ca la acesta însă, și în modul de lucru protejat memoria este împărțită în segmente. Un segment de memorie nu mai este descris însă de o adresă memorată într-un registru de segment ca în cazul modului real, ci de o structură mai complexă având dimensiunea de opt octeți numită *descriptor de segment*. Descriptorii de segment ai tuturor segmentelor definite de către sistem alcătuiesc tabela globală de descriptori - GDT (*Global Descriptor Table*). Pe lângă această tabelă, un proces își poate defini și o tabelă locală de descriptori - LDT (*Local Descriptor Table*) ce conține descriptori ai segmentelor folosite de procesul respectiv.

Redăm în continuare câmpurile care intră în structura unui descriptor de segment:

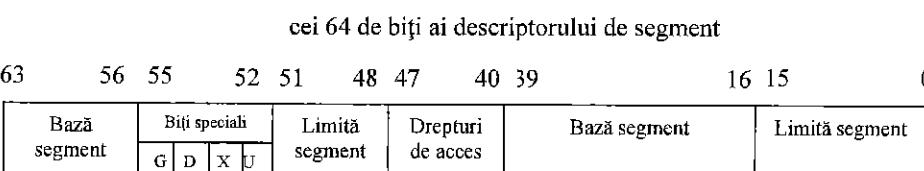


Fig. 10.2. Structura unui descriptor de segment

Se observă că două câmpuri ale descriptorului de segment (bază și limită segment) sunt alcătuite din două componente. Structura descriptorului de segment pentru procesoarele 386 sau mai mari, a fost gândită pentru a fi compatibilă cu descriptorul de segment folosit de procesorul 286 care putea rula în mod protejat pe 16 biți. Structura descriptorului de segment al acestui procesor, avea doar 48 de biți (biți cu rang de la 0 la 47 din figura 10.2.).

Câmpurile din figura 10.2. au următoarea semnificație:

- bază segment ($24 + 8 = 32$ biți) – adresa unde începe segmentul în memorie. Numărul de biți pe care se reprezintă acest câmp este suficient pentru a specifica adresa fizică unde începe segmentul în memorie: 2^{32} pentru procesorul 386 (adică 4 GB RAM căt putea acest procesor să acceseze), respectiv 2^{24} pentru procesorul 286 (16 MB RAM adresabili de către acest procesor);
- limită segment ($16 + 4 = 20$ biți) – dimensiunea segmentului minus un octet. În mod protejat 286, acest procesor putea folosi segment de maxim $2^{16} = 64KB$, în timp ce procesorul 386, în mod protejat poate folosi segment de dimensiune $2^{20} = 1MB$. Dacă bitul G (bit de granularitate) al descriptorului de segment este setat (G = 1), atunci această valoare este shiftată în stânga cu 12 biți. Se permite astfel utilizarea de segmente cu dimensiune cuprinsă între 2^{12} și 2^{32} octeți (între 4KB și 4GB). Practic în acest fel este permisă definirea unui segment care să acopere tot spațiul de memorie adresabil de procesorul 386. De remarcat că bitul G este prezent doar în descriptorul de segment al procesorului 386 (sau mai noi);
- bitii U și X sunt rezervați;
- bitul D dacă are valoarea 0, conținutul segmentului respectiv este interpretat pe 16 biți (instrucțiunile sunt interpretate pe 16 biți dacă este vorba de un segment de cod, stiva este alcătuită din cuvinte, dacă este vorba despre un segment de stivă). Dacă bitul D are valoarea 1, conținutul segmentului este interpretat pe 32 de biți (instrucțiunile sunt pe 32 de biți dacă este vorba despre un segment de cod, stiva este alcătuită din dublu cuvinte în cazul unui segment de stivă). De fapt acest bit face diferență în modul în care este interpretată o codificare a unei instrucțiuni (vezi discuția de la ⁽¹⁾).

Rolul regiștrilor de segment rămâne același, de a localiza un anumit segment de memorie. Valoarea specificată în cadrul unui registru de segment, numită *selector* va fi folosită ca index în cadrul uneia din cele două tabele de descriptori pentru a referi segmentul de memorie dorit. Din cei 16 biți ai unui registru de segment (vezi figura 10.3.), doar 13 se folosesc pentru a forma indexul propriu-zis în tabela de descriptori, un bit se folosește pentru a specifica tabela de descriptori la care se referă indexul (acest bit are valoarea 0 pentru GDT, respectiv valoarea 1 pentru LDT), iar ultimii doi biți se folosesc pentru a se specifica nivelul de protecție. Fiind disponibili pentru index 13 biți, numărul maxim de segmente ce se pot referi este limitat la $2^{13} = 8192$.

index în GTD sau LTD (biți de la 3 la 15)	GDT (0) sau LDT (1)	nivelul de protecție
15	3	2

Fig. 10.3. Structura unui selector memorat într-un registru de segment

Procesoarele, chiar și cele mai noi, la pornire rulează în mod real. Modul real este doar un preambul deoarece, imediat la bootare, sistemele de operare moderne ca Windows sau Linux trec

procesorul în mod de lucru protejat. Modul de lucru real sau protejat, este indicat de valoarea bitului 0 (numit bit PE), într-un registru nou de 32 de biți introdus la procesorul 386 numit CR0 (de fapt, un frate mai mic pe 16 biți al acestui registru, numit MSW – *machine status word*, era prezent și la procesorul 286).

Trecerea la modul de lucru în mod protejat nu se reduce din păcate la simpla setare a bitului mai sus menționat. Este necesară efectuarea următorilor pași:

1. Crearea unei tabele globale valide de descriptori;
2. Dezactivarea intreruperilor (codul majorității rutinelor de tratare este incompatibil cu modul protejat – am văzut în acest capitol că o codificare poate însemna o instrucțiune în mod real și altă instrucțiune în mod protejat);
3. Încărcarea adresei talelei globale de descriptori într-un registru special numit GDTR. Dimensiunea acestui registru este de 48 de biți, din care 32 se folosesc pentru a indica adresa fizică în memorie a talelei globale de descriptori;
4. Setarea bitului 0 a registrului CR0 – abia acum;
5. Încărcarea regiștrilor de segment cu selectori valizi care indică în tabela globală de descriptori;
6. Efectuarea unui *jmp far* pentru încărcarea registrului CS (cu un selector spre un segment de cod) și a registrului EIP cu un offset valid spre acest segment.

Acești pași poți să urmăriți pe exemplu dat la sfârșitul acestui capitol.

Din fericire, sistemele de operare care operează în mod protejat, se îngrijesc de efectuarea acestor pași, efectuează managementul memoriei, al segmentelor și al talelei globale de descriptori. Sub aceste sisteme de operare, programatorul trebuie să scrie doar cod validasm pe 32 de biți. De altfel, pentru aplicațiile utilizator, rulând la nivelul de protecție 3, accesul la tabela globală de descriptori, precum și la regiștri speciali cum ar fi CR0 sau GDTR nu numai că nu este necesar, este și interzis. Sub sistemul de operare DOS, se poate folosi *framework-ul* de programare DPMI (DOS Protected Mode Interface) care efectuează printre altele și trecerea la modul de lucru protejat, managementul memoriei precum și revenirea la modul de lucru real [DPMI].

Odată cu apariția sistemelor de operare care lucrează în mod protejat, a apărut și problema incompatibilității între aceste sisteme de operare și aplicațiile vechi, pe 16 biți, create pentru modul real. Instrucțiunile acestor aplicații erau codificate pentru modul de lucru real, și nu pentru cel protejat. Am văzut că o codificare poate să însemne o instrucțiune în mod real, și altă instrucțiune în mod protejat. Unele sisteme de operare ca OS/2 de exemplu, rulau astfel de aplicații prin trecerea procesorului în mod real la execuția acestor aplicații. Deși simplă, această tehnică avea dezavantajul că expunea sistemul la eventualele erori induse de aceste aplicații care aveau control total asupra sistemului – modul protejat își pierdea tot mai rapiditatea pentru care a fost creat. Pentru rezolvarea acestei probleme, procesoarele începând cu 386, pot rula pe lângă modul real și cel protejat, și în mod virtual 8086. Pentru mai multe detalii privind modul virtual 8086 cititorul poate consulta [VM86].

Am amintit mai sus că la trecerea în mod protejat, trebuie inhibată eventuala execuție a rutinelor de tratare a intreruperilor. În mod protejat, sistemul nu mai folosește vectorul de intreruperi existent în mod real la adresa 0000:0000 ci o tabelă nouă, numită IDT – *Interrupt Descriptor Table*, asemănătoare tabelelor globală și locală de intreruperi, tabelă a cărei adresă este încărcată într-un registru nou, numit IDTR. Rutinele de intrerupere și numerele de ordine a acestora au o cu totul altă semnificație față de cele în mod real. Spre exemplu în mod protejat, intreruperea 10h (16) este rezervată excepțiilor de calcul în virgulă flotantă, pe când în mod real era asociată serviciilor video oferite de către BIOS. Sunt definite de către sistem rutine de tratare a excepțiilor, având numere de ordine de la 0 la 11h, rutinele de la 12h până la 1Fh fiind rezervate de asemenea procesorului. Sistemele de operare își pot defini în tabela descriptorilor de intrerupere rutine având numărul de ordine mai mare de 20h. De obicei aceste rutine sunt redefinite în manieră independentă de către fiecare sistem de operare în parte [PMI].

10.3. NOI INSTRUCȚIUNI ADUȘE DE URMAȘII PROCESORULUI 8086

În cele ce urmează vom trece sumar în revistă instrucțiunile noi introduse de către microprocesoarele de după 8086. Nu vom intra în detaliu, deoarece pe de o parte ar depăși cu mult spațiul tipografic al prezentei lucrări, iar pe de altă parte există pachete de programe accesibile pe orice calculator, în care sunt descrise detaliat aceste instrucțiuni.

Fără a mai reveni, precizăm că la 386 toate instrucțiunile cunoscute pot să opereze pe 32 biți în mod natural. De exemplu, instrucțiunea:

`MOV EAX, 10`

este o instrucțiune corectă. De asemenea, pentru accesarea registrului CR0 se pot folosi instrucțiunile:

`MOV CR0, dublu_cuvant`
`MOV dublu_cuvant, CR0`

Pentru a putea beneficia de elementele de arhitectură nou introduse, utilizatorul are la dispoziție în primul rând limbajul de asamblare. Pentru utilizarea noilor instrucțiuni, asamblorului trebuie să își indice acest fapt prin intermediul unor directive specializate.

Principalele directive utilizate în acest scop sunt:

- .186 admite folosirea instrucțiunilor introduse în plus de către microprocesorul 80186;
- .286 și .286C admite folosirea instrucțiunilor introduse în plus de către microprocesorul 80286, dar nu permit modul de lucru protejat, ci numai modul de lucru real;

.286P permite folosirea tuturor instrucțiunilor introduse în plus la microprocesorul 80286, inclusiv a celor care sunt utilizate în modul de lucru protejat;

.386 și .386C admite folosirea instrucțiunilor introduse în plus de către microprocesorul 80386, dar nu permit modul de lucru protejat, ci numai modul de lucru real;

.386P permite folosirea tuturor instrucțiunilor introduse în plus la microprocesorul 80386, inclusiv a celor care sunt utilizate în modul de lucru protejat;

.8086 este directiva prin care i se transmite asamblorului anularea utilizărilor de facilități suplimentare și restrângerea programării la facilitățile oferite de procesorul 8086;

Pentru 80386, asamblorul mai oferă posibilitatea ca în declarația de segment să se folosească unul dintre argumentele USE16 sau USE32. Prin acestea se indică faptul că în segmentul respectiv se lucrează cu instrucțiuni pe 16 biți, respectiv cu instrucțiuni pe 32 de biți. În mod implicit sunt folosite cele pe 16 biți.

Instrucțiunea IMUL, față de forma cunoscută, mai poate avea forma:

IMUL regprodus, regdinmultit, înmulțitor

Primele două argumente sunt registri. Dacă primul lipsește atunci trebuie să lipsească și al doilea și se va considera implicit AL sau AX. Dacă lipsește al doilea registru, el se va considera implicit egal cu primul. Înmulțitorul poate fi ori o adresă, ori o valoare imediată.

Iată spre exemplu câteva instrucțiuni de înmulțire:

imul bx, ax, A	; bx := ax * A
imul cx, cx, 10	; cx := cx * cx * 10
imul word ptr 10	; ax := ax * 10

Instrucțiunea:

CDQ

extinde, cu păstrarea semnului, dublul cuvântul din EAX într-un număr întreg reprezentat pe 64 de biți, aflat în perechea de registri EDX:EAX.

Procesorul 386, permite ca argument o valoare imediată în instrucțiunea PUSH. De exemplu este validă instrucțiunea:

PUSH 10

Începând cu 80186, s-au introdus, de asemenea perechile de instrucțiuni

POPA - PUSHAD

Instrucțiunea PUSHA are ca efect depunerea în stivă, în ordine, a registrilor: AX, BX, CX, DX, DI, SI, SP și BP. De remarcat faptul că valoarea memorată în stivă a registrului SP este cea avută înaintea instrucțiunii PUSHA. Instrucțiunea POPA reface din stivă cei 8 registri în ordinea inversă depunerii de către PUSHA. La extragerea din stivă, valoarea registrului SP punctează spre vârful curent al stivei, nu primește valoarea care a fost salvată în stivă.

Începând cu 80386 s-au introdus instrucțiunile

PUSHAD - POPAD

care salvează, respectiv restaurează, cei 8 registri extinși: EAX, EBX, ECX, EDX, EDI, ESI, ESP și EBP. Observațiile privitoare la registrul SP de la instrucțiunile PUSA și POPA, rămân valabile aici relativ la registrul ESP.

Tot începând cu 80386 s-a introdus perechea de instrucțiuni

PUSHFD - POPFD

care depune, respectiv extrage în și din stivă registrul extins de flaguri, reprezentat pe 32 de biți.

Instrucțiunile de lucru cu stiva prezентate mai sus, sunt deosebit de utile în cazul în care trebuie salvat contextul de execuție și restaurat ulterior

Începând cu 80386, s-au introdus instrucțiunile SHLD și SHRD, capabile să lucreze cu registrii extinși:

SHLD dest, regs, nrbiti
SHRD dest, regs, nrbiti

Ele deplasează dest, spre stânga sau spre dreapta, cu nrbiti. Biții eliberați în dest nu vor fi completăți cu 0, ci vor fi luați din regs. În cazul deplasării la stânga se iau din regs începând cu bitul 0, iar pentru deplasare spre dreapta începând cu bitul 31. De fapt, aceste instrucțiuni extind într-un fel rotirile cu CF.

De exemplu, dacă dorim să deplasăm cuvântul A spre dreapta cu 5 biți iar biții eliberați să-i înlocuim cu 10101, putem folosi secvența:

```
...
a dw ?
...
mov bx, 10101b
shrd a,bx,5
...
```

nrbiti este fie o constantă, fie registrul CL. **regs** este un registru de 16 sau 32 de biți.

La 80386 au fost introduse instrucțiunile care lucrează pe șiruri de dublu cuvinte:

MOVSD, CMPSD, SCASD, LODSD și STOSD

Acestea lucrează analog cu instrucțiunile pe șiruri de octeți și cuvinte.

Tot începând cu 80386 s-a introdus instrucțiunea

JECXZ destinație

care efectuează saltul atunci când valoarea registrului ECX este zero.

Au fost introduse instrucțiunile de încărcare a unor adrese FAR, analog instrucțiunilor LES și LDS. Offsetul se încarcă într-un registru, iar segmentul într-unul dintre registrii FS, GS sau SS. Aceste instrucțiuni sunt:

LFS reg, adresa

LGS reg, adresa

LSS reg, adresa

Începând cu 80286 au fost introduse instrucțiuni pentru controlul procesorului în modul de lucru protejat. Ele sunt folosite de regulă de către sistemul de operare și mai puțin de către programator.

Controlul tabelelor de descriptori de segment (globală, locală și de intreruperi) este realizat cu ajutorul instrucțiunilor LGDT, LIDT, LLDT, SGDT, SIDT, SLDT. Primele trei încarcă conținutul registriilor descriptori de segmente. Următoarele trei extrag informații din descriptorii de segment

Manevrarea registrului MSW la 80286 se poate realiza cu instrucțiunile:

LMSW sursa

SMSW dest

LMSW încarcă de la sursă conținutul registrului de stare a mașinii (MSW). SMSW salvează conținutul registrului de stare a mașinii la adresa dată de destinație. Pentru 80386, în locul acestor instrucțiuni se vor folosi instrucțiunile

MOV CR0, sursa

MOV destinație, CR0

10.4. EXEMPLU DE PROGRAM CARE LUCREAZĂ CU PROCESORUL ÎN MOD PROTEJAT

Următorul exemplu trece procesorul din modul de lucru real în modul de lucru protejat, afișând un mesaj. După afișarea mesajului, se revine în modul de lucru real. Pentru a scurta lungimea codului sursă și a crea un exemplu cât mai coerent, am scris exemplul în limbajul C, mixat cu cod asamblare inline (vezi capitolul 9).

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <dos.h>

// Numărul de segmente folosite în mod protejat
#define MAX_SEGMENTS 6

// Selectorii care vor fi folosiți în mod protejat. Aceste valori vor fi încărcate în registri de segment
// în mod protejat. Selectorii shiflați în stânga cu 3 pozitii vor fi indici în GDT.

#define NIL_Seg_Selector    0x00 // 0000000000000000 000 - 0
#define CODE_Seg_Selector   0x08 // 0000000000000001 000 - 1
#define DATA_Seg_Selector   0x10 // 0000000000000010 000 - 2 - indici în GDT
#define STACK_Seg_Selector  0x18 // 0000000000000011 000 - 3
#define DATA2_Seg_Selector  0x20 // 000000000000100 000 - 4
#define VIDEO_Seg_Selector 0x28 // 00000000000101 000 - 5

// Constante care ne vor ajuta să construim câmpul drepturi de acces al unui descriptor (vezi
// figura 10.2)
#define DATA_SEG 0x02
#define CODE_SEG 0x0A
#define SEGMENT 0x10
#define DPL0    0x00
#define PRESENT 0x80

// definim tipurile byte, word și dword din comoditate
typedef unsigned char byte;
typedef unsigned short int word;
typedef unsigned long int dword;

// structura unui descriptor de segment pe 8 octeți (vezi figura 10.2)
typedef struct {
    word size_0_15;
    word address_0_15;
```

```

byte address_16_23;
byte type;
byte size_16_19;
byte address_24_31;
} segment_descriptor;

// structură ce va conține dimensiunea și adresa tabelei globale de descriptori. O variabilă de acest
// tip va fi memorată în registrul gdtr.
typedef struct {
    word size;
    dword address;
} GDT_descriptor;

// Tabela globală de descriptori
segment_descriptor GDT[MAX_SEGMENTS];

// variabile în care vom salva valorile regiștrilor de segment din modul real
word backup_cs;
word backup_ds;
word backup_es;
word backup_ss;

// funcție care completează un descriptor de segment. Parametrii funcției sunt:
// descriptor – adresa descriptorului de segment pe care dorim să îl completăm;
// size – dimensiunea segmentului în mod protejat
// segment – adresa segmentului din modul real
// type – drepturile de acces și tipul segmentului
void create_descriptor(segment_descriptor *descriptor, dword size, dword segment,
                      byte type) {
    dword address;

    descriptor->size_0_15 = size & 0xFFFF;
    descriptor->size_16_19 = (size >> 16) & 0xFF;

    descriptor->type = type;

    address = segment << 4;
    descriptor->address_0_15 = address & 0xFFFF;
    descriptor->address_16_23 = (address >> 16) & 0xFF;
    descriptor->address_24_31 = (address >> 24) & 0xFF;
}

```

```

// completăm și încarcăm în registrul gdtr structura ce caracterizează tabela globală de descriptori
void lgdt() {
    GDT_descriptor gdt;
    word segment, offset;

    gdt.size = MAX_SEGMENTS * 8; // dimensiunea GDT

    segment = FP_SEG (&GDT[0]); // adresa de segment a GDT
    offset = FP_OFF (&GDT[0]); // adresa de offset a GDT
    gdt.address = (segment << 4) + offset; // adresa fizică în memorie a GDT este 16 * segment + offset

    asm lgdt gdt
    // încarcăm registrul gdtr cu structura completată
}

// funcție care trece procesorul în modul de lucru protejat
void go_in_protected_mode() {

    // salvăm valorile regiștrilor de segment din modul de lucru real
    backup_cs = _CS;
    backup_ds = _DS;
    backup_es = _ES;
    backup_ss = _SS;

    asm {
        cli // dezactivăm apariția întreruperilor

        mov eax, cr0 // trecem procesorul în mod protejat prin setarea
        or eax, 1 // bitului corespunzător în registrul CR0
        mov cr0, eax

        // completăm regiștrii de segment cu selectori valizi. Cum registrul CS nu se poate
        // modifica direct, construim un jmp far, care va memora un selector valid în acest
        // registru.
        db 0x0ea // codul instrucțiunii jmp far
        dw offset protected_mode // adresa far unde sărim
        dw CODE_Seg_Selector
        protected_mode:

    }

    // completăm celelalte regiștrii de segment cu selectori valizi.
}

```

```

_DS = DATA_Seg_Selector;
_ES = DATA_Seg_Selector;
_SS = STACK_Seg_Selector;
}

// funcție care redusește procesorul în modul de lucru real
void go_in_real_mode() {
    word save_cs;
    word save_ds;

    save_cs = backup_cs;
    save_ds = backup_ss;

    // restaurăm regiștri de segment cu valorile lor din modul de lucru real. Deoarece registrul
    // CS nu poate fi modificat direct, vom construi un jmp far care va completa acest registru
    // cu o adresă de segment validă în mod real. Pentru a construi această instrucțiune, trebuie
    // să accesăm indexat și să modificăm actualul segment de cod. În mod protejat, procesorul
    // nu permite acest lucru, însă folosind selectorul DATA2_Seg_Selector putem accesa
    // actualul segment de cod ca segment de date.

    _DS = DATA2_Seg_Selector;

    asm {
        mov ax, save_cs
        mov di, offset cs:rm_cs
        mov [di], ax

        mov ax, save_ss
        mov di, offset cs:real_mode + 1
        mov [di], ax
    }

    // restaurăm regiștri de segment cu valorile lor din modul de lucru real
    _DS = DATA_Seg_Selector;
    _ES = DATA2_Seg_Selector;

    asm {
        mov eax, cr0      // trecem procesorul în mod protejat prin setarea bitului
        and eax, NOT 1    // corespunzător în registrul CR0
        mov cr0, eax

        db 0x0ea          // jmp far care ne modifică valoarea registrului CS
        dw offset real_mode
    }
}

```

```

rm_cs dw 0

real_mode:
    mov ax, 0
    mov ds, ax
}

_ES = backup_es;           // restaurăm regiștri de segment cu valorile lor din modul de
_SS = backup_ss;           // lucru real

asm sti                   // permitem tratarea rutinelor de întrerupere
}

// tipărim șirul s la linia X, coloana Y, prin copierea fiecărui element al șirului direct în memoria
// video

void print(int x, int y, char *s) {
    char far *video;
    int i;

    // obținem adresa far a memoriei video.
    // VIDEO_Seg_Selector descrie un segment de date suprapus peste memoria video

    video = (char far *) MK_FP(VIDEO_Seg_Selector, y*160 + x*2);
    for (i=0; i<strlen(s); i++) {
        *video = s[i]; // copiem în memoria video un caracter din s
        video += 2;
    }
}

void main() {
    // completăm tabelă globală de descriptori cu
    // - descriptorul NUL (cu acest descriptor trebuie să înceapă orice tabelă globală de descriptori
    // validă)
    create_descriptor(&GDT[0], 0, 0, 0);
    // - un descriptor care să corespundă segmentului de cod din modul real
    create_descriptor(&GDT[1], 0xFFFF, _CS, CODE_SEG | SEGMENT | DPL0 | PRESENT);
    // - un descriptor care să corespundă segmentului de date din modul real
    create_descriptor(&GDT[2], 0xFFFF, _DS, DATA_SEG | SEGMENT | DPL0 | PRESENT);
}

```

```

// - un descriptor care să corespundă segmentului de stivă din modul real
create_descriptor(&GDT[3], 0xFFFF, _SS, DATA_SEG | SEGMENT | DPL0 | PRESENT);

// - un descriptor care să permită accesarea segmentului de cod din modul real ca segment de date în mod protejat
create_descriptor(&GDT[4], 0xFFFF, _CS, DATA_SEG | SEGMENT | DPL0 | PRESENT);

// - un descriptor pentru memoria video
create_descriptor(&GDT[5], 4000, 0xB800, DATA_SEG | SEGMENT | DPL0 | PRESENT);

// Observație: la acest pas al execuției programului, procesorul operează încă în mod real.
// Reîștrîri de segment conțin adrese ale segmentelor de cod, date și stivă. _CS, _DS, _ES și _SS sunt pseudovariabile asociate reîștrîrilor procesorului (vezi 9.4.2).

// Încărcăm în registrul gdtr dimensiunea și adresa tăbelei globale de descriptori lgdt();
// trecem procesorul în mod de lucru protejat
go_in_protected_mode();

// afișăm mesajul la linia 0, coloana 24. Deoarece în mod protejat nu putem folosi întreruperi DOS sau BIOS, iar funcțiile printf și cout se folosesc de funcții ale acestor întreruperi pentru a afișa un sir de caractere, funcția print va tipări sirul prin copierea acestuia caracter cu caracter în memoria video, care în mod protejat va fi accesată ca orice segment de date.
print(0, 24, "Hello from a protected world!");

// revenim în modul de lucru real;
go_in_real_mode();
}

```

CAPITOLUL 11

PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS

11.1. MODUL DE ADRESARE PROTEJAT ȘI MICROPROCESOARE PE 32 DE BIȚI

Trecerea la sistemul de operare Windows a fost facilitată de apariția microprocesorului 80286, care a reprezentat un progres semnificativ în ceea ce privește capabilitățile unui microprocesor, deschizând calea pentru generațiile următoare de microprocesoare. Spațiul maxim de adresare a fost extins de la 1MB la 16MB. Apariția microprocesorului 80286 a dus și la dezvoltarea sistemelor multitasking, inclusiv un mecanism prin care un program poate să evite interferență cu codul sau datele altui program. Toate acestea au devenit posibile prin trecerea la *modul de adresare protejat*.

Spre deosebire de *modul de adresare real* folosit de 8086, în care o adresă de memorie este formată dintr-o adresă de segment și un deplasament în cadrul segmentului (*offset*), în *modul protejat* al 80286 un registru de segment conține o valoare specială numită selector, care este pointer spre un *tabel de descriptori* care conține adrese fizice reprezentate pe 24 biți, adrese care stau la baza formării tuturor adreselor segmentelor de memorie din sistem. Combinând această adresă de bază reprezentată pe 24 biți cu un deplasament reprezentat pe 16 biți, se pot adresa 16MB de memorie fizică.

Un program care rulează în modul real poate să citească sau să scrie în orice zonă de memorie, și astfel nu poate fi controlat accesul unei alte aplicații la memorie, în timp ce o aplicație care rulează în mod protejat poate să acceseze doar adresele de memorie permise de către tăbelele de descriptori. Tăbelele de descriptori sunt controlate de către sistemul de operare. Modul protejat izolează programele unele de altele, nepermittând accesul direct la resursele sistemului.

La baza evitării interferenței cu codul sau datele altui program stă un mecanism prin care fiecare aplicație are un spațiu separat, local, de adrese dar poate și să acceseze spațiul global de adrese al sistemului. Acest mecanism a fost făcut posibil prin trecerea la adresarea indirectă a segmentelor.

Modul de adresare protejat a devenit și mai puternic odată cu apariția primului microprocesor pe 32 de biți, și anume 80386. Offset-ul fiind extins de la 16 biți la 32 biți, iar adresa de bază de la 24 la 32 biți, dimensiunea unui segment poate să crească până la 4GB. Segmentele pot fi divizate în unități mai mici numite *pagini*. Sistemul de memorie virtuală lucrează acum cu pagini de memorie în locul segmentelor de memorie. Acest lucru înseamnă că doar părți din segmente pot fi în memorie la un moment dat, spre deosebire de 80286 cu adresarea pe 16 biți, când fie întregul segment se află în memorie, fie nici o parte din el. Mai multe amănunte referitoare la adresarea pe 32 biți au fost discutate în capitolul 10.

Putem astfel conchide enumerând caracteristicile de bază ale modului de adresare protejat al microprocesoarelor pe 32 de biți:

- **protecție**: fiecărui program îi este alocată o anumită zonă de memorie. Alte programe nu pot folosi această memorie, deci fiecare program este protejat de interferență cu alte programe;
- **memorie extinsă**: permite unui program să acceseze mai mult de 640KB memorie;
- **memorie virtuală**: mărește spațiul de adrese la peste 1GB;
- **multitasking**: mai multe programe pot să ruleze în același timp.

Folosirea acestui mod de adresare a devenit populară odată cu apariția sistemului de operare Windows, care rulează în mod protejat începând cu versiunea 3.1. În prezent, toate sistemele de operare importante rulează în modul protejat cu adresare pe 32 biți.

11.2. PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS

Programele în limbaj de asamblare pot fi scrise pentru orice sistem de operare și orice model de procesor. În acest capitol ne vom concentra asupra programelor scrise în limbaj de asamblare care rulează sub Windows. Mai întâi vom face câteva precizări cu privire la aplicațiile Windows în general, și implicit cu privire la ceea ce se poate scrie în limbaj de asamblare sub Windows.

Arhitectura microprocesoarelor Intel x86 definește patru niveluri de privilegiu. Sistemul de operare Windows folosește nivelul de privilegiu 0 (cel mai privilegiat) pentru modul-nucleu, și nivelul de privilegiu 3 (cel mai puțin privilegiat) pentru modul-utilizator. Fiecare program care rulează în modul-utilizator are spațiul său de memorie. Având cel mai mic nivel de privilegiu, aceste procese nu pot să execute instrucțiuni CPU, au acces limitat și indirect la zona de date sistem și la spațiul de adrese al sistemului și nu au acces direct la echipamentul hardware. Aceste procese sunt considerate ca fiind periculoase în ceea ce privește stabilitatea sistemului, astfel că drepturile lor sunt limitate. O implicație directă a acestor restricții este imposibilitatea apelului direct de intreruperi în programele scrise în limbaj de asamblare sub Windows în modul-utilizator. Spre deosebire de modul-utilizator, în modul-nucleu (cel mai privilegiat) există posibilitatea de a executa orice instrucțiune CPU, există acces nelimitat la zona de date sistem și la resursele hardware. Întreruperile, de exemplu, sunt rezervate pentru a fi folosite la nivel de nucleu.

Sub Windows există un singur model de memorie – modelul *flat*, ce presupune existența a maximum două segmente de memorie, unul pentru cod și unul pentru date. Există câteva reguli importante care trebuie cunoscute de către cei care programează în limbaj de asamblare sub Windows. Windows folosește regiștrii *esi*, *edi*, *ebp* și *ebx* intern și programatorul nu trebuie să schimbe valoarea lor. Aceasta nu înseamnă că acești regiștri nu pot fi folosiți, ci că valoarea lor trebuie restaurată după eventuala lor utilizare.

Sub Windows există următoare modalități de folosire a limbajului de asamblare:

1. programe scrise în întregime în limbaj de asamblare;
2. inserare de cod sursă asamblare în cadrul limbajelor de nivel înalt;
3. programare multimodul, unde cel puțin un modul este scris în limbaj de asamblare.

Aceste modalități vor fi prezentate în cele ce urmează, dar nu înainte de descrierea câtorva generalități cu privire la asamblorul folosit în exemplele prezentate în acest capitol.

În finalul acestei secțiuni să facem precizarea că, deși titlul acestui capitol este „Programare în limbaj de asamblare sub Windows”, el ar trebui să se numească mai degrabă „Programare sub Windows (...și în limbaj de asamblare)”. Aceasta deoarece, datorită modului de adresare protejat sub care rulează SO Windows, programarea sub Windows se caracterizează prin dezvoltarea unor secvențe de apeluri sistem, nebazașându-se pe sistemul de intreruperi propriu modului de lucru real. Ca urmare, „libertatea de mișcare” a unui programator este întotdeauna limitată exclusiv la ceea ce permite programarea Windows, acest lucru neavând practic nici o legătură (din păcate) cu ceea ce ne permite în principiu limbajul de asamblare. Așa cum se va vedea și în exemplele care urmează în acest capitol, acestea nu vor consta decât în identificarea posibilităților limbajului de asamblare de a efectua apeluri sistem Windows. Așadar, fără a descuraja cititorul, ar fi corect să precizăm că cine nu cunoaște programarea Windows, nu va înțelege mare lucru nici din conținutul acestui capitol 😊.

11.3. MICROSOFT MACRO ASSEMBLER

Pentru a putea programa în limbaj de asamblare, vom avea nevoie de un editor de text, un asamblor și un link-editor. Asamblorul convertește codul scris în asamblare în cod mașină. Link-editorul produce executabilul din acest format. Executabilele Windows au extensia .exe. Cele mai folosite asamblări sunt:

MASM – Microsoft Macro Assembler; poate fi obținut de la adresa <http://www.masm32.com/>.

TASM – Borland Turbo Assembler

NASM – Netwide Assembler

În cele ce urmează ne vom referi la asamblorul MASM, considerat a fi printre cele mai populare, datorită „nivelului înalt” care ușurează considerabil dezvoltarea de aplicații scrise în limbaj de asamblare sub Windows. Spre deosebire de alte asamblări, acesta pune la dispoziție biblioteci Windows într-un mod coerent. De asemenea, este bine documentat – este vorba despre pachetul MASM32 dezvoltat de către Steve Hutchesson [MASM32]. Toate exemplele prezentate în cele ce urmează folosesc versiunea 8 a acestui asamblor și funcționează începând cu Windows 2000.

MASM pune la dispoziție câteva macrouri care fac mult mai ușoară programarea în limbaj de asamblare. Câteva dintre ele sunt următoarele:

```
.if, .else, .endif  
.while, .break, .endw
```

Similar cu limbajele de programare de nivel înalt, MASM permite definirea de funcții pentru a ajuta la o mai bună structurare a codului. Sintaxa este descrisă în continuare:

```
<name> proc <var1>:<var1 type>, <var2>:<var2 type>, ...
<function code>
ret
<name> endp
```

Valoarea returnată de către funcție va fi pusă în registrul EAX. Funcția este apelată astfel:

```
invoke <name>, param1, param2, param3
```

Valoarea returnată de către funcție poate fi obținută prin:

```
mov RetVal, eax
```

Invoke nu face altceva decât să pună parametrii în stivă iar apoi să apeleze cu call respectiva funcție, ca în exemplul următor:

```
push param3
push param2
push param1
call name
mov RetVal, eax
```

Variabilele sunt alocate în memorie. Există două tipuri de variabile: variabile globale și variabile locale. Variabilele globale sunt declarate în secțiunea .data dacă sunt inițializate, în secțiunea .data? dacă sunt neinițializate sau în secțiunea .const dacă sunt inițializate și valoarea lor nu se va schimba. Variabilele locale sunt declarate în interiorul funcțiilor și nu pot fi inițializate în momentul creării lor. Sintaxa lor este:

```
local <name>:<type>
```

11.4. MODALITĂȚI DE FOLOSIRE A LIMBAJULUI DE ASAMBLARE SUB WINDOWS

11.4.1. Programe scrise în întregime în limbaj de asamblare

Exemplul 11.4.1.1.

Vom scrie pentru început cel mai comun program – „Hello World!”. Folosiți un editor de text pentru a crea un fișier cu numele „hello.asm” compus dintr-un segment de date și un segment de cod:

```
.386
.model flat, stdcall
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

.data
HelloWorld db "Hello World!", 0
.code
start:
    invoke MessageBox, NULL, addr HelloWorld, addr HelloWorld, MB_OK
    invoke ExitProcess, 0
end start
```

Introduceți următoarele comenzi la linia de comandă:

```
\masm32\bin\ml /c /coff hello.asm
\masm32\bin\link /subsystem:windows hello.obj
```

Prima comandă are ca și efect asamblarea fișierului sursă, în urma căreia va rezulta fișierul cu extensia .obj. Prezența opțiunii /c cauzează invocarea doar a asamblorului, nu și a link-editorului. Opțiunea /coff se referă la formatul fișierului .obj obținut (format COFF, Common Object File Format). Fișierul .obj astfel obținut va conține datele și instrucțiunile în format binar.

A doua comandă efectuează link-editarea, în urma căreia va rezulta un fișier cu extensia .exe. Opțiunea /subsystem:windows informează link-editorul ce fel de program executabil este cel obținut. Rulând programul astfel obținut, se va afișa un MessageBox care va conține textul „Hello world!”.

Să luăm linie cu linie acest fișier:

```
.386
Prezența acestei directive anunță asamblorul să folosească setul de instrucțiuni 386.
```

```
.model flat, stdcall
```

- Directiva .model specifică modelul de memorie al programului.
- flat este modelul pentru programele Windows (nu va mai exista distincție între adresare far și near).
- stdcall se referă la metoda de transmitere a parametrilor folosită de funcțiile Windows. Acest lucru înseamnă că parametrii vor fi puși în stivă de la dreapta spre stânga și programul apelat este responsabil cu eliberarea stivei (similar regulilor limbajului C).

```
option casemap :none
```

Această opțiune este activată pentru a face diferență între literele mari și cele mici (*case sensitive*). Opțiunea complementară (*case insensitive*) este option casemap :all.

```
include \masm32\include\windows.inc
```

```
include \masm32\include\kernel32.inc
```

```
include \masm32\include\user32.inc
```

Fisierele care trebuie incluse pentru programele Windows.

- windows.inc conține declarațiile pentru constantele și definițiile Win32API;
- kernel32.inc conține funcția ExitProcess, necesară pentru terminarea unui proces;
- user32.inc conține funcția MessageBox, utilizată aici pentru afișarea textului dorit.

```
includelib \masm32\lib\kernel32.lib
```

```
includelib \masm32\lib\user32.lib
```

Bibliotecile de care funcțiile de mai sus au nevoie pentru a putea fi folosite sunt de asemenea incluse.

```
.data
```

Această directivă este folosită pentru declararea segmentului de date care conține toate datele inițializate folosite de către program.

```
HelloWorld db "Hello World!", 0
```

db definește variabila HelloWorld ca având valoarea „Hello world!”, string terminat cu caracterul NUL (string ANSI).

```
.code
```

Această directivă marchează începutul codului programului.

```
start:
```

Codul întregului program va fi scris între această etichetă și end start.

```
invoke MessageBox, NULL, addr HelloWorld, addr HelloWorld, MB_OK
```

invoke apeleză o funcție urmată de parametrii acestei funcții. Efectul este apelul funcției MessageBox cu parametrii săi, care va afișa pe ecran un MessageBox având ca și titlu și conținut textul „Hello World!”

```
invoke ExitProcess, 0
```

Apelul acestei funcții cauzează terminarea procesului cu codul de return 0.

Se poate observa la apelul funcției MessageBox modul în care adresa stringului HelloWorld este transmisă ca și parametru. Este momentul să facem distincția între **addr** și **offset**. Diferența principală este faptul că **offset** poate să obțină doar adresa variabilelor globale, în timp ce **addr** poate să obțină atât adresa variabilelor globale cât și pe aceea a variabilelor locale.

Exemplul 11.4.1.2.

Vom prezenta și explica în continuare un program care creează o fereastră simplă.

Poate fi observată structura generală a unei aplicații Windows, și anume cele două părți principale: WinMain și WndProc. WinMain creează fereastra și conține bucla de mesaje. WndProc este funcția care tratează evenimentele.

```
.386
```

```
.model flat, stdcall
```

```
option casemap :none
```

```
include \masm32\include\windows.inc
```

```
include \masm32\include\user32.inc
```

```
include \masm32\include\kernel32.inc
```

```
includelib \masm32\lib\user32.lib
```

```
includelib \masm32\lib\kernel32.lib
```

Necesitatea prezenței acestor directive și opțiuni a fost explicată în exemplul anterior, 11.4.1.1.

```
WinMain proto :DWORD, :DWORD, :DWORD, :DWORD
```

Acesta este prototipul unei funcții care ne va permite apelul funcției WinMain.

```
.data
```

```
ClassName db "WinClass", 0
```

```
AppName db "Simple Window", 0
```

În această secțiune de date sunt declarate două variabile de tip string: **ClassName**, care este numele clasei fereastră, și **AppName** care reprezintă numele ferestrei. Variabilele sunt inițializate.

```
.data?
```

```
hInstance HINSTANCE ?
```

Variabila **hInstance** va conține identificatorul instanței programului. O vom folosi ca și argument la apelul funcției CreateWindow. Directiva **.data?** Este folosită pentru declararea datelor neinițializate.

```
.code
```

```
start:
```

```
invoke GetModuleHandle, NULL ; la începutul segmentului de cod, se obține în eax  
; identificatorul instanței programului prin apelul  
; funcției GetModuleHandle .
```

```
mov hInstance, eax ; variabila hInstance se inițializează cu  
; identificatorul returnat de către funcția GetModuleHandle  
invoke WinMain, hInstance, NULL, NULL, 0 ; apelul funcției WinMain  
invoke ExitProcess, eax ; terminarea aplicației
```

```
WinMain proc hInst:INSTANCE, hPrevInst:INSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
local wc:WNDCLASSEX
local msg:MSG
local hwnd:HWND
```

Se continuă cu declararea și descrierea funcției WinMain. Directiva local aloca spațiu în stivă pentru variabilele folosite în funcție. Sunt declarate trei variabile locale: wc, msg, și hwnd.

- WC ce conține clasa fereastră (*window class*) pe care o creăm. O astfel de clasă reprezintă schema specificațiilor pentru o fereastră (*window*), definind caracteristici importante pentru aceasta.
- msg va conține mesajele pe care bucla de mesaje le captează.
- hwnd conține identificatorul ferestrei.

```
mov wc.cbSize, SIZEOF WNDCLASSEX
mov wc.style, CS_HREDRAW or CS_VREDRAW
mov wc.lpfnWndProc, offset WndProc
mov wc.cbClsExtra, NULL
mov wc.cbWndExtra, NULL
push hInst
pop wc.hInst
mov wc.hbrBackground, COLOR_WINDOW+1
mov wc.lpszMenuName, NULL
mov wc.lpszClassName, offset ClassName
invoke LoadIcon, NULL, IDI_APPLICATION
mov wc.hIcon, eax
mov wc.hIconSm, eax
invoke LoadCursor, NULL, IDC_ARROW
mov wc.hCursor, eax
invoke RegisterClassEx, addr wc
```

Acstea linii de cod completează câmpurile structurii WC declarate anterior. Are loc apoi apelul funcției RegisterClassEx, primind wc ca și argument, pentru a înregistra clasa fereastră.

Câmpurile structurii WNDCLASSEX sunt descrise în cele ce urmează:

```
WNDCLASSEX STRUCT DWORD
cbSize DWORD ?
style DWORD ?
lpfnWndProc DWORD ?
cbClsExtra DWORD ?
cbWndExtra DWORD ?
hInst DWORD ?
```

```
hIcon DWORD ?
hCursor DWORD ?
hbrBackground DWORD ?
lpszMenuName DWORD ?
lpszClassName DWORD ?
hIconSm DWORD ?
WNDCLASSEX ENDS
```

- cbSize: Mărimea în octeți a structurii WNDCLASSEX. Pentru obținerea valorii sale poate fi folosit operatorul SIZEOF.
- style: Stilul ferestrei create din această clasă. Mai multe stiluri pot fi combinate folosind operatorul "or".
- lpfnWndProc: Adresa procedurii responsabile de fereastra creată.
- cbClsExtra: Numărul de octeți suplimentari alocați după structura clasei fereastră. Octeții vor fi inițializați de către sistemul de operare cu valoarea zero. Acești octeți suplimentari pot fi folosiți pentru stocarea unor informații referitoare la clasa fereastră.
- cbWndExtra: Numărul de octeți suplimentari alocați după instanța ferestrei. Acești octeți vor fi inițializați de către sistemul de operare cu valoarea zero. Dacă o aplicație folosește structura WNDCLASS pentru a înregistra o fereastra de dialog creată folosind directiva CLASS în fișierul resursă, trebuie să seteze acest membru al structurii la DLGWINDOWEXTRA.
- hInst: Identificatorul instanței programului.
- hIcon: Identificatorul icon-ului ferestrei. Poate fi obținut prin apelul funcției LoadIcon.
- hCursor: Identificatorul cursorului ferestrei. Poate fi obținut prin apelul funcției LoadCursor.
- hbrBackground: Culoarea de fond a ferestrelor create.
- lpszMenuName: Identificator implicit al meniului ferestrelor create.
- lpszClassName: Numele clasei fereastră.
- hIconSm: Identificatorul unui icon mic, asociat cu clasa fereastră.

```
invoke CreateWindowEx, 0, addr ClassName, addr AppName,
WS_OVERLAPPEDWINDOW or WS_VISIBLE, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL,
NULL, hInst, NULL
mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
```

După înregistrarea clasei fereastră, crearea efectivă a ferestrei este realizată prin apelul funcției CreateWindowEx. Parametrii acestei funcții specifică modul de creare a ferestrei. Identificatorul ferestrei va fi returnat și reținut în variabila hwnd. Fereastra astfel creată nu va fi afișată pe ecran decât după apelul funcției ShowWindow, care primește ca și argumente identificatorul ferestrei și modul de afișare.

Funcția CreateWindowEx are 12 parametri. Acești parametri sunt prezenți în cele ce urmează:

```
CreateWindowExA proto dwExStyle:DWORD,\  
lpClassName:DWORD,\  
lpWindowName:DWORD,\  
dwStyle:DWORD,\  
X:DWORD,\  
Y:DWORD,\  
nWidth:DWORD,\  
nHeight:DWORD,\  
hWndParent:DWORD ,\  
hMenu:DWORD,\  
hInstance:DWORD,\  
lpParam:DWORD
```

Vom examina în continuare fiecare parametru:

- **dwExStyle:** Extra stil pentru fereastră.
- **lpClassName:** Adresa sirului ASCIIZ care conține numele clasei fereastră pe care dorîți să o folosiți ca şablon. Este obligatorie specificarea acestei valori.
- **lpWindowName:** Adresa sirului ASCIIZ care conține numele ferestrei. Aceasta va apărea în bara de titlu a ferestrei.
- **dwStyle:** Stilul ferestrei.
- X,Y: Coordonatele colțului stânga sus al ferestrei. Implicit, aceste valori sunt CW_USEDEFAULT, ceea ce înseamnă că Windows va decide unde să se afișeze fereastra pe ecran.
- **nWidth, nHeight:** Lățimea și înălțimea ferestrei în pixeli. Există o valoare implicită și pentru acest parametru, CW_USEDEFAULT, ceea ce înseamnă că Windows va alege cele mai potrivite valori.
- **hWndParent:** Identificator către părintele ferestrei. Acest parametru informează dacă respectiva fereastra este subordonată altor ferestre.
- **hMenu:** Identificator către meniul ferestrei. Dacă se folosește meniul clasei respective, atunci acest parametru va avea valoare NULL.
- **hInstance:** Identificatorul instanței programului care creează fereastra.
- **lpParam:** Pointer optional către o structură de date trimisă ferestrei. Fereastra poate obține valoarea acestui parametru apelând funcția GetWindowLong.

```
.while TRUE  
    invoke GetMessage, addr msg, NULL, 0, 0  
    .break .if (!eax)  
    invoke TranslateMessage, addr msg  
    invoke DispatchMessage, addr msg  
.endw
```

Aceasta este bucla de mesaje care verifică în mod continuu mesajele primite din partea sistemului de operare apelând funcția **GetMessage**. Apoi, **TranslateMessage** va transforma intrările de la tastatură în noi mesaje pe care le pune în coada de mesaje. **DispatchMessage** trimite mesajele procedurii **WndProc**, unde sunt procesate.

```
mov eax, msg.wParam  
ret  
WinMain endp
```

Valoarea returnată de către funcție este reținută în **msg.wParam**.

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM  
.if uMsg == WM_DESTROY  
    invoke PostQuitMessage, 0
```

WM_DESTROY este mesajul trimis după ce fereastra aplicației a fost închisă (a fost ștearsă de pe ecran). Funcția **PostQuitMessage** pune un mesaj **WM_QUIT** în coada de mesaje a aplicației, anunțând terminarea aplicației. Atunci când va primi mesajul **WM_QUIT**, aplicația va ieși din bucla de mesaje.

```
.else  
    invoke DefWindowProc, hWnd, uMsg, wParam, lParam
```

Această funcție apelează procedura implicită de procesare a mesajelor pentru toate mesajele pe care aplicația nu le procesează.

```
ret  
.endif  
xor eax, eax  
ret  
WndProc endp
```

În această funcție sunt procesate mesajele. În cazul nostru, singurul mesaj care trebuie procesat este **WM_DESTROY**, care apelează funcția **PostQuitMessage** pentru ieșire.

```
end start
```

11.4.2. Inserare de cod sursă asamblare în cadrul limbajelor de nivel înalt (studiu de caz – Visual C++)

Exemplul 11.4.2.1. Asamblorul inline Visual C++

Asamblorul inline poate fi folosit pentru inserarea codului scris în limbaj de asamblare în interiorul programelor scrise în Visual C++. În acest scop, nu vor fi necesari pași suplimentari de asamblare sau link-editare, deci nu vom avea nevoie de un asamblor separat. Asamblorul inline este invocat de prezența cuvântului rezervat `_asm`. Acest cuvânt cheie va fi urmat de o instrucție scrisă în limbaj de asamblare sau un grup de instrucții încadrate de acolade.

În continuare este prezentat un scurt fragment de cod asamblare inclus între acolade:

```
_asm
{
    mov eax, 2
    mov ebx, 10
}
```

Același cod poate fi scris astfel:

```
_asm mov eax, 2
      _asm mov ebx, 10
```

Vom prezenta în continuare o parte dintr-o aplicație Win32, pentru a exemplifica utilizarea asamblorului inline Visual C++, și anume procedura `WndProc`, care procesează toate mesajele pentru fereastra principală. Aplicația afișează într-o fereastră puterea k ($k \leq 255$) a lui 2, apelând pentru aceasta o funcție scrisă în limbaj de asamblare, care primește ca și argument puterea k . Funcția este descrisă în același fișier sursă care conține și descrierea procedurii `WndProc`.

```
.....
int power2(int k)          //funcția calculează și returnează puterea k a lui 2
{
    _asm
    {
        mov eax, 2
        mov cx, k           //în CX se pune argumentul funcției
        dec cx              //print-o shiftare la stânga a lui eax cu (cx-1) poziții se obține
        shl eax, cl         //puterea k a lui 2
    }
    //rezultatul obținut va fi returnat prin intermediul lui eax
.....
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
    LPARAM lParam)
```

//Argumentele acestei proceduri sunt: hWnd – identificator al ferestrei; message – mesaj; wParam și lParam – parametri care diferă în funcție de mesaj;

```
{ int wmid, wmEvent;
    char buf[20];
    char *sir;
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR szHello[MAX_LOADSTRING];
    switch (message)
    {
        case WM_COMMAND:           // WM_COMMAND- procesează meniu aplicației
            wmid = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            switch (wmid)
            {
                case IDM_ABOUT:
                    DialogBox(hInst,           (LPCTSTR)IDD_ABOUTBOX, hWnd,
                               (DLGPROC)About);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:             // WM_PAINT – Scrie în fereastra principală
            hdc = BeginPaint(hWnd, &ps);
            RECT rt;
            GetClientRect(hWnd, &rt);
            sir=(char *)malloc(20);
            sir=itoa(power2(4),buf,10); //aici are loc apelul funcției scrise în limbaj
                                         //de asamblare
            DrawText(hdc, sir, strlen(sir), &rt, DT_CENTER);
                                         //rezultatul întors de către funcție este afișat în
                                         //fereastră
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:           //WM_DESTROY – afișează un mesaj și
            PostQuitMessage(0);
            break;
    }
}
```

```

default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}

```

11.4.3. Programare multimodul

Exemplul 11.4.3.1. Crearea unui proiect Visual C++/Asamblare

Programarea multimodul ia în Visual C++ forma mai multor fișiere sursă conținute într-un același proiect. Putem avea astfel unul sau mai multe fișiere scrise în limbaj de asamblare (MASM) în același proiect cu unul sau mai multe fișiere scrise în Visual C++. Programarea mixtă este posibilă deoarece toate limbajele Microsoft implementează subrutele foarte asemănător. În ceea ce privește modul de apel al subrutinelor, compilatorului trebuie să i se precizeze ce fel de convenții de apel să folosească.

Pentru exemplificarea programării multimodul Visual C++/asamblare, vom dezvolta aceeași aplicație Win32 descrisă în exemplul 11.4.2.1., cu diferența că funcția scrisă în limbaj de asamblare este descrisă într-un fișier sursă separat (având extensia .asm) de cel care conține descrierea procedurii WndProc. Vom prezenta doar procesarea mesajului WM_PAINT, care conține apelul funcției scrisă în asamblare, aceasta fiind singura diferență față de codul procedurii WndProc descris în cadrul exemplului 11.4.2.1.

```

extern "C" int power2(int a);
//declarația extern „C” anunță compilatorul să folosească convențiile de apel C. Aceste convenții
//se referă la modul de transmitere a parametrilor, la faptul dacă funcția apelantă sau funcția
//apelată are responsabilitatea salvării și restaurării regiștrilor. De asemenea, se anunță faptul că
//funcția power2 este definită într-un modul extern și folosită în acest modul.

```

```

.....
case WM_PAINT:           // WM_PAINT – Scrie în fereastra principală
    hdc = BeginPaint(hWnd, &ps);
    RECT rt;
    GetClientRect(hWnd, &rt);
    sir=(char *)malloc(20);
    sir=itoa(power2(4),buf,10); //aici are loc apelul funcției scrise în limbaj de
                                //asamblare
    DrawText(hdc, sir, strlen(sir), &rt, DT_CENTER);
                                //rezultatul întors de funcție este afișat în fereastră
    EndPaint(hWnd, &ps);
    break;
}

```

Se va adăuga la proiect un fișier sursă cu extensia .asm, care va conține funcția scrisă în limbaj de asamblare. Pentru acest fișier avem următorul cod:

.586	specificarea setului de instrucțiuni folosit
.MODEL FLAT, C	;folosirea modelului de memorie FLAT
.CODE	
power2 PROC	
push ebp	
mov ebp, esp	;cod de intrare (izolarea stivei)
mov eax, 2	
mov cx, [ebp+8]	;argumentul k al funcției
dec cx	;printr-o shiftare la stânga a lui eax cu (cx-1) poziții se
shl eax, cl	;obține puterea k a lui 2
pop ebp	;cod de ieșire (refacerea stivei)
ret	
power2 ENDP	;în eax se returnează puterea calculată
END	

Visual C++ nu compilează codul sursă scris în limbaj de asamblare. Pentru aceasta, fișierul sursă asamblare va trebui asamblat separat. Se va folosi în acest scop asamblorul din MASM, ml.exe. Pentru a putea face acest lucru, este nevoie de adăugarea liniei de comandă. Se selectează tab-ul Custom Build iar în textbox-ul Commands se scrie:

```
\masm32\bin\ml /c /coff "$(OutDir)\$(InputPath).asm.obj" "$(InputPath)"
```

Stringul \$(inputpath) este un macro care se va înlocui cu calea spre fișierul care conține codul scris în limbaj de asamblare. Calea spre asamblorul ml.exe poate fi specificată explicit (ca și în exemplul de mai sus) sau poate fi adăugată în căile de căutare.

La Outputs se va scrie:

\$(OutDir)\\$(InputPath).asm.obj
pentru a preciza că rezultatul asamblării va fi un fișier .obj care va avea același nume ca și fișierul .asm. Acum proiectul poate fi compilat și link-editat.

Exemplul 11.4.3.2. Scrierea în limbaj de asamblare a unui DLL (Dynamic-Link Library) pentru Windows

DLL-urile nu rulează sub MS-DOS. Acestea sunt biblioteci speciale folosite în sistemul de operare Windows. MASM32 pune la dispozitie instrumente pentru scrierea DLL-urilor în limbaj de asamblare. Vom descrie în continuare modalitatea de scriere a unui DLL în limbaj de asamblare, punctând pe parcurs elementele distinctive ale acestui tip de bibliotecă.

Pentru ca orice limbaj care suportă DLL-uri să poată face aceste apeluri externe, Microsoft a descris mecanismul de comunicare între DLL-uri și alte module. În continuare sunt prezentate câteva dintre aceste reguli:

- procedurile și funcțiile cu număr fix de parametri folosesc mecanismul de apel Stdcall.
- procedurile și funcțiile cu număr variabil de parametri folosesc mecanismul de apel C.
- parametrii pot fi octeți, cuvinte, dublucuvinte, pointeri sau stringuri.

Procedurile Stdcall pun parametrii în stivă de la stânga spre dreapta. Procedura are responsabilitatea de a scoate din stivă parametrii.

Procedura de intrare trebuie să aibă 3 parametri de tip dublucuvânt, care sunt transmiși prin intermediul stivei. Această funcție trebuie să returneze valoarea TRUE în registrul eax, pentru a continua încărcarea DLL-ului după primirea acestor parametri. O procedură de intrare trebuie să conțină cel puțin următoarele:

```
LibMain proc par1:dword, par2:dword, par3:dword
    mov eax, true
    ret
LibMain endp
```

După funcția LibMain pot fi scrise procedurile de care avem nevoie în DLL, iar la sfârșitul acestor proceduri trebuie să folosim o directivă pentru a anunța sfârșitul fișierului sursă. Un DLL trebuie să aibă deci o procedură de început, iar dacă numele procedurii este LibMain, trebuie să terminăm codul cu end LibMain.

Exportarea procedurilor din DLL

Într-un DLL se face diferență între proceduri care pot fi accesate din exterior și proceduri interne. Acest lucru poate fi controlat prin intermediul unui fișier DEFINITION, care este folosit de către link-editor pentru a se specifica numele fiecărei proceduri care va fi exportată.

Asamblarea codului unui DLL este identică cu a unui fișier obișnuit și se obține astfel:
`\masm32\bin\ml /c /coff MyDll.asm`

Diferența apare însă la link-editare:

`\masm32\bin\link /subsystem:windows /dll /def:MyDll.def MyDll.obj`

Opcțunea /dll spune link-editorului să pună la începutul fișierului codul corect pentru un DLL; opțiunea /def:MyDll.def specifică fișierul unde se află lista procedurilor exportate și numele intern al DLL-ului.

Folosirea DLL-ului dintr-un program apelant

Există două metode care pot fi folosite de către un program care dorește să apeleze proceduri descrise într-un DLL. Alegerea uneia sau a celeilalte depinde de modul în care programul apelant folosește DLL-ul. În cazul în care codul urmează să fie folosit pentru un timp scurt, se recomandă o încărcare dinamică a DLL-ului, prin folosirea funcțiilor API LoadLibrary(), GetProcAddress() și FreeLibrary(). Această metodă are ca și efect un nivel redus de utilizare a memoriei.

Dacă o aplicație are nevoie de DLL pe întregul parcurs al execuției, atunci DLL-ul ar trebui încărcat la pornirea programului. Atunci când link-editorul construiește DLL-ul, mai construiește și o bibliotecă (fișier cu extensia .lib). Această bibliotecă va fi inclusă în faza de link-editare a unui program care dorește să folosească funcțiile DLL-ului. Pentru a putea folosi aceste funcții trebuie ca în programul apelant să se scrie prototipul fiecărei proceduri exportate din DLL și să se includă biblioteca respectivă.

Exemplu:

Vom construi un DLL care va conține o funcție pentru afișarea unui MessageBox cu titlu și textul transmis ca și argumente din programul apelant. Programul scris în limbaj de asamblare (MyDll.asm), din care se va obține DLL-ul, este descris în cele ce urmează:

```
.486
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

```
MyMessageBox PROTO STDCALL :DWORD, :DWORD, :DWORD, :DWORD
;prototipul funcției implementate
.code
```

```
LibMain proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
    mov eax, TRUE
    ret
LibMain Endp
```

```
MyMessageBox proc STDCALL hParent:DWORD, lpMsg:DWORD,
    lpTitle:DWORD, dlgStyle:DWORD, iconID:DWORD
;aceasta este funcția implementată pentru a fi apelată din alte programe
```

390 Arhitectura calculatoarelor. Limbajul de asamblare 80x86.

Modulul secundar (m2.asm), în care apare definită procedura afisare, este descris în continuare:

```
.386
.model flat, c
.data
sir db 'vreau sa afisez acest sir', 0
.code
afisare proc
    mov eax, offset sir
    ret
afisare endp
end
```

Cele două module vor fi asamblate separat:

```
\masm32\bin\ml /c /coff m1.asm
\masm32\bin\ml /c /coff m2.asm
```

După obținerea cu succes a formelor obiect, acestea vor fi link-editate folosind comanda:

```
\masm32\bin\link /subsystem:windows m1.obj m2.obj
```

Va rezulta executabilul m1.exe, prin a căruia rulare se va afișa pe ecran un MessageBox cu textul „vreau să afișez acest și”.

11.5 LIMBAJ DE ASAMBLARE VS. LIMBAJE DE NIVEL ÎNALT

Există numeroase discuții pe tema utilității limbajului de asamblare. Unul dintre avantajele scrierii programelor în limbaj de asamblare este durata execuției, deși unii afirmă că translatoarele moderne pentru limbi de nivel înalt pot genera cod care se execută la fel de rapid. Totuși, există contraexemple care dovedesc că unele calcule se pot efectua mult mai repede dacă sunt scrise în limbaj de asamblare.

Apoi, programarea de nivel scăzut se face de obicei mult mai ușor în asamblare decât în orice alt limbaj de nivel înalt. Cel mai important argument este faptul că unele acțiuni dependente de sistem și întreprinse de către sistemul de operare nu pot fi efectiv exprimate în limbi de nivel înalt.

Limbajul de asamblare este folosit adesea pentru interacțiuni de nivel scăzut între sistemul de operare și hardware, de exemplu în cadrul driverelor. O altă utilizare a limbajului de asamblare este în BIOS-ul unui calculator. Acest cod de nivel scăzut este folosit pentru inițializarea și testarea hardware-ului înainte de încărcarea sistemului de operare și se află rezident în cadrul memoriei ROM. Principalele funcții ale BIOS-ului sunt: furnizarea unui afișaj vizual a sistemului pe monitor la pornirea sistemului, accesul la tastatură și furnizarea de servicii de comunicare de nivel scăzut între componente hardware.

Bibliografie

- [Hyde03] Randall Hyde – *The Art of Assembly Programming*, No Starch Press Inc., 2003
- [Detmer01] Richard C. Detmer - *Introduction to 80X86 Assembly Language and Computer Architecture*, Jones and Bartlett Computer Science, 2001
- [Irvine03] Kip R. Irvine - *Assembly Language for Intel-Based Computers*, Prentice-Hall Inc., 2003
- [Dun00] Jeff Duntemann – *Assembly Language Step-By-Step. Programming with DOS and Linux*, John Wiley & Sons, 2000
- [Boian95] F.M.Boian, Al. Vancea, S. Iurian, M. Iurian – *Arhitectura 80x86. Limbaj de asamblare. Legătura între limbi*, Litografia UBB Cluj, 1995
- [Boian96] Florian Boian - *De la aritmetică la calculatoare*, Ed. Presa Universitară Clujeană, Cluj, 1996
- [Ath92] Irina Athanasiu, Al. Pănoiu - *Microprocesoarele 8086, 286, 386*, Ed. Teora, 1992
- [Musca98] Gh. Muscă – *Programare în limbaj de asamblare*, Ed. Teora, 1998
- [Lungu04] Vasile Lungu - Procesoare Intel. *Programare în limbaj de asamblare*, ed. Teora, 2004
- [Borland98] How to Avoid Memory Corruption,
<http://bdn.borland.com/article/0,1410,18049.00.html>
- [XMS30] eXtended Memory Specification (XMS), ver 3.0,
<http://www.qzx.com/pc-gpe/xms30.txt>
- [ALEXF] Tutorial și exemple simple de programe în mod protejat,
<http://alexfru.chat.ru>
- [PMI] Protected Mode Internals,
<http://www.internals.com/articles/articles.htm>
- [DPMI] DPMI 1.0 Programming API Specification,
<http://www.delorie.com/djgpp/doc/dpmi/>

- [VM86] Tim Robinson, Virtual 8086 Mode,
<http://osdev.berlios.de/v86.html>
- [WENGE] Billy Wenge-Murphy, Learning Assembly,
http://www.doorknobssoft.com/asm_tutorial.html
- [WIN32] Win32 Assembler Coding Tutorial,
<http://www.deinmeister.de/wasmtute.htm>
- [ASMTUT] Assembly Tutorial,
<http://www.xs4all.nl/~smit/asm01001.htm>
- [KETMAN] Ketman Assembly Language Tutorial,
<http://www.btinternet.com/~btketman/tutpage.html>
- [MASM32] Masm32
<http://www.masm32.com>
- [PC] PC Assembly Language,
<http://www.drpaulcarter.com/pcasm/>
- [ASM] Assembly Language,
<http://www.xploiter.com/mirrors/asm/astart.htm>

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

Arhitectura calculatoarelor

Limbajul de asamblare 80x86

